COMPSCI 220: Automata and Pattern Matching

Cristian S. Calude

Term 2 2009

R. Hamming

"The purpose of computing is insight, not numbers."

Thanks to

Elena Calude, Michael Dinneen, Nick Hay, and Radu Nicolescu for stimulating discussions and critical comments.

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler/`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler/`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler/`

Bibliography 3

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, MIT Press, 2001 (second ed.)
- M. Crochemore, W. Rytter. *Jewels of Stringology*, World Scientific, 2002.
- M. J. Dinneen, G. Gimel'farb, M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages*, Prentice Hall, 2009. (textbook)
- JFLAP, `http://www.jflap.org`. (simulation software)
- Regex simulator, `http://osteele.com/tools/reanimator`.
- Animation of the Aho-Corasick Automaton, `http://www-sr.informatik.uni-tuebingen.de/~buehler/`

Assignment and exam 4

- Assignment 1: 21 August 2009; 8:30pm (ADB time)
- Midterm test 18 August (in class): prepare all results discussed in class and tutorials

Assignment and exam 4

- Assignment 1: 21 August 2009; 8:30pm (ADB time)
- Midterm test 18 August (in class): prepare all results discussed in class and tutorials

Assignment and exam 4

- Assignment 1: 21 August 2009; 8:30pm (ADB time)
- Midterm test 18 August (in class): prepare all results discussed in class and tutorials

Question

Are there finite memory machines accepting as input finite binary sequences of any length and deciding whether the sequence has a certain property (for example, it has an even number of 0's)?

Using "states" to remember the 'property' seems a good idea, but don't we have to keep adding newer and newer 'states' as the input gets longer and longer?

Re-phrased: Is a finite memory enough? In general the answer seems to be negative, but . . .

Question

Are there finite memory machines accepting as input finite binary sequences of any length and deciding whether the sequence has a certain property (for example, it has an even number of 0's)?

Using "states" to remember the 'property' seems a good idea, but don't we have to keep adding newer and newer 'states' as the input gets longer and longer?

Re-phrased: Is a finite memory enough? In general the answer seems to be negative, but . . .

Question

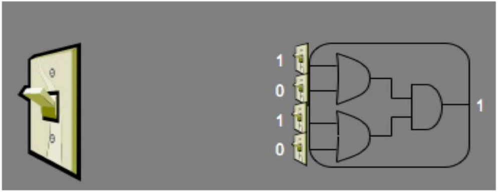
Are there finite memory machines accepting as input finite binary sequences of any length and deciding whether the sequence has a certain property (for example, it has an even number of 0's)?

Using "states" to remember the 'property' seems a good idea, but don't we have to keep adding newer and newer 'states' as the input gets longer and longer?

Re-phrased: Is a finite memory enough? In general the answer seems to be negative, but . . .

A simple example

Probably the simplest finite machine operates a switch as follows:

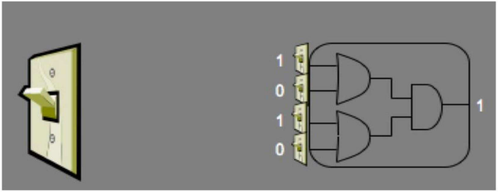


So, if the switch is down, then the light goes on and if the switch is up, then the light goes off.

To this device, the switch position is an input and the light on/off is the output. The machine works with finitely many "states" for any sequence of modifications of the switch.

A simple example

Probably the simplest finite machine operates a switch as follows:



So, if the switch is down, then the light goes on and if the switch is up, then the light goes off.

To this device, the switch position is an input and the light on/off is the output. The machine works with finitely many "states" for any sequence of modifications of the switch.

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$

- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$

- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0$, $|0| = |1| = 1$, $|00| = |01| = |10| = |11| = 2$, ...
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Bits, bit-strings, alphabets

- Bit strings: ε (empty string), $0, 1, 00, 01, 10, 11, \ldots$
- $B = \{0, 1\}$ and B^* is the set of all binary strings
- Strings can be concatenated: from x and y get xy
- The length of the string x is denoted by $|x|$
  - $|\varepsilon| = 0, |0| = |1| = 1, |00| = |01| = |10| = |11| = 2, \ldots$
  - $|xy| = |x| + |y|$
- Other alphabets: $\Sigma = \{a, b, c, d\}$, the set of 7-bit ASCII characters
- Σ^* is the set of all strings over the alphabet Σ

Deterministic finite automata

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

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

Deterministic finite automata

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

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

Deterministic finite automata

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

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

Deterministic finite automata

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

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

Deterministic finite automata

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

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

Deterministic finite automata

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

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

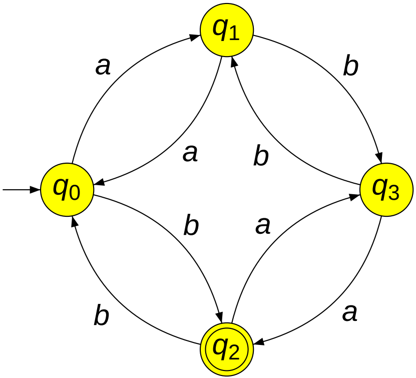
DFA: example 1

$M = (Q, \Sigma, \delta, s, F)$:

$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$

δ	Σ	
Q	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

$s = q_0$
$F = \{q_2\}$



DFA: accepted strings and language

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

such that

$$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$

- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M. If $A = L(M)$, for some DFA M, then A is called *regular*.

DFA: accepted strings and language

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states

  $$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

  $$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$

- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M; if $A = L(M)$, for some DFA M, then A is called *regular*.

DFA: accepted strings and language

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states

  $$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

  $$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$

- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M; if $A = L(M)$, for some DFA M, then A is called *regular*.

DFA: accepted strings and language

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states
$$s_1, s_2, \cdots, s_n, s_{n+1}$$
such that
$$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$
- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M; if $A = L(M)$, for some DFA M, then A is called *regular*.

DFA: accepted strings and language

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states
$$s_1, s_2, \cdots, s_n, s_{n+1}$$
such that
$$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$
- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M; if $A = L(M)$, for some DFA M, then A is called *regular*.

Questions

- Given a DFA M, check which strings M accepts.
- Given a language (set of strings) can we build a DFA M that recognises just them? If the answer is affirmative can we construct a minimal (in the sense of the number of states) DFA recognising the language?
- Which properties of DFAs can be checked algorithmically?

Questions

- Given a DFA M, check which strings M accepts.
- Given a language (set of strings) can we build a DFA M that recognises just them? If the answer is affirmative can we construct a minimal (in the sense of the number of states) DFA recognising the language?
- Which properties of DFAs can be checked algorithmically?

Questions

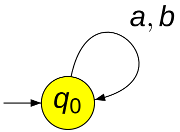
- Given a DFA M, check which strings M accepts.
- Given a language (set of strings) can we build a DFA M that recognises just them? If the answer is affirmative can we construct a minimal (in the sense of the number of states) DFA recognising the language?
- Which properties of DFAs can be checked algorithmically?

Questions

- Given a DFA M, check which strings M accepts.
- Given a language (set of strings) can we build a DFA M that recognises just them? If the answer is affirmative can we construct a minimal (in the sense of the number of states) DFA recognising the language?
- Which properties of DFAs can be checked algorithmically?

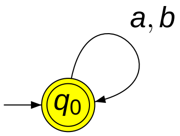
DFA: example 2

The language accepted by this DFA is empty, i.e. the DFA accepts no string.



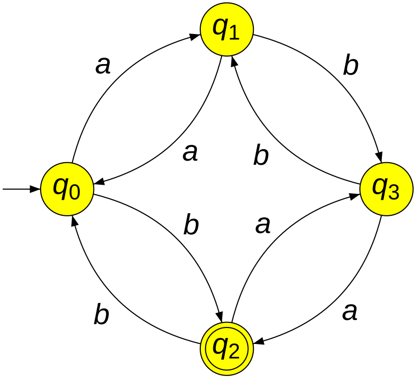
DFA: example 3

The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$, i.e. the language $\Sigma^* = \{a, b\}^*$.



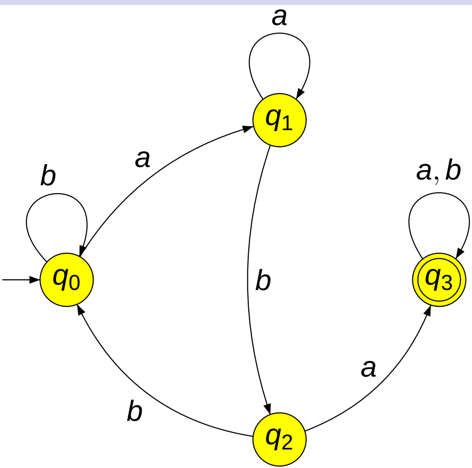
DFA: example 1 continued

The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which contain an even number of a's and an odd number of b's.



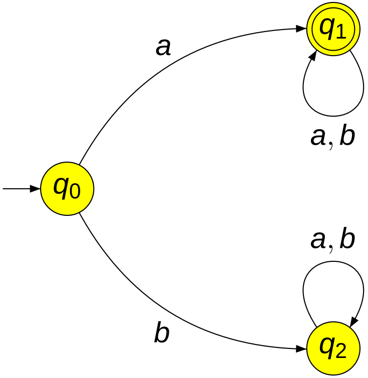
DFA: example 4

The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which contain the substring *aba*, i.e. all the strings of the form *uabav* with $u, v \in \{a, b\}^*$.



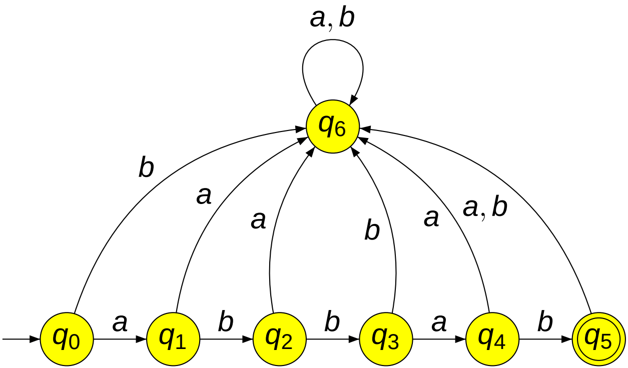
DFA: example 5

The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which start with a, i.e. all the strings of the form av, with $v \in \Sigma^* = \{a, b\}^*$.



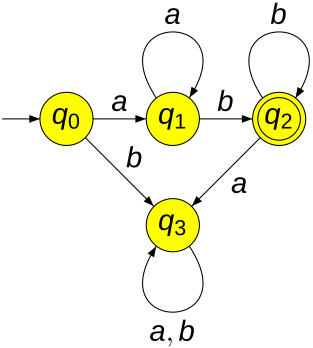
DFA: example 6

The language accepted by this DFA consists of only one string over $\Sigma = \{a, b\}$, namely *abbab*.



DFA: example 7

The language accepted by this DFA is $\{a^m b^n \mid m, n > 0\}$, where a^m means $aa \cdots a$ (m times).



Not all languages are accepted by DFAs

The language

$$L = \{a^n b^n \mid n > 0\}$$

is not accepted by any DFA.

Why?

Informally, because a DFA can 'count' only up to the number of its states.

More formally, because, if n is greater than the number of states of a DFA supposed to accept L, then any trace (path) labelled by a^n passes twice through some state. That is, the there are strings a^i and a^j for $i < j \leq n$ that fall into the same state. Thus both $a^i b^i$ and $a^j b^i$ are accepted/rejected which contradicts the definition of L.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 1

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

Simple properties of DFAs 2

- It is algorithmically decidable whether a DFA M accepts no string.
  Proof: Given the DFA M check whether there is a path from the initial state s (has a trace of a computation) to a final state in F. We have: $L(M) = \emptyset$ if and only if there there is no path from the initial state to a final state.

Simple properties of DFAs 2

- It is algorithmically decidable whether a DFA M accepts no string.
  Proof: Given the DFA M check whether there is a path from the initial state s (has a trace of a computation) to a final state in F.
  We have: $L(M) = \emptyset$ if and only if there there is no path from the initial state to a final state.

Simple properties of DFAs 3

- It is algorithmically decidable whether a DFA M accepts infinitely strings.

  Proof: Given the DFA M, $L(M)$ is infinite if and only if there is a path from the initial state (has a trace of a computation) s to a final state in F having the following additional property: some state q in the path possesses a loop, i.e. there is a path from q to q.

Simple properties of DFAs 3

- It is algorithmically decidable whether a DFA M accepts infinitely strings.

  Proof: Given the DFA M, $L(M)$ is infinite if and only if there is a path from the initial state (has a trace of a computation) s to a final state in F having the following additional property: some state q in the path possesses a loop, i.e. there is a path from q to q.

The reverse operation

The *reverse* of a string

$$w = c_1 c_2 c_3 \cdots c_n$$

is the string

$$R(w) = c_n c_{n-1} \cdots c_2 c_1.$$

For example, $R(abaaa) = aaaba$, $R(abba) = abba$, $R(bac) = cab$.

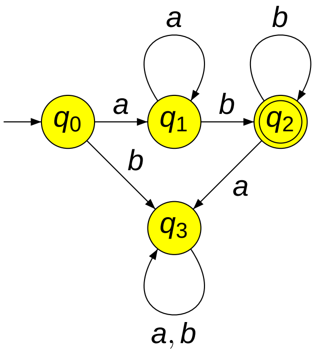
The *reverse* of a language A is the language

$$R(A) = \{R(w) \mid w \in A\}.$$

Problem: Is $R(A)$ regular whenever A is regular?

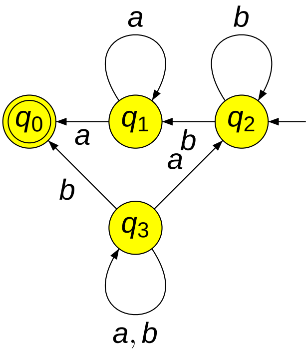
DFA: example 7 revisited

The language accepted by the DFA M is $A = \{a^m b^n \mid m, n > 0\}$.
Is $R(A) = \{b^n a^m \mid m, n > 0\}$ regular?

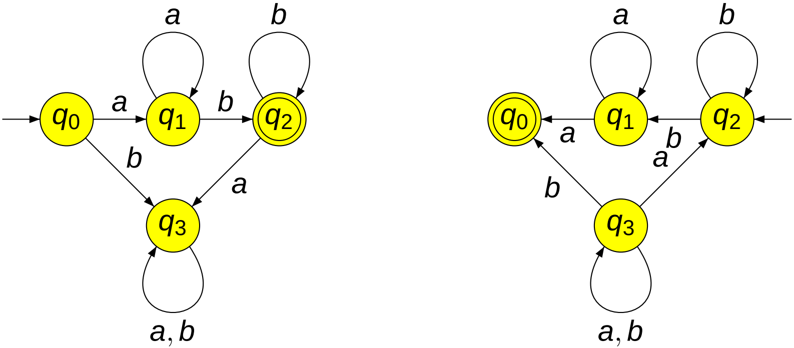


A possible solution?

Is $R(A) = \{b^m a^n \mid m, n > 0\}$ accepted by this machine, M'?



The solution 'under microscope': M vs M' 1



The solution 'under microscope': M vs M' 2

What did we do, in more general terms?

1. The initial state of M becomes the accept state of M'.
2. Every accept state of M becomes an initial state of M'.
3. If $\delta(q_1, c) = q_2$ is in M then $\delta(q_2, c) = q_1$ is in M'. That is, all transitions are reversed.

The solution 'under microscope': M vs M' 2

What did we do, in more general terms?

1. The initial state of M becomes the accept state of M'.
2. Every accept state of M becomes an initial state of M'.
3. If $\delta(q_1, c) = q_2$ is in M then $\delta(q_2, c) = q_1$ is in M'. That is, all transitions are reversed.

The solution 'under microscope': M vs M' 2

What did we do, in more general terms?

1. The initial state of M becomes the accept state of M'.
2. Every accept state of M becomes an initial state of M'.
3. If $\delta(q_1, c) = q_2$ is in M then $\delta(q_2, c) = q_1$ is in M'. That is, all transitions are reversed.

The solution 'under microscope': M vs. M' 3

Do we have a problem with M'?
Answer: yes: M' is not a DFA!

Still, the procedure seems reasonable!

What should we do? Well, let's examine another example.

The solution 'under microscope': M vs. M' 3

Do we have a problem with M'?
Answer: yes: M' is not a DFA!

Still, the procedure seems reasonable!

What should we do? Well, let's examine another example.

The solution 'under microscope': M vs. M' 3

Do we have a problem with M'?
Answer: yes: M' is not a DFA!

Still, the procedure seems reasonable!

What should we do? Well, let's examine another example.

The solution 'under microscope': M vs. M' 3

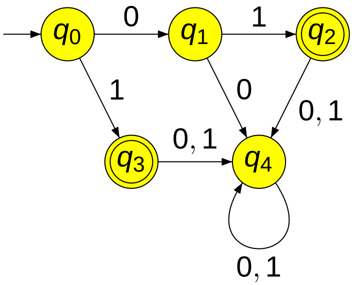
Do we have a problem with M'?
Answer: yes: M' is not a DFA!

Still, the procedure seems reasonable!

What should we do? Well, let's examine another example.

The solution 'under microscope' 4

Transforming this DFA M into M' produces:
a) two initial states: q_2, q_3
b) multiple transitions with the same label (e.g. $\delta(q_4, 0) = \{q_1, q_2, q_3, q_4\}$)



Nondeterministic finite automata

Should we abandon the transformation $M \to M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \rightarrow M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \rightarrow M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \rightarrow M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \to M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \rightarrow M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \to M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Nondeterministic finite automata

Should we abandon the transformation $M \to M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

NFA: accepted strings and language

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

$$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The language accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

NFA: accepted strings and language

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

$$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The *language* accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

NFA: accepted strings and language

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

$$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The *language* accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

NFA: accepted strings and language

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

$$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The *language* accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

NFA: accepted strings and language

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

$$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The *language* accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

NFA: comments

- The state transition function δ is more general for NFAs than DFAs. Besides having transitions to multiple states for a given input symbol, we can have $\delta(q, c)$ empty (undefined) for some $q \in Q$ and $c \in \Sigma$. This means that that we can design automata such that no state moves are possible for when in some state q and the next character read is c (that is, the human designer does not have to worry about all cases).
- Every DFA can be viewed as a special case of an NFA.

NFA: comments

- The state transition function δ is more general for NFAs than DFAs. Besides having transitions to multiple states for a given input symbol, we can have $\delta(q, c)$ empty (undefined) for some $q \in Q$ and $c \in \Sigma$. This means that that we can design automata such that no state moves are possible for when in some state q and the next character read is c (that is, the human designer does not have to worry about all cases).
- Every DFA can be viewed as a special case of an NFA.

NFA: comments

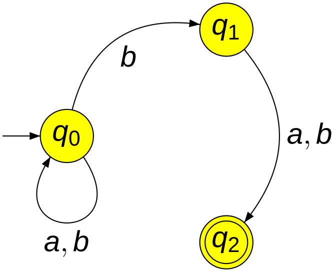
- The state transition function δ is more general for NFAs than DFAs. Besides having transitions to multiple states for a given input symbol, we can have $\delta(q, c)$ empty (undefined) for some $q \in Q$ and $c \in \Sigma$. This means that that we can design automata such that no state moves are possible for when in some state q and the next character read is c (that is, the human designer does not have to worry about all cases).
- Every DFA can be viewed as a special case of an NFA.

NFA: example 1 1

$\Sigma = \{a, b\}$

δ States	Σ a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

$S = \{q_0\}$
$F = \{q_2\}$



NFA: example 1 2

- The string *aba* is accepted: there are two traces,

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2$$

- The string *baa* is not accepted: there are two traces,

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} ?$$

- The language accepted by this NFA is

$$\{uba, ubb \mid u \in \{a, b\}^*\}.$$

NFA: example 1 2

- The string *aba* is accepted: there are two traces,

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} \color{red}{q_2}$$

- The string *baa* is not accepted: there are two traces,

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} ?$$

- The language accepted by this NFA is

$$\{uba, ubb \mid u \in \{a, b\}^*\}.$$

NFA: example 1 2

- The string *aba* is accepted: there are two traces,

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} {\color{red}q_2}$$

- The string *baa* is not accepted: there are two traces,

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} ?$$

- The language accepted by this NFA is

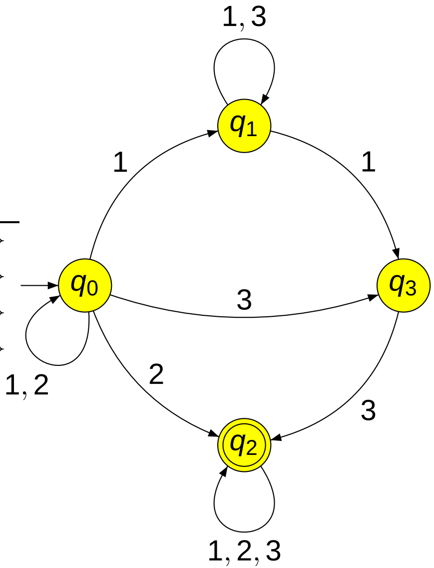
$$\{uba, ubb \mid u \in \{a, b\}^*\}.$$

NFA: example 2

$\Sigma = \{1, 2, 3\}$

δ	Σ		
States	1	2	3
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_3\}$
q_1	$\{q_1, q_3\}$	$\emptyset$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
q_3	$\emptyset$	$\emptyset$	$\{q_2\}$

$S = \{q_0\}$
$F = \{q_2\}$



NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 1

Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

NFA=DFA 2

Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

Note: the algorithm `NFAtoDFA` follows the above construction, but eliminates all non-reachable states.

NFA=DFA 2

Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

Note: the algorithm `NFAtoDFA` follows the above construction, but eliminates all non-reachable states.

NFA=DFA 2

Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

Note: the algorithm `NFAtoDFA` follows the above construction, but eliminates all non-reachable states.

NFA=DFA 2

Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

Note: the algorithm `NFAtoDFA` follows the above construction, but eliminates all non-reachable states.

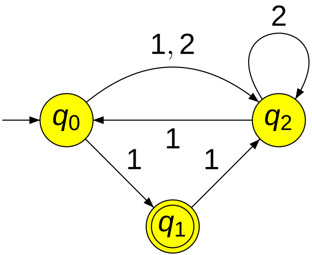
NFA=DFA 2

Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

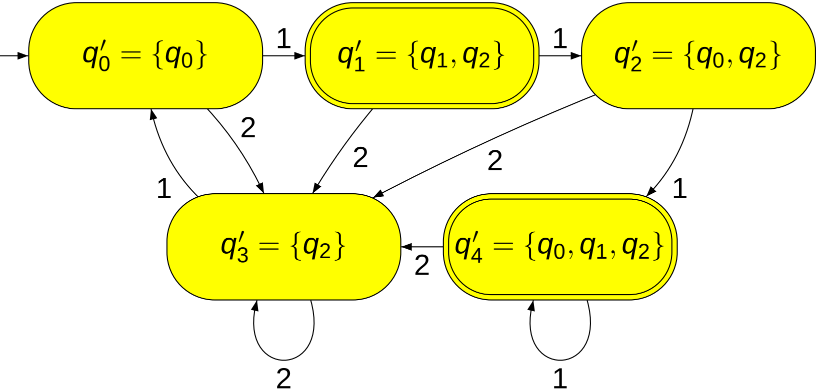
Note: the algorithm `NFAtoDFA` follows the above construction, but eliminates all non-reachable states.

NFAtoDFA: an example 1



The NFA N

NFAtoDFA: an example 2



Equivalent DFA M

Closure properties of regular languages 1

- The union of two regular languages is also regular.

  Proof: Given two NFAs N_A, N_B with no common states such that $A = L(N_A), B = L(N_B)$, the NFA N consisting of the union of all components of N_A, N_B recognises $A \cup B$.

  More precisely, if $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ with $Q_A \cap Q_B = \emptyset$, then $A \cup B$ is recognised by the NFA

  $$N = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B).$$

- The intersection of two regular languages is also regular.

  Proof: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

Closure properties of regular languages 1

- The union of two regular languages is also regular.
  Proof: Given two NFAs N_A, N_B with no common states such that $A = L(N_A), B = L(N_B)$, the NFA N consisting of the union of all components of N_A, N_B recognises $A \cup B$.

  More precisely, if $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ with $Q_A \cap Q_B = \emptyset$, then $A \cup B$ is recognised by the NFA

$$N = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B).$$

- The intersection of two regular languages is also regular.
  Proof: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

Closure properties of regular languages 1

- The union of two regular languages is also regular.
  Proof: Given two NFAs N_A, N_B with no common states such that $A = L(N_A), B = L(N_B)$, the NFA N consisting of the union of all components of N_A, N_B recognises $A \cup B$.

  More precisely, if $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ with $Q_A \cap Q_B = \emptyset$, then $A \cup B$ is recognised by the NFA

$$N = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B).$$

- The intersection of two regular languages is also regular.
  Proof: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

Closure properties of regular languages 1

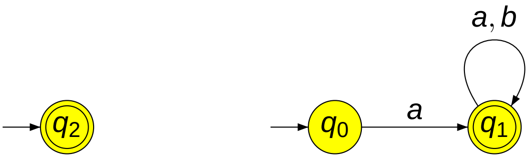
- The union of two regular languages is also regular.
  Proof: Given two NFAs N_A, N_B with no common states such that $A = L(N_A), B = L(N_B)$, the NFA N consisting of the union of all components of N_A, N_B recognises $A \cup B$.

  More precisely, if $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ with $Q_A \cap Q_B = \emptyset$, then $A \cup B$ is recognised by the NFA

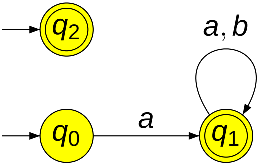
$$N = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B).$$

- The intersection of two regular languages is also regular.
  Proof: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

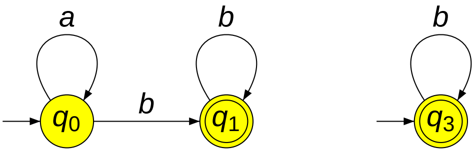
Closure under union: an example 1



Closure under union: an example 2

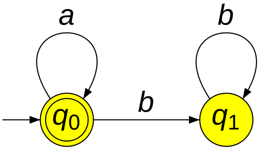


Closure under intersection: an example 1



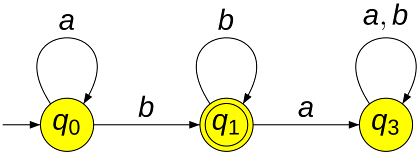
NFA N_1 NFA N_2

Closure under intersection: an example 2



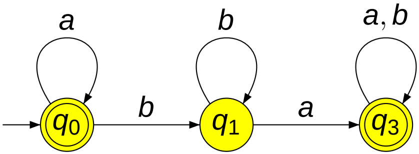
NFA accepting the complement of N_1?

Closure under intersection: an example 3



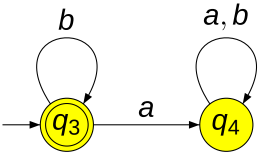
DFA M_1 equivalent to N_1

Closure under intersection: an example 4



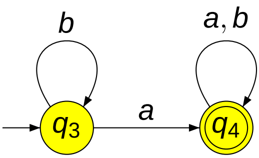
DFA $\overline{M}_1$ recognising the complement of M_1

Closure under intersection: an example 5



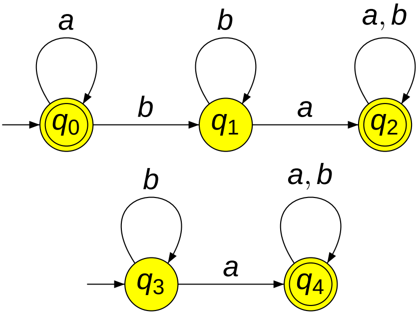
DFA M_2 equivalent to N_2

Closure under intersection: an example 6



DFA $\overline{M}_2$ recognising the complement of M_2

Closure under intersection: an example 7



NFA N_3 recognising $L(\overline{M}_2) \cup L(\overline{M}_2)$

Closure under intersection: an example 8

Last two steps:

- Construct a DFA M_3 equivalent to the NFA N_3
- Construct the complement of $L(M_3) = L(N_1) \cap L(N_2) = \{b^k \mid k \geq 1\}$

Recap:

- $L(N_1) = \{a^n b^m \mid n \geq 0, m \geq 1\}$
- $L(N_2) = \{b^m \mid m \geq 0\}$
- $L(M_3) = \{b^k \mid k \geq 1\}$

Closure under intersection: an example 8

Last two steps:

- Construct a DFA M_3 equivalent to the NFA N_3
- Construct the complement of $L(M_3) = L(N_1) \cap L(N_2) = \{b^k \mid k \geq 1\}$

Recap:

- $L(N_1) = \{a^n b^m \mid n \geq 0, m \geq 1\}$
- $L(N_2) = \{b^m \mid m \geq 0\}$
- $L(M_3) = \{b^k \mid k \geq 1\}$

Closure properties of regular languages 2

The closure (or Kleene star) of a language A, denoted by A^*, is the set of all strings that can be formed by concatenating together any finite number of strings of A.

Examples:

- $\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots, a^n, \ldots\}$
- $\{a, ab\}^* = \{\varepsilon, a, ab, aa, abab, aab, aba, \ldots\}$

- The Kleene star of a regular language is also regular.
  Proof: Given an NFA N_A that recognizes a language A we can build an NFA N_{A^*} that recognises the closure of A by making a start state accept state and, adding transitions, with corresponding labels, from all accept state(s) to the neighbours of the initial state(s).

Closure properties of regular languages 2

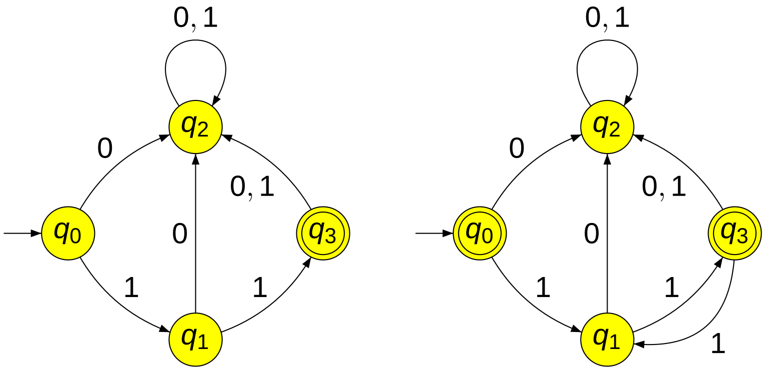
The closure (or Kleene star) of a language A, denoted by A^*, is the set of all strings that can be formed by concatenating together any finite number of strings of A.

Examples:

- $\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots, a^n, \ldots\}$
- $\{a, ab\}^* = \{\varepsilon, a, ab, aa, abab, aab, aba, \ldots\}$

- The Kleene star of a regular language is also regular.
  Proof: Given an NFA N_A that recognizes a language A we can build an NFA N_{A^*} that recognises the closure of A by making a start state accept state and, adding transitions, with corresponding labels, from all accept state(s) to the neighbours of the initial state(s).

Closure operation: an example



Closure properties of regular languages 3

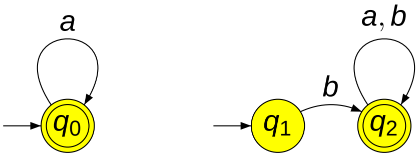
The concatenation of two languages A, B is defined to be the set of strings that can be formed by concatenating all strings of A with all strings of B, i.e.

$$AB = \{xy \mid x \in A, y \in B\}.$$

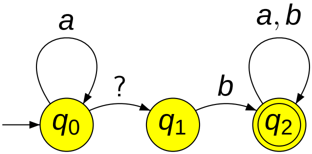
Example: If $A = \{a^n \mid n \geq 0\}$ and $B = \{bw \mid w \in \{a,b\}^*\}$, then

$$AB = \{a^n bw \mid w \in \{a,b\}^*, n \geq 0\} = \{ubv \mid u, v \in \{a,b\}^*\}.$$

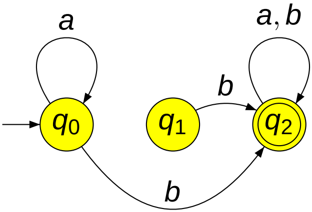
Closure under concatenation: an example 1.1



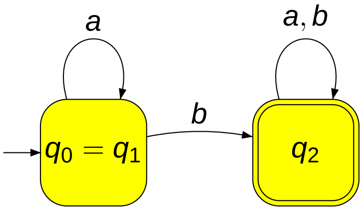
Closure under concatenation: an example 1.2



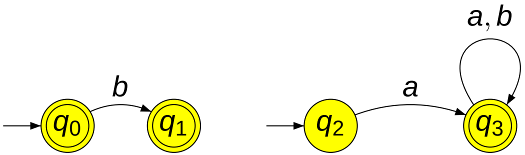
Closure under concatenation: an example 1.3



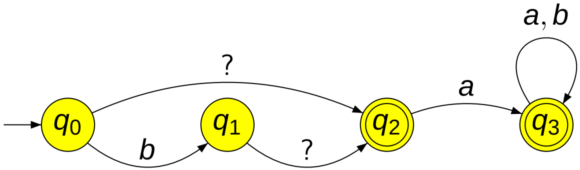
Closure under concatenation: an example 1.3'



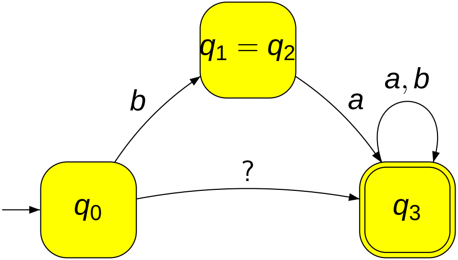
Closure under concatenation: an example 2.1



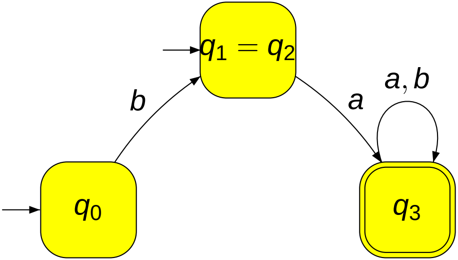
Closure under concatenation: an example 2.2



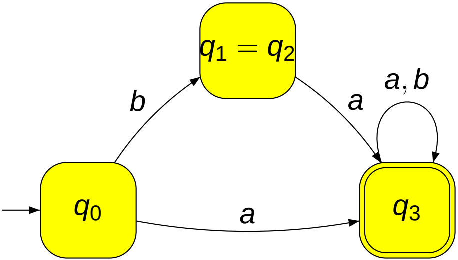
Closure under concatenation: an example 2.3



Closure under concatenation: an example 2.4



Closure under concatenation: an example 2.4'



Closure properties of regular languages 3

- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  -

$$\delta(q,c) = \begin{cases} \delta_A(q,c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q,c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q,c) \cup \{\delta_B(q',c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Closure properties of regular languages 3

- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  -
$$\delta(q,c) = \begin{cases} \delta_A(q,c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q,c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q,c) \cup \{\delta_B(q',c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Closure properties of regular languages 3

- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  -

$$\delta(q,c) = \begin{cases} \delta_A(q,c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q,c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q,c) \cup \{\delta_B(q',c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Closure properties of regular languages 3

- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  -
$$\delta(q, c) = \begin{cases} \delta_A(q, c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q, c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q, c) \cup \{\delta_B(q', c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Closure properties of regular languages 3

- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  - $$\delta(q, c) = \begin{cases} \delta_A(q, c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q, c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q, c) \cup \{\delta_B(q', c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Closure under repeated concatenation

Let A be a language and $n \geq 1$. We define:

$$A^n = \{x_1 x_2 \cdots x_n \mid x_1, x_2, \ldots, x_n \in A\}.$$

- If A is a regular language, then for each $n \geq 1$, A^n is also regular.
  Proof: $A^1 = A, A^2 = AA, \ldots, A^n = \underbrace{AA \cdots A}_{n \text{ times}}$, so the result follows from the closure under concatenation.

Closure under repeated concatenation

Let A be a language and $n \geq 1$. We define:

$$A^n = \{x_1 x_2 \cdots x_n \mid x_1, x_2, \ldots, x_n \in A\}.$$

- If A is a regular language, then for each $n \geq 1$, A^n is also regular. Proof: $A^1 = A, A^2 = AA, \ldots, \underbrace{A^n = AA \cdots A}_{n \text{ times}}$, so the result follows from the closure under concatenation.

More decidable properties of regular languages 1

- It is algorithmically decidable whether two DFAs accept the same language.

  Proof: If A, B are two languages recognised by the DFAs M_A, M_B, respectively, then (using the closure properties of regular languages) we can construct a DFA M such that:

  $$L(M) = A \,\Delta\, B = (A \cap \overline{B}) \cup (B \cap \overline{A}),$$

  and then use the equivalence:

  $$A = B \Leftrightarrow A \,\Delta\, B = \emptyset.$$

More decidable properties of regular languages 1

- It is algorithmically decidable whether two DFAs accept the same language.

  Proof: If A, B are two languages recognised by the DFAs M_A, M_B, respectively, then (using the closure properties of regular languages) we can construct a DFA M such that:

  $$L(M) = A \,\Delta\, B = (A \cap \overline{B}) \cup (B \cap \overline{A}),$$

  and then use the equivalence:

  $$A = B \Leftrightarrow A \,\Delta\, B = \emptyset.$$

More decidable properties of regular languages 1

- It is algorithmically decidable whether two DFAs accept the same language.

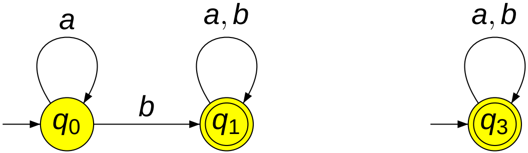
  Proof: If A, B are two languages recognised by the DFAs M_A, M_B, respectively, then (using the closure properties of regular languages) we can construct a DFA M such that:

  $$L(M) = A \,\Delta\, B = (A \cap \overline{B}) \cup (B \cap \overline{A}),$$

  and then use the equivalence:

  $$A = B \Leftrightarrow A \,\Delta\, B = \emptyset.$$

More decidable properties of regular languages: an example 1.1



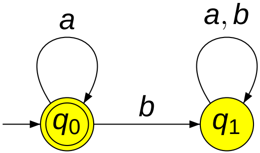
DFA M_1

$\{a^n bu \mid n \geq 0, u \in \{a, b\}^*\}$

DFA M_2

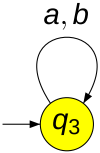
$\{a, b\}^*$

More decidable properties of regular languages: an example 1.2



DFA $\overline{M_1}$

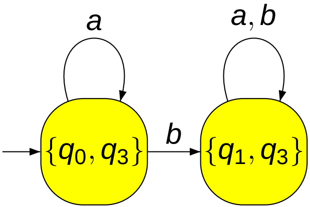
$\{a^n \mid n \geq 0\}$



DFA $\overline{M_2}$

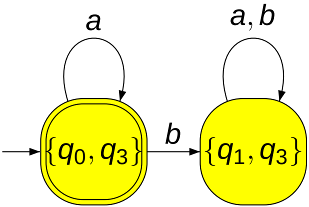
$\emptyset$

More decidable properties of regular languages: an example 1.3



DFA $M_1 \cap \overline{M_2}$

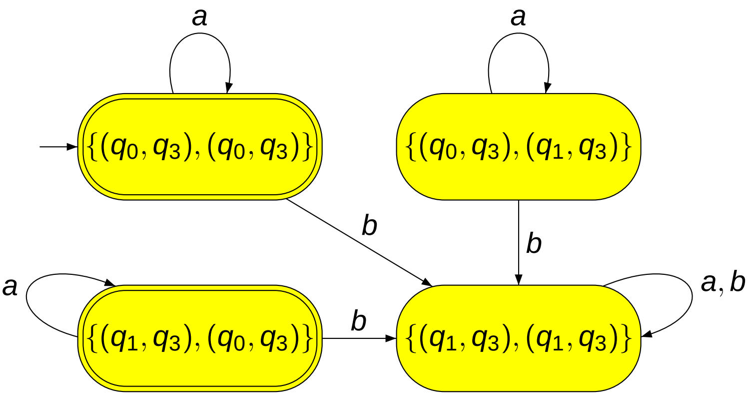
$\emptyset$



DFA $\overline{M_1} \cap M_2$

$\{a^n \mid n \geq 0\}$

More decidable properties of regular languages: an example 1.4



DFA $M_1 \Delta M_2$: $\{a^n \mid n \geq 0\} \neq \emptyset$ implies $L(M_1) \neq L(M_2)$

More decidable properties of regular languages 2

- It is algorithmically decidable whether a DFA M accepts only one a string w.
  Proof: Take $A = L(M)$ and $B = \{w\}$.
- It is algorithmically decidable whether the language accepted by a DFA M includes the language accepted by a DFA M'.
  Proof: We use the equivalence

$$L(M) \subseteq L(M') \Leftrightarrow L(M) \cap L(M') = L(M).$$

More decidable properties of regular languages 2

- It is algorithmically decidable whether a DFA M accepts only one a string w.
  Proof: Take $A = L(M)$ and $B = \{w\}$.

- It is algorithmically decidable whether the language accepted by a DFA M includes the language accepted by a DFA M'.
  Proof: We use the equivalence

$$L(M) \subseteq L(M') \Leftrightarrow L(M) \cap L(M') = L(M).$$

More decidable properties of regular languages 2

- It is algorithmically decidable whether a DFA M accepts only one a string w.
  Proof: Take $A = L(M)$ and $B = \{w\}$.
- It is algorithmically decidable whether the language accepted by a DFA M includes the language accepted by a DFA M'.
  Proof: We use the equivalence

$$L(M) \subseteq L(M') \Leftrightarrow L(M) \cap L(M') = L(M).$$

Minimisation of DFAs 1

We want to minimise the number of states of a DFA, i.e. given a DFA M produce a new DFA M' such that:

- $L(M) = L(M')$,
- M' has less states than M.

Minimisation of DFAs 1

We want to minimise the number of states of a DFA, i.e. given a DFA M produce a new DFA M' such that:

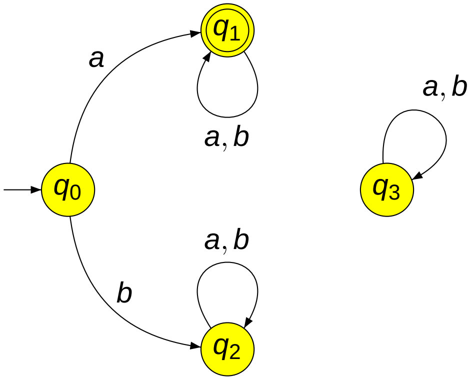
- $L(M) = L(M')$,
- M' has less states than M.

Minimisation of DFAs 1

We want to minimise the number of states of a DFA, i.e. given a DFA M produce a new DFA M' such that:

- $L(M) = L(M')$,
- M' has less states than M.

Minimisation of DFAs 2



The state q_3 can be removed without modifying the accepted language

Minimisation of DFAs 3

From a DFA

$$M = (Q, \Sigma, \delta, s, F)$$

and any state $q \in Q$ we define the new DFA

$$M_q = (Q, \Sigma, \delta, q, F)$$

by simply replacing the initial state s with q.

We say two states p and q of M are distinguishable (k-distinguishable) if there exists a string $w \in \Sigma^*$ (of length k) such that exactly one of M_p or M_q accepts w.

If there is no such string w then we say p and q are equivalent.

Minimisation of DFAs 3

From a DFA

$$M = (Q, \Sigma, \delta, s, F)$$

and any state $q \in Q$ we define the new DFA

$$M_q = (Q, \Sigma, \delta, q, F)$$

by simply replacing the initial state s with q.

We say two states p and q of M are distinguishable (k-distinguishable) if there exists a string $w \in \Sigma^*$ (of length k) such that exactly one of M_p or M_q accepts w.

If there is no such string w then we say p and q are equivalent.

Minimisation of DFAs 3

From a DFA

$$M = (Q, \Sigma, \delta, s, F)$$

and any state $q \in Q$ we define the new DFA

$$M_q = (Q, \Sigma, \delta, q, F)$$

by simply replacing the initial state s with q.

We say two states p and q of M are distinguishable (k-distinguishable) if there exists a string $w \in \Sigma^*$ (of length k) such that exactly one of M_p or M_q accepts w.

If there is no such string w then we say p and q are equivalent.

Minimisation of DFAs 4

Questions:

- Does there exist an algorithm deciding whether two states p and q are distinguishable?
- Does there exist an algorithm deciding whether two states p and q are k-distinguishable?
- Does there exist an algorithm deciding whether two states p and q are equivalent?

Minimisation of DFAs 4

Questions:

- Does there exist an algorithm deciding whether two states p and q are distinguishable?
- Does there exist an algorithm deciding whether two states p and q are k-distinguishable?
- Does there exist an algorithm deciding whether two states p and q are equivalent?

Minimisation of DFAs 4

Questions:

- Does there exist an algorithm deciding whether two states p and q are distinguishable?
- Does there exist an algorithm deciding whether two states p and q are k-distinguishable?
- Does there exist an algorithm deciding whether two states p and q are equivalent?

Minimisation of DFAs: elimination lemma 5

If a DFA M has two equivalent states p and q, then one of these states can be eliminated without modifying the accepted language, hence we can construct a smaller DFA M′ such that $L(M) = L(M')$.

Proof: Assume $M = (Q, \Sigma, \delta, s, F)$ and $p \neq s$. We create an equivalent DFA

$$M' = (Q \setminus \{p\}, \Sigma, \delta', s, F \setminus \{p\}),$$

where δ' is δ with all instances of $\delta(q_i, c) = p$ replaced with $\delta'(q_i, c) = q$, and all instances of $\delta(p, c) = q_i$ deleted.

The resulting automaton M' is deterministic and accepts $L(M)$.

Minimisation of DFAs: distinguish lemma 6

Two states p and q are k-distinguishable if and only if for some $c \in \Sigma$, the states $\delta(p, c)$ and $\delta(q, c)$ are $(k - 1)$-distinguishable.

Proof: Consider all strings $w = cw'$ of length k. If $\delta(p, c)$ and $\delta(q, c)$ are $(k - 1)$-distinguishable by some string w', then p and q must be k-distinguishable by w.

Likewise, if p and q are k-distinguishable by w, then there exist two states $\delta(p, c)$ and $\delta(q, c)$ that are $(k - 1)$-distinguishable by the shorter string w'.

Minimisation of DFAs: the algorithm 7

The algorithm `minimizeDFA` finds the equivalent states of a DFA $M = (Q, \Sigma, \delta, s, F)$. It defines a series of equivalence relations $\equiv_0$, $\equiv_1$, $\ldots$ on the states of Q:

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

It stops generating these equivalence classes when $\equiv_n$ and $\equiv_{n+1}$ are identical.

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is `the algorithm minimizeDFA` optimal?

???

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is the algorithm minimizeDFA optimal?

???

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is the `algorithm minimizeDFA` optimal?

???

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is `the algorithm minimizeDFA` optimal?

???

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is `the algorithm minimizeDFA` optimal?

???

Minimisation of DFAs: correctness of the algorithm 8

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

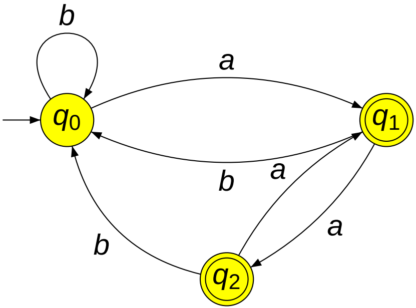
We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

Is `the algorithm minimizeDFA` optimal?

???

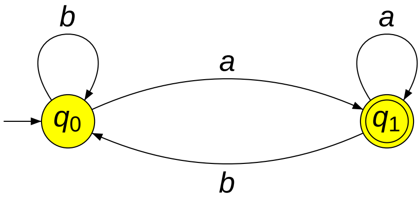
Minimisation of DFAs: example 1 9

The DFA M is not minimal as:
$\equiv_0 = \{\{q_0\}, \{q_1, q_2\}\}$,
$q_1 \equiv_1 q_2$,
$\equiv_1 = \{\{q_0\}, \{q_1, q_2\}\}$,
$\equiv_0 = \equiv_1$
because
$\delta(q_1, a) = q_2 \equiv_0 \delta(q_2, a) = q_1$,
$\delta(q_1, b) = q_0 \equiv_0 \delta(q_2, b) = q_0$



Minimisation of DFAs: example 1 10

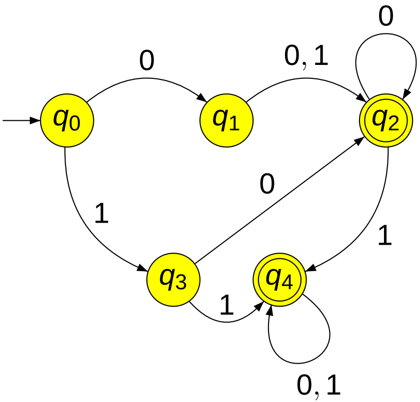
The following DFA is minimal and equivalent to M:



Minimisation of DFAs: example 2 11

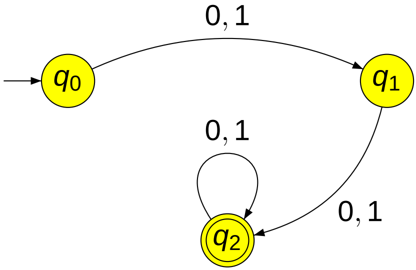
The DFA M is not minimal as:
$\equiv_0 = \{\{q_0, q_1, q_3\}, \{q_2, q_4\}\}$,

$\equiv_1 = \{\{q_0\}, \{q_1, q_3\}, \{q_2, q_4\}\}$,
$\equiv_2 = \equiv_1$,
because
$\delta(q_2, 0) = q_2 \equiv_0 \delta(q_4, 0) = q_4$,
$\delta(q_2, 1) = q_4 \equiv_0 \delta(q_4, 1) = q_4$,
$\delta(q_0, 0) = q_1 \not\equiv_0 \delta(q_1, 0) = q_2$,
$\delta(q_0, 0) = q_1 \not\equiv_0 \delta(q_3, 0) = q_2$,
$\delta(q_1, 0) = q_2 \equiv_0 \delta(q_3, 0) = q_2$,
$\delta(q_1, 1) = q_2 \equiv_0 \delta(q_3, 1) = q_4$



Minimisation of DFAs: example 2 12

The following DFA is minimal and equivalent to M:



Searching with GREP

A grep pattern, also known as a regular expression, describes the text that we are looking for.

For instance, a pattern can describe words that begin with C and end in l. A pattern like this would match “Call”, “Cornwall”, and as well as many other words, but not “Computer”.

Most characters that we type into the Find & Replace dialogue (in your favourite editor) match themselves. For instance, if you are looking for the letter “s”, Grep stops and reports a match when it encounters an “s” in the text.

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

Searching with GREP

A grep pattern, also known as a regular expression, describes the text that we are looking for.

For instance, a pattern can describe words that begin with C and end in l. A pattern like this would match “Call”, “Cornwall”, and as well as many other words, but not “Computer”.

Most characters that we type into the Find & Replace dialogue (in your favourite editor) match themselves. For instance, if you are looking for the letter “s”, Grep stops and reports a match when it encounters an “s” in the text.

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

Searching with GREP

A grep pattern, also known as a regular expression, describes the text that we are looking for.

For instance, a pattern can describe words that begin with C and end in l. A pattern like this would match “Call”, “Cornwall”, and as well as many other words, but not “Computer”.

Most characters that we type into the Find & Replace dialogue (in your favourite editor) match themselves. For instance, if you are looking for the letter “s”, Grep stops and reports a match when it encounters an “s” in the text.

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

Searching with GREP

A grep pattern, also known as a regular expression, describes the text that we are looking for.

For instance, a pattern can describe words that begin with C and end in l. A pattern like this would match “Call”, “Cornwall”, and as well as many other words, but not “Computer”.

Most characters that we type into the Find & Replace dialogue (in your favourite editor) match themselves. For instance, if you are looking for the letter “s”, Grep stops and reports a match when it encounters an “s” in the text.

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

Regular expressions

The Kleene regular expressions over the alphabet Σ and the sets they designate are:

1. Any $c \in \Sigma$ is a regular expression denoting the set $\{c\}$.
2. If E_1, E_2 are regular expressions and E_1 denotes the set S_1, E_2 denotes the set S_2, then so are:
   - $E_1 + E_2$ (or $E_1|E_2$) which denotes the union $S_1 \cup S_2$,
   - E_1E_2 which denotes the concatenation S_1S_2,
   - E_1^* which denotes the Kleene closure S_1^*.

Regular expressions

The Kleene regular expressions over the alphabet Σ and the sets they designate are:

1. Any $c \in \Sigma$ is a regular expression denoting the set $\{c\}$.
2. If E_1, E_2 are regular expressions and E_1 denotes the set S_1, E_2 denotes the set S_2, then so are:
   - $E_1 + E_2$ (or $E_1 | E_2$) which denotes the union $S_1 \cup S_2$,
   - $E_1 E_2$ which denotes the concatenation $S_1 S_2$,
   - E_1^* which denotes the Kleene closure S_1^*.

Regular expressions

The Kleene regular expressions over the alphabet Σ and the sets they designate are:

1. Any $c \in \Sigma$ is a regular expression denoting the set $\{c\}$.
2. If E_1, E_2 are regular expressions and E_1 denotes the set S_1, E_2 denotes the set S_2, then so are:
   - $E_1 + E_2$ (or $E_1|E_2$) which denotes the union $S_1 \cup S_2$,
   - E_1E_2 which denotes the concatenation S_1S_2,
   - E_1^* which denotes the Kleene closure S_1^*.

Regular expressions

The Kleene regular expressions over the alphabet Σ and the sets they designate are:

1. Any $c \in \Sigma$ is a regular expression denoting the set $\{c\}$.
2. If E_1, E_2 are regular expressions and E_1 denotes the set S_1, E_2 denotes the set S_2, then so are:
   - $E_1 + E_2$ (or $E_1|E_2$) which denotes the union $S_1 \cup S_2$,
   - E_1E_2 which denotes the concatenation S_1S_2,
   - E_1^* which denotes the Kleene closure S_1^*.

Examples of regular expressions

Sample regular expressions over $\Sigma = \{a, b, c\}$ and their corresponding sets (languages):

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

Kleene's Theorem 1

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

Kleene's Theorem 1

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

Kleene's Theorem 1

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

Kleene's Theorem 1

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

Kleene's Theorem 1

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

Kleene's Theorem 2

- Verification: If $E = \{c\}$ for some $c \in \Sigma$, then we can take $N = (Q, \Sigma, \delta, S, F)$ where $Q = \{q_0, q_1\}, S = \{q_0\}, F = \{q_1\}$ and there is one transition $\delta(q_0, c) = q_1$.
- Induction:
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under union the NFA N_{union} accepts the language denoted by $E_1 + E_2$:

$$L(N_{union}) = L(N_1) \cup L(N_2).$$

Kleene's Theorem 2

- Verification: If $E = \{c\}$ for some $c \in \Sigma$, then we can take $N = (Q, \Sigma, \delta, S, F)$ where $Q = \{q_0, q_1\}, S = \{q_0\}, F = \{q_1\}$ and there is one transition $\delta(q_0, c) = q_1$.
- Induction:
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under union the NFA N_{union} accepts the language denoted by $E_1 + E_2$:

$$L(N_{union}) = L(N_1) \cup L(N_2).$$

Kleene's Theorem 3

- Induction (continued):
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under concatenation the NFA $N_{concatenation}$ accepts the language denoted by E_1E_2:

$$L(N_{concatenation}) = L(N_1)L(N_2).$$

  - If N_1 is a NFA accepting the language denoted by E_1, then in view of the closure under Kleene closure the NFA N_* accepts the language denoted by E_1^*:

$$L(N_*) = L(N_1)^*.$$

Kleene's Theorem 3

- Induction (continued):
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under concatenation the NFA $N_{concatenation}$ accepts the language denoted by E_1E_2:

$$L(N_{concatenation}) = L(N_1)L(N_2).$$

  - If N_1 is a NFA accepting the language denoted by E_1, then in view of the closure under Kleene closure the NFA N_* accepts the language denoted by E_1^*:

$$L(N_*) = L(N_1)^*.$$

Kleene's Theorem 3

- Induction (continued):
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under concatenation the NFA $N_{concatenation}$ accepts the language denoted by E_1E_2:

$$L(N_{concatenation}) = L(N_1)L(N_2).$$

  - If N_1 is a NFA accepting the language denoted by E_1, then in view of the closure under Kleene closure the NFA N_* accepts the language denoted by E_1^*:

$$L(N_*) = L(N_1)^*.$$

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.

We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: an example 1

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

Kleene's Theorem: other examples 2

- Construct a regular expression denoting the language:

$$A = \{0^n 1^m \mid n, m \geq 0\}.$$

The language L is regular and

$$\begin{aligned} A &= \{0^n 1^m \mid n, m \geq 0\} \\ &= \{0^n \mid n \geq 0\}\{1^m \mid m \geq 0\} \end{aligned}$$

so A is denoted by 0^*1^*.

- There is no a regular expression denoting the language:

$$B = \{0^n 1^n \mid n \geq 0\}$$

because B is not regular.

Kleene's Theorem: other examples 2

- Construct a regular expression denoting the language:

$$A = \{0^n 1^m \mid n, m \geq 0\}.$$

  The language L is regular and

$$\begin{aligned} A &= \{0^n 1^m \mid n, m \geq 0\} \\ &= \{0^n \mid n \geq 0\}\{1^m \mid m \geq 0\} \end{aligned}$$

  so A is denoted by 0^*1^*.

- There is no a regular expression denoting the language:

$$B = \{0^n 1^n \mid n \geq 0\}$$

  because B is not regular.

Kleene's Theorem: other examples 3

There is no a regular expression denoting the language:

$$C = \{uuww \mid u, w \in \{a, b\}^*\}$$

because C is not regular. Prove this fact!

Kleene's Theorem: other examples 3

There is no a regular expression denoting the language:

$$C = \{uuww \mid u, w \in \{a, b\}^*\}$$

because C is not regular. Prove this fact!

The pattern matching problem

The *pattern matching problem*:

> *Given a (short) pattern P and a (long) text T, (over an alphabet Σ) determine whether P appears somewhere in T.*

Example: If $P = aba$ and $T = baababababaaaba$, then the first occurrence of P in T appears at the third character:

$$T = ba\mathbf{aba}babaaaba$$

Of course, there are some other occurrences.

The pattern matching problem

The *pattern matching problem*:

> *Given a (short) pattern P and a (long) text T, (over an alphabet Σ) determine whether P appears somewhere in T.*

Example: If $P = aba$ and $T = baabababaaaba$, then the first occurrence of P in T appears at the third character:

$$T = ba\mathbf{aba}babaaaba$$

Of course, there are some other occurrences.

The pattern matching problem

The *pattern matching problem*:

> *Given a (short) pattern P and a (long) text T, (over an alphabet Σ) determine whether P appears somewhere in T.*

Example: If $P = aba$ and $T = baabababaaaba$, then the first occurrence of P in T appears at the third character:

$$T = ba\mathbf{aba}babaaaba$$

Of course, there are some other occurrences.

Naive string matching 1

Try each possible position the pattern $P[1..m]$ could appear in the text $T[1..n]$:

```
for (i=0; T[i] != ’\0’; i++)
{
         for (j=0; T[i+j] != ’\0’ && P[j] != ’\0’
                          && T[i+j]==P[j]; j++) ;
         if (P[j] == ’\0’) found a match
}
```

There are two nested loops; the inner one takes $O(m)$ iterations and the outer one takes $O(n)$ iterations so the total time is the product, $O(mn)$. This is slow!

Naive string matching 2

An example: if $T[1..n]$ is a^n, and $P[1..m]$ is b, then it takes m comparisons each time to discover that we don't have a match, so mn overall.

The worst case scenario may not be too frequent because the inner loop usually finds a mismatch quickly and moves on to the next position without going through all m steps.

Can we do it better?

Naive string matching 2

An example: if $T[1..n]$ is a^n, and $P[1..m]$ is b, then it takes m comparisons each time to discover that we don't have a match, so mn overall.

The worst case scenario may not be too frequent because the inner loop usually finds a mismatch quickly and moves on to the next position without going through all m steps.

Can we do it better?

Naive string matching 2

An example: if $T[1..n]$ is a^n, and $P[1..m]$ is b, then it takes m comparisons each time to discover that we don't have a match, so mn overall.

The worst case scenario may not be too frequent because the inner loop usually finds a mismatch quickly and moves on to the next position without going through all m steps.

Can we do it better?

Pattern matching and regular languages 1

Solution: Consider the language

$$A(P) = \{x \mid x \text{ contains the pattern } P\}.$$

Assume that $A(P)$ is regular! Let M be a DFA for $A(P)$. When processing an input M must enter an accepting state when it has just finished 'seeing' the first occurrence of P, and thereafter it must remain in some accepting state or other.

Pattern matching and regular languages 2

Is $A(P)$ regular?

Answer: yes.

Example: If $P = aba$ and the alphabet is $\{a, b\}$, then

$$A(P) = \{x \in \{a, b\}^* \mid x = uPv, \text{ for some } u, v \in \{a, b\}^*\},$$

or

$$A(P) = \{uabav \mid u, v \in \{a, b\}^*\}.$$

Pattern matching and regular languages 2

Is $A(P)$ regular?

Answer: yes.

Example: If $P = aba$ and the alphabet is $\{a, b\}$, then

$$A(P) = \{x \in \{a,b\}^* \mid x = uPv, \text{ for some } u, v \in \{a,b\}^*\},$$

or

$$A(P) = \{uabav \mid u, v \in \{a,b\}^*\}.$$

Pattern matching and regular languages 2

Is $A(P)$ regular?

Answer: yes.

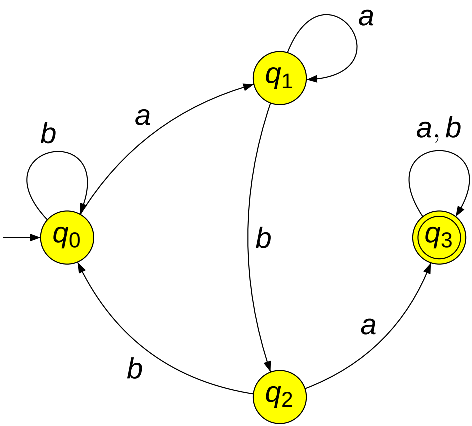
Example: If $P = aba$ and the alphabet is $\{a, b\}$, then

$$A(P) = \{x \in \{a, b\}^* \mid x = uPv, \text{ for some } u, v \in \{a, b\}^*\},$$

or

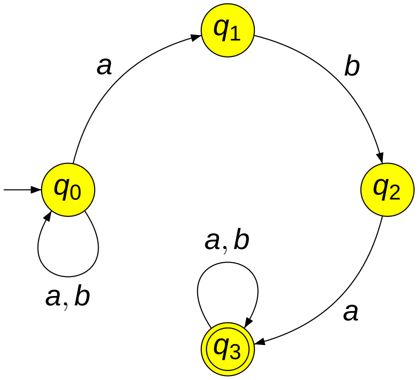
$$A(P) = \{uabav \mid u, v \in \{a, b\}^*\}.$$

Pattern matching and regular languages 3



A DFA for $AP(aba)$

Pattern matching and regular languages 4



An NFA for $A(aba)$

Pattern matching and regular languages 5

For every string P, the language

$$A(P) = \{uPv \mid u, v \in \{a, b\}^*\}$$

is regular.

Proof: Let M be a DFA recognising exactly $\{P\}$. An NFA recognising $A(P)$ can be obtained from a DFA M by adding loops labelled with a and b to the initial and final states of M.

Pattern matching and regular languages 5

For every string P, the language

$$A(P) = \{uPv \mid u, v \in \{a, b\}^*\}$$

is regular.

Proof: Let M be a DFA recognising exactly $\{P\}$. An NFA recognising $A(P)$ can be obtained from a DFA M by adding loops labelled with a and b to the initial and final states of M.

Pattern matching and regular languages 6

Is the fact that $A(P)$ is regular of any use?

Yes, because there is an algorithm testing the membership problem for $A(P)$ which is the same as testing whether P appears in the input text T.

How complex is this algorithm?

Pattern matching and regular languages 6

Is the fact that $A(P)$ is regular of any use?

Yes, because there is an algorithm testing the membership problem for $A(P)$ which is the same as testing whether P appears in the input text T.

How complex is this algorithm?

Pattern matching and regular languages 6

Is the fact that $A(P)$ is regular of any use?

Yes, because there is an algorithm testing the membership problem for $A(P)$ which is the same as testing whether P appears in the input text T.

How complex is this algorithm?

An efficient "automata-theoretic" solution

We will present an efficient "automata-theoretic" solution which consists of:

1. "pre-processing": building a DFA M for each pattern $P[1..m]$, then
2. running M on the text $T[1..n]$.

The complexity of this solution is the sum of the complexities of the above two steps.

An efficient "automata-theoretic" solution

We will present an efficient "automata-theoretic" solution which consists of:

1. "pre-processing": building a DFA M for each pattern $P[1..m]$, then
2. running M on the text $T[1..n]$.

The complexity of this solution is the sum of the complexities of the above two steps.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 1

We introduce the *prefix* and *suffix relations* and the *suffix function*.

A string w is a prefix of the string x, $w \leq_{\text{prefix}} x$ if $x = wz$ for some string z.

Let P_k be $P[1..k]$, the prefix of length $k \leq m$ of $P[1..m]$.

A string w is a suffix of the string x, $w \leq_{\text{suffix}} x$ if $x = yw$ for some string y.

For example: $a \leq_{\text{prefix}} ab$, $ca \leq_{\text{suffix}} aabbca, baab \leq_{\text{suffix}} baab$, but $ca \not\leq_{\text{prefix}} aaaaba$, $ab \not\leq_{\text{suffix}} abb$.

For each string x, $\varepsilon \leq_{\text{prefix}} x, \varepsilon \leq_{\text{suffix}} x$.

Prefix, suffix and the suffix function 2

The *suffix function* associated to the pattern $P[1..m]$ is the function

$$\sigma : \Sigma^* \to \{0, 1, \dots, m\}$$

defined as follows: $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x,

$$\sigma(x) = \max\{k : P_k \leq_{\text{suffix}} x\}.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Prefix, suffix and the suffix function 3

For example, if $P = nano$ and $\Sigma = \{a, b, n, o\}$ then
$P_0 = \varepsilon, P_1 = n, P_2 = na, P_3 = nan, P_4 = nano = P$ (so $m = 4$).

$$\sigma(\varepsilon) = 0$$
$$\sigma(annnao) = 0$$
$$\sigma(aonaaanna) = 2$$
$$\sigma(aon) = 1$$
$$\sigma(aonaaannano) = 4$$
$$\sigma(annnaanan) = 3.$$

Aho-Corasick automaton 1

The automaton states will record partial matches to the pattern.

In particular they will tell whether

- we have already matched $P[1..m]$ in $T[1..n]$, and, if not,
- we could possibly be in the middle of a match.

So we will have $m + 1$ states: the initial and accept states are clear: $0, m$, respectively.

The transition function from (state, character) to state is the longest string that is simultaneously a prefix of the pattern and a suffix of that prefix of the pattern plus the character we have just scanned.

Aho-Corasick automaton 1

The automaton states will record partial matches to the pattern.

In particular they will tell whether

- we have already matched $P[1..m]$ in $T[1..n]$, and, if not,
- we could possibly be in the middle of a match.

So we will have $m + 1$ states: the initial and accept states are clear: $0, m$, respectively.

The transition function from (state, character) to state is the longest string that is simultaneously a prefix of the pattern and a suffix of that prefix of the pattern plus the character we have just scanned.

Aho-Corasick automaton 1

The automaton states will record partial matches to the pattern.

In particular they will tell whether

- we have already matched $P[1..m]$ in $T[1..n]$, and, if not,
- we could possibly be in the middle of a match.

So we will have $m+1$ states: the initial and accept states are clear: $0, m$, respectively.

The transition function from (state, character) to state is the longest string that is simultaneously a prefix of the pattern and a suffix of that prefix of the pattern plus the character we have just scanned.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by
$$\delta(q, x) = \sigma(P_q x),$$
where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

   $$\delta(q, x) = \sigma(P_q x),$$

   where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 2

Given the pattern $P[1..m]$ over Σ, the Aho-Corasick DFA $M = (Q, \Sigma, \delta, s, F)$ is constructed as follows:

1. the set of states: $Q = \{0, 1, \ldots, m\}$,
2. the alphabet: Σ,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $q \in Q, x \in \Sigma$,
4. 0 is the start state,
5. $F = \{m\}$ is the (unique) accepting state.

Aho-Corasick automaton 3

Here is an example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$, so $m = 4$. Aho-Corasick automaton M will have:

1. the set of states: $Q = \{0, 1, 2, 3, 4\}$,
2. the alphabet: $\Sigma = \{a, b, n, o\}$,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

   $$\delta(q, x) = \sigma(P_q x),$$

   where $0 \leq q \leq 4, x \in \{a, b, n, o\}$,
4. 0 is the start state,
5. $F = \{4\}$ is the accepting state.

Aho-Corasick automaton 3

Here is an example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$, so $m = 4$. Aho-Corasick automaton M will have:

1. the set of states: $Q = \{0, 1, 2, 3, 4\}$,
2. the alphabet: $\Sigma = \{a, b, n, o\}$,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $0 \leq q \leq 4, x \in \{a, b, n, o\}$,
4. 0 is the start state,
5. $F = \{4\}$ is the accepting state.

Aho-Corasick automaton 3

Here is an example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$, so $m = 4$. Aho-Corasick automaton M will have:

1. the set of states: $Q = \{0, 1, 2, 3, 4\}$,
2. the alphabet: $\Sigma = \{a, b, n, o\}$,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $0 \leq q \leq 4, x \in \{a, b, n, o\}$,
4. 0 is the start state,
5. $F = \{4\}$ is the accepting state.

Aho-Corasick automaton 3

Here is an example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$, so $m = 4$. Aho-Corasick automaton M will have:

1. the set of states: $Q = \{0, 1, 2, 3, 4\}$,
2. the alphabet: $\Sigma = \{a, b, n, o\}$,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

$$\delta(q, x) = \sigma(P_q x),$$

   where $0 \leq q \leq 4, x \in \{a, b, n, o\}$,
4. 0 is the start state,
5. $F = \{4\}$ is the accepting state.

Aho-Corasick automaton 3

Here is an example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$, so $m = 4$. Aho-Corasick automaton M will have:

1. the set of states: $Q = \{0, 1, 2, 3, 4\}$,
2. the alphabet: $\Sigma = \{a, b, n, o\}$,
3. the transition function δ from $Q \times \Sigma$ to Q is defined by

   $$\delta(q, x) = \sigma(P_q x),$$

   where $0 \leq q \leq 4, x \in \{a, b, n, o\}$,
4. 0 is the start state,
5. $F = \{4\}$ is the accepting state.

Aho-Corasick automaton 4

The transition function δ is calculated as follows:

$$\delta(0,a) = \sigma(P_0 a) = \sigma(\varepsilon a) = \sigma(a) = 0$$
$$\delta(0,b) = \sigma(P_0 b) = \sigma(\varepsilon b) = \sigma(b) = 0$$
$$\delta(0,n) = \sigma(P_0 n) = \sigma(\varepsilon n) = \sigma(n) = 1$$
$$\delta(0,o) = \sigma(P_0 o) = \sigma(\varepsilon o) = \sigma(0) = 0$$

$$\delta(1,a) = \sigma(P_1 a) = \sigma(na) = 2$$
$$\delta(1,b) = \sigma(P_1 b) = \sigma(nb) = 0$$
$$\delta(1,n) = \sigma(P_1 n) = \sigma(nn) = 1$$
$$\delta(1,o) = \sigma(P_1 o) = \sigma(no) = 0$$

$$\delta(2,a) = \sigma(P_2 a) = \sigma(naa) = 0$$
$$\delta(2,b) = \sigma(P_2 b) = \sigma(nab) = 0$$
$$\delta(2,n) = \sigma(P_2 n) = \sigma(nan) = 3$$
$$\delta(2,o) = \sigma(P_2 o) = \sigma(nao) = 0$$

Aho-Corasick automaton 4

The transition function δ is calculated as follows:

$$
\begin{aligned}
\delta(0, a) = \sigma(P_0 a) = \sigma(\varepsilon a) = \sigma(a) = 0 \\
\delta(0, b) = \sigma(P_0 b) = \sigma(\varepsilon b) = \sigma(b) = 0 \\
\delta(0, n) = \sigma(P_0 n) = \sigma(\varepsilon n) = \sigma(n) = 1 \\
\delta(0, o) = \sigma(P_0 o) = \sigma(\varepsilon o) = \sigma(0) = 0
\end{aligned}
$$

$$
\begin{aligned}
\delta(1, a) = \sigma(P_1 a) = \sigma(na) = 2 \\
\delta(1, b) = \sigma(P_1 b) = \sigma(nb) = 0 \\
\delta(1, n) = \sigma(P_1 n) = \sigma(nn) = 1 \\
\delta(1, o) = \sigma(P_1 o) = \sigma(no) = 0
\end{aligned}
$$

$$
\begin{aligned}
\delta(2, a) = \sigma(P_2 a) = \sigma(naa) = 0 \\
\delta(2, b) = \sigma(P_2 b) = \sigma(nab) = 0 \\
\delta(2, n) = \sigma(P_2 n) = \sigma(nan) = 3 \\
\delta(2, o) = \sigma(P_2 o) = \sigma(nao) = 0
\end{aligned}
$$

Aho-Corasick automaton 5

$$\delta(3,a) = \sigma(P_3a) = \sigma(nana) = 2$$
$$\delta(3,b) = \sigma(P_3b) = \sigma(nanb) = 0$$
$$\delta(3,n) = \sigma(P_3n) = \sigma(nann) = 1$$
$$\delta(3,o) = \sigma(P_3o) = \sigma(nano) = 4$$

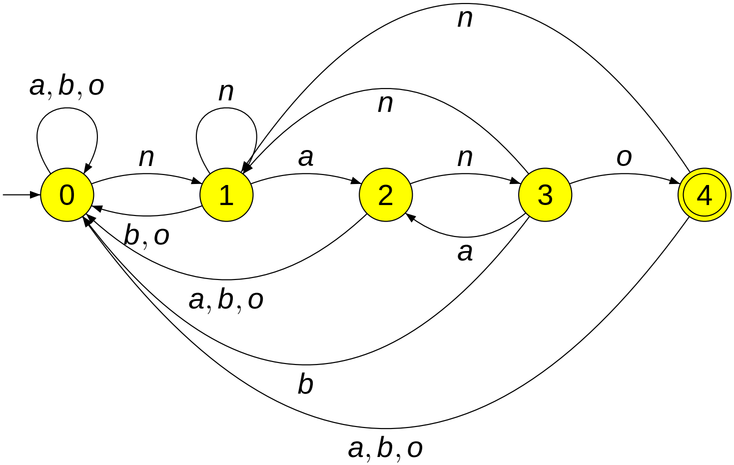
$$\delta(4,a) = \sigma(P_4a) = \sigma(nanoa) = 0$$
$$\delta(4,b) = \sigma(P_4b) = \sigma(nanob) = 0$$
$$\delta(4,n) = \sigma(P_4n) = \sigma(nanon) = 1$$
$$\delta(4,o) = \sigma(P_4o) = \sigma(nanoo) = 0$$

Aho-Corasick automaton 6

A compact presentation of the transition function:

$\delta(q,x)$	a	b	n	o	P
0	0	0	1	0	n
1	2	0	1	0	a
2	0	0	3	0	n
3	2	0	1	4	o
4	0	0	1	0	

Aho-Corasick automaton 7



Aho-Corasick automaton 8

The following procedure computes the transition function:

COMPUTE-TRANSITION-FUNCTION (P, Σ)

$m = length[P]$
for $q = 0$ to m
    do for each character $x \in \Sigma$
        do $k = \min(m + 1, q + 2)$
            repeat $k = k - 1$
                until $P_k \leq_{\text{suffix}} P_q x$
            $\delta(q, a) = k$
return δ

Aho-Corasick automaton 9

The procedure computes $\delta(q, x)$ in a straightforward manner: it starts with the largest possible value for k, which is $\min(m, q+1)$ and decreases k until $P_k \leq_{\text{suffix}} P_q x$.

The running time is $O(m^3 \times$ number of elements in $\Sigma)$: the outer loops contribute a factor of $m\times$ number of elements in Σ, the inner loops can run at most $m+1$ times and the test $P_k \leq_{\text{suffix}} P_q x$ on line 6. can require to compare up to m characters.

A **clever** algorithm requiring $O(m \times$ number of elements in $\Sigma)$ exists!

Aho-Corasick automaton 9

The procedure computes $\delta(q, x)$ in a straightforward manner: it starts with the largest possible value for k, which is $\min(m, q+1)$ and decreases k until $P_k \leq_{\text{suffix}} P_q x$.

The running time is $O(m^3 \times$ number of elements in $\Sigma)$: the outer loops contribute a factor of $m\times$ number of elements in Σ, the inner loops can run at most $m+1$ times and the test $P_k \leq_{\text{suffix}} P_q x$ on line 6. can require to compare up to m characters.

A **clever** algorithm requiring $O(m \times$ number of elements in $\Sigma)$ exists!

Aho-Corasick automaton 9

The procedure computes $\delta(q, x)$ in a straightforward manner: it starts with the largest possible value for k, which is $\min(m, q+1)$ and decreases k until $P_k \leq_{\text{suffix}} P_q x$.

The running time is $O(m^3 \times$ number of elements in $\Sigma)$: the outer loops contribute a factor of $m \times$ number of elements in Σ, the inner loops can run at most $m+1$ times and the test $P_k \leq_{\text{suffix}} P_q x$ on line 6. can require to compare up to m characters.

A **clever** algorithm requiring $O(m \times$ number of elements in $\Sigma)$ exists!

Aho-Corasick automaton 10

The following algorithm runs Aho-Corasick automaton for the pattern P on the text T:

FINITE-AUTOMATON-MATCHER (Σ, δ, T)

1. $n = length[T]$
2. $q = 0$
3.   **for** `i=1` **to** n
4.   **do** $q = \delta(q, T[i])$
5.   **if** $q = m$
6.   **then** `print ‘Pattern occurs at position` $i - m$`’ and` **return**
7. `Print ‘Pattern doesn’t occur’`

Aho-Corasick automaton 11

The simple loop structure of the above algorithm shows that the running time on $T[1..n]$ is $O(n)$. The overall running time, i.e. which includes the pre-processing, is now

$$O(m \times \text{ number of elements in } \Sigma) + O(n).$$

Aho-Corasick automaton 12

Consider the example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$. Running the Aho-Corasick automaton M described above on the text $T = annnaananoaa$ we get:

i	1	2	3	4	5	6	7	8	9	10	**11**	12	13
$T[i]$	a	n	n	n	a	a	n	a	n	o	a	a	
state	0	0	1	1	1	2	0	1	2	3	**4**	0	0
							n	**a**	**n**	**o**			

so the match was found at position $i - m = 11 - 4 = 7$.

Aho-Corasick automaton 12

Consider the example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$. Running the Aho-Corasick automaton M described above on the text $T = annnaananoaa$ we get:

i	1	2	3	4	5	6	7	8	9	10	**11**	12	13
$T[i]$	a	n	n	n	a	a	n	a	n	o	a	a	
state	0	0	1	1	1	2	0	1	2	3	**4**	0	0
							n	**a**	**n**	**o**			

so the match was found at position $i - m = 11 - 4 = 7$.

Aho-Corasick automaton 12

Consider the example for the alphabet $\Sigma = \{a, b, n, o\}$ and pattern $P = nano$. Running the Aho-Corasick automaton M described above on the text $T = annnaananoaa$ we get:

i	1	2	3	4	5	6	7	8	9	10	**11**	12	13
$T[i]$	a	n	n	n	a	a	n	a	n	o	a	a	
state	0	0	1	1	1	2	0	1	2	3	**4**	0	0
							n	**a**	**n**	**o**			

so the match was found at position $i - m = 11 - 4 = 7$.

Knuth-Morris-Pratt algorithm

Knuth-Morris-Pratt algorithm uses the *prefix function* associated to the pattern $P[1..m]$

$$\pi : \{1, 2, \ldots, m\} \to \{0, 1, 2, \ldots, m - 1\}$$

defined by

$$\pi(q) = \max\{k : k < q \text{ and } P_k \leq_{\text{suffix}} P_q\},$$

i.e. the length of the shortest prefix of P that is a proper suffix of P_q.The overall running time is

$$O(m + n).$$

Regexes 1

In the practice of computing regular expressions (abbreviated as regex or regexp, with plural forms regexes) differ from the Kleene definition discussed before.

Regexes are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Regexes 1

In the practice of computing regular expressions (abbreviated as regex or regexp, with plural forms regexes) differ from the Kleene definition discussed before.

Regexes are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Regexes 2

There are various versions of regexes; they provide an expressive power that exceeds the regular languages.

Here is an example. Regexes have the ability to group sub-expressions with parentheses and recall the value they match in the same expression.

Using this feature one can write a pattern that matches strings of repeated words like “papatoetoe” (squares). The regex to match “papatoetoe” is

$$(.^*)\backslash 1(.^*)\backslash 2,$$

where $\backslash 1 =$pa and $\backslash 2 =$toe were the sub-matches. The language associated to this pattern is not regular.

Regexes 2

There are various versions of regexes; they provide an expressive power that exceeds the regular languages.

Here is an example. Regexes have the ability to group sub-expressions with parentheses and recall the value they match in the same expression.

Using this feature one can write a pattern that matches strings of repeated words like "papatoetoe" (squares). The regex to match "papatoetoe" is

$$(.^*)\backslash 1(.^*)\backslash 2,$$

where $\backslash 1 =$pa and $\backslash 2 =$toe were the sub-matches. The language associated to this pattern is not regular.

Regexes 2

There are various versions of regexes; they provide an expressive power that exceeds the regular languages.

Here is an example. Regexes have the ability to group sub-expressions with parentheses and recall the value they match in the same expression.

Using this feature one can write a pattern that matches strings of repeated words like "papatoetoe" (squares). The regex to match "papatoetoe" is

$$(.^*)\backslash 1(.^*)\backslash 2,$$

where $\backslash 1 =$pa and $\backslash 2 =$toe were the sub-matches. The language associated to this pattern is not regular.

Regex and Google search

Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public.

With one exception of Google Code Search:

`http://www.google.com/codesearch`

Regex and Google search

Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public.

With one exception of Google Code Search:

`http://www.google.com/codesearch`

Test distribution

Capacity Room	UPI first letter	Number of students
279 MLT1	A-R	127
122 PLT2	S-Z	57
401		184