PV030 Textual Information Systems

Petr Sojka

Faculty of Informatics
Masaryk University, Brno

Spring 2013

Outline (week three)

1. Summary of the previous lecture, searching with SE.
2. Universal search algorithm.
3. Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)
4. Left-to-right search of infinite patterns algorithms.
5. Regular expressions (RE).
6. Direct construction of (N)FA for given RE.

Outline (week three)

1. Summary of the previous lecture, searching with SE.
2. Universal search algorithm.
3. Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)
4. Left-to-right search of infinite patterns algorithms.
5. Regular expressions (RE).
6. Direct construction of (N)FA for given RE.

Outline (week three)

1. Summary of the previous lecture, searching with SE.
2. Universal search algorithm.
3. Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)
4. Left-to-right search of infinite patterns algorithms.
5. Regular expressions (RE).
6. Direct construction of (N)FA for given RE.

Outline (week three)

① Summary of the previous lecture, searching with SE.

② Universal search algorithm.

③ Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)

④ Left-to-right search of infinite patterns algorithms.

⑤ Regular expressions (RE).

⑥ Direct construction of (N)FA for given RE.

Outline (week three)

1. Summary of the previous lecture, searching with SE.
2. Universal search algorithm.
3. Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)
4. Left-to-right search of infinite patterns algorithms.
5. Regular expressions (RE).
6. Direct construction of (N)FA for given RE.

Outline (week three)

1. Summary of the previous lecture, searching with SE.
2. Universal search algorithm.
3. Left-to-right search of n patterns algorithms. (AC, NFA $\longrightarrow$ DFA.)
4. Left-to-right search of infinite patterns algorithms.
5. Regular expressions (RE).
6. Direct construction of (N)FA for given RE.

Universal search algorithm,

that uses transition table g derived from the searched pattern, (g relates to the transition function δ of FA):

```
var i,T:integer; found: boolean;
text: array[1..T] of char; state,q0: TSTATE;
g:array[1..maxstate,1..maxsymb] of TSTATE;
F: set of TSTATE;...
begin
  found:= FALSE; state:= q0; i:=0;
  while (i <= T) and not found do
    begin
      i:=i+1; state:= g[state,text[i]];
      found:= state in F;
    end;
  end;
```

How to transform pattern into g?

Search engine (SE) for left-to-right search

- ***SE for left-to-right search*** $A = (Q, T, g, h, q_0, F)$
  - Q is a finite set of states.
  - T is a finite input alphabet.
  - $g\colon Q \times T \longrightarrow Q \cup \{\text{fail}\}$ is a forward state-transition function.
  - $h\colon (Q - q_0) \longrightarrow Q$ is a backward state-transition function.
  - q_0 is an initial state.
  - F is a set of final states.
- ***A depth of the state*** q: $d(q) \in N_0$ is a length of the shortest forward sequence of the state transitions from q_0 to q.

Search engine (SE) for left-to-right search

- ***SE for left-to-right search*** $A = (Q, T, g, h, q_0, F)$
  - Q is a finite set of states.
  - T is a finite input alphabet.
  - $g\colon Q \times T \longrightarrow Q \cup \{\text{fail}\}$ is a forward state-transition function.
  - $h\colon (Q - q_0) \longrightarrow Q$ is a backward state-transition function.
  - q_0 is an initial state.
  - F is a set of final states.
- ***A depth of the state*** q: $d(q) \in N_0$ is a length of the shortest forward sequence of the state transitions from q_0 to q.

Search engine (cont.)

☞ Characteristics g, h:

- $g(q_0, a) \neq \text{fail}$ for $\forall a \in T$ (there is no backward transition in the initial state).
- If $h(q) = p$, then $d(p) < d(q)$ (the number of the backward transitions is restricted from the top by a multiple of the maximum depth of the state c and the sum of the forward transitions V). So the speed of searching is linear in relation to V.

Search engine (cont.)

☞ Characteristics g, h:

- $g(q_0, a) \neq \text{fail}$ for $\forall a \in T$ (there is no backward transition in the initial state).
- If $h(q) = p$, then $d(p) < d(q)$ (the number of the backward transitions is restricted from the top by a multiple of the maximum depth of the state c and the sum of the forward transitions V). So the speed of searching is linear in relation to V.

SE configuration, transition

- ***SE configuration*** (q, w), $q \in Q$, $w \in T^*$ the not yet searched part of the text.
- ***An initial configuration of SE*** (q_0, w), w is the entire searched text.
- ***An accepting configuration of SE*** (q, w), $q \in F$, w is the not yet searched text, the found pattern is immediately before w.
- ***SE transition***: relation $\vdash \subseteq (Q \times T^*) \times (Q \times T^*)$:
  - $g(q, a) = p$, then $(q, aw) \vdash (p, w)$ ***forward transition*** for $\forall w \in T^*$.
  - $h(q) = p$, then $(q, w) \vdash (p, w)$ ***backward transition*** for $\forall w \in T^*$.

SE configuration, transition

- ☞ **SE configuration** (q, w), $q \in Q$, $w \in T^*$ the not yet searched part of the text.
- ☞ **An initial configuration of SE** (q_0, w), w is the entire searched text.
- ☞ **An accepting configuration of SE** (q, w), $q \in F$, w is the not yet searched text, the found pattern is immediately before w.
- ☞ **SE transition**: relation $\vdash \subseteq (Q \times T^*) \times (Q \times T^*)$:
  - $g(q, a) = p$, then $(q, aw) \vdash (p, w)$ **forward transition** for $\forall w \in T^*$.
  - $h(q) = p$, then $(q, w) \vdash (p, w)$ **backward transition** for $\forall w \in T^*$.

SE configuration, transition

- ***SE configuration*** (q, w), $q \in Q$, $w \in T^*$ the not yet searched part of the text.
- ***An initial configuration of SE*** (q_0, w), w is the entire searched text.
- ***An accepting configuration of SE*** (q, w), $q \in F$, w is the not yet searched text, the found pattern is immediately before w.
- ***SE transition***: relation $\vdash \subseteq (Q \times T^*) \times (Q \times T^*)$:
  - $g(q, a) = p$, then $(q, aw) \vdash (p, w)$ ***forward transition*** for $\forall w \in T^*$.
  - $h(q) = p$, then $(q, w) \vdash (p, w)$ ***backward transition*** for $\forall w \in T^*$.

SE configuration, transition

- ***SE configuration*** (q, w), $q \in Q$, $w \in T^*$ the not yet searched part of the text.
- ***An initial configuration of SE*** (q_0, w), w is the entire searched text.
- ***An accepting configuration of SE*** (q, w), $q \in F$, w is the not yet searched text, the found pattern is immediately before w.
- ***SE transition***: relation $\vdash \subseteq (Q \times T^*) \times (Q \times T^*)$:
  - $g(q, a) = p$, then $(q, aw) \vdash (p, w)$ ***forward transition*** for $\forall w \in T^*$.
  - $h(q) = p$, then $(q, w) \vdash (p, w)$ ***backward transition*** for $\forall w \in T^*$.

SE configuration, transition

- ***SE configuration*** (q, w), $q \in Q$, $w \in T^*$ the not yet searched part of the text.
- ***An initial configuration of SE*** (q_0, w), w is the entire searched text.
- ***An accepting configuration of SE*** (q, w), $q \in F$, w is the not yet searched text, the found pattern is immediately before w.
- ***SE transition***: relation $\vdash \subseteq (Q \times T^*) \times (Q \times T^*)$:
  - $g(q, a) = p$, then $(q, aw) \vdash (p, w)$ ***forward transition*** for $\forall w \in T^*$.
  - $h(q) = p$, then $(q, w) \vdash (p, w)$ ***backward transition*** for $\forall w \in T^*$.

Searching with SE

During the forward transition, a single input symbol is read and the engine switches to the next state p. However, if $g(q, a) = \text{fail}$, the backward transition is *executed* without reading an input symbol.
$S = O(T)$ (we measure the number of SE transitions).

Construction of the KMP SE for pattern $v_1 v_2 \dots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \dots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) = fail$ for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page ?? by the below mentioned algorithm.

Construction of the KMP SE for pattern $v_1 v_2 \dots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \dots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) = \text{fail}$ for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page ?? by the below mentioned algorithm.

Construction of the KMP SE for pattern $v_1 v_2 \dots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \dots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page ?? by the below mentioned algorithm.

Construction of the KMP SE for pattern $v_1 v_2 \ldots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \ldots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) = \mathit{fail}$ for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page **??** by the below mentioned algorithm.

Construction of the KMP SE for pattern $v_1 v_2 \ldots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \ldots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) = \text{fail}$ for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page **??** by the below mentioned algorithm.

Construction of the KMP SE for pattern $v_1 v_2 \ldots v_V$

1. An initial state q_0.
2. $g(q, v_{j+1}) = q'$, where q' is equivalent to the prefix $v_1 v_2 \ldots v_j v_{j+1}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous step.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.
6. The backward state-transition function h is defined on the page **??** by the below mentioned algorithm.

Part I

Search of a finite set of patterns

Search of n patterns

Aho-Corasick algorithm

Finite automata for searching

Search of a set of patterns

SE for left-to-right search of a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$.

Instead of repeated search of text for every pattern, there is only "one" pass (FA).

Common SE algorithm

```
var text: array[1..T] of char;
  i: integer; found: boolean; state: tstate;
  g: array[1..maxstate,1..maxsymbol] of tstate;
  h: array[1..maxstate] of tstate; F: set of tstate;
found:=false; state:=q0; i:=0;
while (i<=T) and not found do
begin i:=i+1;
  while g[state,text[i]]=fail do state:=h[state];
  state:=g[state,text[i]]; found:=state in F
end
```

Common SE algorithm (cont.)

- Construction of the state-transition functions h, g?
- How about for P patterns? The main idea?
- Aho, Corasick, 1975 (AC search engine).

Common SE algorithm (cont.)

- Construction of the state-transition functions h, g?
- How about for P patterns? The main idea?
- Aho, Corasick, 1975 (AC search engine).

Common SE algorithm (cont.)

- Construction of the state-transition functions h, g?
- How about for P patterns? The main idea?
- Aho, Corasick, 1975 (AC search engine).

Aho-Corasick algorithm I

Construction of g for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

1. An initial state q_0.
2. $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \ldots b_{j+1}$ of the pattern v^i, for $\forall i \in \{1, \ldots, P\}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous steps.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.

An example: $p = \{$he, she, her$\}$ over $T = \{h, e, r, s, x\}$, where x is anything else than $\{h, e, r, s\}$.

Aho-Corasick algorithm I

Construction of g for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

1. An initial state q_0.
2. $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \ldots b_{j+1}$ of the pattern v^i, for $\forall i \in \{1, \ldots, P\}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous steps.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.

An example: $p =$\{he, she, her\} over $T =$\{h, e, r, s, x\}, where x is anything else than \{h, e, r, s\}.

Aho-Corasick algorithm I

Construction of g for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

1. An initial state q_0.
2. $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \ldots b_{j+1}$ of the pattern v^i, for $\forall i \in \{1, \ldots, P\}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous steps.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.

An example: $p = \{$he, she, her$\}$ over $T = \{$h, e, r, s, x$\}$, where x is anything else than $\{$h, e, r, s$\}$.

Aho-Corasick algorithm I

Construction of g for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

1. An initial state q_0.
2. $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \ldots b_{j+1}$ of the pattern v^i, for $\forall i \in \{1, \ldots, P\}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous steps.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.

An example: $p = \{$he, she, her$\}$ over $T = \{$h, e, r, s, x$\}$, where x is anything else than $\{$h, e, r, s$\}$.

Aho-Corasick algorithm I

Construction of g for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

1. An initial state q_0.
2. $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \ldots b_{j+1}$ of the pattern v^i, for $\forall i \in \{1, \ldots, P\}$.
3. For q_0, we define $g(q_0, a) = q_0$ for $\forall a$, for which $g(q_0, a)$ has not been defined in the previous steps.
4. $g(q, a) =$ fail for $\forall q$ and a, for which $g(q, a)$ has not been defined in the previous steps.
5. A state that corresponds to the complete pattern is the final one.

An example: $p =$\{he, she, her\} over $T =$\{h, e, r, s, x\}, where x is anything else than \{h, e, r, s\}.

The failure function h (AC II)

Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \dots, v^P\}$

At first, we define the failure function f inductively relative to the depth of the states this way:

1. For $\forall q$ of the depth 1, $f(q) = q_0$.
2. Let us assume that f is defined for each state of the depth d and lesser. The variable q_D denotes the state of the depth d and $g(q_D, a) = q'$. Then we compute $f(q')$ as follows:

   $q := f(q_D)$;
   while $g(q, a) = \text{fail}$ **do** $q := f(q)$;
   $f(q') := g(q, a)$.

The failure function h (AC II)

Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

At first, we define the failure function f inductively relative to the depth of the states this way:

1. For $\forall q$ of the depth 1, $f(q) = q_0$.
2. Let us assume that f is defined for each state of the depth d and lesser. The variable q_D denotes the state of the depth d and $g(q_D, a) = q'$. Then we compute $f(q')$ as follows:

   $q := f(q_D)$;
   while $g(q, a) = \text{fail}$ **do** $q := f(q)$;
   $f(q') := g(q, a)$.

The failure function h (AC II)

Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$

At first, we define the failure function f inductively relative to the depth of the states this way:

1. For $\forall q$ of the depth 1, $f(q) = q_0$.
2. Let us assume that f is defined for each state of the depth d and lesser. The variable q_D denotes the state of the depth d and $g(q_D, a) = q'$. Then we compute $f(q')$ as follows:

   $q := f(q_D)$;
   while $g(q, a) = \underset{\sim\sim}{\text{fail}}$ **do** $q := f(q)$;
   $f(q') := g(q, a)$.

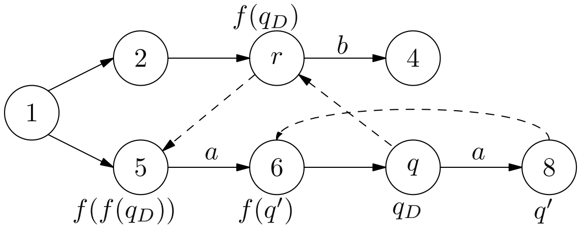
The failure function h (AC II, cont.)

- The cycle terminates, since $g(q_0, a) \neq$ fail.
- If the states q, r represent prefixes u, v of some of the patterns from p, then $f(q) = r \Leftrightarrow v$ is the longest proper suffix u.

The failure function h (AC II, cont.)

- The cycle terminates, since $g(q_0, a) \neq \text{fail}$.
- If the states q, r represent prefixes u, v of some of the patterns from p, then $f(q) = r \Leftrightarrow v$ is the longest proper suffix u.

The failure function h (AC III)



Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^p\}$ (cont.)

- We could use f as the backward state-transition function h, however, redundant backward transitions would be performed.
- We define function h inductively relative to the depth of the states this way:
  - For $\forall$ state q of the depth 1, $h(q) = q_0$.
  - Let us assume that h is defined for each state of the depth d and lesser. Let the depth q be $d + 1$. If the set of letters, for which is in a state $f(q)$ the value of the function g different from fail, is the subset of the set of letters, for which is the value of the function g in a state q different from fail, then $h(q) := h(f(q))$, otherwise $h(q) := f(q)$.

Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^p\}$ (cont.)

- We could use f as the backward state-transition function h, however, redundant backward transitions would be performed.
- We define function h inductively relative to the depth of the states this way:
  - For $\forall$ state q of the depth 1, $h(q) = q_0$.
  - Let us assume that h is defined for each state of the depth d and lesser. Let the depth q be $d + 1$. If the set of letters, for which is in a state $f(q)$ the value of the function g different from fail, is the subset of the set of letters, for which is the value of the function g in a state q different from fail, then $h(q) := h(f(q))$, otherwise $h(q) := f(q)$.

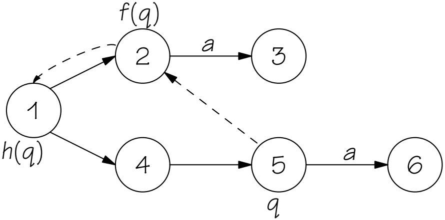
Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^P\}$ (cont.)

- We could use f as the backward state-transition function h, however, redundant backward transitions would be performed.
- We define function h inductively relative to the depth of the states this way:
  - For $\forall$ state q of the depth 1, $h(q) = q_0$.
  - Let us assume that h is defined for each state of the depth d and lesser. Let the depth q be $d + 1$. If the set of letters, for which is in a state $f(q)$ the value of the function g different from fail, is the subset of the set of letters, for which is the value of the function g in a state q different from fail, then $h(q) := h(f(q))$, otherwise $h(q) := f(q)$.

Construction of h for AC SE for a set of patterns $p = \{v^1, v^2, \ldots, v^p\}$ (cont.)

- We could use f as the backward state-transition function h, however, redundant backward transitions would be performed.
- We define function h inductively relative to the depth of the states this way:
    - For $\forall$ state q of the depth 1, $h(q) = q_0$.
    - Let us assume that h is defined for each state of the depth d and lesser. Let the depth q be $d + 1$. If the set of letters, for which is in a state $f(q)$ the value of the function g different from fail, is the subset of the set of letters, for which is the value of the function g in a state q different from fail, then $h(q) := h(f(q))$, otherwise $h(q) := f(q)$.

Construction of h for AC SE (cont.)



Finite automata for searching

Deterministic finite automaton (DFA) $M=(K,T,\delta,q_0,F)$

1. K is a finite set of inner states.
2. T is a finite input alphabet.
3. δ is a projection from $K \times T$ to K.
4. $q_0 \in K$ is an initial state.
5. $F \subseteq K$ is a set of final states.

Finite automata for searching

Deterministic finite automaton (DFA) $M=(K,T,\delta,q_0,F)$

1. K is a finite set of inner states.
2. T is a finite input alphabet.
3. δ is a projection from $K \times T$ to K.
4. $q_0 \in K$ is an initial state.
5. $F \subseteq K$ is a set of final states.

Finite automata for searching

Deterministic finite automaton (DFA) $M=(K,T,\delta,q_0,F)$

1. K is a finite set of inner states.
2. T is a finite input alphabet.
3. δ is a projection from $K \times T$ to K.
4. $q_0 \in K$ is an initial state.
5. $F \subseteq K$ is a set of final states.

Finite automata for searching

Deterministic finite automaton (DFA) $M=(K,T,\delta,q_0,F)$

1. K is a finite set of inner states.
2. T is a finite input alphabet.
3. δ is a projection from $K \times T$ to K.
4. $q_0 \in K$ is an initial state.
5. $F \subseteq K$ is a set of final states.

Finite automata for searching

Deterministic finite automaton (DFA) $M=(K,T,\delta,q_0,F)$

1. K is a finite set of inner states.
2. T is a finite input alphabet.
3. δ is a projection from $K \times T$ to K.
4. $q_0 \in K$ is an initial state.
5. $F \subseteq K$ is a set of final states.

Finite automata for searching

1. **Completely specified automaton** if δ is defined for every pair $(q, a) \in K \times T$, otherwise **incompletely specified automaton**.
2. **Configuration** M is a pair (q, w), where $q \in K$, $w \in T^*$ is the not yet searched part of the text.
3. **An initial configuration** M is (q_0, w), where w is the entire text to be searched.
4. **An accepting configuration** M is (q, w), where $q \in F$ and $w \in T^*$.

Finite automata for searching

1. ***Completely specified automaton*** if δ is defined for every pair $(q, a) \in K \times T$, otherwise ***incompletely specified automaton***.
2. ***Configuration*** M is a pair (q, w), where $q \in K$, $w \in T^*$ is the not yet searched part of the text.
3. ***An initial configuration*** M is (q_0, w), where w is the entire text to be searched.
4. ***An accepting configuration*** M is (q, w), where $q \in F$ and $w \in T^*$.

Finite automata for searching

1. ***Completely specified automaton*** if δ is defined for every pair $(q, a) \in K \times T$, otherwise ***incompletely specified automaton***.
2. ***Configuration*** M is a pair (q, w), where $q \in K$, $w \in T^*$ is the not yet searched part of the text.
3. ***An initial configuration*** M is (q_0, w), where w is the entire text to be searched.
4. ***An accepting configuration*** M is (q, w), where $q \in F$ and $w \in T^*$.

Finite automata for searching

1. ***Completely specified automaton*** if δ is defined for every pair $(q, a) \in K \times T$, otherwise ***incompletely specified automaton***.
2. ***Configuration*** M is a pair (q, w), where $q \in K$, $w \in T^*$ is the not yet searched part of the text.
3. ***An initial configuration*** M is (q_0, w), where w is the entire text to be searched.
4. ***An accepting configuration*** M is (q, w), where $q \in F$ and $w \in T^*$.

Searching with FA M

During the transition, a single input symbol is read and the engine switches to the next state p.

- **Transition** M: is defined by a state and an input symbol; relation $\vdash \subseteq (K \times T^*) \times (K \times T^*)$; if $\delta(q, a) = p$, then $(q, aw) \vdash (p, w)$ for every $\forall w \in T^*$.
- ***The*** k***th power***, ***transitive*** or more precisely ***transitive reflexive closure*** of the relation $\vdash$: $\vdash^k$, $\vdash^+$, $\vdash^*$.
- $L(M) = \{w \in T^* : (q_0, w) \vdash^* (q, w') \text{ for some } q \in F, w' \in T^*\}$ ***the language accepted by FA*** M.
- time complexity $O(T)$ (we measure the number of transitions of FA M).

Searching with FA M

During the transition, a single input symbol is read and the engine switches to the next state p.

- ☞ ***Transition*** M: is defined by a state and an input symbol; relation $\vdash \subseteq (K \times T^*) \times (K \times T^*)$; if $\delta(q, a) = p$, then $(q, aw) \vdash (p, w)$ for every $\forall w \in T^*$.
- ☞ ***The*** k***th power***, ***transitive*** or more precisely ***transitive reflexive closure*** of the relation $\vdash$: $\vdash^k$, $\vdash^+$, $\vdash^*$.
- ☞ $L(M) = \{w \in T^* : (q_0, w) \vdash^* (q, w') \text{ for some } q \in F, w' \in T^*\}$ ***the language accepted by FA*** M.
- ☞ time complexity $O(T)$ (we measure the number of transitions of FA M).

Searching with FA M

During the transition, a single input symbol is read and the engine switches to the next state p.

- ***Transition*** M: is defined by a state and an input symbol; relation $\vdash \subseteq (K \times T^*) \times (K \times T^*)$; if $\delta(q, a) = p$, then $(q, aw) \vdash (p, w)$ for every $\forall w \in T^*$.
- ***The kth power***, ***transitive*** or more precisely ***transitive reflexive closure*** of the relation $\vdash$: $\vdash^k$, $\vdash^+$, $\vdash^*$.
- $L(M) = \{w \in T^* : (q_0, w) \vdash^* (q, w')$ for some $q \in F, w' \in T^*\}$ ***the language accepted by FA*** M.
- time complexity $O(T)$ (we measure the number of transitions of FA M).

Searching with FA M

During the transition, a single input symbol is read and the engine switches to the next state p.

- ***Transition*** M: is defined by a state and an input symbol; relation $\vdash \subseteq (K \times T^*) \times (K \times T^*)$; if $\delta(q, a) = p$, then $(q, aw) \vdash (p, w)$ for every $\forall w \in T^*$.
- ***The*** k***th power***, ***transitive*** or more precisely ***transitive reflexive closure*** of the relation $\vdash$: $\vdash^k$, $\vdash^+$, $\vdash^*$.
- $L(M) = \{w \in T^* : (q_0, w) \vdash^* (q, w') \text{ for some } q \in F, w' \in T^*\}$ ***the language accepted by FA*** M.
- time complexity $O(T)$ (we measure the number of transitions of FA M).

Nondeterministic FA

Definition: ***Nondeterministic finite automaton*** (NFA) is $M = (K, T, \delta, q_0, F)$, where K, T, q_0, F are the same as those in the deterministic version of FA, but $\delta : K \times T \longrightarrow 2^K$ $\delta(q, a)$ is now ***a set*** of states.

Definition: $\vdash \in (K \times T^*) \times (K \times T^*)$ ***transition***: if $p \in \delta(q, a)$, then $(q, aw) \vdash (p, w)$ for $\forall w \in T^*$.

Definition: a final state, $L(M)$ analogically as in DFA.

Nondeterministic FA

Definition: ***Nondeterministic finite automaton*** (NFA) is $M = (K, T, \delta, q_0, F)$, where K, T, q_0, F are the same as those in the deterministic version of FA, but $\delta : K \times T \longrightarrow 2^K$ $\delta(q, a)$ is now ***a set*** of states.

Definition: $\vdash \in (K \times T^*) \times (K \times T^*)$ ***transition***: if $p \in \delta(q, a)$, then $(q, aw) \vdash (p, w)$ for $\forall w \in T^*$.

Definition: a final state, $L(M)$ analogically as in DFA.

Nondeterministic FA

Definition: ***Nondeterministic finite automaton*** (NFA) is $M = (K, T, \delta, q_0, F)$, where K, T, q_0, F are the same as those in the deterministic version of FA, but $\delta : K \times T \longrightarrow 2^K$ $\delta(q, a)$ is now ***a set*** of states.

Definition: $\vdash \in (K \times T^*) \times (K \times T^*)$ ***transition***: if $p \in \delta(q, a)$, then $(q, aw) \vdash (p, w)$ for $\forall w \in T^*$.

Definition: a final state, $L(M)$ analogically as in DFA.

Construction of SE (DFA) from NFA

Theorem: for every nondeterministic finite automaton M=(K,T,δ,q_0,F), we can build deterministic finite automaton M'=(K',T,δ',q'_0, F') such that $L(M) = L(M')$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
   - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for, $\forall p \in q'$ and $a \in T$;
   - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
   - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}$; $F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
   - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for, $\forall p \in q'$ and $a \in T$;
   - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
   - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}$; $F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
   - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for, $\forall p \in q'$ and $a \in T$;
   - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
   - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}$; $F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
   - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for,$\forall p \in q'$ and $a \in T$;
   - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
   - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}; F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
   - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for, $\forall p \in q'$ and $a \in T$;
   - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
   - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}$; $F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of SE (DFA) from NFA (cont.)

A constructive proof (of the algorithm):
Input: nondeterministic FA $M = (K, T, \delta, q_0, F)$.
Output: deterministic FA.

1. $K' = \{\{q_0\}\}$, state $\{q_0\}$ in unmarked.
2. If there are in K' all the states marked, continue to the step 4.
3. We choose from K' unmarked state q':
    - $\delta'(q', a) = \bigcup\{\delta(p, a)\}$ for,$\forall p \in q'$ and $a \in T$;
    - $K' = K' \cup \delta'(q', a)$ for $\forall a \in T$;
    - we mark q' and continue to the step 2.
4. $q'_0 = \{q_0\}$; $F' = \{q' \in K' : q' \cap F \neq \emptyset\}$.

Construction of g for SE

Construction g' for SE for a set of patterns $p = \{v^1, v^2, \dots, v^P\}$

1. We create NFA M:
   - An initial state q_0.
   - For $\forall a \in T$, we define $g(q_0, a) = q_0$.
   - For $\forall i \in \{1, \dots, P\}$, we define $g(q, b_{j+1}) = q'$, where q' is equivalent to the prefix $b_1 b_2 \dots b_{j+1}$ of the pattern v^i.
   - The state corresponding to the entire pattern is the final one.
2. …and its corresponding DFA M' with g'.

Part II

Search for an infinite set of patterns

Regular expression (RE)

Definition: ***Regular expression** E **over the alphabet** A*:

1. $\varepsilon, \mathbf{0}$ are RE and for $\forall a \in A$ is a RE.
2. If x, y are RE over A, then:
   - $(x + y)$ is RE (union);
   - $(x.y)$ is RE (concatenation);
   - $(x)^*$ is RE (iteration).

A convention about priority of regular operations:
union < concatenation < iteration.
Definition: Thereafter, we consider as a ***(generalized) regular expression*** even those terms that do not contain, with regard to this convention, the unnecessary parentheses.

Value of RE

1. $h(\mathbf{O}) = \emptyset, h(\varepsilon) = \{\varepsilon\}, h(a) = \{a\}$
2. - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \ldots$

☞ The value of RE is a regular language (RL).

☞ Every RL can be represented as RE.

☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Value of RE

1. $h(\mathbf{O}) = \emptyset, h(\varepsilon) = \{\varepsilon\}, h(a) = \{a\}$
2. - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \ldots$

☞ The value of RE is a regular language (RL).

☞ Every RL can be represented as RE.

☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Value of RE

1. $h(\mathbf{O}) = \emptyset, h(\varepsilon) = \{\varepsilon\}, h(a) = \{a\}$
2. - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \ldots$

☞ The value of RE is a regular language (RL).

☞ *Every RL can be represented as RE.*

☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Value of RE

1. $h(\mathbf{O}) = \emptyset, h(\varepsilon) = \{\varepsilon\}, h(a) = \{a\}$
2.
   - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \ldots$

☞ The value of RE is a regular language (RL).

☞ Every RL can be represented as RE.

☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Value of RE

1. $h(\mathbf{O}) = \emptyset$, $h(\varepsilon) = \{\varepsilon\}$, $h(a) = \{a\}$
2.
   - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

- ☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \dots$
- ☞ The value of RE is a regular language (RL).
- ☞ Every RL can be represented as RE.
- ☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Value of RE

1. $h(\mathbf{O}) = \emptyset$, $h(\varepsilon) = \{\varepsilon\}$, $h(a) = \{a\}$
2.
   - $h(x + y) = h(x) \cup h(y)$
   - $h(x.y) = h(x).h(y)$
   - $h(x^*) = (h(x))^*$

☞ $h(x^*) = \varepsilon \cup x \cup x.x \cup x.x.x \cup \ldots$

☞ The value of RE is a regular language (RL).

☞ Every RL can be represented as RE.

☞ For $\forall$ RE V $\exists$ FA M: $h(V) = L(M)$.

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Axiomatization of RE (Salomaa 1966)

A1: $x + (y + z) = (x + y) + z = x + y + z$ associativity of union

A2: $x.(y.z) = (x.y).z = x.y.z$ associativity of concatenation

A3: $x + y = y + x$ commutativity of union

A4: $(x + y).z = x.z + y.z$ right distributivity

A5: $x.(y + z) = x.y + x.z$ left distributivity

A6: $x + x = x$ idempotence of union

A7: $\varepsilon.x = x$ identity element for concatenation

A8: $\mathbf{0}.x = \mathbf{0}$ inverse element for concatenation

A9: $x + \mathbf{0} = x$ identity element for union

A10: $x^* = \varepsilon + x^*x$

A11: $x^* = (\varepsilon + x)^*$

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Outline (week three)

1. Summary of the previous lecture.
2. Regular expressions, value of RE, characteristics.
3. Derivation of regular expressions.
4. Direct construction of equivalent DFA for given RE by derivation.
5. Derivation of regular expressions by position vector.
6. Right-to-left search (BMH, CW, BUC).

Similarity of regular expressions

Theorem: the axiomatization of RE is complete and consistent.

Definition: regular expressions are termed as ***similar***, when they can be mutually conversed using axioms A1 to A11.

Theorem: similar regular expressions have the same value.

Similarity of regular expressions

Theorem: the axiomatization of RE is complete and consistent.

Definition: regular expressions are termed as ***similar***, when they can be mutually conversed using axioms A1 to A11.

Theorem: similar regular expressions have the same value.

Length of a regular expression

Definition: ***the length*** $d(E)$ ***of the regular expression*** E:

1. If E consists of one symbol, then $d(E) = 1$.
2. $d(V_1 + V_2) = d(V_1) + d(V_2) + 1$.
3. $d(V_1.V_2) = d(V_1) + d(V_2) + 1$.
4. $d(V^*) = d(V) + 1$.
5. $d((V)) = d(V) + 2$.

Note: the length corresponds to the syntax of a regular expression.

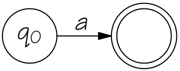
Construction of NFA for given RE

Definition: ***a generalized NFA*** allows ε-transitions (transitions without reading of an input symbol).

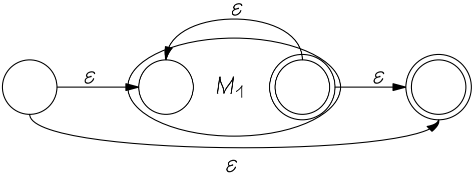
Theorem: for every RE E, we can create FA M such that $h(E) = L(M)$.
Proof: by structural induction relative to the RE E:

Construction of NFA for given RE (a proof)

1. $E = a$

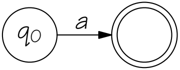


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

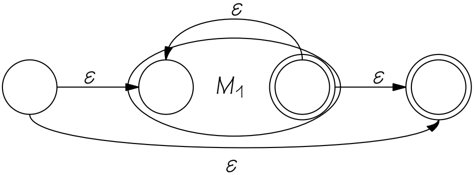


Construction of NFA for given RE (a proof)

1. $E = a$

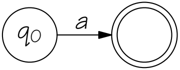


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

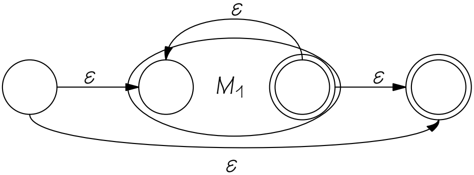


Construction of NFA for given RE (a proof)

1. $E = a$

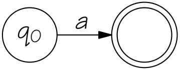


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

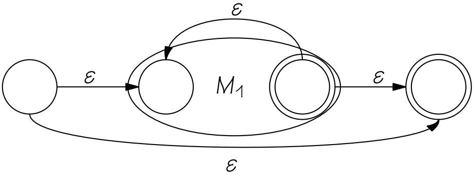


Construction of NFA for given RE (a proof)

1. $E = a$

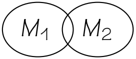


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

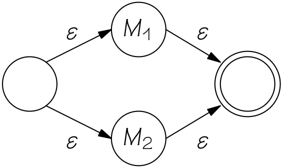


Construction of NFA for given RE (cont. of a proof)

3. $E = E_1 \cdot E_2$



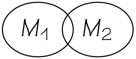
4. $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$,



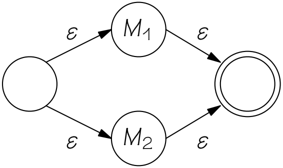
$h(E_2) = L(M_2)$)

Construction of NFA for given RE (cont. of a proof)

③ $E = E_1 \cdot E_2$

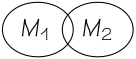


④ $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$, $h(E_2) = L(M_2)$)

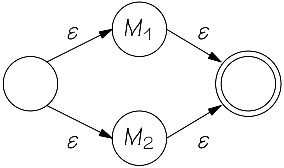


Construction of NFA for given RE (cont. of a proof)

③ $E = E_1 \cdot E_2$

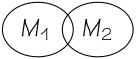


④ $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 $(h(E_1) = L(M_1)$, $h(E_2) = L(M_2))$

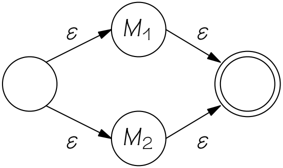


Construction of NFA for given RE (cont. of a proof)

③ $E = E_1 \cdot E_2$



④ $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$,



$h(E_2) = L(M_2)$)

Construction of NFA for given RE (cont.)

- No more than two edges come out of every state.
- No edges come out of the final states.
- The number of the states $M \leq 2 \cdot d(E)$.
- The simulation of automaton M is performed in $O(d(E)T)$ time and in $O(d(E))$ space.

Construction of NFA for given RE (cont.)

- No more than two edges come out of every state.
- No edges come out of the final states.
- The number of the states $M \leq 2 \cdot d(E)$.
- The simulation of automaton M is performed in $O(d(E)T)$ time and in $O(d(E))$ space.

Construction of NFA for given RE (cont.)

- No more than two edges come out of every state.
- No edges come out of the final states.
- The number of the states $M \leq 2 \cdot d(E)$.
- The simulation of automaton M is performed in $O(d(E)T)$ time and in $O(d(E))$ space.

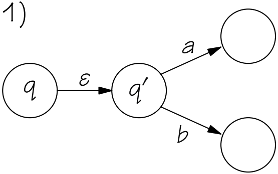
Construction of NFA for given RE (cont.)

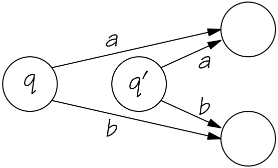
- No more than two *edges* come out of every state.
- No *edges* come out of the final states.
- The number of the states $M \leq 2 \cdot d(E)$.
- The simulation of automaton M is performed in $O(d(E)T)$ time and in $O(d(E))$ space.

NFA simulation

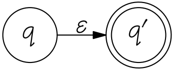
For the following methods of NFA simulation, we must remove the ε-transitions. We can achieve it with the well-known procedure:

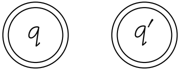
1)





2)





NFA simulation (cont.)

We represent a state with a Boolean vector and we pass through all the paths at the same time. There are two approaches:

- The general algorithm that use a transition table.
- Implementation of the automaton in a form of (generated) program for the particular automaton.

NFA simulation (cont.)

We represent a state with a Boolean vector and we pass through all the paths at the same time. There are two approaches:

- ☞ The general algorithm that use a transition table.
- ☞ Implementation of the automaton in a form of (generated) program for the particular automaton.

Direct construction of (N)FA for given RE

Let E is a RE over the alphabet T. Then we create FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$ this way:

1. We assign different natural numbers to all the occurrences of the symbols of T in the expression E. We get E'.
2. A set of starting symbols $Z = \{x_i :$ a string of $h(E')$ can start with the symbol x_i, $x_i \neq \varepsilon\}$.
3. A set of neighbours $P = \{x_i y_j :$ symbols $x_i \neq \varepsilon \neq y_j$ can be next to each other in a string of $h(E')\}$.
4. A set of ending symbols $F = \{x_i :$ a string of $h(E')$ can end with the symbol $x_i \neq \varepsilon\}$.
5. A set of states $K = \{q_0\} \cup Z \cup \{y_j : x_i y_j \in P\}$.
6. A transition function δ:
    - $\delta(q_0, x)$ contains x_i for,$\forall x_i \in Z$ that originate from numbering of x.
    - $\delta(x_i, y)$ contains y_j for,$\forall x_i y_j \in P$ such that y_j originates from numbering of y.
7. F is a set of final states, a state that corresponds to E is q_0.

Direct construction of (N)FA for given RE (cont.)

Example 1: $R = ab^*a + ac + b^*ab^*$.

Example 2: $R = ab^* + ac + b^*a$.

Derivation of a regular expression

Definition: ***derivation*** $\frac{dE}{dx}$ ***of the regular expression*** E ***by a string*** $x \in T^*$:

1. $\frac{dE}{d\varepsilon} = E$.
2. For $a \in T$, these statements are true:

$$\frac{d\varepsilon}{da} = 0$$

$$\frac{db}{da} = \begin{cases} 0 & \text{if } a \neq b \\ \varepsilon & \text{if } a = b \end{cases}$$

$$\frac{d(E+F)}{da} = \frac{dE}{da} + \frac{dF}{da}$$

$$\frac{d(E.F)}{da} = \begin{cases} \frac{dE}{da} \cdot F + \frac{dF}{da} & \text{if } \varepsilon \in h(E) \\ \frac{dE}{da} \cdot F & \text{otherwise} \end{cases}$$

$$\frac{d(E^*)}{da} = \frac{dE}{da} \cdot E^*$$

Derivation of a regular expression

Definition: ***derivation*** $\frac{dE}{dx}$ ***of the regular expression*** E ***by a string*** $x \in T^*$:

1. $$\frac{dE}{d\varepsilon} = E.$$
2. For $a \in T$, these statements are true:

$$\begin{aligned}
\frac{d\varepsilon}{da} &= 0 \\
\frac{db}{da} &= \begin{cases} 0 & \text{if } a \neq b \\ \varepsilon & \text{if } a = b \end{cases} \\
\frac{d(E+F)}{da} &= \frac{dE}{da} + \frac{dF}{da} \\
\frac{d(E.F)}{da} &= \begin{cases} \frac{dE}{da} \cdot F + \frac{dF}{da} & \text{if } \varepsilon \in h(E) \\ \frac{dE}{da} \cdot F & \text{otherwise} \end{cases} \\
\frac{d(E^*)}{da} &= \frac{dE}{da} \cdot E^*
\end{aligned}$$

Derivation of a regular expression

Definition: ***derivation*** $\frac{dE}{dx}$ ***of the regular expression*** E ***by a string*** $x \in T^*$:

1. $$\frac{dE}{d\varepsilon} = E.$$
2. For $a \in T$, these statements are true:

$$\begin{aligned}
\frac{d\varepsilon}{da} &= \mathbf{0} \\
\frac{db}{da} &= \begin{cases} \mathbf{0} & \text{if } a \neq b \\ \varepsilon & \text{if } a = b \end{cases} \\
\frac{d(E+F)}{da} &= \frac{dE}{da} + \frac{dF}{da} \\
\frac{d(E.F)}{da} &= \begin{cases} \frac{dE}{da} \cdot F + \frac{dF}{da} & \text{if } \varepsilon \in h(E) \\ \frac{dE}{da} \cdot F & \text{otherwise} \end{cases} \\
\frac{d(E^*)}{da} &= \frac{dE}{da} \cdot E^*
\end{aligned}$$

Derivation of a regular expression (cont.)

③ For $x = a_1 a_2 \ldots a_n$, $a_i \in T$, these statements are true

$$\frac{dE}{dx} = \frac{d}{da_n}\left(\frac{d}{da_{n-1}}\left(\cdots\frac{d}{da_2}\left(\frac{dE}{da_1}\right)\cdots\right)\right).$$

Characteristics of regular expressions

Example: Derive $E = fi + fi^* + f^*ifi$ by i and f.
Example: Derive $(o^*sle)^*cno$ by o, s, l, c and $osle$.

Theorem: $h\left(\frac{dE}{dx}\right) = \{y : xy \in h(E)\}$.
Example: Prove the above-mentioned statement. Instruction: use structural induction relative to E and x.

Definition: ***Regular expressions** x, y **are similar*** if one of them can be transformed to the other one with axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has length 7, 15?

Characteristics of regular expressions

Example: Derive $E = fi + fi^* + f^*ifi$ by i and f.
Example: Derive $(o^*sle)^*cno$ by o, s, l, c and $osle$.

Theorem: $h\left(\frac{dE}{dx}\right) = \{y : xy \in h(E)\}$.
Example: Prove the above-mentioned statement. Instruction: use structural induction relative to E and x.

Definition: ***Regular expressions*** x, y ***are similar*** if one of them can be transformed to the other one with axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has length 7, 15?

Characteristics of regular expressions

Example: Derive $E = fi + fi^* + f^*ifi$ by i and f.
Example: Derive $(o^*sle)^*cno$ by o, s, l, c and $osle$.

Theorem: $h\left(\frac{dE}{dx}\right) = \{y : xy \in h(E)\}$.
Example: Prove the above-mentioned statement. Instruction: use structural induction relative to E and x.

Definition: ***Regular expressions** x, y **are similar*** if one of them can be transformed to the other one with axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has length 7, 15?

Characteristics of regular expressions

Example: Derive $E = fi + fi^* + f^*ifi$ by i and f.
Example: Derive $(o^*sle)^*cno$ by o, s, l, c and $osle$.

Theorem: $h\left(\frac{dE}{dx}\right) = \{y : xy \in h(E)\}$.
Example: Prove the above-mentioned statement. Instruction: use structural induction relative to E and x.

Definition: ***Regular expressions** x, y **are similar*** if one of them can be transformed to the other one with axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has length 7, 15?

Characteristics of regular expressions

Example: Derive $E = fi + fi^* + f^*ifi$ by i and f.
Example: Derive $(o^*sle)^*cno$ by o, s, l, c and $osle$.

Theorem: $h\left(\frac{dE}{dx}\right) = \{y : xy \in h(E)\}$.
Example: Prove the above-mentioned statement. Instruction: use structural induction relative to E and x.

Definition: ***Regular expressions** x, y **are similar*** if one of them can be transformed to the other one with axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has length 7, 15?

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}$, $Q_0 = \{E\}$, $i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{dxa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}$, $Q_0 = \{E\}$, $i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{dxa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}$, $Q_0 = \{E\}$, $i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{d\,xa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}$, $Q_0 = \{E\}$, $i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{dxa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}$, $Q_0 = \{E\}$, $i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{d\,xa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Direct construction of DFA for given RE (by RE derivation)

Brzozowski (1964, Journal of the ACM)
Input: RE E over T.
Output: FA $M = (K, T, \delta, q_0, F)$ such that $h(E) = L(M)$.

1. Let us state $Q = \{E\}, Q_0 = \{E\}, i := 1$.
2. Let us create the derivation of all the expressions of Q_{i-1} by all the symbols of T. Into Q_i, we insert all the expressions created by the derivation of the expressions of Q_{i-1} that are not similar to the expressions of Q.
3. If $Q_i \neq \emptyset$, we insert Q_i into Q, set $i := i + 1$ a move to the step 2.
4. For $\forall \frac{dF}{dx} \in Q$ and $a \in T$, we set $\delta\left(\frac{dF}{dx}, a\right) = \frac{dF}{dx'}$, in case that the expression $\frac{dF}{dx'}$ is similar to the expression $\frac{dF}{d\,xa}$. (Concurrently $\frac{dF}{dx'} \in Q$.)
5. The set $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

Example: RE= $R = (0+1)^*1$.

$Q = Q_0 = \{(0+1)^*1\}, i = 1$

$Q_1 = \{\frac{dR}{d0} = R, \frac{dR}{1}\} = \{(0+1)^*1 + \varepsilon\}$

$Q_2 = \{\frac{(0+1)^*1+\varepsilon}{d0} = R, \frac{(0+1)^*1+\varepsilon}{d1} = (0+1)^*1 + \varepsilon\} = \emptyset$

Example: RE= $(10)^*(00)^*1$.

For more, see Watson, B. W.: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. `citeseer.ist.psu.edu/watson94taxonomy.html`

Example: RE= $R = (0+1)^*1$.

$Q = Q_0 = \{(0+1)^*1\}, i = 1$

$Q_1 = \{\frac{dR}{d0} = R, \frac{dR}{1}\} = \{(0+1)^*1 + \varepsilon\}$

$Q_2 = \{\frac{(0+1)^*1+\varepsilon}{d0} = R, \frac{(0+1)^*1+\varepsilon}{d1} = (0+1)^*1 + \varepsilon\} = \emptyset$

Example: RE= $(10)^*(00)^*1$.

For more, see Watson, B. W.: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. `citeseer.ist.psu.edu/watson94taxonomy.html`

Example: RE$= R = (0+1)^*1$.

$Q = Q_0 = \{(0+1)^*1\}, i = 1$

$Q_1 = \{\frac{dR}{d0} = R, \frac{dR}{1}\} = \{(0+1)^*1 + \varepsilon\}$

$Q_2 = \{\frac{(0+1)^*1+\varepsilon}{d0} = R, \frac{(0+1)^*1+\varepsilon}{d1} = (0+1)^*1 + \varepsilon\} = \emptyset$

Example: RE$= (10)^*(00)^*1$.

For more, see Watson, B. W.: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. `citeseer.ist.psu.edu/watson94taxonomy.html`

Example: RE= $R = (0+1)^*1$.
$Q = Q_0 = \{(0+1)^*1\}, i = 1$
$Q_1 = \{\frac{dR}{d0} = R, \frac{dR}{1}\} = \{(0+1)^*1 + \varepsilon\}$
$Q_2 = \{\frac{(0+1)^*1+\varepsilon}{d0} = R, \frac{(0+1)^*1+\varepsilon}{d1} = (0+1)^*1 + \varepsilon\} = \emptyset$

Example: RE= $(10)^*(00)^*1$.

For more, see Watson, B. W.: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.
`citeseer.ist.psu.edu/watson94taxonomy.html`

Exercise

Example : let us have a set of the patterns P= {tis, ti, iti}:

- Create NFA that searches for P.
- Create DFA that corresponds to this NFA and minimize it. Draw the transition graphs of both the automata (DFA and the minimal DFA) and describe the procedure of minimization.
- Compare it to the result of the search engine SE.
- Solve the exercise using the algorithm of direct construction of DFA (by deriving) and discuss whether the result automata are isomorphic.

Derivation of RE by position vector I

Definition: Position vector is a set of numbers that correspond to the positions of those symbols of alphabet which can occur in the beginning of the tail of the string that is a part of the value of the given RE.

Example: let us have a regular expression:

$$a \quad . \quad b^* \quad . \quad c \tag{1}$$

To denote the position, we are going to use the wedge symbol $\wedge$. So the expression (1) is represented as:

$$\underset{\wedge}{a} \quad . \quad b^* \quad . \quad c \tag{2}$$

By deriving a denoted expression, we get a new denoted regular expression. The basic rule of derivation is this:

1. If the operand, by which we derive, is denoted, then we denote the positions right after this operand. Subsequently, we remove its denotation. It means that, by deriving the expression (2) by the operand a, we get:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad c \tag{3a}$$

Derivation of RE by position vector I

Definition: Position vector is a set of numbers that correspond to the positions of those symbols of alphabet which can occur in the beginning of the tail of the string that is a part of the value of the given RE.

Example: let us have a regular expression:

$$a \quad . \quad b^* \quad . \quad c \tag{1}$$

To denote the position, we are going to use the wedge symbol $\wedge$. So the expression **(1)** is represented as:

$$\underset{\wedge}{a} \quad . \quad b^* \quad . \quad c \tag{2}$$

By deriving a denoted expression, we get a new denoted regular expression. The basic rule of derivation is this:

1. If the operand, by which we derive, is denoted, then we denote the positions right after this operand. Subsequently, we remove its denotation. It means that, by deriving the expression **(2)** by the operand a, we get:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad c \tag{3a}$$

Derivation of RE by position vector I

Definition: Position vector is a set of numbers that correspond to the positions of those symbols of alphabet which can occur in the beginning of the tail of the string that is a part of the value of the given RE.

Example: let us have a regular expression:

$$a \quad . \quad b^* \quad . \quad c \tag{1}$$

To denote the position, we are going to use the wedge symbol $\wedge$. So the expression **(1)** is represented as:

$$\underset{\wedge}{a} \quad . \quad b^* \quad . \quad c \tag{2}$$

By deriving a denoted expression, we get a new denoted regular expression. The basic rule of derivation is this:

1. If the operand, by which we derive, is denoted, then we denote the positions right after this operand. Subsequently, we remove its denotation. It means that, by deriving the expression **(2)** by the operand a, we get:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad c \tag{3a}$$

Derivation of RE by position vector I

Definition: Position vector is a set of numbers that correspond to the positions of those symbols of alphabet which can occur in the beginning of the tail of the string that is a part of the value of the given RE.

Example: let us have a regular expression:

$$a \quad . \quad b^* \quad . \quad c \tag{1}$$

To denote the position, we are going to use the wedge symbol $\wedge$. So the expression **(1)** is represented as:

$$\underset{\wedge}{a} \quad . \quad b^* \quad . \quad c \tag{2}$$

By deriving a denoted expression, we get a new denoted regular expression. The basic rule of derivation is this:

1. If the operand, by which we derive, is denoted, then we denote the positions right after this operand. Subsequently, we remove its denotation. It means that, by deriving the expression **(2)** by the operand a, we get:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad c \tag{3a}$$

Derivation of RE by position vector II

2. Since the construction, which generates also the empty string, is denoted, we denote the following construction as well:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \tag{3b}$$

Now, by deriving by the operand b of the expression **(3b)**, we get:

$$a \quad . \quad b^* \quad . \quad \underset{\wedge}{c} \tag{4a}$$

3. Since the construction following the construction in iteration is denoted, the previous constructions have to be also denoted.

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \tag{4b}$$

By deriving the expression **(4b)** by the operand c, we get:

$$a \quad . \quad b^* \quad . \quad c \quad \underset{\wedge}{} \tag{5}$$

When a regular expression is denoted this way, it corresponds to the empty regular expression ε.

Derivation of RE by position vector II

2. Since the construction, which generates also the empty string, is denoted, we denote the following construction as well:

$$\begin{array}{ccccc} a & . & \underset{\wedge}{b^*} & . & \underset{\wedge}{c} \end{array} \qquad \textbf{(3b)}$$

Now, by deriving by the operand b of the expression **(3b)**, we get:

$$\begin{array}{ccccc} a & . & b^* & . & \underset{\wedge}{c} \end{array} \qquad \textbf{(4a)}$$

3. Since the construction following the construction in iteration is denoted, the previous constructions have to be also denoted.

$$\begin{array}{ccccc} a & . & \underset{\wedge}{b^*} & . & \underset{\wedge}{c} \end{array} \qquad \textbf{(4b)}$$

By deriving the expression **(4b)** by the operand c, we get:

$$\begin{array}{cccccc} a & . & b^* & . & c & \underset{\wedge}{} \end{array} \qquad \textbf{(5)}$$

When a regular expression is denoted this way, it corresponds to the empty regular expression ε.

Derivation of RE by position vector II

2. Since the construction, which generates also the empty string, is denoted, we denote the following construction as well:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \qquad \textbf{(3b)}$$

Now, by deriving by the operand b of the expression **(3b)**, we get:

$$a \quad . \quad b^* \quad . \quad \underset{\wedge}{c} \qquad \textbf{(4a)}$$

3. Since the construction following the construction in iteration is denoted, the previous constructions have to be also denoted.

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \qquad \textbf{(4b)}$$

By deriving the expression **(4b)** by the operand c, we get:

$$a \quad . \quad b^* \quad . \quad c \quad \underset{\wedge}{} \qquad \textbf{(5)}$$

When a regular expression is denoted this way, it corresponds to the empty regular expression ε.

Derivation of RE by position vector II

2. Since the construction, which generates also the empty string, is denoted, we denote the following construction as well:

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \qquad \textbf{(3b)}$$

Now, by deriving by the operand b of the expression **(3b)**, we get:

$$a \quad . \quad b^* \quad . \quad \underset{\wedge}{c} \qquad \textbf{(4a)}$$

3. Since the construction following the construction in iteration is denoted, the previous constructions have to be also denoted.

$$a \quad . \quad \underset{\wedge}{b^*} \quad . \quad \underset{\wedge}{c} \qquad \textbf{(4b)}$$

By deriving the expression **(4b)** by the operand c, we get:

$$a \quad . \quad b^* \quad . \quad c \quad \underset{\wedge}{} \qquad \textbf{(5)}$$

When a regular expression is denoted this way, it corresponds to the empty regular expression ε.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector III

- For every syntactic construction, we make a list of the starting positions at the initials of the members.
- If a construction symbol equals to the symbol we use for deriving, and it is located in the denoted position, then we move the denotation in front of the following position.
- If an iteration operator is located after the construction, and the denotation is at the end of the construction, then we append the list of the starting positions, which belong to this construction, to the resulting list.
- If the denotation is located before a construction, then we append the list of the starting positions of this construction to the resulting list.
- If the denotation is before the construction which generates also an empty string, then we append the list of the starting positions of the following construction to the resulting list.
- When we want to denote a construction inside parentheses, we must denote all the initials of the members inside the parentheses.

Derivation of RE by position vector: an example

Example: $a.b^*.c$, derived by a, b, c.

Part III

Right-to-left search

Right-to-left search of one pattern

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- one pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- n patterns—Commentz-Walter (CW, 1979)
- an infinite set of patterns: reversed regular expression—Buczиłowski (BUC)

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- one pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- *n* patterns—Commentz-Walter (CW, 1979)
- an infinite set of patterns: reversed regular expression—Buczilowski (BUC)

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- one pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- *n* patterns—Commentz-Walter (CW, 1979)
- an infinite set of patterns: reversed regular expression—Bucziłowski (BUC)

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- one pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- *n* patterns—Commentz-Walter (CW, 1979)
- an infinite set of patterns: reversed regular expression—Buczıłowski (BUC)

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- ☞ *one* pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- ☞ *n* patterns—Commentz-Walter (CW, 1979)
- ☞ an infinite set of patterns: reversed regular expression—Bucziłowski (BUC)

Right-to-left search

Right-to-left search—principles.
Could the direction of the search be significant?
In which cases?

- one pattern—Boyer-Moore (BM, 1977), Boyer-Moore-Horspool (BMH, 1980), Boyer-Moore-Horspool-Sunday (BMHS, 1990)
- n patterns—Commentz-Walter (CW, 1979)
- an infinite set of patterns: reversed regular expression—Bucziłowski (BUC)

Boyer-Moore-Horspool algorithm

```
var: TEXT: array[1..T] of char;
    PATTERN: array[1..P] of char; I,J: integer; FOUND: boolean;
FOUND := false;    I := P;
while (I ≤ T) and not FOUND do
        J := 0;
        while (J < P) and (PATTERN[P − J] = TEXT[I − J]) do
                J := J + 1;
        end while
        FOUND := (J = P);

        if not FOUND then
                I := I + SHIFT(TEXT[I − J], J)
        end if
end while
```

$\texttt{SHIFT}(A, J)$ = **if** A does not occur in the not yet compared part of the pattern **then** $P - J$ **else** the smallest $0 \le K < P$ such that $PATTERN[P - (J + K)] = A$;

When is it faster than KMP? When $O(T/P)$?
The time complexity $O(T+P)$.

Example: searching for the pattern `BANANA` in text
`I-WANT-TO-FLAVOR-NATURAL-BANANAS`.

When is it faster than KMP? When $O(T/P)$?

The time complexity $O(T+P)$.

Example: searching for the pattern `BANANA` in text `I-WANT-TO-FLAVOR-NATURAL-BANANAS`.

When is it faster than KMP? When $O(T/P)$?
The time complexity $O(T+P)$.

Example: searching for the pattern `BANANA` in text
`I-WANT-TO-FLAVOR-NATURAL-BANANAS`.

When is it faster than KMP? When $O(T/P)$?
The time complexity $O(T + P)$.

Example: searching for the pattern `BANANA` in text
`I-WANT-TO-FLAVOR-NATURAL-BANANAS`.

CW algorithm

The idea: AC + right-to-left search (BM) [1979]

```
const LMIN=/the length of the shortest pattern/
var TEXT: array [1..T] of char; I, J: integer;
    FOUND: boolean;  STATE: TSTATE;
    g: array [1..MAXSTATE,1..MAXSYMBOL] of TSTATE;
    F: set of TSTATE;
begin
 FOUND:=FALSE; STATE:=q0; I:=LMIN; J:=0;
 while (I<=T) & not (FOUND) do
  begin
   if g[STATE, TEXT[I-J]]=fail
    then begin I:=I+SHIFT[STATE, TEXT[I-J]];
               STATE:=q0; J:=0;
         end
    else begin STATE:=g[STATE, TEXT[I-J]]; J:=J+1 end
   FOUND:=STATE in F
  end
end
```

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: *we construct the function g and introduce the evaluation of the individual states w:*

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $a b_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) = \mathit{fail}$ for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: *we construct the function g and introduce the evaluation of the individual states w:*

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $a b_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) =$ fail for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: we construct the function g and introduce the evaluation of the individual states w:

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $a b_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) =$ fail for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: *we construct the function g and introduce the evaluation of the individual states w:*

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $ab_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) =$ fail for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: *we construct the function g and introduce the evaluation of the individual states w:*

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $ab_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) = \text{fail}$ for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \ldots, v_k\}$
OUTPUT: CW search engine
METHOD: we construct the function g and introduce the evaluation of the individual states w:

1. An initial state q_0; $w(q_0) = \varepsilon$.
2. Each state of the search engine corresponds to the suffix $b_m b_{m+1} \ldots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $ab_m b_{m+1} \ldots b_n$ of a pattern v_i: $w(q) = b_n \ldots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) =$ fail for every q and a, for which $g(q, a)$ was not defined in the step 2.
4. Each state, that correspond to the full pattern, is a final one.

CW—the function *shift*

Definition: $shift[STATE, TEXT[I-J]] = \min\{A, shift2(STATE)\}$, where $A = \max\{shift1(STATE), char(TEXT[I-J]) - J - 1\}$.

The functions are defined this way:

1. $char(a)$ is defined for all the symbols from the alphabet T as the least depth of a state, to that the CW search engine passes through a symbol a. If the symbol a is not in any pattern, then $char(a) = \text{LMIN} + 1$, where LMIN is the length of the shortest pattern. Formally:
$char(a) = \min\{\text{LMIN} + 1, \min\{d(q) | w(q) = xa,\ x \in T^*\}\}$.
2. Function $shift1(q_0) = 1$; for the other states, the value is $shift1(q) = \min\{\text{LMIN}, A\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is its own suffix $w(q')$ and a state q' has higher depth than $q\}$.
3. Function $shift2(q_0) = \text{LMIN}$; for the other states, the value is $shift2(q) = \min\{A, B\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is a proper suffix $w(q')$ and q' is a final state$\}$, $B = shift2(q') | q'$ is a predecessor of q.

CW—the function *shift*

Definition: $shift[STATE, TEXT[I-J]] = \min\{A, shift2(STATE)\}$, where $A = \max\{shift1(STATE), char(TEXT[I-J]) - J - 1\}$.

The functions are defined this way:

1. $char(a)$ is defined for all the symbols from the alphabet T as the least depth of a state, to that the CW search engine passes through a symbol a. If the symbol a is not in any pattern, then $char(a) = \text{LMIN} + 1$, where LMIN is the length of the shortest pattern. Formally: $char(a) = \min\{\text{LMIN} + 1, \min\{d(q) | w(q) = xa,\ x \in T^*\}\}$.
2. Function $shift1(q_0) = 1$; for the other states, the value is $shift1(q) = \min\{\text{LMIN}, A\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is its own suffix $w(q')$ and a state q' has higher depth than $q\}$.
3. Function $shift2(q_0) = \text{LMIN}$; for the other states, the value is $shift2(q) = \min\{A, B\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is a proper suffix $w(q')$ and q' is a final state$\}$, $B = shift2(q') | q'$ is a predecessor of q.

CW—the function *shift*

Definition: $shift[STATE, TEXT[I-J]] = \min\{A, shift2(STATE)\}$, where $A = \max\{shift1(STATE), char(TEXT[I-J]) - J - 1\}$.

The functions are defined this way:

1. $char(a)$ is defined for all the symbols from the alphabet T as the least depth of a state, to that the CW search engine passes through a symbol a. If the symbol a is not in any pattern, then $char(a) = \mathrm{LMIN} + 1$, where LMIN is the length of the shortest pattern. Formally: $char(a) = \min\{\mathrm{LMIN} + 1, \min\{d(q) | w(q) = xa,\ x \in T^*\}\}$.
2. Function $shift1(q_0) = 1$; for the other states, the value is $shift1(q) = \min\{\mathrm{LMIN}, A\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is its own suffix $w(q')$ and a state q' has higher depth than $q\}$.
3. Function $shift2(q_0) = \mathrm{LMIN}$; for the other states, the value is $shift2(q) = \min\{A, B\}$, where $A = \min\{k | k = d(q') - d(q)$, where $w(q)$ is a proper suffix $w(q')$ and q' is a final state$\}$, $B = shift2(q') | q'$ is a predecessor of q.

CW—the function *shift*

Example: $P = \{cacbaa, aba, acb, acbab, ccbab\}$.

LMIN = 3,

	a	b	c	X
char	1	1	2	4

$w(q)$	*shift1*	*shift2*
ε	1	3
a	1	2
b	1	3
aa	3	2
ab	1	2
bc	2	3
ba	1	1
aab	3	2
aba	3	2
bca	2	2
bab	3	1
aabc	3	2
babc	3	1
aabca	3	2
babca	3	1
babcc	3	1
aabcac	3	2

CW—the function *shift*

Example: $P = \{cacbaa, aba, acb, acbab, ccbab\}$.

LMIN = 3,

	a	b	c	X
char	1	1	2	4

$w(q)$	*shift1*	*shift2*
ε	1	3
a	1	2
b	1	3
aa	3	2
ab	1	2
bc	2	3
ba	1	1
aab	3	2
aba	3	2
bca	2	2
bab	3	1
aabc	3	2
babc	3	1
aabca	3	2
babca	3	1
babcc	3	1
aabcac	3	2

CW—the function *shift*

Example: $P = \{cacbaa, aba, acb, acbab, ccbab\}$.

LMIN = 3,

	a	b	c	X
char	1	1	2	4

$w(q)$	*shift1*	*shift2*
ε	1	3
a	1	2
b	1	3
aa	3	2
ab	1	2
bc	2	3
ba	1	1
aab	3	2
aba	3	2
bca	2	2
bab	3	1
aabc	3	2
babc	3	1
aabca	3	2
babca	3	1
babcc	3	1
aabcac	3	2