PV030 Textual Information Systems

Petr Sojka

Faculty of Informatics
Masaryk University, Brno

Spring 2013

Outline (week four)

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 four)

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 four)

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 four)

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 four)

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 four)

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.

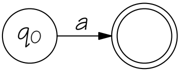
Construction of NFA for given RE

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

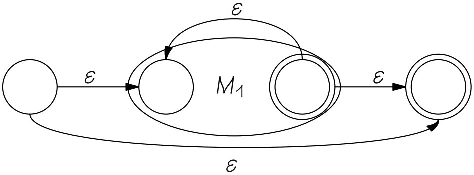
Theorem: for every RE E, we can create FA M such that $h(E) = L(M)$.
Proof: by structural induction relative to the RE E:

Construction of NFA for given RE (a proof)

① $E = a$

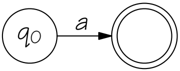


② $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

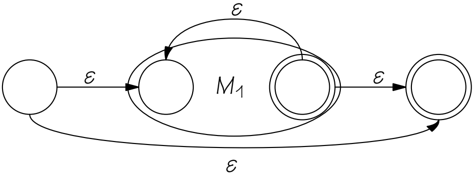


Construction of NFA for given RE (a proof)

1. $E = a$

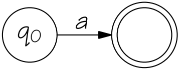


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

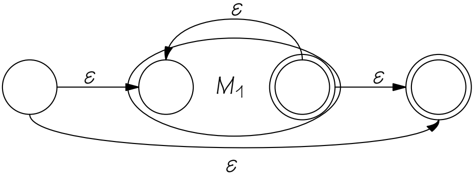


Construction of NFA for given RE (a proof)

1. $E = a$

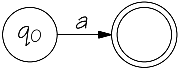


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

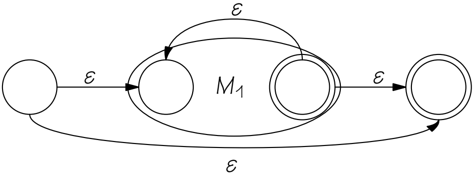


Construction of NFA for given RE (a proof)

1. $E = a$

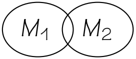


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

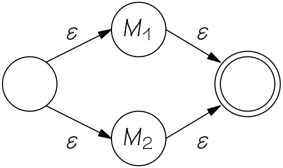


Construction of NFA for given RE (cont. of a proof)

③ $E = E_1 \cdot E_2$

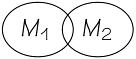


④ $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$, $h(E_2) = L(M_2)$)

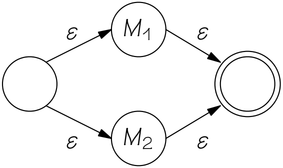


Construction of NFA for given RE (cont. of a proof)

3. $E = E_1 \cdot E_2$



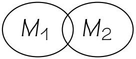
4. $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$,



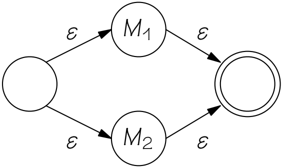
$h(E_2) = L(M_2)$)

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

3. $E = E_1 \cdot E_2$



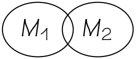
4. $E = E_1 + E_2$ M_1, M_2 automata for E_1, E_2 ($h(E_1) = L(M_1)$,



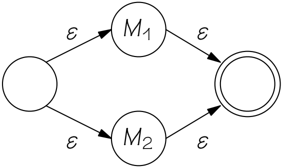
$h(E_2) = L(M_2)$)

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:

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

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:

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

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

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

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

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

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:

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

Derivation of a regular expression (cont.)

③ For $x = a_1 a_2 \dots a_n$, $a_i \in T$, 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 $\frac{dF}{d\,xa}$. (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 $F = \left\{\frac{dF}{dx} \in Q : \varepsilon \in h\left(\frac{dF}{dx}\right)\right\}$.

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: <u>Position vector</u> 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:

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

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:

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

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:

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

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:

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

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

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

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

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

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

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

   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:

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

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 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 I

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—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—Buczi łowski (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—Buczikówski (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—Buczіłowski (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*—Buczіłowski (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
```

SHIFT(A,J) = **if** A does not occur in the not yet compared part of the pattern **then** $P - J$ **else** the smallest $0 \leq K < P$ 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. $g(q, a) =$ 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.

Construction of the CW search engine

INPUT: a set of patterns $P = \{v_1, v_2, \dots, v_k\}$
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 $b_m b_{m+1} \dots b_n$ of a pattern v_i of the set P. Let us define $g(q, a) = q'$, where q' corresponds to the suffix $ab_m b_{m+1} \dots b_n$ of a pattern v_i: $w(q) = b_n \dots b_{m+1} b_m$; $w(q') = w(q)a$.
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 $a b_m b_{m+1} \ldots b_n$ 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. $g(q, a) = \text{fail}$ 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.

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*

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*

Definition: $shift[STATE, TEXT[I-J]] = \min\left\{A, shift2(STATE)\right\}$,
where $A = \max\left\{shift1(STATE), char(TEXT[I-J]) - J - 1\right\}$.
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:
$char(a) = \min\left\{\mathrm{LMIN}+1, \min\{d(q) | w(q) = xa,\ x \in T^*\}\right\}$.
2. Function $shift1(q_0) = 1$; for the other states, the value is $shift1(q) = \min\left\{\mathrm{LMIN}, A\right\}$, 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

Outline (week four)

1. Right-to-left search of an infinite set of patterns
2. Two-way jump automaton – a generalization of the so far learned left-to-right and right-to-left algorithms.
3. Hierarchy of the exact search engines.

Outline (week four)

1. Right-to-left search of an infinite set of patterns
2. Two-way jump automaton – a generalization of the so far learned left-to-right and right-to-left algorithms.
3. Hierarchy of the exact search engines.

Outline (week four)

1. Right-to-left search of an infinite set of patterns
2. Two-way jump automaton – a generalization of the so far learned left-to-right and right-to-left algorithms.
3. Hierarchy of the exact search engines.

Part II

Search for an infinite set of patterns

Right-to-left search for an inf. set of patterns

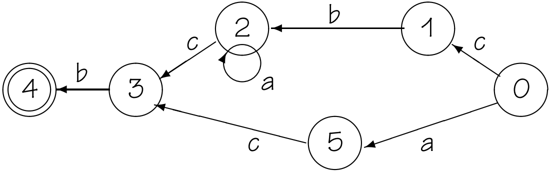
Generalization of SE

Search engine hierarchy

Right-to-left search for an inf. set of patterns

Definition: ***reversed regular expression*** is created by reversion of all concatenation in the expression.

Example: reversed RE for $E = bc(a + a^*bc)$ is $E^R = (a + cba^*)cb$:



Right-to-left search for an inf. set of patterns (cont.)

Bucziłowski: *we search for E such that we create E^R* and we use it for *determination of shift[STATE, SYMBOL] for each state and undefined* transition analogically as in the CW algorithm:

	a	b	c	X
0		1		3 .
1	1		1	2 (3!) .
2		1		
3	1		1	1
4	1	1	1	1
5	1	1		1
	.	.		.

Two-way jump automaton I

Definition: ***2DFAS*** is $M = (Q, \Sigma, \delta, q_0, k, \uparrow, F)$, where

Q a set of states

Σ an input alphabet

δ a projection. $Q \times \Sigma \longrightarrow Q \times \{-1, 1, \ldots, k\}$

$q_0 \in Q$ an initial state

$k \in N$ max. length of a jump

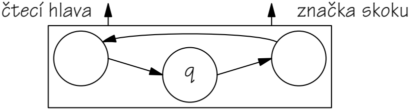
$\uparrow \notin Q \cup \Sigma$ a jump symbol

$F \subseteq Q$ a set of final states

Definition: ***a configuration of 2DFAS*** is a string of $\Sigma^* Q \Sigma^* \uparrow \Sigma^*$.

Definition: we denote ***a set of configurations 2DFAS*** M as $K(M)$.

Example: $a_1 a_2 \ldots a_{i-1}\ q\ a_i \ldots a_{j-1} \uparrow a_j \ldots a_n \in K(M)$:



Two-way jump automaton II

Definition: ***a transition of 2DFAS*** is a relation $\vdash \subseteq K(M) \times K(M)$ such that

- $a_1 \dots a_{i-1} a_i\ q\ a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_{i-1}\ q'\ a_i a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n$ for $i > 1$, $\delta(q, a_{i+1}) = (q', -1)$ (right-to-left comparison),
- $a_1 \dots a_i\ q\ a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_i a_{i+1} \dots a_{t-1}\ q' \uparrow a_t \dots a_n$ for $\delta(q, a_{i+1}) = (q', m), \quad m \geq 1, \quad t = \min\{j + m, n + 1\}$ (right-to-left jump),
- $a_1 \dots a_j\ q\ a_{j+1} \dots a_{i-1} \uparrow a_i \dots a_n \vdash a_1 \dots a_j a_{j+1} \dots a_{t-1}\ q' \uparrow a_t \dots a_n$ for $\delta(q, a_i) = (q', m), \quad m \geq 1, \quad t = \min\{i + m, n + 1\}$ (left-to-right jump), .
- $a_1 \dots a_{j-1}\ q\ a_j \dots a_{i-1} \uparrow a_i a_{i+1} \dots a_n \vdash a_1 \dots a_{j-1}\ q'\ a_j \dots a_{i-1} a_i \uparrow a_{i+1} \dots a_n$ for $i > 1$, $\delta(q, a_i) = (q', 1)$ (left-to-right comparison).

(Left-to-right rules are for the left-to-right engines and vice versa.)

Definition: $\vdash^k$, $\vdash^*$ analogically as in the SE.

Two-way jump automaton II

Definition: ***a transition of 2DFAS*** is a relation $\vdash \subseteq K(M) \times K(M)$ such that

- $a_1 \dots a_{i-1} a_i\ q\ a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_{i-1}\ q'\ a_i a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n$ for $i > 1$, $\delta(q, a_{i+1}) = (q', -1)$ (right-to-left comparison),
- $a_1 \dots a_i\ q\ a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_i a_{i+1} \dots a_{t-1}\ q' \uparrow a_t \dots a_n$ for $\delta(q, a_{i+1}) = (q', m), \quad m \geq 1, \quad t = \min\{j + m, n + 1\}$ (right-to-left jump),
- $a_1 \dots a_j\ q\ a_{j+1} \dots a_{i-1} \uparrow a_i \dots a_n \vdash a_1 \dots a_j a_{j+1} \dots a_{t-1}\ q' \uparrow a_t \dots a_n$ for $\delta(q, a_i) = (q', m), \quad m \geq 1, \quad t = \min\{i + m, n + 1\}$ (left-to-right jump), .
- $a_1 \dots a_{j-1}\ q\ a_j \dots a_{i-1} \uparrow a_i a_{i+1} \dots a_n \vdash a_1 \dots a_{j-1}\ q'\ a_j \dots a_{i-1} a_i \uparrow a_{i+1} \dots a_n$ for $i > 1$, $\delta(q, a_i) = (q', 1)$ (left-to-right comparison).

(Left-to-right rules are for the left-to-right engines and vice versa.)

Definition: $\vdash^k$, $\vdash^*$ analogically as in the SE.

Two-way jump automaton II

Definition: ***a transition of 2DFAS*** is a relation $\vdash \subseteq K(M) \times K(M)$ such that

- $a_1 \ldots a_{i-1} a_i\ q\ a_{i+1} \ldots a_{j-1} \uparrow a_j \ldots a_n \vdash a_1 \ldots a_{i-1}\ q'\ a_i a_{i+1} \ldots a_{j-1} \uparrow a_j \ldots a_n$ for $i > 1$, $\delta(q, a_{i+1}) = (q', -1)$ (right-to-left comparison),
- $a_1 \ldots a_i\ q\ a_{i+1} \ldots a_{j-1} \uparrow a_j \ldots a_n \vdash a_1 \ldots a_i a_{i+1} \ldots a_{t-1}\ q' \uparrow a_t \ldots a_n$ for $\delta(q, a_{i+1}) = (q', m), \quad m \geq 1, \quad t = \min\{j + m, n + 1\}$ (right-to-left jump),
- $a_1 \ldots a_j\ q\ a_{j+1} \ldots a_{i-1} \uparrow a_i \ldots a_n \vdash a_1 \ldots a_j a_{j+1} \ldots a_{t-1}\ q' \uparrow a_t \ldots a_n$ for $\delta(q, a_i) = (q', m), \quad m \geq 1, \quad t = \min\{i + m, n + 1\}$ (left-to-right jump), .
- $a_1 \ldots a_{j-1}\ q\ a_j \ldots a_{i-1} \uparrow a_i a_{i+1} \ldots a_n \vdash a_1 \ldots a_{j-1}\ q'\ a_j \ldots a_{i-1} a_i \uparrow a_{i+1} \ldots a_n$ for $i > 1$, $\delta(q, a_i) = (q', 1)$ (left-to-right comparison).

(Left-to-right rules are for the left-to-right engines and vice versa.)

Definition: $\vdash^k$, $\vdash^*$ analogically as in the SE.

Two-way jump automaton II

Definition: ***a transition of 2DFAS*** is a relation $\vdash \subseteq K(M) \times K(M)$ such that

- $a_1 \dots a_{i-1} a_i \; q \; a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_{i-1} \; q' \; a_i a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n$ for $i > 1$, $\delta(q, a_{i+1}) = (q', -1)$ (right-to-left comparison),
- $a_1 \dots a_i \; q \; a_{i+1} \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_i a_{i+1} \dots a_{t-1} \; q' \uparrow a_t \dots a_n$ for $\delta(q, a_{i+1}) = (q', m), \quad m \geq 1, \quad t = \min\{j+m, n+1\}$ (right-to-left jump),
- $a_1 \dots a_j \; q \; a_{j+1} \dots a_{i-1} \uparrow a_i \dots a_n \vdash a_1 \dots a_j a_{j+1} \dots a_{t-1} \; q' \uparrow a_t \dots a_n$ for $\delta(q, a_i) = (q', m), \quad m \geq 1, \quad t = \min\{i+m, n+1\}$ (left-to-right jump), .
- $a_1 \dots a_{j-1} \; q \; a_j \dots a_{i-1} \uparrow a_i a_{i+1} \dots a_n \vdash a_1 \dots a_{j-1} \; q' \; a_j \dots a_{i-1} a_i \uparrow a_{i+1} \dots a_n$ for $i > 1$, $\delta(q, a_i) = (q', 1)$ (left-to-right comparison).

(Left-to-right rules are for the left-to-right engines and vice versa.)

Definition: $\vdash^k$, $\vdash^*$ analogically as in the SE.

Search engine hierarchy

Definition: ***the language accepted by the two-way automaton*** $M = (Q, \Sigma, \delta, q_0, k, \uparrow, F)$ is a set $L(M) = \{w \in \Sigma^* : q_0 \uparrow T \vdash^* w' f x w \uparrow,$ where $f \in F, w' \in \Sigma^*, x \in \Sigma\}$.

Theorem: $L(M)$ for 2DKAS M is regular.

Example: formulate a right-to-left search of the pattern BANANA in the text I-WANT-TO-FLAVOUR-NATURAL-BANANAS using BM as 2DFAS and trace the search as a sequence of configurations of the 2DFAS.

Search engine hierarchy

Definition: ***the language accepted by the two-way automaton*** $M = (Q, \Sigma, \delta, q_0, k, \uparrow, F)$ is a set $L(M) = \{w \in \Sigma^* : q_0 \uparrow T \vdash^* w'fxw \uparrow,$ where $f \in F, w' \in \Sigma^*, x \in \Sigma\}$.

Theorem: $L(M)$ for $2DKAS$ M is regular.
Example: formulate a right-to-left search of the pattern BANANA in the text I-WANT-TO-FLAVOUR-NATURAL-BANANAS using BM as 2DFAS and trace the search as a sequence of configurations of the 2DFAS.

Exercise

Let us have a regular expression $R = 1(0 + 1^*02)$ over the alphabet $A = \{0, 1, 2\}$.

- ☞ Construct a right-to-left DFA R (Buczilowski) and compute the failure function. Draw the transition graph of this automaton including the failure function visualization.
- ☞ Express the resulting automaton as 2DFAS and trace searching in the text 11201012102.

Summary of the exact search

$$
\begin{array}{cccccc}
 & & \text{2DFAS} & & & \\
 & \swarrow & & \searrow & & \\
\text{DFA} & & & & \text{BUC} & \infty \\
\downarrow & & & & \downarrow & \\
\text{AC} & & & & \text{CW} & k \\
\downarrow & & & & \downarrow & \\
\text{KMP} & & & & \text{BM} & 1 \\
\rightarrow & & & & \leftarrow \text{ direction} & \text{— \# of patterns.}
\end{array}
$$

Outline (week four)

1. Fuzzy (proximity) search. Metrics for measurement of distance of strings.
2. Classification of search: 6D space of search problems.
3. Examples of creation of search engines.
4. Completion of the chapter about searching without text preprocessing.
5. Indexing basics.

Outline (week four)

1. Fuzzy (proximity) search. Metrics for measurement of distance of strings.
2. Classification of search: 6D space of search problems.
3. Examples of creation of search engines.
4. Completion of the chapter about searching without text preprocessing.
5. Indexing basics.

Outline (week four)

1. Fuzzy (proximity) search. Metrics for measurement of distance of strings.
2. Classification of search: 6D space of search problems.
3. Examples of creation of search engines.
4. Completion of the chapter about searching without text preprocessing.
5. Indexing basics.

Outline (week four)

1. Fuzzy (proximity) search. Metrics for measurement of distance of strings.
2. Classification of search: 6D space of search problems.
3. Examples of creation of search engines.
4. Completion of the chapter about searching without text preprocessing.
5. Indexing basics.

Outline (week four)

1. Fuzzy (proximity) search. Metrics for measurement of distance of strings.
2. Classification of search: 6D space of search problems.
3. Examples of creation of search engines.
4. Completion of the chapter about searching without text preprocessing.
5. Indexing basics.

Part III

Proximity search

Fuzzy search: metrics

Classification of search problems

Metrics (for proximity search)

How to measure (metrics) the similarity of strings?

Definition: we call $d : S \times S \to R$ ***metrics*** if the following is true:

1. $d(x, y) \geq 0$
2. $d(x, x) = 0$
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, y) = 0 \Rightarrow x = y$ (identity of indiscernibles)
5. $d(x, y) + d(y, z) \geq d(x, z)$ (triangle inequality)

We call the values of the function d (distance).

Metrics (for proximity search)

How to measure (metrics) the similarity of strings?

Definition: we call $d: S \times S \to R$ ***metrics*** if the following is true:

1. $d(x, y) \geq 0$
2. $d(x, x) = 0$
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, y) = 0 \Rightarrow x = y$ (identity of indiscernibles)
5. $d(x, y) + d(y, z) \geq d(x, z)$ (triangle inequality)

We call the values of the function d (distance).

Metrics for proximity search

Definition: let us have strings X and Y over the alphabet Σ. The minimal number of editing operation for transformation X to Y is

- ***Hamming distance***, R-distance, when we allow just the operation Replace,
- ***Levenshtein distance***, DIR-distance, when we allow the operations Delete, Insert and Replace,
- ***Generalized Levenshtein distance***, $DIRT$-distance, when we allow the operations Delete, Insert, Replace and Transpose. Transposition is possible at the neighbouring characters only.

They are ***metrics***, Hamming must be performed over strings of the same length, Levenshtein can be done over the different lengths.

Metrics for proximity search

Definition: let us have strings X and Y over the alphabet Σ. The minimal number of editing operation for transformation X to Y is

- ***Hamming distance***, R-distance, when we allow just the operation **R**eplace,
- ***Levenshtein distance***, DIR-distance, when we allow the operations **D**elete, **I**nsert and **R**eplace,
- ***Generalized Levenshtein distance***, $DIRT$-distance, when we allow the operations **D**elete, **I**nsert, **R**eplace and **T**ranspose. Transposition is possible at the neighbouring characters only.

They are ***metrics***, Hamming must be performed over strings of the same length, Levenshtein can be done over the different lengths.

Metrics for proximity search

Definition: let us have strings X and Y over the alphabet Σ. The minimal number of editing operation for transformation X to Y is

- ***Hamming distance***, R-distance, when we allow just the operation **R**eplace,
- ***Levenshtein distance***, DIR-distance, when we allow the operations **D**elete, **I**nsert and **R**eplace,
- ***Generalized Levenshtein distance***, $DIRT$-distance, when we allow the operations **D**elete, **I**nsert, **R**eplace and **T**ranspose. Transposition is possible at the neighbouring characters only.

They are ***metrics***, Hamming must be performed over strings of the same length, Levenshtein can be done over the different lengths.

Metrics for proximity search

Definition: let us have strings X and Y over the alphabet Σ. The minimal number of editing operation for transformation X to Y is

- ***Hamming distance***, R-distance, when we allow just the operation **R**eplace,
- ***Levenshtein distance***, DIR-distance, when we allow the operations **D**elete, **I**nsert and **R**eplace,
- ***Generalized Levenshtein distance***, $DIRT$-distance, when we allow the operations **D**elete, **I**nsert, **R**eplace and **T**ranspose. Transposition is possible at the neighbouring characters only.

They are ***metrics***, Hamming must be performed over strings of the same length, Levenshtein can be done over the different lengths.

Proximity search—examples

Example: Find such an example of strings X and Y, that simultaneously holds $R(X,Y) = 5$, $DIR(X,Y) = 5$, and $DIRT(X,Y) = 5$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y, that holds simultaneously $R(X,Y) = 5$, $DIR(X,Y) = 4$, and $DIRT(X,Y) = 3$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y of the length $2n$, $n \in N$, that $R(X,Y) = 2n$ and a) $DIR(X,Y) = 2$; b) $DIRT(X,Y) = \lceil \frac{n}{2} \rceil$

Proximity search—examples

Example: Find such an example of strings X and Y, that simultaneously holds $R(X, Y) = 5$, $DIR(X, Y) = 5$, and $DIRT(X, Y) = 5$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y, that holds simultaneously $R(X, Y) = 5$, $DIR(X, Y) = 4$, and $DIRT(X, Y) = 3$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y of the length $2n$, $n \in N$, that $R(X, Y) = 2n$ and a) $DIR(X, Y) = 2$; b) $DIRT(X, Y) = \lceil \frac{n}{2} \rceil$

Proximity search—examples

Example: Find such an example of strings X and Y, that simultaneously holds $R(X, Y) = 5$, $DIR(X, Y) = 5$, and $DIRT(X, Y) = 5$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y, that holds simultaneously $R(X, Y) = 5$, $DIR(X, Y) = 4$, and $DIRT(X, Y) = 3$, or prove the non-existence of such strings.

Example: find such an example of strings X and Y of the length $2n$, $n \in N$, that $R(X, Y) = 2n$ and a) $DIR(X, Y) = 2$; b) $DIRT(X, Y) = \lceil \frac{n}{2} \rceil$

Classification of search problems

Definition: Let $T = t_1 t_2 \ldots t_n$ and pattern $P = p_1 p_2 \ldots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \ldots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing **don't care symbol** $\emptyset$ (*) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing **don't care symbol** $\emptyset$ (*) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing ***don't care symbol*** $\varnothing$ (*) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing ***don't care symbol*** $\emptyset$ (*) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing ***don't care symbol*** $\emptyset$ (*) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing ***don't care symbol*** $\emptyset$ (*****) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.

Classification of search problems

Definition: Let $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$. For example, we can ask:

1. is P a substring of T?
2. is P a subsequence of T?
3. is a substring or a subsequence P in T?
4. is P in T such that $D(P, X) \leq k$ for $k < m$, where $X = t_i \dots t_j$ is a part of T (D is R, DIR or $DIRT$)?
5. is a string P containing ***don't care symbol*** $\emptyset$ (*****) in T?
6. is a sequence of patterns P in T?

Furthermore, the variants for more patterns, plus instances of the search problem yes/no, the first occurrence, all the overlapping occurrences, all the also non-overlapping occurrences.