Outline (week four)

① Summary of the previous lecture.

② Regular expressions, the 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.

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.

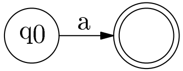
Construction of NFA for given RE

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

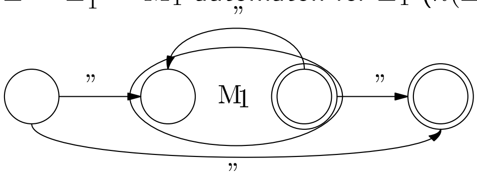
Theorem: for every RE E, we can create FA M such that $h(E) = L(M)$.
Proof: by structural induction relative to the RE E:

Construction of NFA for given RE (a proof)

1. $E = a$

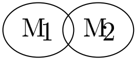


2. $E = E_1^*$ M_1 automaton for E_1 $(h(E_1) = L(M_1))$

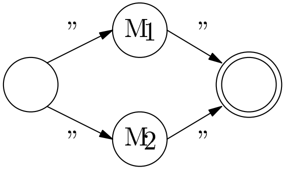


Construction of NFA for given RE (cont. of a proof)

3. $E = E_1 \cdot E_2$



4. $E = E_1 + E_2 \quad M_1, M_2$ automata for E_1, E_2



$(h(E_1) = L(M_1),\ h(E_2) = L(M_2))$

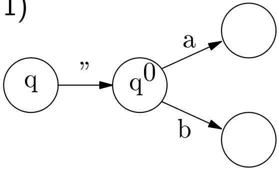
Construction of NFA for given RE (cont.)

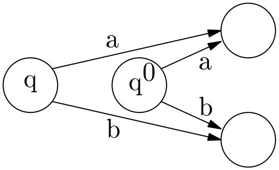
- ☞ No more than two edges come out of every state.
- ☞ No edges come out of the final states.
- ☞ The number of the states $M \leq 2 \cdot d(E)$.
- ☞ The simulation of automaton M is performed in $O(d(E)T)$ time and in $O(d(E))$ space.

NFA simulation

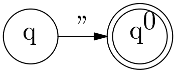
For the following methods of NFA simulation, we must remove the ε-transitions. We can achieve it with the well-known procedure:

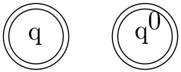
1)





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.

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 <u>the occurrences</u> 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:

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

Derivation of a regular expression (cont.)

③ For $x = a_1a_2 \dots a_n$, $a_i \in T$, 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 $(\mathrm{o}^*\mathrm{sle})^*\mathrm{cno}$ 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 by axioms of the axiomatic theory of RE (Salomaa).

Example: Is there a RE similar to $E = fi + fi^* + f^*ifi$ that has lengths 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 $\frac{dF}{d\,xa}$. (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`

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 sign $\wedge$. 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 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 VII

Right-to-left search

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: reverse regular expression—Bucziłowski (BUC).

Boyer-Moore-Horspool algorithm

```
1: var: TEXT: array[1..T] of char;
2:    $PATTERN$: array[1..P] of char; I,J: integer; $FOUND$: boolean;
3: $FOUND := false$;    $I := P$;
4: while $(I \leq T)$ and not $FOUND$ do
5:       $J := 0$;
6:       while $(J < P)$ and $(PATTERN[P - J] = TEXT[I - J])$ do
7:             $J := J + 1$;
8:       end while
9:       $FOUND := (J = P)$;
10:
11:       if not FOUND then
12:             $I := I + SHIFT(TEXT[I - J], J)$
13:       end if
14: 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`.

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 $P = \{v_1, v_2, \dots, v_k\}$
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 $a b_m b_{m+1} \dots b_n$ of a pattern v_i: $w(q) = b_n \dots b_{m+1} b_m$; $w(q') = w(q)a$.
3. $g(q, a) = \utilde{\text{fail}}$ 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 *shift*function

Definition: $\textit{shift}[STATE, TEXT[I-J]] =$
$\min\{A, \textit{shift}2(STATE)\}$, where
$A = \max\{\textit{shift}1(STATE), \textit{char}(TEXT[I-J]) - J - 1\}$.
Particular functions are defined this way:

1. $\textit{char}(a)$ 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 $\textit{char}(a) = \text{LMIN} + 1$, where LMIN is the length of the shortest pattern. Formally: $\textit{char}(a) = \min\{\text{LMIN} + 1, \min\{d(q) | \, w(q) = xa, \, x \in T^*\}\}$.
2. Function $\textit{shift}1(q_0) = 1$; for the other states, the value is $\textit{shift}1(q) = \min\{\text{LMIN}, A\}$, 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 $\textit{shift}2(q_0) = \text{LMIN}$; for the other states, the value is $\textit{shift}2(q) = \min\{A, B\}$, where $A = \min\{k | \, k = d(q') - d(q)$, where $w(q)$ is a proper suffix $w(q')$ and q' is a final state$\}$, $B = \textit{shift}2(q') | \, q'$ is a predecessor of q.

CW—the *shift*function

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

LMIN $= 3$,

	a	b	c	X
char	1	1	2	4

$w(q)$	*shift*1	*shift*2
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