Advanced Algorithmics (6EAP)


MTAT.03.238

Regular expressions and automata

Jaak Vilo

2010 Spring
Contents

• Regular languages
• Automata
  – Deterministic finite automata DFA
  – Nondeterministic finite automata NFA
• Regular expressions
• Mapping to NFA
• NFA to DFA
• Matching
• ...
Links

• **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002), ch. 5: Regular Expression Matching (pp. 99--143)

• Regular expression search using a DFA (relative difficulty: medium-hard) [ASU1986, pp. 92-105, 113-146], [NaRa2002, ch. 5], [Orponen1994, ch. 2]


• **Teoreetiline Informaatika** (Jaan Penjam, TTÜ), Peatükk 5.

• **Regulaarsest avaldisest mittedetermineeritud automaadi moodustamine** (Meelis Roos) (kohalik)

• Google – *Query*


• **GNU grep manual** (grep = Global Search for Regular Expression and Print)

• **FSA Utilities toolbox** FSA Utilities toolbox: a collection of utilities to manipulate regular expressions, finite-state automata and finite-state transducers. (Gertjan van Noord)

• **Finnish-language course Models for Programming and Computing** - essential regular expressions and automata theory...

• [http://www.regular-expressions.info/](http://www.regular-expressions.info/)
Regular expression

- **Definition:** A *regular expression* RE is a string on the set of symbols $\Sigma \cup \{ \varepsilon, |, \cdot, *, (, ) \}$, which is recursively defined as follows. RE is
  - an empty character $\varepsilon$,
  - a character $\alpha \in \Sigma$,
  - $(RE_1)$,
  - $(RE_1 \cdot RE_2)$,
  - $(RE_1 | RE_2)$, and
  - $(RE_1*)$,
- where $RE_1$ and $RE_2$ are regular expressions
Example

$((((A \cdot T) \mid (G \cdot A)) \cdot (((A \cdot G) \mid ((A \cdot A) \cdot A))^{*}))$

• we can simplify

$(AT \mid GA)((AG \mid AAA)^{*})$

• Often also this is used:

$RE^{+} = RE \cdot RE^{*}$
Why?

• Regular expression defines a language

• A set of words from $\Sigma^*$

• A convenient short-hand

• $(AT|GA)((AG|AAA)^*) \Rightarrow AT, ATAG, GAAAA, GAAGAAAAAAA, ...$

• Infinite set
Language represented by RE

**Definition:** A *language represented by a regular expression* RE is a set of strings over $\Sigma$, which is defined recursively on the structure of RE as follows:

- if RE is $\varepsilon$, then $L(\text{RE})=\{\varepsilon\}$, the empty string
- if RE is $\alpha \in \Sigma$, then $L(\text{RE})=\{\alpha\}$, a single string of one character
- if RE is of the form $(\text{RE}_1)$, then $L(\text{RE})=L(\text{RE}_1)$
- if RE is of the form $(\text{RE}_1 \cdot \text{RE}_2)$, then $L(\text{RE})=L(\text{RE}_1) \cdot L(\text{RE}_2)$, where $w=w_1w_2$ is in $L(\text{RE})$ if $w_1 \in L(\text{RE}_1)$ and $w_2 \in L(\text{RE}_2)$. (We call $\cdot$ the concatenation operator)
- if RE is of the form $(\text{RE}_1 \mid \text{RE}_2)$, then $L(\text{RE})=L(\text{RE}_1) \cup L(\text{RE}_2)$, the union of two languages. (We call $\mid$ the union operator)
- if RE is of the form $(\text{RE}_1^*)$, then $L(\text{RE}) = L(\text{RE})^* = \cup_{i \geq 0} L(\text{RE}_1)^i$, where $L^0 = \{\varepsilon\}$ and $L^i = L \cdot L^{i-1}$. (We call $^*$ the star operator)
<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language ( L(\text{RE}) )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>{\varepsilon}</td>
<td>Empty string</td>
</tr>
<tr>
<td>( \alpha \in \Sigma )</td>
<td>{\alpha}</td>
<td>Single character</td>
</tr>
<tr>
<td>( (\text{RE}_1) )</td>
<td>( L(\text{RE}_1) )</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>( (\text{RE}_1 \cdot \text{RE}_2) )</td>
<td>( (\text{RE}_1 \cdot \text{RE}_2) ) ( w = w_1w_2 ) is in ( L(\text{RE}) ) if ( w_1 \in L(\text{RE}_1) ) and ( w_2 \in L(\text{RE}_2) )</td>
<td>Concatenation</td>
</tr>
<tr>
<td>( (\text{RE}_1</td>
<td>\text{RE}_2) )</td>
<td>( L(\text{RE})=L(\text{RE}_1) L(\text{RE}_2) )</td>
</tr>
<tr>
<td>( (\text{RE}_1^*) )</td>
<td>( L(\text{RE}) = L(\text{RE})^* = \bigcup_{i \geq 0} L(\text{RE}_1)^i ) ( L^0 = { \varepsilon } ) and ( L^i = L \cdot L^{i-1} ).</td>
<td>The star operator (Kleene star)</td>
</tr>
<tr>
<td>( (\text{RE}_1^+) )</td>
<td>( L(\text{RE}) = (\text{RE}_1) \cdot (\text{RE}_1^*) )</td>
<td>Kleene plus</td>
</tr>
</tbody>
</table>
• $L( (AT|GA)((AG|AAA)*) ) = \{ AT, GA, ATAG, GAAG, ATAAA, GAAAA, ATAGAG, ATAGAAA, ATAAAAAG, \ldots \}$

• $\Sigma^*$ denotes all strings over alphabet $\Sigma$

• The size of a regular expression RE is the number of characters of $\Sigma$ in it.

• Many complexities are based on this measure.
A different example definition

• Just as finite automata are used to recognize patterns of strings, regular expressions are used to generate patterns of strings. A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression.
• Operands in a regular expression can be:
  • characters from the alphabet over which the regular expression is defined.
  • variables whose values are any pattern defined by a regular expression.
  • epsilon which denotes the empty string containing no characters.
  • null which denotes the empty set of strings.
• Operators used in regular expressions include:
  • * Concatenation: If R1 and R2 are regular expressions, then R1R2 (also written as R1.R2) is also a regular expression.
    L(R1R2) = L(R1) concatenated with L(R2).
  • * Union: If R1 and R2 are regular expressions, then R1 | R2 (also written as R1 U R2 or R1 + R2) is also a regular expression.
    L(R1|R2) = L(R1) U L(R2).
  • * Kleene closure: If R1 is a regular expression, then R1* (the Kleene closure of R1) is also a regular expression.
    L(R1*) = epsilon U L(R1) U L(R1R1) U L(R1R1R1) U ...
• Closure has the highest precedence, followed by concatenation, followed by union.
• The problem of searching regular expression RE in a text T is to find all the factors of T that belong to the language L(RE).
  • Parsing
  • Thompsons NFA construction (1968)
    Glushkov NFA construction (1961)
  • Search with the NFA
  • Determinization
  • Search with the DFA
  • Or, search by extracting set of strings, multi-pattern matching, and verification of real occurrences.
Matching of RE-s

- Regular expression
- Parse
- NFA
- DFA
- Occurrences
Parse tree

( A T I G A )(( A G I A A A A )*)
Q: what is the language?
Function IsDigit(c) { if ( c ∈ { 0,1,...,9 } ) return 1 else return 0 }

int q=0 ; // current state
int sign=1 ; // sign of the value int
val=0 ; // value of the number
while( ( c=getc() ) != EOF )
    switch ( q ) {
        case 0 : if c ∈ { '+', '-' } q = 1
                if ( c == '-' ) sign = -1
                elsif IsDigit(c)
                    val = c - '0' // numeric value of c
                    q = 2
                else q = 99
                break ;
        case 1 : if IsDigit(c)
                    val = c - '0'
                    q = 2
                else q = 99
                break ;
        case 2 :
            if IsDigit(c)
                val = 10*val + ( c - '0')
                q = 2
            else q = 99
            break ;
        case 99 : break ;
    }
if( q == 2 )
    then print 'The value of the number is ', sign*val
else print ‘Does not match the automaton for signed integers’
Lõplik automaat (näide)

Deterministic finite automaton DFA

**Definition** DFA is a quintuple $M=\langle Q, \Sigma, \delta, q_0, F \rangle$, where

- $Q$ is the finite set of states of an automaton
- $\Sigma$ is the input alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting final states

**Usage:**

- Transition step - $(q, aw) \vdash (q', w)$ if $\delta(q,a) = q'$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0, w) \vdash^* (q, \varepsilon), q \in F \}$
Non-deterministic finite automaton NFA

Definition NFA is a quintuple $M=(Q, \Sigma, \delta, q_0, F)$, where

- $Q$ is the finite set of states of an automaton
- $\Sigma$ is the input alphabet
- $\delta : Q \times \Sigma \rightarrow P(Q)$ is the transition function (a set)
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting final states

Usage:

- Transition step - $(q, aw) \ | - (q', w)$ if $q' \in \delta(q,a)$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{w \mid (q_0, w) \ | - * (q, \varepsilon), q \in F\}$
Q = \{ S_0, S_1, S_2 \}
Σ = \{ a, b \}
δ:

<table>
<thead>
<tr>
<th>State</th>
<th>Char</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

\( q_0 \in S_0 \)
\( F = \{ S_2 \} \)

\( a \ a \ a \ a \ b \ b \)
(AA)*AT

NFA

AAAAAT
• (AA)*AT

NFA

AAAAAT
• \((AA)^*AT\)

\[
\begin{align*}
S_0 & \rightarrow a S_1 \\
S_1 & \rightarrow a S_0 \\
S_0 & \rightarrow a S_2 \\
S_2 & \rightarrow t S_3
\end{align*}
\]
NFA

- \((AA)^*AT\)

\[ S_0 \rightarrow a S_1 \]
\[ S_1 \rightarrow a S_0 \]
\[ S_0 \rightarrow a S_2 \]
\[ S_2 \rightarrow t S_3 \]
• \((AA)^*AT\)

\[
\begin{align*}
S_0 &\rightarrow a S_1 \\
S_1 &\rightarrow a S_0 \\
S_0 &\rightarrow a S_2 \\
S_2 &\rightarrow t S_3
\end{align*}
\]
• \((AA)^*AT\)

- \(S_0 \rightarrow a S_1\)
- \(S_1 \rightarrow a S_0\)
- \(S_0 \rightarrow a S_2\)
- \(S_2 \rightarrow t S_3\)
NFA – simultaneously in all reachable states

- \((AA)^*AT\)

\[
\begin{align*}
S_0 &\rightarrow a S_1 \\
S_1 &\rightarrow a S_0 \\
S_0 &\rightarrow a S_2 \\
S_2 &\rightarrow b S_3
\end{align*}
\]

<table>
<thead>
<tr>
<th>State</th>
<th>Char</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
Simulation of an NFA

**Input:** NFA $M_A=(Q, \Sigma, \delta, q_0, F)$, Text $S = s[1..n]$

**Output:** States after each character read $Q_0, Q_1, ... Q_n$

NB: $S \in L(M_A)$ only if $F \subseteq Q_n$.

Initially queue and sets $Q_i$ are empty

1. **for** $i = 0$ to $n$ do  
   // for each symbol of text
2. mark all $q \in Q$ unreached
3. if ($i == 0$)
   // Initialise start state
4. then
   $Q_0 = q_0$; queue = $q_0$; mark $q_0$ as reached
5. else
6. foreach $q \in Q_{i-1}$  // Main transitions on $s[i]$
7. foreach $p \in \delta(q, s[i])$  // All transitions on $s[i]$
8. if $p$ not yet reached
9. $Q_i = Q_i \cup p$
10. push(queue, $p$)
11. mark $p$ as reached
12. while queue $\neq \emptyset$
13. $q = \text{take(queue)}$  // Follow up on all $\varepsilon$ - transitions
14. foreach $p \in \delta(q, \varepsilon)$  // All $\varepsilon$ - transitions
15. if $p$ not yet reached
16. $Q_i = Q_i \cup p$
17. push(queue, $p$)
18. mark $p$ as reached
Regexp -> NFA / DFA

• Construction of an automaton from the regular expression
• Regular expressions are mathematical and human-readable descriptions of the language
• Automata represent computational mechanisms to evaluate the language
• One needs to be able to parse the regular expression and to construct an automaton for matching it.
• See **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). pp. 103.
• Automaadi konstruktsioon: [Regulaarset avaldisest mittedetermineeritud automaadi moodustamine](Meelis Roos) (kohalik)
• Tsitaat:
• Nii saadud lõplik automaat pole determineeritud, kuna me kasutame juba primitiivsetes automaatides mittedetermineeritud. Lõpliku automaadi võib hiljem muidugi eraldi determineerida.
Thompson construction

- Symbol $\varepsilon$:

- Terminal symbol $a$:
Union and Concatenation

- **s|t**

- **st**
Closure

• $s^*$
Example

- $a^* (ba | c)$
Fig. 5.5. Thompson automaton construction for the regular expression $(AA|AT)((AG|AAA)^*)$. 
Thompson construction:

• Produces up to 2m states, but it has interesting properties, such as ensuring a linear number of edges, constant indegree and outdegree, etc.
• **Theorem** Time complexity of the NFA simulation is $O(||M_A|| \cdot n)$ where $||M_A||$ is the total number of states and transitions of $M_A$, $||M_A|| \leq 6 |A|$.

• **Proof** - During one step all states are manipulated only once, since all states are marked reached. There is at most $n$ steps. The size of the automaton is at most $6 |A|$ where $|A|$ is the length of the regular expression.
Simulation of an NFA

Input: NFA $M_A= (Q, \Sigma, \delta, q_0, F)$, Text $S = s[1..n]$ 
Output: States after each character read $Q_0, Q_1, \ldots, Q_n$ 
NB: $S \in L(M_A)$ only if $F \subseteq Q_n$.

Initially queue and sets $Q_i$ are empty

1. for $i = 0$ to $n$ do // for each symbol of text 
2. mark all $q \in Q$ unreached 
3. if ( $i == 0$ ) 
4. then // Initialise start state 
5. $Q_0 = q_0; \text{ queue } = q_0; \text{ mark } q_0 \text{ as reached}$ 
6. else 
7. foreach $q \in Q_{i-1}$ // Main transitions on $s[i]$ 
8. foreach $p \in \delta(q, s[i])$ // All transitions on $s[i]$ 
9. if $p$ not yet reached 
10. $Q_i = Q_i \cup p$ 
11. push( queue, $p$ ) 
12. mark $p$ as reached 
13. while queue $\neq \emptyset$ // Follow up on all $\epsilon$ - transitions 
14. $q = \text{take( queue )}$ 
15. foreach $p \in \delta(q, \epsilon)$ // All $\epsilon$ - transitions 
16. if $p$ not yet reached 
17. $Q_i = Q_i \cup p$ 
18. push( queue, $p$ ) 
19. mark $p$ as reached
Glushkov construction

\[(A_1 \; T_2 \; I \; G_3 \; A_4)^* (A_5 \; G_6 \; I \; A_7 \; A_8 \; A_9)^*\]

Fig. 5.6. Marked Glushkov automaton built on the marked regular expression \((A_1 \; T_2[G_3; A_4])((A_5[G_6; A_7; A_8; A_9])^*\). The state 0 is initial. Double-circled states are final.

To obtain the Glushkov automaton of the original \(RE\), we simply erase the position indices in the marked automaton. At this step, the automaton usually becomes nondeterministic. The new automaton recognizes the language \(L(RE)\). The Glushkov automaton of our example \((AT[GA])((AG\; AAA)^*)\) is shown in Figure 5.7.

Fig. 5.7. Glushkov automaton built on the regular expression \((AT[GA])((AG\; AAA)^*)\). The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.
Fig. 5.6. Marked Glushkov automaton built on the marked regular expression \((A_1 T_2[G_3 A_4])(A_5 G_6 A_7 A_8 A_9)^*\). The state 0 is initial. Double-circled states are final.

To obtain the Glushkov automaton of the original RE, we simply erase the position indices in the marked automaton. At this step, the automaton usually becomes nondeterministic. The new automaton recognizes the language \(L(RE)\). The Glushkov automaton of our example \((AT|GA)((AG|AAA)^*)\) is shown in Figure 5.7.

Fig. 5.7. Glushkov automaton built on the regular expression \((AT|GA)((AG|AAA)^*)\). The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.
Fig. 5.19. Glushkov automaton built on the regular expression \(((GA\mid AAA)^*)\) \((TA\mid AG)\).
• No $\varepsilon$ links
• All incoming arcs have the same character label
• To reach a certain state always the same character from text had to be read.
• Construction: worst case is $O(m^3)$ since poor performance for star closures...
• But this has been speeded up a bit
NFA -> DFA

• Why?

• More straightforward (i.e. faster) to match/simulate
Determinization of a NFA into a DFA

- Maintain at each stage a set of states reachable from previous set on the given character. (Remove $\varepsilon$ transitions.)
- Represent every reachable combination of states of a NFA as a new state of DFA
- From each state there has to be only one transition on a given character.
- *Automata for Matching Patterns Handbook of Formal Languages* (kohalik)
Maintain at each stage a set of states reachable from previous set on the given character. (Remove $\varepsilon$ transitions.)

Represent every reachable combination of states of a NFA as a new state of DFA.

From each state there can be only one transition on a given character.
Regular expression matching

\textbf{BuildState}(S)

1. \textbf{If} \( S \cap F \neq \emptyset \) \textbf{Then} \( F_d \leftarrow F_d \cup \{S\} \)
2. \textbf{For} \( \sigma \in \Sigma \) \textbf{Do}
3. \( T \leftarrow \emptyset \)
4. \textbf{For} \( s \in S \) \textbf{Do}
5. \( \text{For} (s, \sigma, s') \in \Delta \) \textbf{Do} \( T \leftarrow T \cup E(s') \)
6. \textbf{End of for}
7. \( \delta(S, \sigma) \leftarrow T \)
8. \textbf{If} \( T \not\in Q_d \) \textbf{Then}
9. \( Q_d \leftarrow Q_d \cup \{T\} \)
10. \textbf{BuildState}(T)
11. \textbf{End of if}
12. \textbf{End of for}

\textbf{BuildDFA}(N = (Q, \Sigma, I, F, \Delta))

13. \textbf{EpsClosure}(N)
14. \( I_d \leftarrow E(I) \) /* initial DFA state */
15. \( F_d \leftarrow \emptyset \) /* final DFA states */
16. \( Q_d \leftarrow \{I_d\} \) /* all the DFA states */
17. \textbf{BuildState}(I_d)
18. \textbf{Return} \( (Q_d, \Sigma, I_d, F_d, \delta) \)

Fig. 5.11. Classical computation of the DFA from the NFA.
Fig. 5.5. Thompson automaton construction for the regular expression $(AA|AT)((AG|AAA)^*)$. 

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(2)</td>
<td>3,7,8,9,12,17</td>
<td></td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression \((AA|AT)((AG|AAA)*)\).

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(2) F</td>
<td>3,7,8,9,12,17</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression $(AA|AT)((AG|AAA)^*)$. 

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(2) F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression 
(AA|AT)(AG|AAA)*.

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(2) F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
</tr>
<tr>
<td>E(3)</td>
<td>10,13</td>
<td></td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AAA)*).

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
</tr>
<tr>
<td>E(2) F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
</tr>
<tr>
<td>E(3)</td>
<td>10,13</td>
<td>14</td>
<td>-</td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression \((AA|AT)((AG|AAA)^*)\).

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(0)</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E(1)</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
</tr>
<tr>
<td>E(2)F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
</tr>
<tr>
<td>E(3)</td>
<td>10,13</td>
<td>14</td>
<td>-</td>
</tr>
</tbody>
</table>
**DFA state** | **NFA States** | **A** | **T** | **G**
---|---|---|---|---
0 | 0,1,4 | 2 | - | -
1 | 2 | - | 3,7,8,9,12,17 | -
2 F | 3,7,8,9,12,17 | 10,13 | - | -
3 | 10,13 | 14 | - | 8,9,11, 12, 16,17
4 | 14 | 15,16,17,8,9,12 | - | |
5 | 8,9,12,15,16,17 | 10,13 | - | -
6 | 8,9,11,12,16,17 | 10,13 | - | -
7 | 10,13 | 14 | - | 11,16,17,8,9,12

Fig. 5.5. Thompson automaton construction for the regular expression (AA | AT)((AG | AAA)*)

---
Fig. 5.5. Thompson automaton construction for the regular expression \((AA|AT)((AG|AAA)*).\)

<table>
<thead>
<tr>
<th>DFA state</th>
<th>NFA States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-</td>
<td></td>
<td>3,7,8,9,12,17</td>
</tr>
<tr>
<td>2 F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>10,13</td>
<td>14</td>
<td>-</td>
<td>8,9,11, 12, 16,17</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8,9,12,15,16,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>8,9,11,12,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>8,9,12,15,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>6,7,8,9,12,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>6,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFA state</td>
<td>NFA States</td>
<td>A</td>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>-------</td>
<td>-------</td>
<td>------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
<td>-</td>
</tr>
<tr>
<td>2 F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>10,13</td>
<td>14</td>
<td>-</td>
<td>8,9,11,12,16,17</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8,9,12,15,16,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>8,9,11,12,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>8,9,12,15,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>6,7,8,9,12,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>6,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2 F</td>
<td>3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5 F</td>
<td>3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>6 F</td>
<td>3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>8 F</td>
<td>4</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Fig. 5.5. Thompson automaton construction for the regular expression \((AA|AT)(AG|AAA)^*\).

<table>
<thead>
<tr>
<th>DFA state</th>
<th>NFA States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
<td></td>
</tr>
<tr>
<td>2 F</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>10,13</td>
<td>14</td>
<td>-</td>
<td>8,9,11,12,16,17</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8,9,12,15,16,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>8,9,11,12,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>8,9,12,15,16,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>6,7,8,9,12,17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>6,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5.3 Classical approaches to regular expression searching

From Thompson’s NFA
5.3 Classical approaches to regular expression searching

From Thompson's NFA

From Glushkov's NFA

Fig. 5.12. The DFAs resulting from Thompson's and Glushkov's NFAs.
Minimization of automata

• DFA construction does not always produce the minimal automaton

• Smaller -> better(?)

• Must still represent equivalent languages!
Minimization of automata

- Language is simply a subset of all possible strings of $\Sigma^*$
- Regular language is the language that can be described using regular expressions and recognized by a finite automaton (no pushdown, multi-tape, or Turing machines)
- Two automatons that recognize exactly the same language are equivalent
- The automaton that has fewest possible states from the same equivalence class, is the minimal
- Automaton with more states is called redundant
- The automaton construction techniques do not create the minimal automatons
- It would be easier to understand nonredundant automatons
- Smaller automaton consumes less memory
- The manipulation is faster
Minimization

• A compiler course subject
• Minimization description (L4_RegExp/min-fa.html)

A: Merge all equivalent states until minimum achieved
B: Start from minimal possible (2-state) and split states until no conflicts
• Fact. Equivalent states go to equivalent states under all inputs.
• Recognizer for \((aa \mid b)^*ab(bb)^*\)
Step 1: Generate 2 equivalence classes: final and other states

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3:B</td>
</tr>
<tr>
<td>7</td>
<td>3:B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1:B</th>
<th>4:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A</td>
</tr>
<tr>
<td>3</td>
<td>3:B</td>
<td>3:B</td>
</tr>
<tr>
<td>4</td>
<td>1:B</td>
<td>4:B</td>
</tr>
<tr>
<td>5</td>
<td>1:B</td>
<td>4:B</td>
</tr>
<tr>
<td>6</td>
<td>3:B</td>
<td>7:A</td>
</tr>
</tbody>
</table>
Step 2: Create new class from 1 and 6 (conflict on b)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3:B</td>
<td>6:C</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>3:B</td>
<td>6:C</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1:C</td>
<td>4:B</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3:B</td>
<td>3:B</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>1:C</td>
<td>4:B</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1:C</td>
<td>4:B</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>3:C</td>
<td>7:A</td>
<td></td>
</tr>
</tbody>
</table>
Step 3: Create new class from 3

\[
\begin{array}{ccc}
1 & 2 & 3 \\
2 & 3:D & 6:C & A \\
7 & 3:D & 6:C \\
\end{array}
\]

---

\[
\begin{array}{ccc}
0 & 1:C & 4:B \\
4 & 1:C & 4:B & B \\
5 & 1:C & 4:B \\
\end{array}
\]

---

\[
\begin{array}{ccc}
1 & 2 & 3 \\
6 & 3:D & 7:A \\
\end{array}
\]

---

\[
\begin{array}{ccc}
3 & 3:D & 3:D \\
\end{array}
\]
Step 4: Create new class from 6

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3:D</td>
<td>6:E A</td>
</tr>
<tr>
<td>7</td>
<td>3:D</td>
<td>6:E</td>
</tr>
</tbody>
</table>

All states are consistent
Minimal automaton

Step 4: Create new class from 6

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3:D</td>
<td>6:E A</td>
</tr>
<tr>
<td>7</td>
<td>3:D</td>
<td>6:E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1:C</td>
<td>4:B A</td>
</tr>
<tr>
<td>4</td>
<td>1:C</td>
<td>4:B B</td>
</tr>
<tr>
<td>5</td>
<td>1:C</td>
<td>4:B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3:D</td>
<td>3:D D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3:D</td>
<td>7:A E</td>
</tr>
</tbody>
</table>

All states are consistent
• Recognizer for \((aa \mid b)^*ab(bb)^*\)
$(aa \mid b)^*ab(bb)^*$
(aa | b)*ab(bb)*
Construct a DFA from the regular expression

- Usually NFA is constructed, and then determinized
- McNaughton and Yamada proposed a method for direct construction of a DFA
\((a \mid b)^* \text{aba}\#\) to DFA
• Example: Let's analyze RE = (a U b)*aba
• Add end symbol #: (a U b)*aba#
• Make a parse tree
  – Leaves represent symbols of $\Sigma$ from RE
  – Internal nodes - operators
• Give a unique numbering of leaves
• Position nr is **active** if this can represent the next symbol
• DFA states and transitions are made from the tree:
• A state of DFA corresponds to a set of positions that are active after reading some prefix of the input
• Initial state is (1,2,3) (when nothing has been read yet)
• DFA contains transitions $q \rightarrow_a q'$, where $q'$ are position nrs that are activated when in positions of $q$ the character $a$ is read.
• Final states are those containing the position number of #
\((a \mid b)^* \text{aba# to DFA}\)
Construction of regular expressions from the automata

• It is possible to start from an automaton and then generate the regular expression that describes the language recognized by the automaton.

• **Example.**


• In the bottom of page there are links to "current version".
Moodusta reg. avaldis sellest automaadist
Moodusta reg. avalcis sellest automaadist

\[ a, b, \text{ab}, \text{ba}, \text{aa}, \text{bb} \]
Moodusta reg. avaldus seelegest automaadist
Moodusta reg. avalcis sellest automaadist

(aa | b) (ba)* (bb | a)
Moodusta reg. avalcis sellest automaadist

ab | (aa | b)(ba)* (bb | a)
Moodusta reg. avaldis sellest automaadist

\[(ab | (aa | b)(ba)^* (bb | a))^*\]
Filtering approaches for regular expression searches

• Identify a (sub)set of prefixes or factors that are necessarily present in the language represented by regexp.
• Use multi-pattern matching techniques for matching them all simultaneously
• In case of a match use the automaton to verify the occurrence
• Prefixes
• lmin - the shortest occurrence length (to avoid missing short occurrences)
• \(((\text{GA}\mid\text{AAA})\ast(\text{TA}\mid\text{AG}))\) the set of 2-long prefixes is \{ GA, AA, TA, AG \}
• \((\text{AT}\mid\text{GA})(\text{AG}\mid\text{AAA})((\text{AG}\mid\text{AAA})+)\) \(l_{\text{min}}=6\)
• \{ ATAGAG, ATAGAA, ATAAAA, GAAGAG, GAAGAA, GAAAAA \}
Fig. 5.24. Automaton to recognize all the reverse prefixes of the regular expression ((GA|AAA)*)(TA|AG).
• (AG|GA)ATA((TT)*)
• The string ATA is a necessary factor.
• Gnu grep uses such heuristics
• Can be developed to utilise a lot of knowledge about possible frequencies of occurrences, speed of multi-pattern matchers etc.
Summary

Regular expression -> Parse -> NFA -> DFA -> Occurrences

minimize
Back-references

• (regexp)\1+

• True or false?
  – “111111111111” =~ /^1(1+?)\1+$/
Detecting primality

• Is 7 prime?
• To know this, the function first generates “1111111” (from “1” * 7) and tries to see if that string does not match
  /^1?($)|(11+?)\1+$/.  
• If there is no match, then the number is prime.
vilo@muhu:~/Algorithmics$ cat regexp_primes_short.pl

($F,$T) = @ARGV ;

foreach $i ( $F..$T ) {
    $ii = "a"x$i ; # string with $i times a
    print "$i\n" unless $ii =~ /^(aa+?)\1+$/  ;
}

vilo@muhu:~/Algorithmics$ perl regexp_primes_short.pl 1990 2050
1993
1997
1999
2003
2011
2017
2027
2029
2039
vilo@muhu:~/Algorithmics$
Learning languages

Grammatical inference

• AGAGGAT +
• ATGAGAA +
• ATGATTA –
• AA –
• AAATGA –
• AGATAG +

Q: What is the language represented by the positive examples?

A1: List of positive examples

A2: Minimal automaton that recognises + examples, and none of the – examples?

Finding A2 in general a computationally hard problem
NFA/DFA

• Create an automaton for matching a word approximately

• Allow 0,1,…n errors
Approximate search: Problem statement

• Let $S=s_1s_2\ldots s_n \in \Sigma^*$ be a text and $P=p_1p_2\ldots p_m$ the pattern. Let $k$ be a pregiven constant.

• Main problems

• $k$ mismatches
  – Find from $S$ all substrings $X$, $|X|=|P|$, that differ from $P$ at max $k$ positions (Hamming distance)

• $k$ differences
  – Find from $S$ all substrings $X$, where $D(X,P) \leq k$ (Edit distance)

• best match
  – Find from $S$ such substrings $X$, that $D(X,P)$ is minimal

• Distance $D$ can be defined using one of the ways from previous chapters
Algorithm for approximate search, $k$ edit operations

Input: P, S, k
Output: Approximate occurrences of P in S (with edit distance $\leq k$)

for $j=0$ to $m$ do $h_{j,0} = j$ // Initialize first column

for $i=1$ to $n$ do
  $h_{0,i} = 0$
  for $j=1$ to $m$ do
    $h_{j,i} = \min(h_{i-1,j-1} + (\text{if } p_j == s_i \text{ then } 0 \text{ else } 1),$
    $h_{i-1,j} + 1, h_{i,j-1} + 1)$

if $h_{m,i} \leq k$ Report match at i

Trace back and report the minimizing path (from-to)
|   | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| 1 | S=| A | B | C | C | B | C | B | C | B | A | B | C | A | B | C | A | C | A | A | A |
| 5 |   | A | B | C | C | B | C | B | C | B | A | B | C | A | B | C | A | C | C | C | C |
| 6 |   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | A | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8 | B | 2 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 2 | 1 | 2 | 1 | 2 |
| 9 | C | 3 | 2 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 10 | A | 4 | 3 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 11 | B | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 2 | 3 | 2 | 3 | 2 | 2 | 1 | 0 | 1 | 1 | 1 | 2 | 2 |
| 12 | C | 6 | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 3 | 2 | 2 | 1 | 0 | 1 | 1 | 1 | 1 |

=MIN(S11+IF($A12=T$5,0,1),S12+1,T11+1)
Fig. 6.4. An NFA for approximate string matching of the pattern "annual" with two errors. The shaded states are those active after reading the text "anneal".

The original proposal of [Ukk85] was to make this automaton deterministic using the classical algorithm to convert an NFA into a DFA. This way, \(O(n)\) worst-case search time is obtained, which is optimal. The main problem then becomes the construction and storage requirements of the DFA. An upper bound to the number of states of the DFA is \(O(\min(3^m, m(2m|\Sigma|^k)))\) [Ukk85]. In practice, this automaton cannot be used for \(m > 20\), and
Fig. 6.19. The graph for the regular expression "(a|b)a*" on the text "baa". Bold arrows show an optimal path, of cost zero.

The idea of the shortest path can still be applied quite easily if the graph is acyclic, that is, if the regular expression does not contain the "*" or the "+" operator. On acyclic regular expressions we can find a topological order to evaluate the graph so as to find the shortest paths in overall time $O(mn)$. This requires Thompson's guarantee that there are $O(m)$ edges on
Regular expressions

Fig. 6.20. Glushkov’s NFAs for the regular expression "ab cd (d|ε)(e|f) de" searched with two insertions, deletions, or substitutions. To simplify the figure, the dashed lines represent deletions and substitutions (i.e., they move by $\Sigma \cup \{\epsilon\}$), while the vertical lines represent insertions (i.e., they move by $\Sigma$).
Fast matching

• Use bit-parallelism

• I.e. keep a bit vector expressing a set of reachable states

• Manipulate bit vectors by mask vectors based on the input character.

• agrep, shift-or ...
Automaton toolbox

- Roman Tehkov, Kristjan Vedel
  - [Link](http://biit.cs.ut.ee/~vilo/Photos/2008_12_17_Tekstialgoritmid_Posterid/IMG_3585.JPG)
  - [Link](http://wiki.cs.ut.ee/atiwiki/Main/TA2008AutomatonToolbox)
  - [Link](http://code.google.com/p/ta-asm/)

- Features: NFA construction from regular expressions (Thompson and Glushkov)
- Determinisation of NFA
- Minimization of DFA
- Constructing automaton allowing errors from input automaton
1. Abstract

The main goal stated for the work was to implement various regular expression related automata
construction algorithms and convert them. These include building a parse tree from a regular
expression, constructing NFA (Chompski and Grune) from a parse tree. Determination of NFA
minimization of DFA and finally conversion of a directly matching automaton to an approximately
matching one.

The programs are written in Java and use slightly different libraries for developing
applications. Visualization is done using JGraphX software. (http://jgraph.com/)

2. Regexp → Parse tree

3. Parse tree → NFA

Regular expression tree is then converted to Normalized Form Automaton (NFA) using either
Thompson or Chompski-Grune construction.

Both algorithms first construct a parse tree which is then converted to a NFA. The
algorithm for each operation

4. NFA → DFA

To implement better matching performance NFA can be transformed into a Deterministic
Finite Automaton (DFA) which has an memoryless transition to the next state for any given state
and input.

The minimum procedure is used that builds a state table. Then state table is converted to
the state graph that is necessary. The transition is defined by the following algorithm:

5. DFA minimization

The DFA constructed from NFA is a standard, with the exception that because some transition
states can be present, 2 states are required if the sets of states leading to them are equivalent
and the same 2 symbol sequence of input reads to equal states. A standard equivalent DFA
can be constructed from the initial one using the following algorithm:

6. Allowing errors

Over an input automaton for exact matching we can produce an automaton for some
error. Each symbol becomes 6 symbol, addition, subtraction, or error symbol. Depending

7. Visualization

All automata constructed using Java and JGraphX libraries. The program comes with
two visualization software. The program comes with

http://biit.cs.ut.ee/~vilo/Photos/2008_12_17_Tekstialgoritmid_Posterid/IMG_3585.JPG
3. Parse tree → NFA

Regular expression tree is then converted to Non-deterministic Finite Automaton (NFA) using either Thompson or Champarnaud-Glushkov construction.

Both algorithms first construct a finite state automaton for symbol nodes and then inductively merge the created automata for each operation:

1) symbol 2) union 3) concatenation 4) Kleene closure

Thompson

Glushkov

Thompson automaton

Glushkov automaton