Algorithmics (6EAP)

Regular Expressions and Automata

Jaak Vilo

2020 fall
Contents

• Regular expressions and regular languages
• Automata
  – Deterministic finite automata **DFA**
  – Nondeterministic finite automata **NFA**
• Regular expressions to NFA
• NFA to DFA
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 Informatika** (Jaan Penjam, TTÜ), Peatükk 5.

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

• Google – Query


• **GNU grep manual** (grep = Global Search for Regular Expression and Print) [http://www.hmug.org/man/1/grep.php](http://www.hmug.org/man/1/grep.php)

• **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/)
**Definition:** A *regular expression* RE is a string on the set of symbols $\Sigma \cup \{ \varepsilon, |, \cdot, \ast, (, ) \}$, 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

\[
((A \cdot T) | (G \cdot A)) \cdot ((A \cdot G) | ((A \cdot A) \cdot A))^*
\]

• we can simplify

\[
(AT | GA)((AG | AAA)^*)
\]

• 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

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

• 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 $L(RE) = \{\varepsilon\}$, the empty string
- if $RE$ is $\alpha \in \Sigma$, then $L(RE) = \{\alpha\}$, a single string of one character
- if $RE$ is of the form $(RE_1)$, then $L(RE) = L(RE_1)$
- if $RE$ is of the form $(RE_1 \cdot RE_2)$, then $L(RE) = L(RE_1) \cdot L(RE_2)$, where $w = w_1w_2$ is in $L(RE)$ if $w_1 \in L(RE_1)$ and $w_2 \in L(RE_2)$. (We call $\cdot$ the concatenation operator)
- if $RE$ is of the form $(RE_1 | RE_2)$, then $L(RE) = L(RE_1) \cup L(RE_2)$, the union of two languages. (We call $|$ the union operator)
- if $RE$ is of the form $(RE_1^*)$, then $L(RE) = L(RE)^* = \bigcup_{i \geq 0} L(RE_1)^i$, where $L^0 = \{\varepsilon\}$ and $L^i = L \cdot L^{i-1}$. (We call $\ast$ the star operator)
<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language $L(RE)$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>${\epsilon}$</td>
<td>Empty string</td>
</tr>
<tr>
<td>$\alpha \in \Sigma$</td>
<td>${\alpha}$</td>
<td>Single character</td>
</tr>
<tr>
<td>$(RE_1)$</td>
<td>$L(RE_1)$</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>$(RE_1 \cdot RE_2)$</td>
<td>$L(RE_1)$</td>
<td>Concatenation</td>
</tr>
<tr>
<td>$(RE_1</td>
<td>RE_2)$</td>
<td>$L(RE_1) \cup L(RE_2)$</td>
</tr>
<tr>
<td>$(RE_1^*)$</td>
<td>$L(RE_1^*) = \cup_{i \geq 0} L(RE_1)^i$</td>
<td>The star operator (Kleene star)</td>
</tr>
<tr>
<td>$(RE_1^+)$</td>
<td>$(RE_1) \cdot (RE_1^*)$</td>
<td>Kleene plus</td>
</tr>
</tbody>
</table>
• \( L( (AT|GA)((AG|AAA)^*) ) \) = \{ AT, GA, ATAG, GAAG, ATAAA, GAAAA, ATAGAG, ATAGAAA, ATAAAAA, ... \}

• \( \Sigma^* \) 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.
Regexp Matching

- 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 ) * )
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
            elseif ( 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)

Löppolekud:
2: INTEGER
5: REAL
8: Scientific
9: Identifier
12: Identifier List

Deterministic finite automaton DFA

**Definition** DFA 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
- $\delta : Q \times \Sigma \rightarrow Q$ 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 - $(q, aw) \rightarrow (q', w)$ if $q' = \delta(q,a), w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0, w) \rightarrow^* (q, \varepsilon), q \in F \}$
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 - $(q, aw) \rightarrow (q', w)$ if $q' \in \delta(q,a), a \in \Sigma \cup \{ \varepsilon \}, w \in \Sigma^*$
- Accepted language: $L(M) = \{ w | (q_0, w) \rightarrow^* (q, \varepsilon), q \in F \}$
DFA

Q = \{ S_0, S_1, S_2 \}
\Sigma = \{ a, b \}
\delta :

<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 \}

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

a a a a b b
(AA)*AT

AAAAAT
• (AA)^*AT

AAAAAT
• (AA)*AT

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

$S_0 \rightarrow a S_1$
$S_1 \rightarrow a S_0$
$S_0 \rightarrow a S_2$
$S_2 \rightarrow t S_3$
(AA)*AT

1. $S_0 \rightarrow a S_1$
2. $S_1 \rightarrow a S_0$
3. $S_0 \rightarrow a S_2$
4. $S_2 \rightarrow t S_3$

A AAA AT
NFA – simultaneously in all reachable states

• \((AA)^*AT\)

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

<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>
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: Regulaarest 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

- Primitive automata
- Composition
- No optimality, no compression, etc.
Thompson construction: 2 primitive automata

• Symbol $\varepsilon$ :

```
   i --\varepsilon/ -- f
```

• Terminal symbol $a$ :

```
   i --\a/ -- f
```
Union and Concatenation

- $s|t$

- $st$

\[
\begin{align*}
\text{N}(s) & \quad \varepsilon \\
\text{N}(t) & \quad \varepsilon \\
f & \quad \varepsilon
\end{align*}
\]
Closure

• $s^*$
Example

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

Regular expression matching
• Produces up to 2m states, but it has interesting properties, such as ensuring a linear number of edges, constant indegree and outdegree, etc.
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
5.                 $Q_0 = q_0$; queue = q0; mark q0 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 $\neq \emptyset$
14.         q = take( queue )  // Follow up on all $\epsilon$ - transitions
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

1 or more? 

Epsilons?
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.
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 \((A_1 \ T_2 \ G_3 \ A_4) ((A_5 \ G_6 \ I \ A_7 \ A_8 \ A_9)^*)\). 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 \((A \ T \ G \ A) \ ((A \ G \ A) \ A A)^*\) is shown in Figure 5.7.

Fig. 5.7. Glushkov automaton built on the regular expression \((A \ T \ G \ A) \ ((A \ G \ A) \ A A)^*\). The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.
Regular expression matching

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 \(((GA|AAA)*)\) \((TA|AG)\).
• No ε 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
Matching of RE-s

Regular expression → Parse → NFA → DFA → Occurrences
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)

• Navarro and Raffinot Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). pp. 106 -> pp 115
Maintain at each stage a set of states reachable from previous set on the given character. (Remove $\epsilon$ 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

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

BuildDFA($N = (Q, \Sigma, I, F, \Delta)$)
13. 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. BuildState($I_d$)
18. 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 
\((AA|AT)((AG|AAA)*)\).

| States | A  | T
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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 \((AA|AT)((AG|AAA)*)\).
Fig. 5.5. Thompson automaton construction for the regular expression \((AA|AT)((AG|AAA)*)\).
Fig. 5.5. Thompson automaton construction for the regular expression
\((AA | AT)(AM | AAA)*\).

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

Fig. 5.5. Thompson automaton construction for the regular expression 
(\text{AA}|\text{AT})\langle(\text{AG}|\text{AAA})\rangle.

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

---

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

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

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

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

--------------------------------

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

--------------------------------

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

--------------------------------

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3:D</td>
<td>3:D D</td>
</tr>
</tbody>
</table>
Step 4: Create new class from 6

\[
\begin{array}{cc}
a & b \\
2 & 3:D & 6:E & A \\
7 & 3:D & 6:E \\
\end{array}
\]

--------------------------------

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

--------------------------------

\[
\begin{array}{cc}
1 & 5:B & 2:A & C \\
\end{array}
\]

--------------------------------

\[
\begin{array}{cc}
3 & 3:D & 3:D & D \\
\end{array}
\]

--------------------------------

\[
\begin{array}{cc}
6 & 3:D & 7:A & E \\
\end{array}
\]

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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3:D</td>
<td>3:D D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3:D</td>
<td>7:A E</td>
</tr>
</tbody>
</table>

All states are consistent
(aa | 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 | b)* aba# to DFA
• Example: Lets analyze RE = (a ∪ b)*aba
• Add end symbol #: (a ∪ b)*aba#
• Make a parse tree
  – Leaves represent symbols of Σ 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 transuitions $q \rightarrow_a q'$, whwere $q'$ 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 | b)* aba# to DFA

<table>
<thead>
<tr>
<th>state</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 (start)</td>
<td>1234</td>
<td>123</td>
</tr>
<tr>
<td>1234</td>
<td>1234</td>
<td>1235</td>
</tr>
<tr>
<td>1235</td>
<td>12346</td>
<td>123</td>
</tr>
<tr>
<td>12346</td>
<td>1234</td>
<td>1235</td>
</tr>
</tbody>
</table>
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.


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

\[ ab \xrightarrow{aa \mid b} \xrightarrow{bb \mid a} ba \]
Moodusta reg. avaldis sellest automaadist

\[(aa \mid b)(ba)^* (bb \mid a)\]
Moodusta reg. avaldis sellest automaadist

\[ ab \mid (aa \mid b)(ba)^* (bb \mid a) \]
Moodusta reg. avaldis seljest 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
• \( l_{\text{min}} \) - the shortest occurrence length (to avoid missing short occurrences)
• \(((GA|AAA)^\ast(TA|AG))\) the set of 2-long prefixes is \{ GA, AA, TA, AG \}
• \((AT|GA)(AG|AAA)((AG|AAA)^\ast)\) \( l_{\text{min}}=6 \)
• \{ 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
How to find primes

$ perl -e 'foreach $i ( 1000..1050 ) { print "$i\n" if ("a"x$i) !~ /^(aa+)(\1+)$/ }'  
1009
1013
1019
1021
1031
1033
1039
1049
$
Learning languages

Grammatical inference

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

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