Advanced Algorithmics
& Text Algorithms
Regular expressions and automata

Jaak Vilo
2009 Spring

Contents

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

2.04.2009
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.
- Regulaarset 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
- 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/

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 \( \alpha \in \Sigma \),
  - \( (\text{RE}_1) \),
  - \( (\text{RE}_1 \cdot \text{RE}_2) \),
  - \( (\text{RE}_1 | \text{RE}_2) \), and
  - \( (\text{RE}_1 *) \),
  - where \( \text{RE}_1 \) and \( \text{RE}_2 \) 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

• \((AT \mid GA)((AG \mid AAA)^*)\) \(\Rightarrow\) \(AT\), \(ATAG\), \(GAAAA\), \(GAAGAAAAA\), ...

• 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 $\epsilon$, then $L(RE)=\{\epsilon\}$, 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 $(RE_1 \cup RE_2)$, then $L(RE)=L(RE_1) \cup L(RE_2)$, the union of two languages. (We call $\cup$ the union operator)
- if RE is of the form $(RE_1^*)$, then $L(RE)=L(RE_1)^* = \cup_{i \geq 0} L(RE_1)^i$, where $L^0 = \{ \epsilon \}$ and $L^i = L \cdot L^{i-1}$. (We call $^*$ 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>$L(RE_1) \cdot L(RE_2)$</td>
<td>Concatenation</td>
</tr>
<tr>
<td>$(RE_1 \cup RE_2)$</td>
<td>$L(RE_1) \cup L(RE_2)$</td>
<td>Union</td>
</tr>
<tr>
<td>$(RE_1^*)$</td>
<td>$L(RE_1)^*$</td>
<td>The star operator (Kleene star)</td>
</tr>
<tr>
<td>$(RE_1^+)$</td>
<td>$L(RE_1) \cdot (RE_1^*)$</td>
<td>Kleene plus</td>
</tr>
</tbody>
</table>
• \( L( (AT|GA)((AG|AAA)^*) ) = \{ AT, GA, ATAG, GAAG, ATAAA, GAAAA, ATAGAG, ATAGAAA, ATAAAAG, \ldots \} \)

• \( \Sigma^* \) 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.
  • \( \text{null} \) which denotes the empty set of strings.
• Operators used in regular expressions include:
  • \( \ast \) Concatenation: If \( R_1 \) and \( R_2 \) are regular expressions, then \( R_1R_2 \) (also written as \( R_1.R_2 \)) is also a regular expression.
    \( L(R_1R_2) = L(R_1) \text{ concatenated with } L(R_2) \).
  • \( \ast \) Union: If \( R_1 \) and \( R_2 \) are regular expressions, then \( R_1 \cup R_2 \) (also written as \( R_1 \cup R_2 \) or \( R_1 + R_2 \)) is also a regular expression.
    \( L(R_1 \cup R_2) = L(R_1) \cup L(R_2) \).
  • \( \ast \) Kleene closure: If \( R_1 \) is a regular expression, then \( R_1^\ast \) (the Kleene closure of \( R_1 \)) is also a regular expression.
    \( L(R_1^\ast) = \epsilon \cup L(R_1) \cup L(R_1R_1) \cup L(R_1R_1R_1) \cup \ldots \).  
• 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
Parse tree

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 ( (q == 0) || (q == 99) )
    switch (q)
    { case 0 : if (c ∈ {',','.'}) q = 1
        else 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'

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 $\delta(q, a) = q'$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0, w) \rightarrow^* (q, \epsilon), q \in F \}$

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 $q' \in \delta(q, a)$, $a \in \Sigma \cup \{ \epsilon \}$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0, w) \rightarrow^* (q, \epsilon), q \in F \}$
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>

\[ q_0 \in S_0 \]
\[ F = \{ S_2 \} \]

\[ S_0 \to a S_1 \]
\[ S_1 \to a S_1 \]
\[ S_1 \to b S_2 \]
\[ S_2 \to b S_2 \]

\[ a a a a b b \]
• (AA)*AT

```
S₀ → a S₁
S₁ → a S₀
S₀ → a S₂
S₂ → t S₃
```

A AAAAT
• (AA)*AT

S₀ → a S₁
S₁ → a S₀
S₀ → a S₂
S₂ → t S₃

• (AA)*AT

S₀ → a S₁
S₁ → a S₀
S₀ → a S₂
S₂ → t S₃
• (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 – 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 $Q_0, Q_1, \ldots Q_n$

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. $Q_0 = q_0$; queue = $q_0$; mark $q_0$ 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 $\varepsilon$ - transitions
14. $q = \text{take}(\text{queue})$  // All $\varepsilon$ - transitions
15. foreach $p \in \delta(q, \varepsilon)$  // All $\varepsilon$ - transitions
16. if $p$ not yet reached
17. $Q_i = Q_i \cup p$
18. push( queue, p )
19. mark p as reached

Regexp $\rightarrow$ 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: Regulaargest 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 ε :

```
  i   ε   f
  \---\---\---
```

• Terminal symbol a :

```
  i   a   f
     \---\---\---
```

Union and Concatenation

• $s \cup t$

• $st$

Closure

• $s^*$
Example

- $a^*(ba|c)$
• 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.
Glushkov construction

\((A_1 T_2 \mid G_3 A_4)(A_5 G_6 \mid A_7 A_8 A_9)^*\)
• 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

Fig. 5.19. Glushkov automaton built on the regular expression \(((GA|AAA)^*)\) (\(TA|AG\)).
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 \( \epsilon \) 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 $\varepsilon$ 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.
### Table 1

<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. 3.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AA)+).**

### Table 2

<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>
</tbody>
</table>

**Fig. 3.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AA)+).**
<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)*).**
### States of the Thompson Automaton

<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>

---

### States of the Thompson Automaton

<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>11,16,17,8,9,12</td>
</tr>
</tbody>
</table>
### DFA state vs NFA States

<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>

---

### Thompson Automaton Construction for the Regular Expression

\[(AA|AT)((AG|AAA)*\]

---

### Thompson Automaton Construction for the Regular Expression

\[(AA|AT)((AG|AAA)*\]
### DFA state | NFA States | A | T | G
---|---|---|---|---
0 | 0,1,4 | 2 | - | 5
1 | 2 | - | 3,7,8,9,12,17 |
2 | 3,7,8,9,12,17 | 10,13 | - | - |
3 | 10,13 | 14 | - | 8,9,11,12,16,17 |
4 | 14 | 8,9,12,15,16,17 | - | - |
5 | 8,9,11,12,16,17 | 10,13 | - | - |
6 | 8,9,12,15,16,17 | 10,13 | - | - |
7 | 5 | 6,7,8,9,12,17 | - | - |
8 | 6,7,8,9,12,17 | 10,13 | - | - |

### DFA state | NFA States | A | T | G
---|---|---|---|---
0 | 1 | - | 7 |
1 | - | 2 |
2 | 3 | - | - |
3 | 4 | - | 6 |
4 | 5 | - | - |
5 | 3 | - | - |
6 | 3 | - | - |
7 | 8 | - | - |
8 | 4 | - | - |

**Fig. 1.5. Thompson’s construction for the regular expression.**

### DFA state | NFA States | A | T | G
---|---|---|---|---
0 | 0,1,4 | 2 | - | 5 |
1 | 2 | - | 3,7,8,9,12,17 |
2 | 3,7,8,9,12,17 | 10,13 | - | - |
3 | 10,13 | 14 | - | 8,9,11,12,16,17 |
4 | 14 | 8,9,12,15,16,17 | - | - |
5 | 8,9,11,12,16,17 | 10,13 | - | - |
6 | 8,9,12,15,16,17 | 10,13 | - | - |
7 | 5 | 6,7,8,9,12,17 | - | - |
8 | 6,7,8,9,12,17 | 10,13 | - | - |
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 automata 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 automata
- It would be easier to understand nonredundant automata
- 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>
<th>2</th>
<th>3:B</th>
<th>6:B A</th>
<th>7</th>
<th>3:B</th>
<th>6:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1:B</td>
<td>4:B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3:B</td>
<td>3:B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3:B</td>
<td>3:B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4:B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4:B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3:B</td>
<td>7:A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Step 2: Create new class from 1 and 6 (conflict on b)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>6:C</th>
<th>A</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>A</td>
</tr>
</tbody>
</table>

Step 3: Create new class from 3

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

---

Step 2:

- 2: 3:B 6:C A
- 7: 3:B 6:C

---

Step 3:

- 2: 3:B 6:C A
- 7: 3:D 6:C

---

Diagram for Step 2 and Step 3.
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</td>
</tr>
<tr>
<td>7</td>
<td>3:D</td>
<td>6:E</td>
</tr>
<tr>
<td>0</td>
<td>1:C</td>
<td>4: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>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A</td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3:D</td>
<td>3:D</td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3:D</td>
<td>7:A</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</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</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>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5:B</td>
<td>2:A</td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3:D</td>
<td>3:D</td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3:D</td>
<td>7:A</td>
</tr>
</tbody>
</table>

All states are consistent
(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
• Example: Let's analyze $RE = (a \cup b)^*aba$
• Add end symbol $\# : (a \cup b)^*aba\#$
• Make a parse tree
  — Leaves represent symbols of $I$ 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 $q \rightarrow q'$, 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 $\#$
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".
Moodale reg. validis sedes automoedistis

\[ ab \mid (aa \mid b)(ba)^* (bb \mid a)^* \]
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)
- $((GA|AAA)*(TA|AG))$ the set of 2-long prefixes is \{ GA, AA, TA, AG \}
- $(AT|GA)(AG|AAA)((AG|AAA)+)$ $l_{\text{min}}=6$
- \{ ATAGAG, ATAGAA, ATAAAA, GAAGAG, GAAGAA, GAAAAA \}
• (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

Occurrences

DFA

minimize

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
Graph algorithms?

• Shortest path from start to end?

• Minimal cost path? (what would be the weights?)

NFA/DFA

• Create an automaton for matching a word approximately

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

- Let $S=s_1s_2...s_n \in \Sigma^*$ be a text and $P=p_1p_2...p_m$ 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 $\leq k$)

```plaintext
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} + (if $p_j=s_i$ then 0 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)
```
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 NFA’s for the regular expression "abcd(d|e)(a|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

• Features: NFA construction from regular expressions (Thompson and Glushkov)
• Determinisation of NFA
• Minimization of DFA
• Constructing automaton allowing errors from input automaton
3. Parse tree → NFA

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

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

Thompson

Glushkov

Thompson automaton

Glushkov automaton