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
• ...
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 $a \in \Sigma$, then $L(RE) = \{a\}$, a single string of one character.
- If $RE$ is of the form $(RE_1) \wedge (RE_2)$, then $L(RE_1) \wedge L(RE_2)$, where $\wedge$ is the concatenation operator.
- If $RE$ is of the form $(RE_1) \vee (RE_2)$, then $L(RE_1) \vee L(RE_2)$, the union of two languages. (We call this the union operator).
- If $RE$ is of the form $(RE_1)*$, then $L(RE_1)*$, $\cdot$ star operator (Kleene star).
- If $RE$ is of the form $(RE_1)$, then $L(RE_1)$, single character.
- Parenthesis

### Table

<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>$a \in \Sigma$</td>
<td>${a}$</td>
<td>Single character</td>
</tr>
<tr>
<td>$(RE_1)$</td>
<td>$L(RE_1)$</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>$(RE_1) \wedge (RE_2)$</td>
<td>$L(RE_1) \wedge L(RE_2)$</td>
<td>Concatenation</td>
</tr>
<tr>
<td>$(RE_1) \vee (RE_2)$</td>
<td>$L(RE_1) \vee L(RE_2)$</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>$L(RE_1)$</td>
<td></td>
</tr>
</tbody>
</table>

### 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.
- Operators 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.
  - the empty string containing no characters.
  - an operator which denotes the empty set of strings.
- Operators used in regular expressions include:
  - $\wedge$ (concatenation) if $R_1$ and $R_2$ are regular expressions, then $R_1 \wedge R_2$ (also written as $R_1R_2$) is also a regular expression.
  - $\vee$ (union) if $R_1$ and $R_2$ are regular expressions, then $R_1 \vee R_2$ (also written as $R_1 \cup R_2$) is also a regular expression.
  - $(R_1)*$ (Kleene closure) if $R_1$ is a regular expression, then $(R_1)*$ (also written as $R_1^*$) is also a regular expression.
  - $\epsilon$ (epsilon) if $R_1$ and $R_2$ are regular expressions, then $R_1 \epsilon R_2$ (also written as $R_1 \cup R_2$ or $R_1 + R_2$) is also a regular expression.
- Closure has the highest precedence, followed by concatenation, followed by union.

### Matching of RE-s

- 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.

**Example:**

- $L\{(\text{AT} \mid \text{GA})((\text{AG} \mid \text{AAA})^*)\} = \{\text{AT, GA, ATAG, GAAG, ATAGAA, ATAAAAG, ...}\}$
- $\Sigma^*$ 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.
**Parse tree**

- **Q:** what is the language?

**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: $L(M) = \{ w | (q_0, w) \rightarrow^* (q, \epsilon), q \in F \}$

**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: $L(M) = \{ w | (q_0, w) \rightarrow^* (q, \epsilon), q \in F \}$
DFA

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

\[ Q = \{ S_0, S_1, S_2 \} \]
\[ \Sigma = \{ a, b \} \]

\[ \delta: \text{State} \times \text{Char} \rightarrow \text{Next State} \]
\[ q_0 \rightarrow S_0 \rightarrow a S_1 \rightarrow a S_1 \rightarrow a S_1 \rightarrow a S_1 \rightarrow a S_0 \rightarrow b S_2 \rightarrow b S_2 \rightarrow S_2 \rightarrow b S_2 \rightarrow S_2 \rightarrow b S_2 \rightarrow \]

NFA

\[ (AA)^{\ast}AT \]

\[ A \ A \ A \ A \]
\[ A \ A \ A \ A \]
\[ A \ A \ A \ A \]

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

\[ a \ a \ a \ b \ b \]

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

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]

\[ A \ A \ A \ A \]
Simulation of an NFA

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

Output: States after each character read \( Q_0, Q_1, ... Q_n \)

1. \( S \notin L(M) \) only if \( F \notin Q_n \).

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 \( i == 0 \) then   // Initialise start state
4.     \( Q_0 = q_0; \) 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 ≠ \( \emptyset \) // Follow up on all \( \varepsilon \) - transitions
13.        q = take( queue ) // 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
19. Thompson construction

- Symbol \( \varepsilon \):

- Terminal symbol \( a \):


Automata construction: Regular expression automata construction (Meelis Roos) (in Estonian)

```
• Automata konstruktsioon: Regulaarse avaldisest mitmedetermīnētu automaadi mondustrājānu (Meelis Roos) (in balto)
• Hitaat:
  • Ni jaadud lõplik automaat pole determinētu, kuna mes kasutame juba primitīvēcē automaļā mitmedetermīnētu. Lõpliku automaadi vēl hiljem mudugi eraldīt determinētu.
```
Union and Concatenation

- \( s | t \)

- \( st \)

Closure

- \( s^* \)

Example

- \( a^*(ba|c) \)

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 MA=( Q, Σ, δ, q0, F ), Text S = s[1..n]
Output: States after each character read Q0, Q1, ... Qn.
NB: S ∈ L(MA) only if F ∈ Qn.
Initially queue and sets Qi are empty
1. for i = 0 to n do // for each symbol of text
2. mark all q ∈ Q unreached
3. if ( i == 0 )
4. then // Initialise start state
5. Q0 = q0; queue = q0; mark q0 as reached
6. else
7. for each q ∈ Qi-1 // Main transitions on s[i]
8. for each p ∈ δ(q, s[i]) // All transitions on s[i]
9. if p not yet reached
10. Qi = Qi ∪ p
11. push( queue, p )
12. mark p as reached
13. while queue ≠ ∅
14. q = take( queue ) // Follow up on all ε - transitions
15. for each p ∈ δ(q, ε) // All ε - transitions
16. if p not yet reached
17. Qi = Qi ∪ p
18. push( queue, p )
19. mark p as reached

Glushkov construction
• 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(m³) 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 ε 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 (Kuželka)

Fig. 1.1: Classical computation of the DFA from the NFA.

Maintain at each stage a set of states reachable from previous set on the given character. (Remove ε 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>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
</tr>
<tr>
<td>E2</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
</tr>
<tr>
<td>E3</td>
<td>10,13</td>
<td>14</td>
<td>11,16,17,8,12</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>0,1,4</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>-</td>
<td>3,7,8,9,12,17</td>
</tr>
<tr>
<td>E2</td>
<td>3,7,8,9,12,17</td>
<td>10,13</td>
<td>-</td>
</tr>
<tr>
<td>E3</td>
<td>10,13</td>
<td>14</td>
<td>11,16,17,8,12</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>States</th>
<th>A</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</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>8,9,11,12,16,17</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>-</td>
<td>8,9,11,12,16,17</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>8,9,11,12,16,17</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>
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 Σ*
- 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 automations 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 automations
- It would be easier to understand nonredundant automations
- 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 | 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>0</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>1</td>
<td>5B</td>
<td>2A</td>
</tr>
<tr>
<td>2</td>
<td>3D</td>
<td>3D</td>
</tr>
<tr>
<td>3</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>4</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>5</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>6</td>
<td>3B</td>
<td>7A</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>0</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>1</td>
<td>5B</td>
<td>2A</td>
</tr>
<tr>
<td>2</td>
<td>3D</td>
<td>6C</td>
</tr>
<tr>
<td>3</td>
<td>1B</td>
<td>3B</td>
</tr>
<tr>
<td>4</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>5</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>6</td>
<td>3C</td>
<td>7A</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>0</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>1</td>
<td>5B</td>
<td>2A</td>
</tr>
<tr>
<td>2</td>
<td>3D</td>
<td>6C</td>
</tr>
<tr>
<td>3</td>
<td>3D</td>
<td>3D</td>
</tr>
<tr>
<td>4</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>5</td>
<td>1C</td>
<td>4B</td>
</tr>
<tr>
<td>6</td>
<td>3D</td>
<td>7A</td>
</tr>
</tbody>
</table>

Step 4: Create new class from 6

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

Minimal automaton

All states are consistent

Recognizer for \((aa \mid b)^*ab(bb)^*\)
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 U b)*aba
- Add end symbol #: (a U 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 transitions q → 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 #
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.
- The program JFLAP for transforming FSA to regular expressions can be downloaded from http://www.jflap.org/ or http://www.cs.duke.edu/~rodger/tools/jflap/indexold.html
- In the bottom of page there are links to "current version". 

![Diagram of automata with transitions labeled by symbols such as 'b', 'a', 'ab', and 'bb'. The current version of JFLAP can be downloaded from the specified URLs.](image)
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
- \((GA|AAA)*TA(AG)\) 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

Back-references

• (regexp)\1+

• True or false?
  – "111111111111" =~ /^{11+}\1+$/

Detecting primality

• Is 7 prime?
  • To know this, the function first generates "11111111111" (from "1" * 7) and tries to see if that string **does not match** /
    /^1?\1|^\1+$/.
  • If there is no match, then the number is prime.

Learning languages

Grammatical inference

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

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...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 ≤ k)

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

for i=1 to n do
    h_{i,0} = 0

for j=1 to m do
    h_{j,i} = \min( h_{i-1,j-1} + (\text{if } p_i == s_j \text{ then } 0 \text{ else } 1),
                    h_{i-1,j} + 1, h_{i,j-1} + 1 )

if h_{m,n} ≤ k Report match at i
Trace back and report the minimizing path (from-to)

Regular expressions

- 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...

Fast matching
Automaton toolbox

- Roman Tehkov, Kristjan Vedel
  - http://code.google.com/p/ta-as/

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