Lectures Outline

1. Pattern Matching
   - Exact pattern matching
     - Searching for finite set of patterns
     - Searching for (Regular) Infinite Set of Patterns in Text
   - Approximate pattern matching
1. How to describe infinite set of pattern i.e. string?
   Regular Expressions

2. What shall we use to perform matching?
   Finite Automata

### Regular Expressions and Languages

<table>
<thead>
<tr>
<th>Regular expression $R$</th>
<th>Value of expression $h(R)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic expressions</td>
<td></td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$a, a \in \Sigma$</td>
<td>${a}$</td>
</tr>
<tr>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>$U \cdot V$</td>
<td>${uv</td>
</tr>
<tr>
<td>$U + V$</td>
<td>$h(U) \cup h(V)$</td>
</tr>
<tr>
<td>$V^k$</td>
<td>$V \cdot V \cdot \ldots \cdot V$</td>
</tr>
<tr>
<td>$V^+$</td>
<td>$V^1 + V^2 + V^3 + \ldots$</td>
</tr>
<tr>
<td>$V^*$</td>
<td>$V^0 + V^1 + V^2 + \ldots$</td>
</tr>
</tbody>
</table>

### Deterministic Finite Automaton

**Definition**

Deterministic Finite Automaton (DFA) is a quintuple $A = (Q, \Sigma, q_0, \delta, F)$, where

- $Q$ is a finite set of states
- $\Sigma$ is an alphabet
- $q_0 \in Q$ is an initial state
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function
- $F \subseteq Q$ is a set of final states
Deterministic Finite Automaton (cont.)

Configuration of Finite Automaton

\[(q, w) \in Q \times \Sigma^*\]

Transition of Finite Automaton is a relation

\[\rightarrow : (Q \times \Sigma^*) \times (Q \times \Sigma^*)\]

such as

\[(q, aw) \rightarrow (q', w) \iff \delta(q, a) = q'\]

Automaton accepts word \(w\) if

\[(q_0, w) \rightarrow^* (q, \varepsilon), q \in F\]

Nondeterministic Finite Automaton

Definition

Nondeterministic Finite Automaton (NFA) is a quintuple \(A = (Q, \Sigma, q_0, \delta, F)\), where

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

Alternatively NFA can be defined as \(A = (Q, \Sigma, S, \delta, F)\), where \(S \subseteq Q\) is a set of initial states.

For each NFA, there is a DFA such that it recognizes the same formal language.

NFA \(\rightarrow\) DFA Conversion

The DFA can be constructed using the powerset construction. NFA \(A = (Q, \Sigma, S, \delta, F)\) \(\rightarrow\) DFA \(A' = (Q', \Sigma, q'_0, \delta', F')\)

- \(Q' \subseteq P(Q)\)
- \(\Sigma' = \Sigma\)
- \(q'_0 = S\)
- \(\delta'(q', x) = \cup \delta(q, x)\) for all \(q \in q'\)
- \(F' = \{q' \in Q' | q' \cap F \neq \emptyset\}\)
Derivation of Regular Expression

For given regular expression $R$, derivation is defined as

$$h\left(\frac{dR}{dx}\right) = \{y \mid xy \in h(R)\}$$

Example

For $R = a + shell + stop + plot$ and its value $h(R) = \{a, shell, stop, plot\}$ derivations are

$$h\left(\frac{dR}{da}\right) = \{\varepsilon\}$$

$$h\left(\frac{dR}{ds}\right) = \{hell, top\}$$

$$h\left(\frac{dR}{dt}\right) = \emptyset$$

Derivation of Regular Expression – properties

\[
\begin{align*}
\frac{d\emptyset}{da} &= \emptyset, \forall a \in \Sigma \\
\frac{d\varepsilon}{da} &= \emptyset, \forall a \in \Sigma \\
\frac{d(U + V)}{da} &= \frac{dU}{da} + \frac{dV}{da} \\
\frac{d(U \cdot V)}{da} &= \frac{dU}{da} \cdot V + \frac{dV}{da} \cdot \varepsilon \notin U \\
\frac{d\varepsilon^*}{da} &= \frac{dV}{da} \cdot V^* \\
\frac{d\varepsilon^*}{da} &= \frac{d}{da} \left(\frac{d}{da} \left(\frac{d}{da} \left(\frac{d}{da} \left(\ldots \frac{d}{da} (\frac{dV}{da}) \right) \right) \right) \right), \text{ for } X = a_1a_2 \ldots a_n
\end{align*}
\]
Construction of DFA Derivations of RE

- Derivation of regular expressions allows directly and algorithmically build DFA for any regular expression.
- Let \( V \) is given regular expression in alphabet \( \Sigma \).
- Each state of DFA defines a set of words, that move the DFA from this state to any of final states.
  
  So, every state can be associated with regular expression, defining this set of words

\[
q_0 = V \\
\delta(q, x) = \frac{dq}{dx} \\
F = \{q \in Q | \epsilon \in h(q)\}
\]

Construction of DFA Derivations of RE – example

Lest’s have \( V = (0 + 1)^* \cdot 01 \) over alphabet \( \Sigma\{0, 1\} \).

Then \( q_0 = (0 + 1)^* \cdot 01 \)

Example of derivations:

\[
\frac{d((0 + 1)^* \cdot 01)}{d0} = \frac{d((0 + 1)^*) \cdot 01 + d01}{d0}
\]

\[
= \frac{d(0 + 1)}{d0} \cdot (0 + 1)^* \cdot 01 + \emptyset
\]

\[
= \left(\frac{d0}{d0} + \frac{d1}{d1}\right) \cdot (0 + 1)^* \cdot 01
\]

\[
= (\emptyset + \epsilon) \cdot (0 + 1)^* \cdot 01
\]

\[
= (0 + 1)^* \cdot 01
\]

Construction of DFA Derivations of RE – example (cont.)

\[
\frac{d((0 + 1)^* \cdot 01)}{d1} = \frac{d((0 + 1)^*) \cdot 01 + d01}{d1}
\]

\[
= \frac{d(0 + 1)}{d1} \cdot (0 + 1)^* \cdot 01 + \emptyset
\]

\[
= \left(\frac{d0}{d1} + \frac{d1}{d1}\right) \cdot (0 + 1)^* \cdot 01
\]

\[
= \emptyset \cdot (0 + 1)^* \cdot 01
\]

\[
= (0 + 1)^* \cdot 01
\]
Pattern Matching

Approximate pattern matching

String metric (string distance function) is a metric that measures distance between two text strings for approximate string matching.

- String metric can be considered as “inverse similarity” – how two strings are dissimilar.
- There are two classic metrics
  1. Hamming distance
  2. Levenshtein distance
- Yes, string dissimilarity, distance can be measured. Both distances are metrics from mathematical point of view – non-negativity, identity, symmetry, and triangle inequality.

---

**Hamming distance**

**Definition**

Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different.

In other words, it measures the minimum number of substitutions required to change one string into the other.

**Example**

Hamming distance of “karolin” and “kathrin” is 3.

<table>
<thead>
<tr>
<th>k</th>
<th>a</th>
<th>r</th>
<th>o</th>
<th>l</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>a</td>
<td>t</td>
<td>h</td>
<td>r</td>
<td>i</td>
<td>n</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Levenshtein distance**

**Definition**

Levenshtein distance (1965) between two strings is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one string into the other.
**Example**

Levenshtein distance between “kitten” and “sitting” is 3:

1. kitten → sitten (substitution of “s” for “k”)
2. sitten → sittin (substitution of “i” for “e”)
3. sittin → sitting (insertion of “g” at the end).

There is no way to do it with fewer than three edits.

**Upper and lower bounds**

The Levenshtein distance has several simple upper and lower bounds:

- It is at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- It is zero if and only if the strings are equal.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance.
- The Levenshtein distance between two strings is no greater than the sum of their Levenshtein distances from a third string (triangle inequality).

\[d(i,j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min(d(i-1,j) + 1, d(i,j-1) + 1, d(i-1,j-1) + c(i,j)) \end{cases}\]

where

\[c(i,j) = \begin{cases} 0, & \text{if } a_i = b_j \\ 1, & \text{otherwise} \end{cases}\]

First element in the minimum corresponds to deletion (from \(a\) to \(b\)), the second to insertion and the third to match or mismatch.
/* return minimum of delete char from s, delete char from t, and delete char from both */
return minimum
{
LevenshteinDistance(s, len_s-1, t, len_t) + 1,
LevenshteinDistance(s, len_s, t, len_t-1) + 1,
LevenshteinDistance(s, len_s-1, t, len_t-1) + cost
};
Approximate pattern matching using finite automata (cont.)

NFA for the approximate string matching using the Levenshtein distance \((m = 4, k = 3)\)