VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Methods of Analysis of Textual Data (MATD)

Jiří Dvorský

March 7, 2023

Department of Computer Science
VSB – TU Ostrava

Lectures Outline

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Pattern Matching

Jiří Dvorský

Department of Computer Science
VSB – TU Ostrava

Pattern Matching

Exact Pattern Matching

Pattern Matching Algorithms – common declarations

```
const size_t PatternNotFound = -1;

const int AlphabetSize = 256;
```


Brute Force Algorithm – source code

```
size_t BruteForce(const string& Pattern, const string& Text, const size_t StartPosition)
{
  for (size_t i = StartPosition; i < Text.length() - Pattern.length() + 1; i++)
  {
    size_t j = 0;
    while (j < Pattern.length())
    {
      if (Text[i + j] != Pattern[j])
        break;
      j += 1;
    }
    if (j == Pattern.length())
      return i;
```


Brute Force Algorithm – source code (cont.)

```
  }
  return PatternNotFound;
}
```


Brute Force Algorithm – example

First attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
1 2 3 4
G C A G A G A G
```

Shift by 1

Second attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
  1
  G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Third attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
    1
    G C A G A G A G
```

Shift by 1

Fourth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
      1
      G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Fifth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
        1
        G C A G A G A G
```

Shift by 1

Sixth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
          1 2 3 4 5 6 7 8
          G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Seventh attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
            1
            G C A G A G A G
```

Shift by 1

Eighth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
              1
              G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Nineth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                1 2
                G C A G A G A G
```

Shift by 1

Tenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                  1
                  G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Eleventh attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                    1 2
                    G C A G A G A G
```

Shift by 1

Twelfth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                      1
                      G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Thirteenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                        1 2
                        G C A G A G A G
```

Shift by 1

Fourteenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                          1
                          G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Fifteenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                            1
                            G C A G A G A G
```

Shift by 1

Sixteenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                              1
                              G C A G A G A G
```

Shift by 1

Brute Force Algorithm – example (cont.)

Seventeenth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                                1
                                G C A G A G A G
```

Shift by 1

The algorithm performs 30 character comparisons.

Morris-Pratt Algorithm – preprocessing

```
void MorrisPrattPreprocessing(const string& Pattern,
    vector<int>& Next)
{
  int i = 0;
  int j = Next[0] = -1;
  while (i < Pattern.length())
  {
    while (j > -1 && Pattern[i] != Pattern[j])
    {
      j = Next[j];
    }
    Next[++i] = ++j;
  }
}
```


Morris-Pratt Algorithm – source code

```
size_t MorrisPratt(const string& Pattern, const string&
    Text, const size_t StartPosition)
{
  vector<int> Next(Pattern.length() + 1, 0);
  MorrisPrattPreprocessing(Pattern, Next);
  int i = 0;
  size_t j = StartPosition;
  while (j < Text.length())
  {
    while (i > -1 && Pattern[i] != Text[j])
    {
      i = Next[i];
    }
    i += 1;
    j += 1;
```


Morris-Pratt Algorithm – source code (cont.)

```
    if (i >= Pattern.length())
    {
      return j - i;
    }
  }
  return PatternNotFound;
}
```


Morris-Pratt Algorithm

First attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
1 2 3 4
G C A G A G A G
```

Shift by $i - Next[i] = 3 - 0 = 3$

Second attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
      1
      G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Morris-Pratt Algorithm (cont.)

Third attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
        1
        G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Fourth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
          1 2 3 4 5 6 7 8
          G C A G A G A G
```

Shift by $i - Next[i] = 8 - 1 = 7$

Morris-Pratt Algorithm (cont.)

Fifth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                        1 2
                        G C A G A G A G
```

Shift by $i - Next[i] = 1 - 0 = 1$

Sixth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                          1
                          G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Morris-Pratt Algorithm (cont.)

Seventh attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                            1
                            G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Eighth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                              1
                              G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Morris-Pratt Algorithm (cont.)

Nineth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                                1
                                G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

The algorithm performs 19 character comparisons.

Knuth-Morris-Pratt Algorithm

First attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
1 2 3 4
G C A G A G A G
```

Shift by $i - Next[i] = 3 - (-1) = 4$

Second attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
        1
        G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Knuth-Morris-Pratt Algorithm (cont.)

Third attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
          1 2 3 4 5 6 7 8
          G C A G A G A G
```

Shift by $i - Next[i] = 8 - 1 = 7$

Fourth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                        1 2
                        G C A G A G A G
```

Shift by $i - Next[i] = 1 - 0 = 1$

Knuth-Morris-Pratt Algorithm (cont.)

Fifth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                          1
                          G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Sixth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                            1
                            G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Knuth-Morris-Pratt Algorithm (cont.)

Seventh attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                              1
                              G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

Eighth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
                                1
                                G C A G A G A G
```

Shift by $i - Next[i] = 0 - (-1) = 1$

The algorithm performs 18 character comparisons.

Knuth-Morris-Pratt Algorithm – preprocessing

```
void KnuthMorrisPrattPreprocessing(const string&
    Pattern, vector<int>& Next)
{
  int i = 0;
  int j = Next[0] = -1;
  while (i < Pattern.length())
  {
    while (j > -1 && Pattern[i] != Pattern[j])
    {
      j = Next[j];
    }
    i += 1;
    j += 1;
    Next[i] = Pattern[i] == Pattern[j] ? Next[j] : j;
  }
```


Knuth-Morris-Pratt Algorithm – preprocessing (cont.)

```
}
```


Knuth-Morris-Pratt Algorithm – source code

```
size_t KnuthMorrisPratt(const string& Pattern, const
    string& Text, const size_t StartPosition)
{
  vector<int> Next(Pattern.length() + 1, 0);
  KnuthMorrisPrattPreprocessing(Pattern, Next);
  int i = 0;
  size_t j = StartPosition;
  while (j < Text.length())
  {
    while (i > -1 && Pattern[i] != Text[j])
    {
      i = Next[i];
    }
    i += 1;
    j += 1;
```


Knuth-Morris-Pratt Algorithm – source code (cont.)

```
    if (i >= Pattern.length())
    {
      return j - i;
    }
  }
  return PatternNotFound;
}
```

Searching for (Regular) Infinite Set of Patterns in Text

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

Regular expression R	Value of expression $h(R)$
Atomic expressions	
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
$a,\ a \in \Sigma$	$\{a\}$
Operations	
$U \cdot V$	$\{uv \mid u \in h(U) \wedge v \in h(V)\}$
$U + V$	$h(U) \cup h(V)$
$V^k = \underbrace{V \cdot V \cdot \ldots \cdot V}_{k\ times}$	
$V^+ = V^1 + V^2 + V^3 + \ldots$	
$V^* = V^0 + V^1 + V^2 + \ldots$	

Regular Expression Features

$$
\begin{aligned}
U + (V + W) &= (U + V) + W \\
U \cdot (V \cdot W) &= (U \cdot V) \cdot W \\
U + V &= V + U \\
(U + V) \cdot W &= (U \cdot W) + (V \cdot W) \\
U \cdot (V + W) &= (U \cdot V) + (U \cdot W) \\
U + U &= U \\
\varepsilon \cdot U &= U \\
\emptyset \cdot U &= \emptyset \\
U + \emptyset &= U \\
U^* &= \varepsilon + U^+
\end{aligned}
$$

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
- Σ is an alphabet
- $q_0 \in Q$ is an initial state
- $\delta : Q \times \Sigma \to Q$ 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

$$\mapsto : (Q \times \Sigma^*) \times (Q \times \Sigma^*)$$

such as

$$(q, aw) \mapsto (q', w) \iff \delta(q, a) = q'$$

Automaton accepts word w if

$$(q_0, w) \mapsto^* (q, \varepsilon), \ q \in F$$

Nondeterministic Finite Automaton

Definition

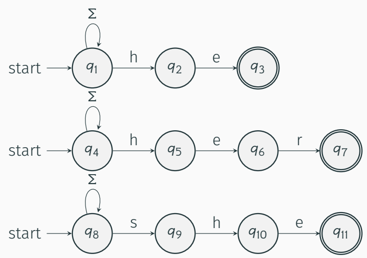
Nondeterministic Finite Automaton (NFA) is a quintuple $A = (Q, \Sigma, q_0, \delta, F)$, where

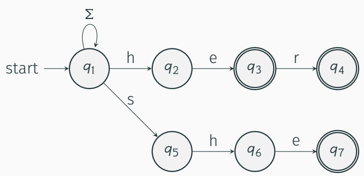
- Q is a finite set of states
- Σ is an alphabet
- $q_0 \in Q$ is an initial state
- $\delta : Q \times \Sigma \to P(Q)$ is a transition function
- $F \subseteq Q$ is a set of final states

- 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.

Nondeterministic Finite Automaton – example

Set of patterns $P = \{\text{he}, \text{her}, \text{she}\}$





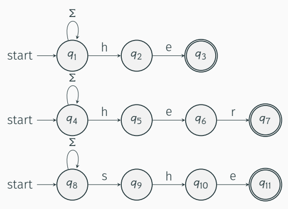
NFA $\longrightarrow$ DFA Conversion

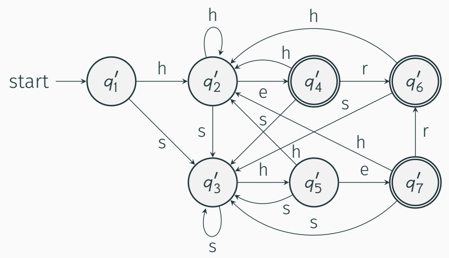
The DFA can be constructed using the **powerset construction**.
NFA $A = (Q, \Sigma, S, \delta, F) \longrightarrow$ DFA $A' = (Q', \Sigma', q_0', \delta', F')$

- $Q' \subseteq P(Q)$
- $\Sigma' = \Sigma$
- $q_0' = S$
- $\delta'(q', x) = \cup\delta(q, x)$ for all $q \in q'$
- $F' = \{q' \in Q' | q' \cap F \neq \emptyset\}$

NFA $\longrightarrow$ DFA Conversion I

State	Label	e	h	r	s	other
{1, 4, 8}	q_1'	{1, 4, 8}	{1, 2, 4, 5, 8}	{1, 4, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 2, 4, 5, 8}	q_2'	{1, 3, 4, 6, 8}	{1, 2, 4, 5, 8}	{1, 4, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 4, 8, 9}	q_3'	{1, 4, 8}	{1, 2, 4, 5, 8, 10}	{1, 4, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 3, 4, 6, 8}	q_4'	{1, 4, 8}	{1, 2, 4, 5, 8}	{1, 4, 7, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 2, 4, 5, 8, 10}	q_5'	{1, 3, 4, 6, 8, 11}	{1, 2, 4, 5, 8}	{1, 4, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 4, 7, 8}	q_6'	{1, 4, 8}	{1, 2, 4, 5, 8}	{1, 4, 8}	{1, 4, 8, 9}	{1, 4, 8}
{1, 3, 4, 6, 8, 11}	q_7'	{1, 4, 8}	{1, 2, 4, 5, 8}	{1, 4, 7, 8}	{1, 4, 8, 9}	{1, 4, 8}

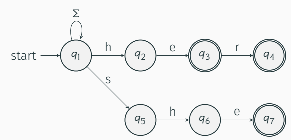


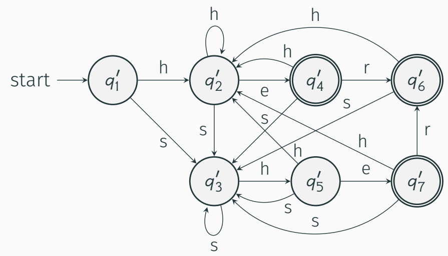


Only reachable states, transitions to state q_1 are not shown.

NFA $\longrightarrow$ DFA Conversion II

State	Label	e	h	r	s	other
{1}	q_1'	{1}	{1, 2}	{1}	{1, 5}	{1}
{1, 2}	q_2'	{1, 3}	{1, 2}	{1}	{1, 5}	{1}
{1, 5}	q_3'	{1}	{1, 2, 6}	{1}	{1, 5}	{1}
{1, 3}	q_4'	{1}	{1, 2}	{1, 4}	{1, 5}	{1}
{1, 2, 6}	q_5'	{1, 3, 7}	{1, 2}	{1}	{1, 5}	{1}
{1, 4}	q_6'	{1}	{1, 2}	{1}	{1, 5}	{1}
{1, 3, 7}	q_7'	{1}	{1, 2}	{1, 4}	{1, 5}	{1}





Derivation of Regular Expression

For given regular expression R, derivation is defined as

$$h\left(\frac{\mathrm{d}R}{\mathrm{d}x}\right) = \{y \mid xy \in h(R)\}$$

Example

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

$$\begin{aligned} h\left(\frac{\mathrm{d}R}{\mathrm{d}a}\right) &= \{\varepsilon\} \\ h\left(\frac{\mathrm{d}R}{\mathrm{d}s}\right) &= \{hell, top\} \\ h\left(\frac{\mathrm{d}R}{\mathrm{d}t}\right) &= \emptyset \end{aligned}$$

Derivation of Regular Expression – properties

$$\begin{aligned} \frac{\mathrm{d}\emptyset}{\mathrm{d}a} &= \emptyset, \forall a \in \Sigma & \frac{\mathrm{d}(U+V)}{\mathrm{d}a} &= \frac{\mathrm{d}U}{\mathrm{d}a} + \frac{\mathrm{d}V}{\mathrm{d}a} \\ \frac{\mathrm{d}\varepsilon}{\mathrm{d}a} &= \emptyset, \forall a \in \Sigma & \frac{\mathrm{d}(U \cdot V)}{\mathrm{d}a} &= \frac{\mathrm{d}U}{\mathrm{d}a} \cdot V, \varepsilon \notin U \\ \frac{\mathrm{d}a}{\mathrm{d}a} &= \varepsilon, \forall a \in \Sigma & \frac{\mathrm{d}(U \cdot V)}{\mathrm{d}a} &= \frac{\mathrm{d}U}{\mathrm{d}a} \cdot V + \frac{\mathrm{d}V}{\mathrm{d}a}, \varepsilon \in U \\ \frac{\mathrm{d}b}{\mathrm{d}a} &= \emptyset, \forall b \neq a & \frac{\mathrm{d}V^*}{\mathrm{d}a} &= \frac{\mathrm{d}V}{\mathrm{d}a} \cdot V^* \end{aligned}$$

$$\frac{\mathrm{d}V}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}a_n}\left(\frac{\mathrm{d}}{\mathrm{d}a_{n-1}}\left(\cdots \frac{\mathrm{d}}{\mathrm{d}a_2}\left(\frac{\mathrm{d}V}{\mathrm{d}a_1}\right)\right)\right), \text{ for } x = a_1 a_2 \ldots a_n$$

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 Σ.
- 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

$$\begin{aligned} q_0 &= V \\ \delta(q,x) &= \frac{\mathrm{d}q}{\mathrm{d}x} \\ F &= \{q \in Q | \varepsilon \in h(q)\} \end{aligned}$$

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:

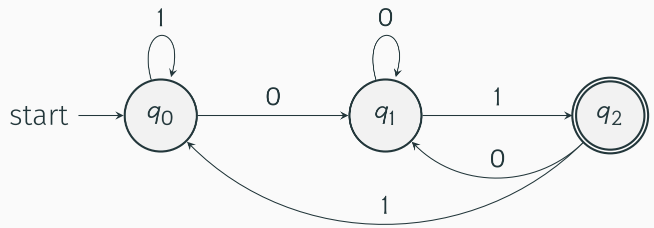
$$\begin{aligned} \frac{\mathrm{d}((0+1)^* \cdot 01)}{\mathrm{d}0} &= \frac{\mathrm{d}((0+1)^*)}{\mathrm{d}0} \cdot 01 + \frac{\mathrm{d}01}{\mathrm{d}0} \\ &= \frac{\mathrm{d}(0+1)}{\mathrm{d}0} \cdot (0+1)^* \cdot 01 + 1 \\ &= \left(\frac{\mathrm{d}0}{\mathrm{d}0} + \frac{\mathrm{d}1}{\mathrm{d}0}\right) \cdot (0+1)^* \cdot 01 + 1 \\ &= (\varepsilon + \emptyset) \cdot (0+1)^* \cdot 01 + 1 \\ &= (0+1)^* \cdot 01 + 1 \end{aligned}$$

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

$$\begin{aligned} \frac{\mathrm{d}((0+1)^* \cdot 01)}{\mathrm{d}1} &= \frac{\mathrm{d}((0+1)^*)}{\mathrm{d}1} \cdot 01 + \frac{\mathrm{d}01}{\mathrm{d}1} \\ &= \frac{\mathrm{d}(0+1)}{\mathrm{d}1} \cdot (0+1)^* \cdot 01 + \emptyset \\ &= \left(\frac{\mathrm{d}0}{\mathrm{d}1} + \frac{\mathrm{d}1}{\mathrm{d}1}\right) \cdot (0+1)^* \cdot 01 \\ &= (\emptyset + \varepsilon) \cdot (0+1)^* \cdot 01 \\ &= (0+1)^* \cdot 01 \end{aligned}$$

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

Regular Expression	State	0	1
$(0+1)^* \cdot 01$	q_0	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01$
$(0+1)^* \cdot 01 + 1$	q_1	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01 + \varepsilon$
$(0+1)^* \cdot 01 + \varepsilon$	q_2	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01$



Pattern Matching

Approximate 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.

k	a	r	o	l	i	n
k	a	t	h	r	i	n
0	0	1	1	1	0	0

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.

Levenshtein distance (cont.)

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.

Levenshtein distance (cont.)

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).

Levenshtein distance (cont.)

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

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), the second to insertion (to b)and the third to match or mismatch.

Levenshtein distance (cont.)

```c
int LevenshteinDistance(const char *s, int len_s, const
    char *t, int len_t)
{
 int cost;

 /* base case: empty strings */
 if (len_s == 0) return len_t;
 if (len_t == 0) return len_s;

 /* test if last characters of the strings match */
 if (s[len_s-1] == t[len_t-1])
  cost = 0;
 else
  cost = 1;

```

Levenshtein distance (cont.)

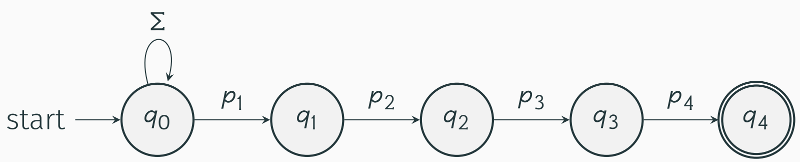
```
/* return minimum of delete char from s, delete char
    from t, and delete char from both */
return minimum
  (
    LevenshteinDistance(s, len_s-1, t, len_t) + 1,
    LevenshteinDistance(s, len_s, t, len_t-1) + 1,
    LevenshteinDistance(s, len_s-1, t, len_t-1) + cost
  );
}
```


Levenshtein distance (cont.)

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	**1**[a]	2	3	4	5	6
i	2	2	**1**[b]	2	3	4	5
t	3	3	2	**1**[c]	2	3	4
t	4	4	3	2	**1**[d]	2	3
i	5	5	4	3	2	**2**[e]	3
n	6	6	5	4	3	3	**2**[f]
g	7	7	6	5	4	4	**3**[g]

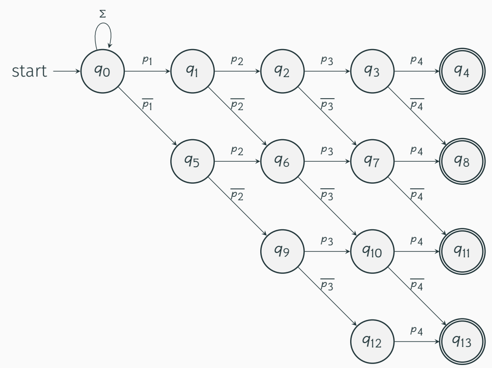
[a] subst. of k for s
[b] i is equal i
[c] t is equal t
[d] t is equal t
[e] subst. of e for i
[f] n is equal n
[g] delete g

Approximate pattern matching using finite automata



NFA for the exact string matching ($m = 4$)

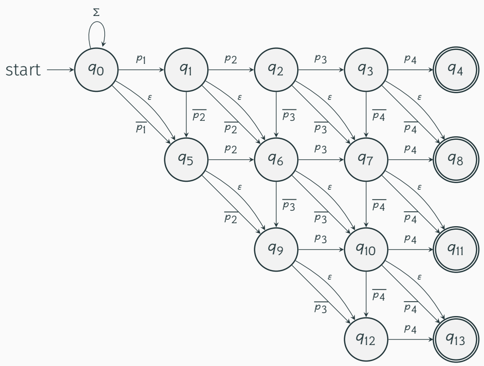
Approximate pattern matching using finite automata (cont.)



left to right – match
diagonal – replace

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

Approximate pattern matching using finite automata (cont.)



left to right – match
diagonal – replace
down – insert
diagonal ε– delete

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