Research Report DC-98-04

Simulation of Nondeterministic Finite Automata in Approximate String and Sequence Matching

Jan Holub

April 1998

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

This report was prepared as a part of the project

Approximate Pattern Matching

This research has been supported by grants FRVŠ 0892/97 "Approximate Pattern Matching in Text" and GAČR 201/98/1155 "Pattern Matching in Text".

This report has been defended as the Postgraduate Study Report DC-PSR-98-01 at the Department of Computer Science and Engineering on March 19, 1998.

Table of contents

Simulation of Nondeterministic Finite Automata in Approximate String and Sequence Matching

Jan Holub
holub@cs.felk.cvut.cz
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Abstract

We present a detailed description of the simulation of nondeterministic finite automata (*NFA*) for the approximate string matching. Two methods of the simulation are presented — one is based on arithmetic operations and the other uses bit parallelism. The first method is called dynamic programming and the second one is called Shift-Or algorithm. Using the knowledge of simulation of *NFA* by Shift-Or algorithm we have designed the modification of Shift-Or algorithm for the approximate string matching using the generalized Levenshtein distance and the modification for the exact and approximate sequence matching. For the case that we are not interested in the number of errors in the found string we have also designed the reduction of states of *NFA* as well as the modification of Shift-Or algorithm for this case.

Keywords

sequence matching, string matching, approximate pattern matching, finite automata, simulation of nondeterministic finite automata, dynamic programming, Shift-Or algorithm, bitwise parallelism

1 Introduction

There are many problems to be solved in Computer Science. Some of them can be solved using the finite automata. If we construct the nondeterministic finite automaton (*NFA*) for some problem we cannot use it directly because of its nondeterminism. The way of using this *NFA* is to transform it to the corresponding deterministic finite automaton (*DFA*) but such *DFA* can have $\mathcal{O}(2^m)$ states where m is a number of states of *NFA*. This fact discourages many people from using the finite automata but in the reality the increase of states is not always exponential. We show another way of usage of *NFA*. This way is the simulation of *NFA* and we show it in the area of the approximate string matching.

The approximate string matching is a problem which has been studied since the beginning of 1980's. Several algorithms have been developed to solve that problem among them the most important are "dynamic programming" [Ukk85] and "Shift-Or" based algorithms [BYG92, WM92]. These two algorithms have been developed in an intuitive way. We show that these two algorithms simulate *NFA*. This view make these algorithms easier to understand.

The approximate string matching algorithms are utilized in various areas: text information systems, computer vision, molecular biology, genetics, ... etc.

This report is organized as follows. Section 2 introduces the basic definitions and the terminology. In Section 3 we show the nondeterministic finite automata for the exact and approximate string matching. Section 4 describes "dynamic programming" and Section 5 describes "Shift-Or" based algorithms and both sections also show how these algorithms simulate *NFA*s. Section 7 describes the reduction of finite automata for the case that we are not interested in the number of errors in the found string and in Section 6 we present the application of finite automata and "Shift-Or" based algorithms to the exact and approximate sequence matching. Section 8 concludes with outlines for future work.

2 Basic Definitions

In this section we define basic notions which are used in this report.

Definition 2.1 (Alphabet)
An *alphabet* Σ is a finite non-empty set of *symbols*.

Definition 2.2 (Complement of symbol)
A *complement* of symbol a over Σ, where $a \in \Sigma$, is a set $\Sigma - \{a\}$ and is denoted $\overline{a}$.

Definition 2.3 (String)
A *string* over Σ is any sequence of symbols from Σ.

Definition 2.4 (Set of all strings)
The *set of all strings* over Σ is denoted Σ^*.

Definition 2.5 (Substring)
String x is a *substring* of string y if $y = uxv$ where $x, y, u, v \in \Sigma^*$.

Definition 2.6 (Length of string)
The *length of string* w is the number of symbols in string $w \in \Sigma^*$ and is denoted $|w|$.

Definition 2.7 (Empty string)
An *empty string* is a string of length 0 and is denoted ε.

Definition 2.8 (Powerset)
For any set A the set of all subsets of A is called *powerset* of A and is denoted $\mathcal{P}(A)$.

Definition 2.9 (Replace)
Edit operation *replace* is an operation which converts string wav to string wbv where $w, v \in \Sigma^*$, $a, b \in \Sigma$, $a \neq b$ (one symbol is replaced by another).

Definition 2.10 (Insert)
Edit operation *insert* is an operation which converts string wv to string wav where $w, v \in \Sigma^*$, $a \in \Sigma$ (a symbol is inserted into a string).

Definition 2.11 (Delete)
Edit operation *delete* is an operation which converts string wav to string wv where $w, v \in \Sigma^*$, $a \in \Sigma$ (a symbol is removed from a string).

Definition 2.12 (Transpose)
Edit operation *transpose* is an operation which converts $wabv$ to string $wbav$ where $w, v \in \Sigma^*$, $a, b \in \Sigma$ (two adjacent symbols are exchanged).

Definition 2.13 (Hamming distance)
Hamming distance $D_H(v, w)$ between two strings $v, w \in \Sigma^*$, $|v| = |w|$ is a minimal number of edit operations *replace* needed to convert v to w.

Definition 2.14 (Levenshtein distance)
Levenshtein distance $D_L(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimal number of edit operations *replace*, *insert*, and *delete* needed to convert v to w.

Definition 2.15 (Generalized Levenshtein distance)
Generalized Levenshtein distance $D_G(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimal number of edit operations *replace*, *insert*, *delete*, and *transpose* needed to convert v to w.

Definition 2.16 (Approximate string matching)
Approximate string matching is defined as a searching for all occurrences of pattern $P = p_1 p_2 \ldots p_m$ in text $T = t_1 t_2 \ldots t_n$ with at most k errors allowed. The number of errors allowed in a found substring is determined by the Hamming, Levenshtein, and generalized Levenshtein distances. The approximate string matching using the Hamming distance is called *approximate string matching with k mismatches* and the approximate string matching using the Levenshtein distance is called *approximate string matching with k differences*.

Definition 2.17 (Approximate sequence matching)
Approximate sequence matching is defined as a searching for all occurrences of pattern $P = p_1 p_2 \ldots p_m$ in text $T = t_1 t_2 \ldots t_n$ with at most k errors allowed such that between symbols p_i and p_{i+1}, $0 < i < m$, in text T can be located any number of symbols. For the approximate sequence matching we can also use the Hamming, Levenshtein, and generalized Levenshtein distances.

Definition 2.18 (Exact string (sequence) matching)
Exact string (sequence) matching is an approximate string (sequence) matching where $k = 0$.

Definition 2.19 (Nondeterministic finite automaton)
Nondeterministic finite automaton (*NFA*) is a quintuple $(Q, \Sigma, \delta, Q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $Q_0 \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of final states.

Definition 2.20 ($\varepsilon CLOSURE$)
Let $M = (Q, \Sigma, \delta, Q_0, F)$ be an *NFA* and $P \subseteq Q$. $\varepsilon CLOSURE(P)$ is a set $\{q' \mid q' \in \delta(q, \varepsilon), q \in P\} \cup P$.

Definition 2.21 (Configuration of *NFA*)
A *configuration of NFA* $M = (Q, \Sigma, \delta, Q_0, F)$ is a pair $(q, p) \in Q \times \Sigma^*$. An *initial configuration of NFA* is a pair (q_0, p) where $q_0 \in Q_0$ and a *final (accepting) configuration of NFA* is a pair (q_f, ε) where $q_f \in F$.

Definition 2.22 (Move of *NFA*)
A *move of NFA* $M = (Q, \Sigma, \delta, Q_0, F)$ is a relation $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$ where $q_2 \in \delta(q_1, a)$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a transitive and reflexive closure of relation $\vdash_M$.

Definition 2.23 (Deterministic finite automaton)
Deterministic finite automaton (*DFA*) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a mapping $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states.

Definition 2.24 (Configuration of *DFA*)
A *configuration of DFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a pair $(q, p) \in Q \times \Sigma^*$. An *initial configuration of DFA* is a pair (q_0, p) and *final (accepting) configuration of DFA* is a pair (q_f, ε) where $q_f \in F$.

Definition 2.25 (Move of *DFA*)
A *move of DFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a relation $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$ where $\delta(q_1, a) = q_2$, $a \in \Sigma$, $w \in \Sigma^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a transitive and reflexive closure of relation $\vdash_M$.

Definition 2.26 (Finite Automaton)
A *finite automaton* (*FA*) is *DFA* or *NFA*.

Definition 2.27 (Language accepted by *FA*)
A *language accepted by FA M* is a set $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \varepsilon)$, q_0 is an initial state of M and q_f is a final state of $M\}$.

Definition 2.28 (Equivalence of *FA*s)
Two *FA*s M and M' are *equivalent* if $L(M) = L(M')$.

Definition 2.29 (Inaccessible state)
An *inaccessible state* of *FA M* is a state q such that there does not exist any string $w \in \Sigma^*$ such that $(q_0, w) \vdash_M^* (q, \varepsilon)$ where q_0 is an initial state of M.

Definition 2.30 (Active state)
An *active state* of *FA M* after reading input string $w \in \Sigma^*$ is each state q such that $(q_0, w) \vdash_M^* (q, \varepsilon)$ where q_0 is an initial state of M.

3 Finite Automata

In this chapter we introduce *NFA*s for the exact and approximate string matching and their transformation to the equivalent *DFA*s.

3.1 Nondeterministic Finite Automata

We construct *NFA*s searching for the first occurrence of pattern P in text T. They can also continue the searching and find all other occurrences of the pattern in the text. We check in each step of run of the automaton whether any final state is active. Therefore these *NFA*s can determine:

1. whether the pattern is in the text or not,

2. the first occurrence of the pattern in the text,
3. number of all occurrences of the pattern in the text,
4. all occurrences of the pattern in the text.

3.1.1 Exact String Matching

In the exact string matching we search for all occurrences of pattern $P = p_1p_2 \dots p_m$ in text $T = t_1t_2 \dots t_n$ and we do not care the string before string P. Therefore we construct such *NFA* M that accepts language $L(M) = \{wP \mid w \in \Sigma^*\}$. In this *NFA* each state q_i except the initial one q_0 represents that first i symbols of the pattern has been found in the text. To skip the symbols before the occurrence of the pattern we add a self-loop into the initial state for each symbol of input alphabet Σ.

Algorithm 3.1 (Construction of *NFA* for the exact string matching)
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *NFA* M accepting language $L(M) = \{wP \mid w \in \Sigma^*\}$.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, \{q_0\}, \{q_m\})$, where the mapping δ is constructed in the following way:

$\delta(q_0, a) := \{q_0\},\ a \in \Sigma - \{p_1\}$
$\delta(q_0, p_1) := \{q_0, q_1\}$
$\delta(q_i, p_{i+1}) := \{q_{i+1}\},\ 0 < i < m$
$\delta(q_m, a) := \emptyset,\ a \in \Sigma$

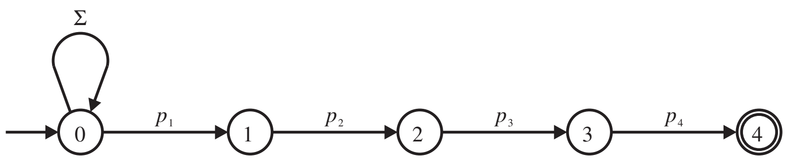

Figure 1: *NFA* for the exact string matching ($m = 4$).

The resulting *NFA* has $m + 1$ states and an example of such *NFA* for $m = 4$ is shown in Figure 1*.

3.1.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance for pattern $P = p_1p_2 \dots p_m$ we have to consider maximum number of errors allowed k. Therefore we use $k + 1$ parallel *NFA*s for the exact string matching $(M_0, M_1, \dots, M_k)$ — one for 'no error' (level 0), one for 'one error' (level 1), ..., one for 'k errors' (levels k). These $k + 1$ automata are connected by transitions representing edit operation *replace*. Each such transition is labeled by symbol $\overline{p}_{j+1}$ (mismatching the symbol p_{j+1} in pattern P) and leads from state q_j of automaton M_i to state q_{j+1} of automaton M_{i+1}, $0 \le i < k$, $0 \le j < m$ — depth of automaton (corresponding to a pointer in the pattern) is increased as well as the minimum number of errors. The initial state of such automaton is the initial state of automaton M_0. Inaccessible states of the resulting automaton are removed.

*In figures 'q' symbols are omitted and states are labeled only by indices. Initial states are denoted by arrow leading from no state and final states are denoted by double circle.

This automaton can be constructed by connecting $k+1$ parallel *NFA*s as described above or directly as shown by Algorithm 3.2.

Algorithm 3.2 (Construction of *NFA* for the approximate string matching using the Hamming distance)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_H(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, \{q_0\}, F)$, where Q, δ, and F are constructed in the following way:

$Q := \{q_0, q_1, \dots, q_{|Q|}\}$, $|Q| = (k+1)(m+1-\frac{k}{2})$
$l := 0$ /* number of level */
$r := 1$ /* order of symbol in P */
for $i := 0, 1, \dots, |Q|$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) := \emptyset$, $a \in \Sigma$ /* final state */
    $F := F \cup \{q_i\}$
    $l := l + 1$; $r := l$
  else
    $\delta(q_i, p_r) := \{q_{i+1}\}$ /* matching transition */
    if $l < k$ **then**
      $s := i + m + 1 - l$
      $\delta(q_i, a) := \{q_s\}$, $a \in \Sigma - \{p_r\}$ /* *replace* transition */
    endif
    $r := r + 1$
  endif
endfor
$\delta(q_0, a) := \delta(q_0, a) \cup \{q_0\}$, $a \in \Sigma$ /* self loop of initial state */

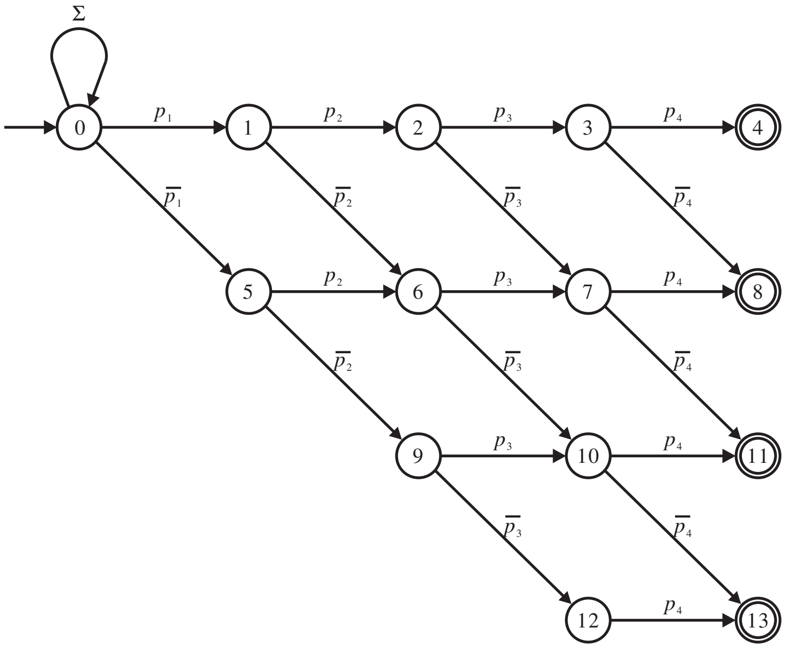

Figure 2: *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 3$).

The resulting *NFA* has $(m+1)+(m+1-1)+(m+1-2)+\cdots = \sum_{i=0}^{k}(m+1-i) = (k+1)(m+1-\frac{k}{2})$ states and an example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 2. The resulting *NFA* was also presented in [Dar92] and [Mel95].

3.1.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance we can use also $k+1$ parallel *NFA*s for the exact string matching $(M_0, M_1, \ldots, M_k)$ and connect them by the transitions representing edit operation *replace*. Since in the Levenshtein distance besides edit operation *replace* there are two new edit operations allowed we have to insert new transitions for these two edit operations into *NFA* for the Hamming distance.

The transition representing edit operation *insert* is labeled by symbol $\overline{p}_{j+1}$ and leads from state q_j of automaton M_i to state q_j of automaton M_{i+1}, $0 \leq i < k$, $0 < j < m$ — the minimum number of errors is increased but the depth of automaton is the same. The transition representing edit operation *delete* is labeled by ε and leads from state q_j of automaton M_i to state q_{j+1} of automaton M_{i+1}, $0 \leq i < k$, $0 \leq j < m$ — the depth of automaton is increased as well as the minimum number of errors but no symbol is read from the input. The initial state of resulting *NFA* is the initial state of automaton M_0 and inaccessible states are removed.

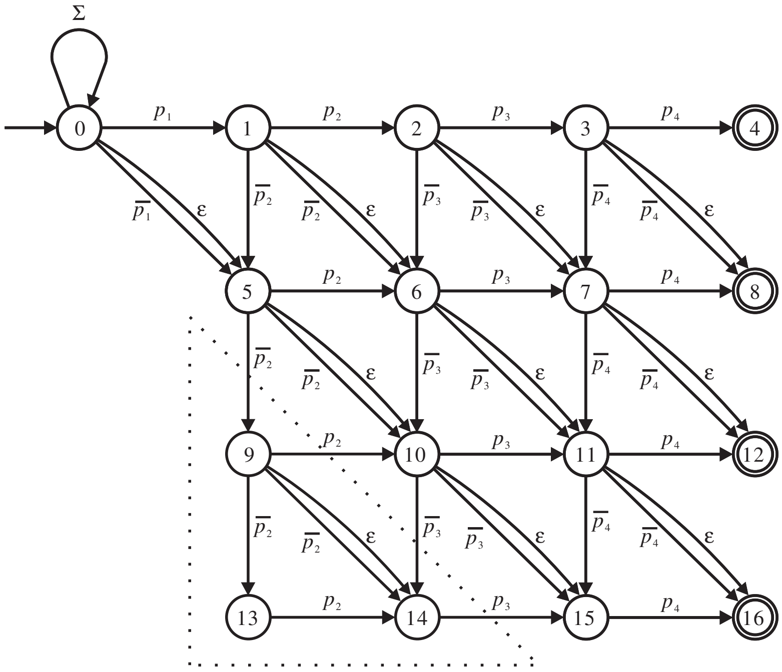

Figure 3: *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 3$).

This automaton, constructed by connecting $k + 1$ parallel *NFA*s as described above, was presented in [Mel96b] and an example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3. But in this *NFA* there are redundant states (in Figure 3 bordered by the dotted line).

Because of self loop of the initial state of the *NFA* the initial state is always active. Since states located on diagonal leading from the initial state are connected with the initial state by ε-transitions they are also active all the time. Therefore the states with transitions leading only into the states that are always active are redundant and can be removed. Moreover when transforming this *NFA* to *DFA* such states are eliminated automatically.

To eliminate such states from *NFA* during construction of *NFA* we put *replace* transition only to states q_j of automata M_i, $0 \le i < k$, $i < j < m$.

The *NFA* with redundant states eliminated can be constructed as described above, by modification of *NFA* for the Hamming distance or directly as shown by Algorithm 3.3.

Algorithm 3.3 (Construction of *NFA* for the approximate string matching using the Levenshtein distance)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_L(P, u) \le k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, \{q_0\}, F)$, where Q, δ, and F are constructed in the following way:

```
Q := {q_0, q_1, ..., q_|Q|}, |Q| = (k+1)(m+1-k/2)
l := k                                                /* number of level */
r := m                                                /* order of symbol in P */
for  i := |Q|, |Q|-1, ..., 0  do
  if  r = m  then
    δ(q_i, a) := ∅, a ∈ Σ                             /* final state */
    F := F ∪ {q_i}
  else
    δ(q_i, p_{r+1}) := {q_{i+1}}                      /* matching transition */
    if  l < k  then
      s := i + m + 1 - l
      δ(q_i, a) := {q_s}, a ∈ Σ - {p_{r+1}}           /* replace transition */
      if  r > l  then
        δ(q_i, a) := δ(q_i, a) ∪ {q_{s-1}}, a ∈ Σ - {p_{r+1}}   /* insert transition */
      endif
      δ(q_i, a) := δ(q_i, a) ∪ δ(q_s, a), a ∈ Σ       /* delete transition */
    else
      δ(q_i, a) := ∅, a ∈ Σ - {p_{r+1}}
    endif
  endif
  r := r - 1
  if  r < l  then
    r := m
    l := l - 1
  endif
endfor
δ(q_0, a) := δ(q_0, a) ∪ {q_0}, a ∈ Σ                 /* self loop of initial state */
```

The resulting *NFA* has $(k+1)(m+1-\frac{k}{2})$ states and an example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 4.

3.1.4 Approximate String Matching Using Generalized Levenshtein Distance

In the approximate string matching using the generalized Levenshtein distance we can use also $k+1$ parallel *NFA*s for the exact string matching $(M_0, M_1, \dots, M_k)$ and connect them by transitions representing edit operation *replace*, *insert*, and *delete*. Since in the generalized Levenshtein distance there is besides edit operations *replace*, *insert*, and *delete* a new edit operation allowed we have to insert new transitions for this edit operation into *NFA* for the Levenshtein distance.

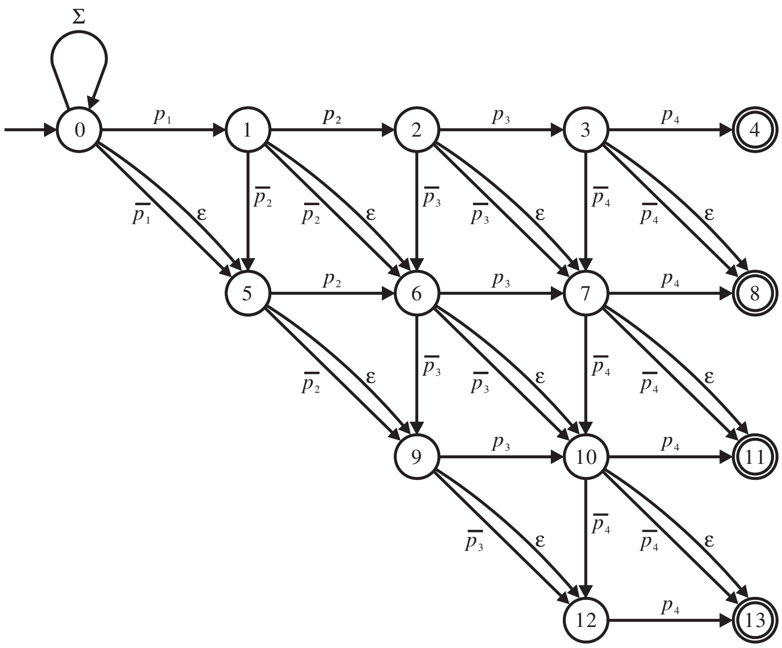

Figure 4: *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 3$).

The transition representing edit operation *transpose* leads from state q_j, $0 < j < m - 1$, of automaton M_i, $0 < i < k$ to new state q' and is labeled by symbol p_{j+2}. From state q' then leads transition to state q_{j+2} of automaton M_{i+1} and is labeled by symbol p_{j+1}. So two adjacent symbols q_{j+1} and q_{j+2} are accepted in reverse order, the depth of automaton is increased of 2 and the minimum number of errors is increased of 1.

Let us note that edit operation *transpose* can be replaced by two adjacent edit operations *replace* but it increases the number of errors of 2. Therefore if we want to label edit operation *transpose* only by 1 error as well as for *replace*, *insert*, and *delete* we have to use the generalized Levenshtein distance.

The *NFA* for the approximate string matching using the generalized Levenshtein distance can be constructed by connecting $k + 1$ parallel *NFA*s as described above or directly as shown by Algorithm 3.4.

Algorithm 3.4 (Construction of *NFA* for the approximate string matching using the generalized Levenshtein distance)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_G(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, \{q_0\}, F)$, where Q, δ, and F are constructed in the following way:

$Q := \{q_0, q_1, \dots, q_{|Q|}\}$, $|Q| = (m + 1) + k(2m - k)$
$l := k$ /* number of level */
$r := m$ /* order of symbol in P */
$j := |Q|$ /* index of last *transpose* state */
for $i := (k + 1)(m + 1 - \frac{k}{2}), (k + 1)(m + 1 - \frac{k}{2}) - 1, \dots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) := \emptyset$, $a \in \Sigma$ /* final state */
    $F := F \cup \{q_i\}$

else
$\delta(q_i, p_{r+1}) := \{q_{i+1}\}$ /∗ matching transition ∗/
if $l < k$ **then**
$s := i + m + 1 - l$
$\delta(q_i, a) := \{q_s\},\ a \in \Sigma - \{p_{r+1}\}$ /∗ *replace* transition ∗/
if $r > l$ **then**
$\delta(q_i, a) := \delta(q_i, a) \cup \{q_{s-1}\},\ a \in \Sigma - \{p_{r+1}\}$ /∗ *insert* transition ∗/
endif
if $r < m - 1$ **then**
$\delta(q_i, p_{r+2}) := \delta(q_i, p_{r+2}) \cup \{q_j\}$ /∗ *transpose* transition ∗/
$\delta(q_j, p_{r+1}) := \{q_{s+1}\}$
$j := j + 1$
endif
$\delta(q_i, a) := \delta(q_i, a) \cup \delta(q_s, a),\ a \in \Sigma$ /∗ *delete* transition ∗/
else
$\delta(q_i, a) := \emptyset,\ a \in \Sigma - \{p_{r+1}\}$
endif
endif
$r := r - 1$
if $r < l$ **then**
$r := m$
$l := l - 1$
endif
endfor
$\delta(q_0, a) := \delta(q_0, a) \cup \{q_0\},\ a \in \Sigma$ /∗ self loop of initial state ∗/

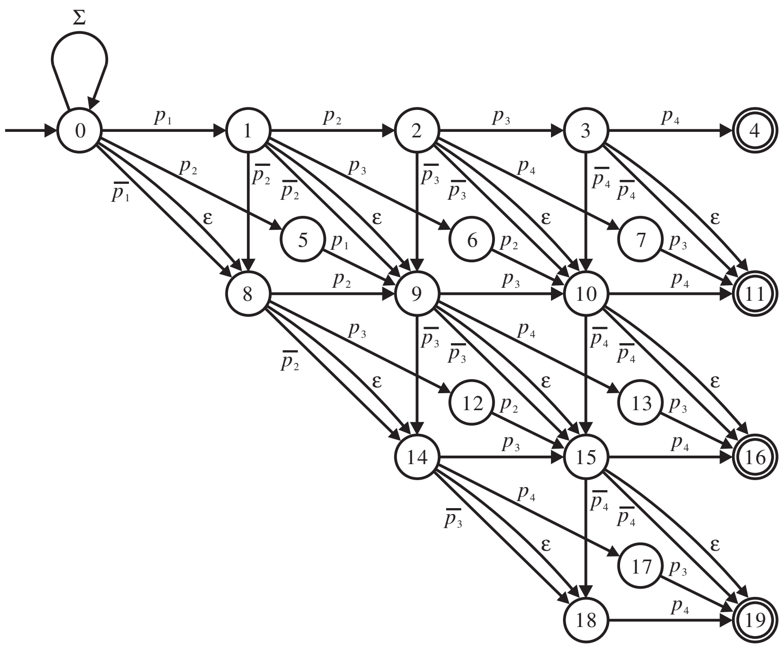

Figure 5: *NFA* for the approximate string matching using the generalized Levenshtein distance $(m = 4,\ k = 3)$.

The resulting *NFA* has $(m+1)+(m-1+m+1-1)+(m-2+m+1-2)+\cdots=(m+1)+\sum_{i=1}^{k}(2m+1-2i)=(m+1)+k(2m-k)$ states and an example of such *NFA* for $m=4$ and $k=3$ is shown in Figure 5.

3.2 Deterministic Finite Automata

NFA cannot be directly used because of its nondeterminism. There are two ways of using it:

1. transform it to the equivalent *DFA*,
2. simulate the run of *NFA*.

In this section we discuss the first way and the second one is discussed in the next section.

The transformation of any *NFA* to equivalent *DFA* can be done by subset construction [HU79, Koz97] as shown by Algorithm 3.5.

Algorithm 3.5 (Transformation of *NFA* to equivalent *DFA*)
Input: *NFA* $M=(Q,\Sigma,\delta,Q_0,F)$
Output: *DFA* M' accepting language $L(M)$
Method: $M'=(Q',\Sigma,\delta',q'_0,F')$ where Q', δ', q'_0, and F' are constructed in the following way:

$Q' := \mathcal{P}(Q)$
$\delta'(A,a) := \bigcup_{q\in A} \varepsilon CLOSURE(\delta(q,a)), a\in\Sigma, A\in Q'$
$q'_0 := \varepsilon CLOSURE(Q_0)$
$F' := \{A \mid A\in Q', A\cap F\neq\emptyset\}$

Algorithm 3.5 constructs equivalent *DFA* that has always $2^{|Q|}$ states. In many cases the resulting *DFA* contains a lot of inaccessible states. Therefore we use Algorithm 3.6 which does not insert the inaccessible states into the resulting *DFA*.

Algorithm 3.6 (Transformation of *NFA* to equivalent *DFA*)
Input: *NFA* $M=(Q,\Sigma,\delta,Q_0,F)$
Output: *DFA* M' accepting language $L(M)$
Method: $M'=(Q',\Sigma,\delta',q'_0,F')$ where Q', δ', q'_0, and F' are constructed in the following way:

$q'_0 := \varepsilon CLOSURE(Q_0)$
$Q' := \emptyset$
$S := \{q'_0\}$ /* set of not processed states */
for each $q'\in S$ **do**
  for each $a\in\Sigma$ **do**
    $\delta'(q',a) := \bigcup_{q\in q'} \varepsilon CLOSURE(\delta(q,a))$
    if $\delta'(q',a)\notin Q'$ **and** $\delta'(q',a)\notin S$ **then**
      $S := S\cup\delta'(q',a)$
    endif
  endfor
  if $q'\cap F\neq\emptyset$ **then**
    $F' := F'\cup\{q'\}$
  endif
  $S := S-\{q'\}$
  $Q' := Q'\cup\{q'\}$
endfor

The number of states of *DFA* for the exact string matching is m. The maximum number of states of *DFA* for the approximate string matching using the Hamming distance is $\frac{(k+1)!}{2}(k+2)^{m-k+1}$ [Mel96a], for the approximate string matching using the Levenshtein distance is $(k+1)!(k+1)^{m-k}$ [Mel96a] and for the approximate string matching using the generalized Levenshtein distance the maximum number of states has not been found yet.

The space complexity is determined by the number of states and the size of input alphabet Σ. But in implementation of *DFA* we can reduce alphabet Σ to set A of symbols contained in pattern P and one special symbol ϖ representing any other symbol of alphabet. Of course if $|\Sigma| > |A|+1$. When *DFA* runs mapping $\Sigma \mapsto A \cup \{\varpi\}$ is used. Therefore the space complexity is $|Q'|\min(|\Sigma|, |A|+1)$ where Q' is a number of states of *DFA*.

The time complexity of the run of *DFA* is $\mathcal{O}(|Q'|+n)$ where the first part of the formula is the time of the construction of *DFA* (dependent on the number of states) and the second part is dependent on the size of input text T.

When *DFA*s for the string matching are running it is checked in each step whether *DFA* is in a final state. If *DFA* is in a final state that represents that the pattern P has been found with at most j errors it is reported and the computation continues.

3.3 Simulation of Nondeterministic Finite Automata

In the case that the space complexity of *DFA* makes *DFA* unusable we can use some of simulation methods. In Algorithm 3.7 there is shown naive algorithm which is very similar to the transformation of *NFA* to the equivalent *DFA* in Algorithm 3.6. In each step of the run of *NFA* a new set of active states is computed by evaluating transitions from all states of the previous set of active states.

Algorithm 3.7 (Simulation of run of *NFA* — naive method)
Input: *NFA* $M = (Q, \Sigma, \delta, Q_0, F)$, input text $T = t_1t_2 \dots t_n$
Output: Output of run of *NFA*
Method: Set S of active states is used.

$S := Q_0$
$i := 1$
while $i \leq n$ **and** $S \neq \emptyset$ **do**
  $S := \bigcup_{q \in S} \varepsilon CLOSURE(\delta(q, t_i))$
  if $S \cap F \neq \emptyset$ **then**
    write(information associated with each final state in S)
  endif
  $i := i + 1$
endwhile

All other simulation methods are based on this naive method presented and they differ only in the implementation of the set of active states.

4 Dynamic Programming Algorithm

The dynamic programming is well known method that was used for the approximate string matching in [Sel80, Ukk85, Mel95].

4.1 Algorithm

The dynamic programming uses matrix D of size $(m+1)\times(n+1)$. Each element $d_{j,i}$, $0 \le j \le m$, $0 < i \le n$, contains the edit distance between string ending at i-th position in text T and first j symbols of pattern P.

4.1.1 Exact String Matching

The dynamic programming for the exact string matching is the same as the dynamic programming for the approximate string matching using the Hamming distance in which $k = 0$. See the below section.

4.1.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance each element $d_{j,i}$, $0 < i \le n$, $0 \le j \le m$, contains the Hamming distance between string $t_{i-j+1} \dots t_i$ and string $p_1 \dots p_j$. Since the Hamming distance compares two strings of equal length element $d_{j,i}$ is valid only if $j \le i$ and $i - n \le j - m$ — the pattern cannot start before the first symbol of T and cannot end after last symbol of T. Elements of matrix D are computed as follows:

$$
\begin{array}{lcll}
d_{j,0} & := & j, & 0 \le j \le m \\
d_{0,i} & := & 0, & 0 \le i \le n \\
d_{j,i} & := & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \textbf{else } d_{j-1,i-1} + 1, & 0 < i \le n, 0 < j \le m
\end{array}
\tag{1}
$$

In Formula (1) part $d_{j-1,i-1}$ represents matching — position i in text T is increased, position j in pattern P is increased and edit distance d is the same — and part $d_{j-1,i-1} + 1$ represents edit operation *replace* — position i in text T is increased, position j in pattern P is increased and edit distance d is increased.

Pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$ if $d_{m,i} \le k$, $m \le i \le n$. The maximum number of mismatches of the found string is $D_H(P, t_{i-m+1} \dots t_i) = d_{m,i}$.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 1.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	2	0	2	2	1	2	2	1	1	2	0	2	2	2	2
b	3	3	3	1	3	2	2	3	3	1	2	3	0	2	3	3
b	4	4	4	4	2	3	3	3	4	3	2	3	3	0	3	4
c	5	5	5	4	5	3	3	4	4	5	4	3	4	4	0	4
a	6	5	6	6	4	6	4	3	4	5	5	5	4	5	5	0

Table 1: Matrix D for pattern $P = adbbca$ and text $T = adcabcaabadbbca$ using the Hamming distance.

4.1.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance each element $d_{j,i}$, $0 < i \le n$, $0 \le j \le m$, contains the Levenshtein distance between string ending at position i in T and string $p_1 \dots p_j$. Since the Levenshtein distance compares two strings of not necessary equal length element $d_{j,i}$ is always valid and we can determine only ending position of the found string in text. The string can start before position $i - m + 1$ — at most k symbols inserted — or after position $i - m + 1$ — at most k symbols deleted — therefore the string can start at position $i - l + 1$, $m - k \leq l \leq m + k$. Elements of matrix D are computed as follows [Sel80, Ukk85]:

$$
\begin{array}{lcll}
d_{j,0} & := & j, & 0 \leq j \leq m \\
d_{0,i} & := & 0, & 0 \leq i \leq n \\
d_{j,i} & := & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \textbf{else } \mathbf{min}(d_{j-1,i-1} + 1, d_{j,i-1} + 1, d_{j-1,i} + 1) & 0 < i \leq n, 0 < j \leq m
\end{array}
\tag{2}
$$

In Formula (2) part $d_{j-1,i-1}$ represents matching, part $d_{j-1,i-1} + 1$ represents edit operation *replace*, part $d_{j,i-1} + 1$ represents edit operation *insert* — position i in text T is increased, position j in pattern P is not increased and edit distance d is increased — and part $d_{j-1,i} + 1$ represents edit operation *delete* — position i in text T is not increased, position j in pattern P is increased and edit distance d is increased.

Pattern P is found with at most k differences ending at position i if $d_{m,i} \leq k$, $1 \leq i \leq n$. The maximum number of differences of the found string is $D_L(P, t_{i-l+1} \dots t_i) = d_{m,i}$, $m - k \leq l \leq m + k$, $l < i$.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 2.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
a	6	5	4	3	2	3	3	2	3	4	3	4	3	2	1	0

Table 2: Matrix D for pattern $P = adbbca$ and text $T = adcabcaabadbbca$ using the Levenshtein distance.

4.1.4 Approximate String Matching Using Generalized Levenshtein Distance

For the approximate string matching using the generalized Levenshtein distance we have modified formula for the Levenshtein distance such that we have added the part for edit transition *transpose*. The resulting formula is as follows:

$$
\begin{array}{lcll}
d_{j,0} & := & j, & 0 \leq j \leq m \\
d_{0,i} & := & 0, & 0 \leq i \leq n \\
d_{j,i} & := & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \textbf{else } \mathbf{min}(d_{j-1,i-1} + 1, d_{j,i-1} + 1, d_{j-1,i} + 1) & i = 1, 0 < j \leq m \\
 & & & \text{or } 0 < i \leq n, j = 1 \\
d_{j,i} & := & \mathbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \quad \textbf{else } d_{j-1,i-1} + 1, d_{j,i-1} + 1, d_{j-1,i} + 1, & \\
 & & \quad \textbf{if } t_{i-1} = p_j \textbf{ and } t_i = p_{j-1} \textbf{ then } d_{j-2,i-2} + 1) & 1 < i \leq n, 1 < j \leq m
\end{array}
\tag{3}
$$

In Formula (3) part $d_{j-1,i-1}$ represents matching, part $d_{j-1,i-1}+1$ represents edit operation *replace*, part $d_{j,i-1}+1$ represents edit operation *insert*, part $d_{j-1,i}+1$ represents edit operation *delete* and part $d_{j-2,i-2}+1$ represents edit operation *transpose* — position i in text T is increased of 2, position j in pattern P is increased of 2 and edit distance d is increased of 1.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ is shown in Table 3.

D	-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	2	2	1	1	1	1	0	1	2	2	1
b	3	2	1	0	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	1	1	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	1	1	2	3	3	3	3	2	1	0	1
a	6	5	4	3	2	2	1	2	3	3	4	3	2	1	0

Table 3: Matrix D for pattern $P = adbbca$ and text $T = adbcbaabadbbca$ using the generalized Levenshtein distance.

4.2 Simulation of *NFA*

We have found out that the dynamic programming algorithm simulates run of *NFA* for the approximate string matching [Mel95, Hol96a, Mel96b]. In the dynamic programming there is for each depth j, $0 < j \leq m$, of *NFA* in each step i of the run one integer variable $d_{j,i}$ that contains the edit distance between string $p_1 \dots p_j$ and the string in text T ending at position i. Since for each value l of the edit distance there is one level of states in *NFA* the integer variable $d_{j,i} = l$ and contains number of level of active state in j-th depth of *NFA*. Each value of $d_{i,j}$ greater than $k+1$ can be replaced by value $k+1$ and represents that there is no active state in j-th depth of *NFA* in i-th step of the run. So in dynamic programming, the set of active states from Section 3.3 is implemented by the vector of integer variables — each variable for one depth of *NFA*.

In Formulae (1), (2), and (3) part $d_{j-1,i-1}$ represents matching transition — active state is moved from depth $j-1$ to the next depth j within level $d_{j-1,i-1}$ and symbol t_i is read from the input. In Formulae (1), (2), and (3) part $d_{j-1,i-1}+1$ represents transition *replace* — active state is moved from depth $j-1$ and level $d_{j-1,i-1}$ to the next depth j and the next level $d_{j-1,i-1}+1$ and symbol t_i is read from the input. In Formulae (2) and (3) part $d_{j,i-1}+1$ represents transition *insert* — active state is moved from level $d_{j-1,i-1}$ to the next level $d_{j-1,i-1}+1$ within depth $j-1$ and symbol t_i is read from the input. In Formulae (2) and (3) part $d_{j-1,i}+1$ represents transition *delete* — active state is moved from depth $j-1$ and level $d_{j-1,i-1}$ to the next depth j and the next level $d_{j-1,i-1}+1$ and no symbol is read from the input. In Formula (3) part $d_{j-2,i-2}+1$ represents transition *transpose* — active state is moved from depth $j-2$ and level $d_{j-2,i-2}$ to depth j and level $d_{j-2,i-2}+1$ and symbols t_{i-1} and t_i are read from the input. When representing transition *transpose* we need not new integer variable for states on transition *transpose* in *NFA*.

5 Shift-Or Based Algorithms

The Shift-Or algorithm [BYG92] for the approximate string matching is well known method that uses bitwise operations. There are also modifications of this algorithm called Shift-Add algorithm [BYG92] and Shift-And algorithm [WM92].

5.1 Algorithm

The Shift-Or algorithm uses matrices R^l, $0 \le l \le k$ of size $m \times (n+1)$ and mask matrix D of size $m \times |\Sigma|$. Each element $r^l_{j,i}$, $0 < j \le m$, $0 \le i \le n$, contains 0, if the edit distance between string $p_1 \dots p_j$ and string ending at position i in text T is $\le l$, or 1, otherwise. Each element $d_{j,x}$, $0 < j \le m$, $x \in \Sigma$, contains 0, if $p_j = x$, or 1, otherwise. The matrices are implemented as tables of bit vectors as follows:

$$R^l_i = \begin{bmatrix} r^l_{1,i} \\ r^l_{2,i} \\ \vdots \\ r^l_{m,i} \end{bmatrix} \quad \text{and} \quad D[x] = \begin{bmatrix} d_{1,x} \\ d_{2,x} \\ \vdots \\ d_{m,x} \end{bmatrix} \tag{4}$$

5.1.1 Exact String Matching

In the exact string matching vectors R^0_i, $0 \le i \le n$, are computed as follows [BYG92]:

$$\begin{array}{lcll} r^l_{j,0} & := & 1, & 0 < j \le m, 0 \le l \le k \\ R^0_i & := & \mathbf{shl}(R^0_{i-1}) \ \mathbf{or} \ D[t_i], & 0 < i \le n \end{array} \tag{5}$$

In Formula (5) operation **shl** is bitwise operation `left shift` that inserts 0 at the beginning of vector and operation **or** is bitwise operation `or`.

In Formula (5) part $\mathbf{shl}(R^0_{i-1})$ **or** $D[t_i]$ represents matching — position i in text T is increased, position in pattern P is increased by operation **shl** and the edit distance is the same. Pattern P is found at position $t_{i-m+1} \dots t_i$ if $r^0_{m,i} = 0$, $0 < i \le n$.

An example of mask matrix D for pattern $P = adbbca$ is shown in Table 4 and an example of matrix R^0 for exact searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 5.

D	a	b	c	d	$\Sigma - \{a,b,c,d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 4: Matrix D for pattern $P = adbbca$.

5.1.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance vectors R_i^l, $0 \le l \le k$, $0 \le i \le n$, are computed as follows:

$$
\begin{array}{lcll}
r_{j,0}^l & := & 1, & 0 < j \le m, 0 \le l \le k \\
R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \ \mathbf{or} \ D[t_i], & 0 < i \le n \\
R_i^l & := & (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or} \ D[t_i]) \ \mathbf{and} \ \mathbf{shl}(R_{i-1}^{l-1}), & 0 < i \le n, 0 < l \le k
\end{array}
\tag{6}
$$

In Formula (6) operation **and** is bitwise operation and.

In Formula (6) part $\mathbf{shl}(R_{i-1}^l)$ **or** $D[t_i]$ represents matching and part $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace* — position i in text T is increased, position in pattern P is increased and edit distance l is increased.

Pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$ if $r_{m,i}^k = 0$, $0 < i \le n$. The maximum number of mismatches of the found string is $D_H(P, t_{i-m+1} \dots t_i) = l$, where l is the minimum number such that $r_{m,i}^l = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ mismatches is shown in Table 5.

5.1.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance vectors R_i^l, $0 \le l \le k$, $0 \le i \le n$, are computed as follows [WM92]:

$$
\begin{array}{lcll}
r_{j,0}^l & := & 0, & 0 < j \le l, 0 < l \le k \\
r_{j,0}^l & := & 1, & l < j \le m, 0 \le l \le k \\
R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \ \mathbf{or} \ D[t_i], & 0 < i \le n \\
R_i^l & := & (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or} \ D[t_i]) & \\
 & & \quad \mathbf{and} \ \mathbf{shl}(R_{i-1}^{l-1} \ \mathbf{and} \ R_i^{l-1}) \ \mathbf{and} \ R_{i-1}^{l-1}, & 0 < i \le n, 0 < l \le k
\end{array}
\tag{7}
$$

In Formula (7) part $\mathbf{shl}(R_{i-1}^l)$ **or** $D[t_i]$ represents matching, part $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace*, part $\mathbf{shl}(R_i^{l-1})$ represents edit operation *delete* — position in pattern P is increased, position in text T is not increased and edit distance l is increased — and part R_{i-1}^{l-1} represents edit operation *insert* — position in pattern P is not increased, position in text T is increased and edit distance l is increased.

Pattern P is found with at most k differences ending at position i if $r_{m,i}^k = 0$, $0 < i \le n$. The maximum number of differences of the found string is l, where l is the minimum number such that $r_{m,i}^l = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ mismatches is shown in Table 6.

5.2 Simulation of *NFA*

The Shift-And algorithm [WM92] has exchanged bitwise operations **and** and **or** and bits 0 and 1 are also exchanged. Shift-Add algorithm [BYG92] uses instead bitwise operation **or** arithmetic operation **add**. Both modifications are equivalent to the Shift-Or algorithm.

We have found out that the Shift-Or algorithm simulates the run of *NFA* for the approximate string matching [Mel95, Hol96a, Mel96b, Hol97].

In the Shift-Or algorithm there is for each level l, $0 \le l \le k$, of states of *NFA* one bit vector R_i^l, $0 \le i \le n$. In the vectors, 0 represents active state and 1 represents nonactive state in the

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1
b	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1
b	1	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
c	1	1	1	1	1	0	0	1	1	1	1	0	1	1	0	1
a	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0

Table 5: Matrices R^l for pattern $P = adbbca$, $k = 3$ and text $T = adcabcaabadbbca$ using the Hamming distance.

NFA. So in Shift-Or algorithm, the set of active states from Section 3.3 is implemented by bit vectors — each vector for one level of states.

In Formulae (5), (6), and (7) part $\mathbf{shl}(R^l_{i-1})$ **or** $D[t_i]$ represents matching transition — each active state is moved to the next position in the right* in the same level. All active states are moved at once and only transitions corresponding to the read symbol t_i are selected by mask vector $D[t_i]$ which change 0 to 1 in each state such that matching transition leading to it is not labeled by t_i. The initial state of *NFA* is not in vector R^0 and is implemented by inserting 0 at the beginning of the vector in operation **shl** — initial state is always active because of its self-loop. In Formulae (6) and (7) part $\mathbf{shl}(R^{l-1}_{i-1})$ represents transition *replace* — each active state of level $l-1$ is moved to the next position in the right in level l. In Formulae (7) part R^{l-1}_{i-1} represents transition *insert* — each active state of level $l-1$ is moved to the same position in level l. In Formula (7) part $\mathbf{shl}(R^{l-1}_i)$ represents transition *delete* — each active state of

*In the Shift-Or algorithm, transitions from states in our figures are implemented by operation **shl** (shift to the left) because of easier implementation in the case that number of states of *NFA* is greater than length of computer word and vectors R^l have to be divided into two or more bit vectors.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
b	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
c	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0	0
a	1	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
a	1	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0

Table 6: Matrices R^l for pattern $P = adbbca$, $k = 3$ and text $T = adcabcaabadbbca$ using the Hamming distance.

level $l-1$ is moved to the next position in the right in level l while no symbol is read from the input (position i in text T is the same).

Let us note that transitions *replace* and *insert* are labeled by Σ and not by $\overline{p}_j$ as in *NFA*s described in Section 3. If we want Formulae (6) and (7) to exactly correspond to the *NFA*s we should mask transitions *replace* and *insert* by vector $\mathbf{not}D[t_i]$ that we get from vector $D[t_i]$ by exchanging 0s and 1s. This modification would add several operations and moreover is not necessary. Without the modification, if a state of depth j and level l is active all the states of the same depth j and levels greater than l are also active. That does not influence the result of searching because in Shift-Or algorithm we search for minimum number l such that $r^l_{m,i} = 0$.

5.2.1 Approximate String Matching Using Generalized Levenshtein Distance

Having the knowledge how the Shift-Or algorithm simulates the run of *NFA* we modify the algorithm for the approximate string matching using the generalized Levenshtein distance.

We modify Formula (7) for the approximate string matching using the Levenshtein distance such that we add the part representing edit operation *transpose*. Since *NFA* for the approximate string matching using the generalized Levenshtein distance has on each transition *transpose* one auxiliary state (see Figure 5) we introduce new bit vectors S_i^l, $0 \le l < k$, $0 \le i < n$, as follows:

$$S_i^l = \begin{bmatrix} s_{1,i}^l \\ s_{2,i}^l \\ \vdots \\ s_{m,i}^l \end{bmatrix} \tag{8}$$

Now, we construct the part of formula that simulates transition *transpose*. In this transition all states of level l are moved to the next position in the right ($\mathbf{shl}(R_{i-1}^l)$) of auxiliary level l'. Then only transitions corresponding to input symbol t_i have to be selected. It is provided by mask vector $D[t_i]$. Since each transition leading to state of depth j is labeled by symbol p_{j+1} we have to shift the mask vector in opposite direction than in which vector R_{i-1}^l is shifted ($\mathbf{shr}(D[t_i])$). All states of auxiliary level l' are stored in vector $S_i^l := \mathbf{shl}(R_{i-1}^l)$ **or** $\mathbf{shr}(D[t_i])$, $0 < i < n$, $0 \le l < k$. In next step $i+1$ of the computation all states of auxiliary level l' are moved to the next position in the right of level $l+1$ and only transitions corresponding to input symbol t_{i+1} are selected by mask vector $D[t_{i+1}]$. Since each transition leading to state of depth $j+1$ is labeled by symbol p_j we have to shift mask vector $D[t_{i+1}]$ in the same direction in which vector S_i^l is shifted. Therefore we insert part $\mathbf{shl}(S_{i-1}^{l-1}$ **or** $D[t_i])$ in Formula (7) as well as the formula for computation of vector S_i^l. The resulting formula is as follows:

$$\begin{array}{lcll} r_{j,0}^l & := & 0, & 0 < j \le l, 0 < l \le k \\ r_{j,0}^l & := & 1, & l < j \le m, 0 \le l \le k \\ R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \ \mathbf{or}\ D[t_i], & 0 < i \le n \\ R_i^l & := & (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or}\ D[t_i]) & \\ & & \ \mathbf{and}\ \mathbf{shl}(R_{i-1}^{l-1} \ \mathbf{and}\ R_i^{l-1} \ \mathbf{and}\ (S_{i-1}^{l-1} \ \mathbf{or}\ D[t_i])) & \\ & & \ \mathbf{and}\ R_{i-1}^{l-1}, & 0 < i \le n, 0 < l \le k \\ s_{j,0}^l & := & 1, & 0 < j \le m, 0 \le l < k \\ S_i^l & := & \mathbf{shl}(R_{i-1}^l) \ \mathbf{or}\ \mathbf{shr}(D[t_i]), & 0 < i < n, 0 \le l < k \end{array} \tag{9}$$

The time and space analysis of the Shift-Or algorithm is as follows [WM92]. Denote the computer word size by w. The preprocessing requires $\mathcal{O}(m|\Sigma|)$ time plus $\mathcal{O}(k\lceil \frac{m}{w} \rceil)$ to initialize the k vectors. The running time is $\mathcal{O}(nk\lceil \frac{m}{w} \rceil)$. The space complexity is $\mathcal{O}(|\Sigma|\lceil \frac{m}{w} \rceil)$ for mask matrix D plus $\mathcal{O}(k\lceil \frac{m}{w} \rceil)$ for the k vectors.

6 Approximate Sequence Matching

In Section 3 we have presented *NFA*s for the string matching. Since the sequence matching is very similar to the string matching we can modify *NFA*s for the string matching. The sequence matching differs from the string matching only in such a way that between occurrences of two adjacent symbols p_j and p_{j+1}, $0 < j < m$, of pattern P in text T there can be located any number of symbols x, $x \in \Sigma - \{p_{j+1}\}$. Therefore we add self-loop labeled by $\overline{p}_{j+1}$ to each state of depth j, $0 < j < m$, of *NFA*s for the string matching and we get *NFA*s for the sequence matching [Hol97].

An example of *NFA* for the exact sequence matching is shown in Figure 6 where $m = 4$.

We can also modify the Shift-Or algorithm for the approximate string matching to provide the approximate sequence matching. To do that we have to add simulation of self-loops to Formulae (6), (7), and (9). It is provided by adding the previous value R_{i-1}^l of vector in which

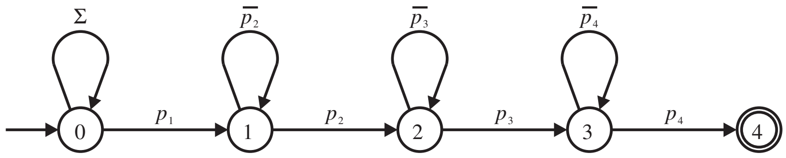

Figure 6: *NFA* for the exact sequence matching ($m = 4$).

we have to select only such states that matching transitions leading from them are not labeled by input symbol t_i. This selection is performed by mask vector $\mathbf{not}D[t_i]$ which has to be shifted in opposite direction than in which matching transitions are shifted.

Formula for the exact sequence matching is as follows:

$$\begin{array}{lcll} r^l_{j,0} & := & 1, & 0 < j \leq m, 0 \leq l \leq k \\ R^0_i & := & (\mathbf{shl}(R^0_{i-1}) \text{ } \mathbf{or} \text{ } D[t_i]) \text{ } \mathbf{and} \text{ } (R^0_{i-1} \text{ } \mathbf{or} \text{ } \mathbf{shr}(\mathbf{not}D[t_i])), & 0 < i \leq n \end{array} \tag{10}$$

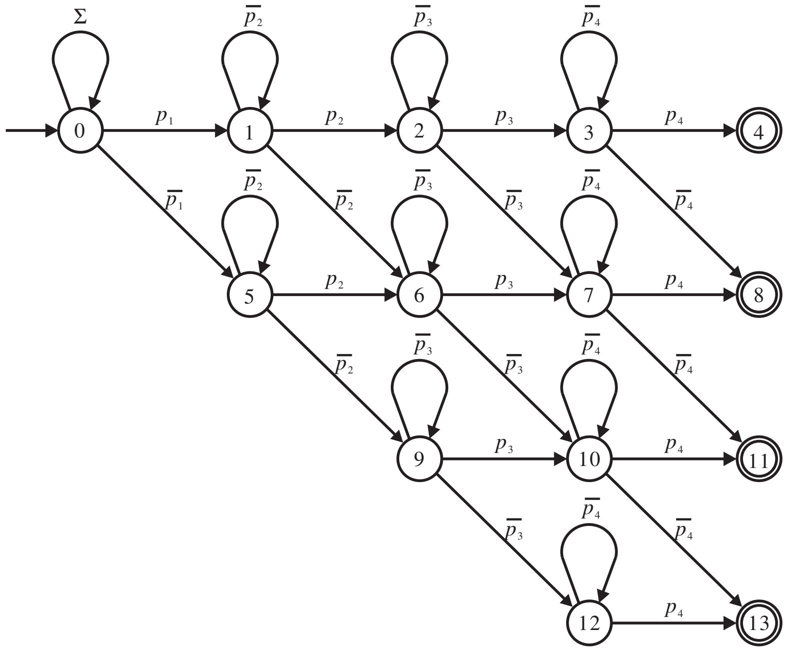

Figure 7: *NFA* for the approximate sequence matching using the Hamming distance ($m = 4$, $k = 3$).

NFA for the approximate sequence matching using the Hamming distance is shown in Figure 7 and the formula for the Shift-Or algorithm is as follows:

$$\begin{array}{lcll} r^l_{j,0} & := & 1, & 0 < j \leq m, 0 \leq l \leq k \\ R^0_i & := & \mathbf{shl}(R^0_{i-1}) \text{ } \mathbf{or} \text{ } D[t_i], & 0 < i \leq n \\ R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \text{ } \mathbf{or} \text{ } D[t_i]) \text{ } \mathbf{and} \text{ } \mathbf{shl}(R^{l-1}_{i-1}) & \\ & & \quad \mathbf{and} \text{ } (R^0_{i-1} \text{ } \mathbf{or} \text{ } \mathbf{shr}(\mathbf{not}D[t_i])), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{11}$$

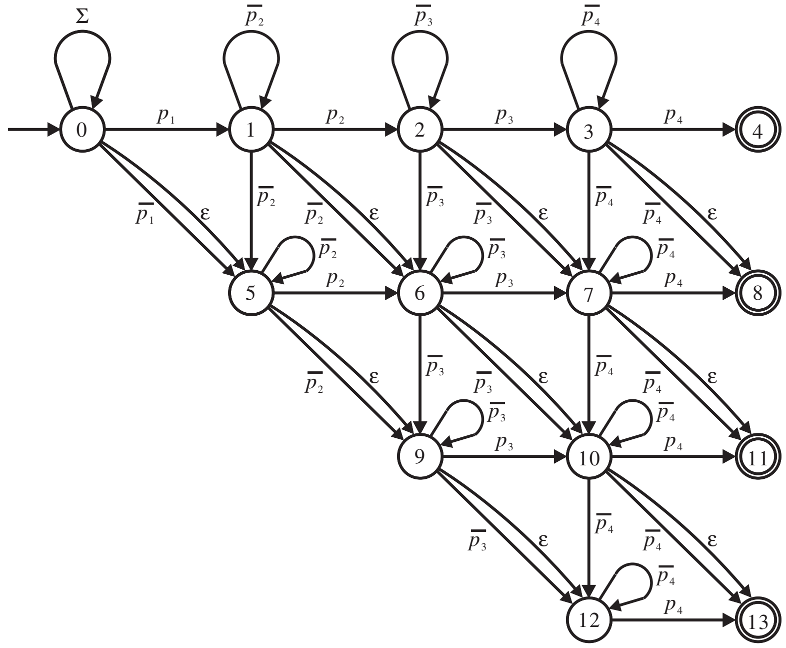

Figure 8: *NFA* for the approximate sequence matching using the Levenshtein distance ($m = 4$, $k = 3$).

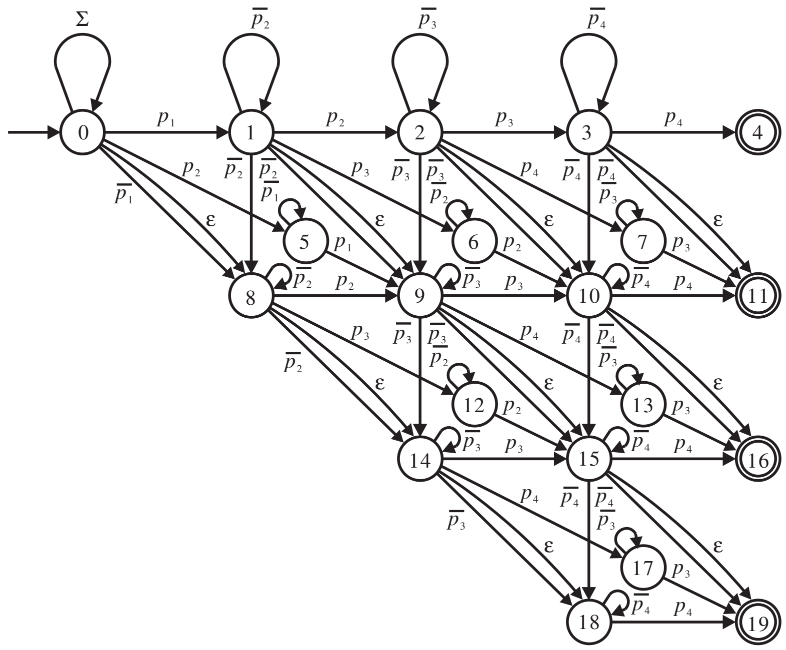

Figure 9: *NFA* for the approximate sequence matching using the generalized Levenshtein distance ($m = 4$, $k = 3$).

NFA for the approximate sequence matching using the Levenshtein distance is shown in Figure 8 and the formula for the Shift-Or algorithm is as follows:

$$
\begin{array}{lcll}
r^l_{j,0} & := & 0, & 0 < j \leq l, 0 < l \leq k \\
r^l_{j,0} & := & 1, & l < j \leq m, 0 \leq l \leq k \\
R^0_i & := & \mathbf{shl}(R^0_{i-1}) \ \mathbf{or} \ D[t_i], & 0 < i \leq n \\
R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \ \mathbf{or} \ D[t_i]) & \\
 & & \quad \mathbf{and\ shl}(R^{l-1}_{i-1} \ \mathbf{and} \ R^{l-1}_i) \ \mathbf{and} \ R^{l-1}_{i-1} & \\
 & & \quad \mathbf{and} \ (R^0_{i-1} \ \mathbf{or\ shr}(\mathbf{not} D[t_i])), & 0 < i \leq n, 0 < l \leq k
\end{array}
\tag{12}
$$

NFA for the approximate sequence matching using the generalized Levenshtein distance is shown in Figure 9 and the formula for the Shift-Or algorithm is as follows:

$$
\begin{array}{lcll}
r^l_{j,0} & := & 0, & 0 < j \leq l, 0 < l \leq k \\
r^l_{j,0} & := & 1, & l < j \leq m, 0 \leq l \leq k \\
R^0_i & := & \mathbf{shl}(R^0_{i-1}) \ \mathbf{or} \ D[t_i], & 0 < i \leq n \\
R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \ \mathbf{or} \ D[t_i]) & \\
 & & \quad \mathbf{and\ shl}(R^{l-1}_{i-1} \ \mathbf{and} \ R^{l-1}_i \ \mathbf{and} \ (S^{l-1}_{i-1} \ \mathbf{or} \ D[t_i])) & \\
 & & \quad \mathbf{and} \ R^{l-1}_{i-1} \ \mathbf{and} \ (R^0_{i-1} \ \mathbf{or\ shr}(\mathbf{not} D[t_i])), & 0 < i \leq n, 0 < l \leq k \\
s^l_{j,0} & := & 1, & 0 < j \leq m, 0 \leq l < k \\
S^l_i & := & \mathbf{shl}(R^l_{i-1}) \ \mathbf{or\ shr}(D[t_i]), & 0 < i < n, 0 \leq l < k
\end{array}
\tag{13}
$$

7 Reduced Finite Automata

There are some situations in which we want to know all occurrences of pattern P in text T with at most k errors but we are not interested in knowing the number of errors in the found strings. In such cases the states that are needed only to recognize how many errors are in the found strings can be removed from *NFA*s. They are states of depth j and level l such that $j > m-k+l$. The number of states of reduced *NFA* for the approximate string matching using the Hamming distance as well as for the approximate string matching using the Levenshtein distance is $(m+1-k)(k+1)$.

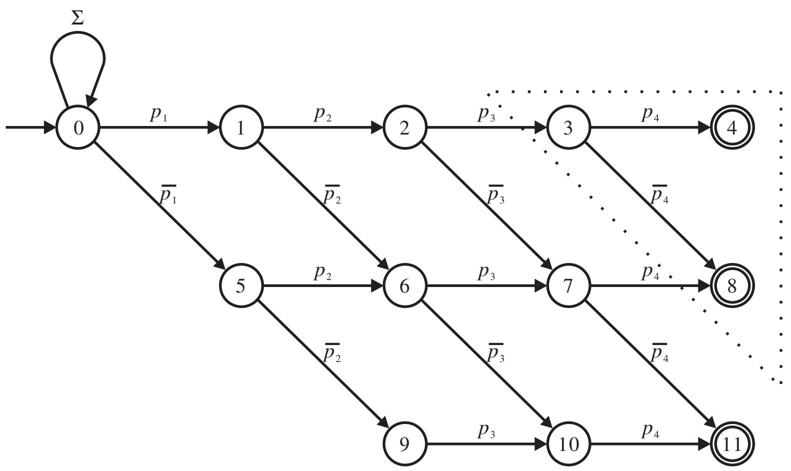

Figure 10: *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

In the approximate string matching using the Hamming distance, the states that can be removed are in *NFA* in Figure 10 bordered by the dotted line. If we remove these states we get reduced *NFA* in which each state of level l and depth $m-k+l$ become a final state as shown in Figure 11. If such state of level l is active in step i of searching that reports that pattern P will be found with at most k mismatches at location $t_{i+k-l-m+1}\dots t_{i+k-l}$. Moreover we can say that $l \le D_H(P, t_{i+k-l-m+1}\dots t_{i+k-l}) \le k$. Of course we have to check whether $i+k-l \le n$ — ending position is not located behind the end of text T.

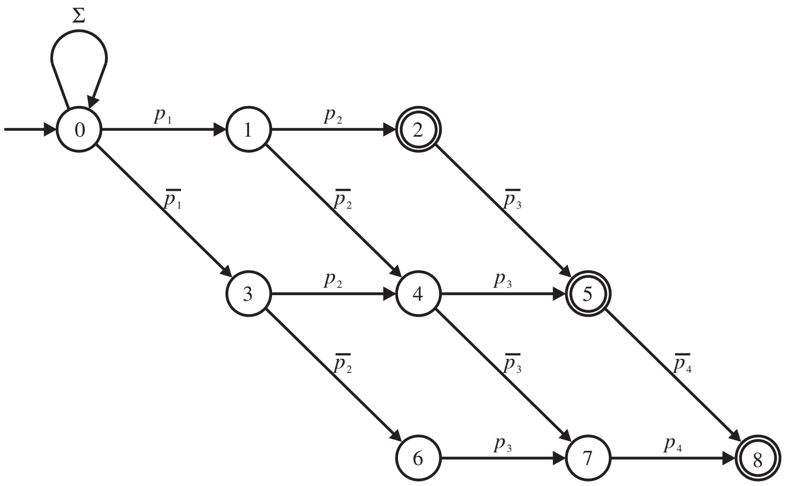

Figure 11: Reduced *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

Having the reduced *NFA* for the approximate string matching using the Hamming distance we can shorten vectors R^l in the Shift-Or algorithm. The new vectors R^l contain only $m-k$ bits. In each vector R^l we remove first l bits and last $k-l$ bits. New formula for computing vectors R^l is then as follows:

$$\begin{array}{rcll} r^l_{j,0} & := & 1, & 0 < j \le m-k, 0 \le l \le k \\ R^0_i & := & \mathbf{shl}(R^0_{i-1}) \textbf{ or } D^0[t_i], & 0 < i \le n \\ R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \textbf{ or } D^l[t_i]) \textbf{ and } R^{l-1}_{i-1}, & 0 < i \le n, 0 < l \le k \end{array} \quad (14)$$

Bit vectors $D^l[x]$ are defined as follows:

$$D^l[x] = \begin{bmatrix} d_{1-l,x} \\ d_{2-l,x} \\ \vdots \\ d_{m-k+l,x} \end{bmatrix}, \quad 0 \le l \le k, x \in \Sigma \quad (15)$$

Since first l bits of each vector R^l, $0 \le l \le k$, are removed a new problem has appeared. Because of inserting 0s at the beginning of each vector R^l by operation **shl**, state of depth $l+1$ is active every time $p_{l+1} = t_i$. It represents that there are l mismatching symbols in input text T followed by one matching symbol $p_{l+1} = t_i$. It is true except the case that $i < l+1$ — pattern cannot start before first symbol of text T. Therefore we can say that if $r^l_{m-k,i} = 0$, $0 \le l \le k$, $m-k+l \le i \le n-k+l$ then pattern P is found with at most k mismatches at position $t_{i+k-l-m+1}\dots t_{i+k-l}$.

In the approximate string matching using the Hamming distance this reduction of states shortens vectors R^l and simplifies the formula for their computing — one operation **shl** is omitted. On the other hand new operation appears and it is shift of mask vector $D[t_i]$ by l positions in order to get vector $D^l[t_i]$. This operation is not needed in the case that we accept higher space complexity and store k matrices D^l, $0 \leq l \leq k$, each shifted by l positions.

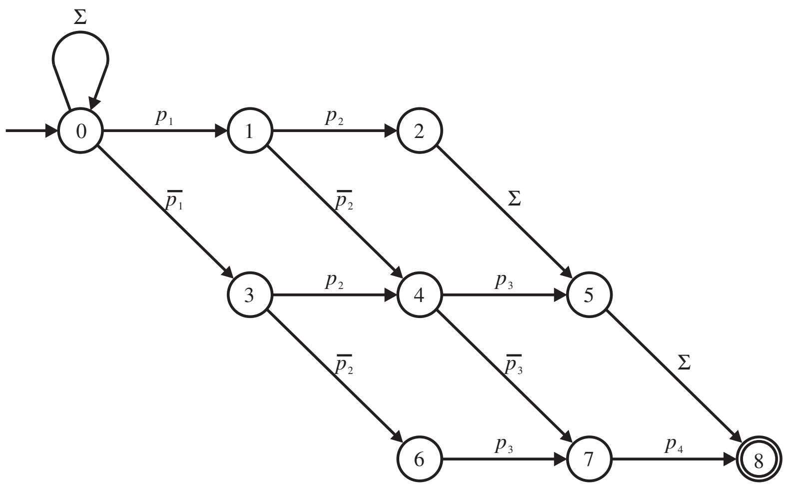

Figure 12: Reduced *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

In reduced *NFA* for the approximate string matching using the Hamming distance we do not need each state of level l and depth $m - k + l$ to become a final state. We can connect all such states by transitions labeled by Σ such that from state of level l and depth $m - k + l$ leads transition labeled by Σ to state of level $l + 1$ and depth $m - k + l + 1$, $0 \leq l < k$, as shown in Figure 12. This *NFA* has only one final state that, if it is active in step i, reports that pattern P is found in text T at position $t_{i-m+1} \dots t_i$ with at most k mismatches.

This *NFA* can be simulated by the Shift-Or algorithm that uses the same Formula (14) as for the case above because the Shift-Or algorithm labels *replace* transitions by Σ. In this case we have also the problem at the beginning of input text T as in the case above but we have not the problem at the end of text T. Therefore we can say that if $r^k_{m-k,i} = 0$, $m - k + l \leq i \leq n$ then pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$.

NFA for the approximate string matching using the Levenshtein distance can be also reduced as described above. Such reduced *NFA* for $m = 4$ and $k = 2$ is shown in Figure 13. In this case we do not need to change final states since ε-transition makes each state of level l and depth $m - k + l$ also final state. The reduced *NFA* for the approximate string matching using the Levenshtein distance can be also simulated by using the Shift-Or algorithm. The formula for this simulation is as follows:

$$
\begin{array}{lcll}
r^l_{j,0} & := & 1, & 0 < j \leq m - k, 0 \leq l \leq k \\
R^0_i & := & \mathbf{shl}(R^0_{i-1}) \textbf{ or } D^0[t_i], & 0 < i \leq n \\
R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \textbf{ or } D^l[t_i]) & \\
 & & \quad \textbf{and } R^{l-1}_{i-1} \textbf{ and } R^{l-1}_i \textbf{ and } \mathbf{shr}(R^{l-1}_{i-1}), & 0 < i \leq n, 0 < l \leq k
\end{array}
\qquad (16)
$$

Since in *NFA* for the approximate string matching using the Levenshtein distance there can be at the beginning of text T at most k deletions we have not problems with inserting 0s at the beginning of vectors R^l, $0 < l \leq k$, as it was in the case of *NFA* for the approximate string matching using the Hamming distance. Therefore if $r^k_{m-k,i} = 0$, $0 < i \leq n$, then pattern P is found with at most k differences at position $t_{i-m+1} \dots t_i$.

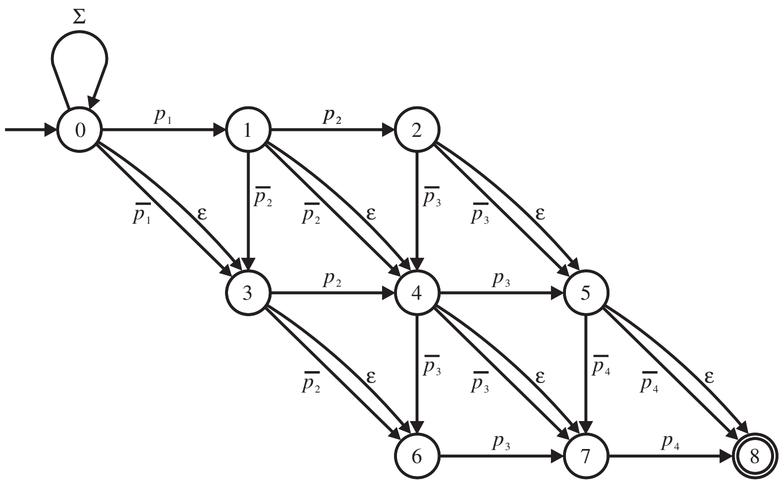

Figure 13: Reduced *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$).

In the case of the approximate string matching using the Levenshtein distance the reduction of states of *NFA* only shortens vectors R^l and does not simplify the formula for the Shift-Or algorithm.

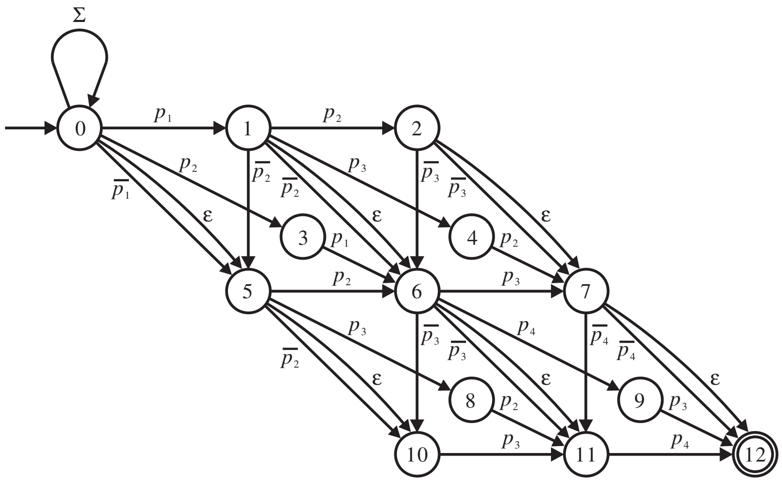

Figure 14: Reduced *NFA* for the approximate string matching using the generalized Levenshtein distance ($m = 4$, $k = 2$).

NFA for the approximate string matching using the generalized Levenshtein distance can be also reduces as in the previous two cases. The reduced *NFA* is shown in Figure 14 and the formula for simulation by the Shift-Or algorithms is as follows:

$$
\begin{array}{lcll}
r_{j,0}^l & := & 1, & 0 < j \le m-k, 0 \le l \le k \\
R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \ \mathbf{or}\ D^0[t_i], & 0 < i \le n \\
R_i^l & := & (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or}\ D^l[t_i]) & \\
 & & \quad \mathbf{and}\ R_{i-1}^{l-1} \ \mathbf{and}\ R_i^{l-1} \ \mathbf{and}\ \mathbf{shr}(R_{i-1}^{l-1}) & \\
 & & \quad \mathbf{and}\ (S_{i-1}^{l-1} \ \mathbf{or}\ D^{l-1}[t_i]), & 0 < i \le n, 0 < l \le k \\
s_{j,0}^l & := & 1, & 0 < j \le m-k, 0 \le l \le k \\
S_i^l & := & \mathbf{shl}(R_{i-1}^l) \ \mathbf{or}\ D^{l+1}[t_i], & 0 < i < n, 0 \le l < k
\end{array}
\tag{17}
$$

The reduction of states of *NFA* for the approximate string matching can be also applied to *NFA* for the approximate sequence matching.

8 Conclusions and Future Work

In this report we have presented nondeterministic finite automata (*NFA*s) for the exact and approximate string matching and for the exact and approximate sequence matching. *NFA* for approximate string matching using the Hamming distance and the initial version of *NFA* for approximate string matching using the Levenshtein distance were published in [Dar92, Mel95, Mel96b]. We have constructed the others and some of them have been published in [Hol96b, Hol97, MH97].

Then we have presented the description of the dynamic programming that was for the approximate string matching using the Hamming distance published in [Sel80, Ukk85]. We have extended this algorithm to the exact string matching and to the approximate string matching using the Hamming distance and shown how the dynamic programming simulates *NFA*s for the exact and approximate string matching.

We have also presented the Shift-Or based algorithms for the exact and approximate string matching. The Shift-Or algorithm for the exact string matching was presented in [BYG92] and the Shift-And algorithm for the approximate string matching using the Levenshtein distance was presented in [WM92]. We have shown how the Shift-Or based algorithms simulates *NFA*s for the exact and approximate string matching. Having this knowledge we have modified the Shift-Or algorithm to provide the approximate string matching using the generalized Levenshtein distance and the exact and approximate sequence matching.

For the case that we are not interested in the number of errors in the found string, we have reduced *NFA*s for the approximate string matching and shown the impact of this reduction on the Shift-Or algorithm.

Our future plans are to extend presented methods to other pattern matching problems and to try formalize general methods of the simulation of *NFA* based on the Shift-Or based algorithms and on the dynamic programming.

9 Acknowledgement

The author is very grateful to prof. Bořivoj Melichar for his comments on earlier versions of this report and some stimulating remarks.

10 References

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[Dar92] T. Darabont. Approximate string matching with k mismatches. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 1992.

[Hol96a] J. Holub. Approximate string matching in text. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, February 1996.

[Hol96b] J. Holub. Reduced nondeterministic finite automata for approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 19–27, Prague, Czech Republic, 1996. Collaborative Report DC-96-10.

[Hol97] J. Holub. Simulation of NFA in approximate string and sequence matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations.* Addison-Wesley, Reading, MA, 1979.

[Koz97] D. C. Kozen. *Automata and Computability.* Springer-Verlag, Berlin, 1997.

[Mel95] B. Melichar. Approximate string matching by finite automata. In V. Hlaváč and R. Šára, editors, *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, pages 342–349. Springer-Verlag, Berlin, 1995.

[Mel96a] B. Melichar. Space complexity of linear time approxiamte string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 28–36, Prague, Czech Republic, 1996. Collaborative Report DC-96-10.

[Mel96b] B. Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., pages 256–260, Vienna, Austria, 1996. IEEE Computer Society Press.

[MH97] B. Melichar and J. Holub. 6D classification of pattern matching problems. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 24–32, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[Sel80] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[Ukk85] E. Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1–3):100–118, 1985.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.