Jan Holub

Dynamic programming for reduced NFAs for approximate string and sequence matching

Kybernetika, Vol. 38 (2002), No. 1, [81]--90

Persistent URL: http://dml.cz/dmlcz/135447

KYBERNETIKA — VOLUME 38 (2002), NUMBER 1, PAGES 81 - 90

DYNAMIC PROGRAMMING FOR REDUCED NFAs FOR APPROXIMATE STRING AND SEQUENCE MATCHING[1]

JAN HOLUB

Approximate string and sequence matching is a problem of searching for all occurrences of a pattern (string or sequence) in some text, where the pattern can occur with some limited number of errors given by edit distance. Several methods were designed for the approximate string matching that simulate nondeterministic finite automata (*NFA*) constructed for this problem. This paper presents reduced *NFAs* for the approximate string matching usable in case, when we are interested only in occurrences having edit distance less than or equal to a given integer, but we are not interested in exact edit distance of each found occurrence. Then an algorithm based on the dynamic programming that simulates these reduced *NFAs* is presented. It is also presented how to use this algorithm for the approximate sequence matching.

1. INTRODUCTION

The task of approximate string and sequence matching is to search for all occurrences of a pattern P (string or sequence) in some text T, where the pattern can occur with some limited number of errors given by edit distance. In this paper the Levenshtein and generalized Levenshtein distances are considered. Several methods [1, 2, 6, 7, 8] were designed for the approximate string matching. These methods simulate nondeterministic finite automata (*NFA*) constructed for this problem as discovered in [3, 5].

This paper presents reduced *NFAs* for the approximate string matching usable in case, when we are interested only in occurrences having edit distance less than or equal to a given integer, but we are not interested in exact edit distance of each found occurrence. Then we present an algorithm based on the dynamic programming that simulates these reduced *NFAs*. We also present how to use this algorithm for the approximate sequence matching.

Given a string $T = t_1 t_2 \dots t_n$ over alphabet Σ, a pattern $P = p_1 p_2 \dots p_m$ over alphabet Σ, and an integer k, $k \leq m \leq n$. The approximate string matching is defined as a searching for all occurrences of pattern P in text T such that edit distance $D(P, X)$ between pattern P and string $X = t_i t_{i+1} \dots t_j$, $0 < i \leq j \leq n$,

[1]This research was partially supported by Grant 201/98/1155 of the Grant Agency of the Czech Republic and by Internal Grant 3098098/336 of the Czech Technical University.

found in the text is less than or equal to k. The approximate sequence matching is defined in the same way as the approximate string matching, but any number of symbols can be located between the occurrences of two adjacent symbols of the pattern in the text. In this paper we consider two types of distances called the Levenshtein distance and the generalized Levenshtein distance.

The Levenshtein distance $D_L(P, X)$ between strings P and X not necessarily of the same length is the minimum number of edit operations *replace* (one character is replaced by another), *insert* (one character is inserted), and *delete* (one character is removed) needed to convert string P to string X. The generalized Levenshtein distance $D_G(P, X)$ between strings P and X not necessarily of the same length is the minimum number of edit operations *replace*, *insert*, *delete*, and *transpose* (two adjacent characters are exchanged) needed to convert string P to string X.

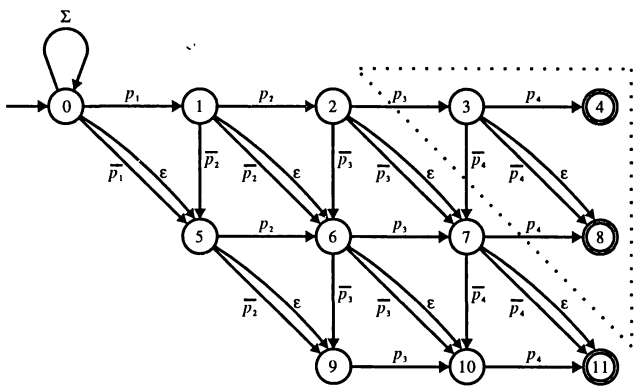

Fig. 1. *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$).

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 (Σ^* denotes the set of all strings over Σ and ε the empty string), δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. An *extended transition function* $\hat{\delta}$ is defined as $\forall q \in Q, w \in \Sigma^*, a \in \Sigma \cup \{\varepsilon\}$, $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$, $\hat{\delta}(q, \varepsilon) = q$. An *active state* of *NFA* after reading input string $w \in \Sigma^*$ is each state q such that $q \in \hat{\delta}(q_0, w)$. A *level of state* q in *NFA*, $q \in Q$, is the minimum among the numbers of errors associated with all final states reachable from q. A *depth of state* q in *NFA*, $q \in Q$, is the minimum number of transitions that are needed to get from q_0 to this state q without using ε-transitions. We say that an *algorithm* $\mathcal{A}$ *simulates a run of an NFA* M, if $\forall w, w \in \Sigma^*$, it holds that $\mathcal{A}$ with given w at the input reports all information associated with each final state q_f, $q_f \in F$, after processing w, if and only if $q_f \in \hat{\delta}(q_0, w)$.

The *NFA* for the approximate string matching using the Levenshtein distance has been presented in [3, 5]. In the *NFA* there is for each edit distance l, $0 \leq l \leq k$, one

level[2] of states. An example of such *NFA* for $m = 4$ and $k = 2$ is shown in Figure 1[3].

When string $p_1p_2p_3p_4$ is being read, the states of level 0 are active according to the found prefix – the horizontal transitions representing *match* are used. When the last symbol p_4 is read from the input, the final state q_4 is active, which reports "no error in the found string". One can get to other level of states only using vertical (*insert*) or diagonal (*delete, replace*) transition. Each of such transitions increases number of errors (edit distance). Transition *replace* (diagonal labeled by $\overline{p}$) changes position in pattern P as well as in text T. Transition *delete* (diagonal labeled by ε) changes position in P but not in T. Transition *insert* (vertical) changes position in T but not in P. The self-loop of the initial state provides that any string preceding an occurrence is omitted.

Among the algorithms for the approximate string matching there were recognized [5, 4] two methods of simulation of a run of *NFA* for the approximate string matching. The first method is called bit parallelism (Shift-Or algorithm [1] and its variations – Shift-Add [1] and Shift-And [8]). The second method is called dynamic programming [6, 7].

2. REDUCED *NFA*s

If we are interested only in all occurrences of the pattern in the text with edit distance less than or equal to k and we do not want to know the edit distance between the found string and the pattern, we can remove such states from the *NFA* for the approximate string matching that are needed only to determine the edit distance of the found string [3]. Such states are bordered by the dotted line in Figure 1. The resulting reduced *NFA* is shown in Figure 2 and has only one final state that represents that the pattern has been found with the edit distance less than or equal to k.

The modification of Shift-Or algorithm for the reduced *NFA*s was presented in [3] and the modification of the dynamic programming is discussed in the following sections.

3. DYNAMIC PROGRAMMING

The basic idea of the dynamic programming [6, 7] was to compute matrix D of size $(m \times n)$ of edit distances ($d_{j,i}$ is edit distance between prefix of P $(p_1p_2 \ldots p_j)$ and substring of T ending at position i). [2] then made some some optimization of storing and computing matrix D.

From the *NFA* simulation point of view, the dynamic programming computes in each step i of the run of the *NFA* ith column of matrix D: one element of the column is for each depth of the *NFA* and it contains the number of level of the highest active state of this depth. If there is no active state in this depth, then the element contains number of a level not existing in this depth. Since each *NFA* for the approximate string matching has $m + 1$ depths, it needs space $\mathcal{O}(m)$ and runs in time $\mathcal{O}(mn)$.

[2]In figures, the states of the same level are in the same row.

[3]Symbol $\overline{p}_j$, $0 < j \leq m$, represents $\Sigma - \{p_j\}$ in figures.

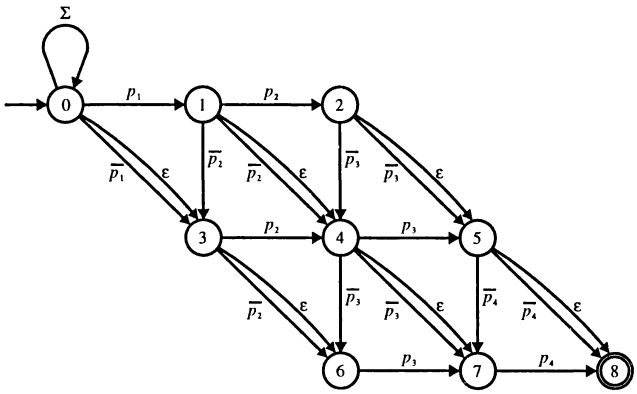

Fig. 2. Reduced *NFA* for the approximate string matching using the Levenshtein distance($m = 4$, $k = 2$).

Since the last k depths of the reduced *NFA* do not have states on all $k + 1$ levels, this method is not suitable for the reduced *NFA*s for the approximate string matching. Instead of having one element of the column for each depth of the reduced *NFA* we have one element for each diagonal of the reduced *NFA*; these diagonals are formed by the ε-transitions and are of the same length. If any state on a diagonal is active, then all states located lower on this diagonal are also active because of ε-transitions connecting them. Therefore in the element for each diagonal l, $0 \leq l \leq m-k$, we store only the number of the level of the highest active state on diagonal l. In this way we get for each step i, $0 \leq i \leq n$, of the run of the *NFA* the column $D_i = d_{0,i}, d_{1,i}, \ldots, d_{m-k,i}$ of length $m - k + 1$. Each element of the column can contain a value ranging from 0 to $k + 1$, where value $k + 1$ represents that there is no active state on the corresponding diagonal. The formula for computing columns D_i is as follows:

$$
\begin{array}{lcll}
d_{0,i} & := & 0, & 0 \leq i \leq n \\
d_{j,0} & := & k+1, & 0 < j \leq m-k \\
d_{j,i} & := & \textbf{min}(k+1, & \\
 & & \quad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, & \textit{delete \& match} \\
 & & \quad \textbf{if } p_{d_{j,i-1}+j+1} \neq t_i & \\
 & & \quad \textbf{then } d_{j,i-1}+1 & \textit{replace} \\
 & & \quad \textbf{else } k+1, & \\
 & & \quad \textbf{if } p_{d_{j+1,i-1}+j+2} \neq t_i & \\
 & & \quad \textbf{then } d_{j+1,i-1}+1 & \textit{insert} \\
 & & \quad \textbf{else } k+1), & 0 < j < m-k, 0 < i \leq n \\
d_{j,i} & := & \textbf{min}(k+1, & \\
 & & \quad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, & \textit{delete \& match} \\
 & & \quad \textbf{if } p_{d_{j,i-1}+j+1} \neq t_i & \\
 & & \quad \textbf{then } d_{j,i-1}+1 & \textit{replace} \\
 & & \quad \textbf{else } k+1), & j = m-k, 0 < i \leq n
\end{array}
\tag{1}
$$

The first line in the formula says that the initial state lying on the 0th diagonal of the *NFA* is always active because of its self-loop.

The second one says that at the beginning of the searching there is no active state on diagonals l, $0 < l \leq m - k$, because there is no initial state on such diagonals.

Term $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ represents *match* and *delete* transitions. The *match* is represented by the horizontal transitions and edit operation *delete* is represented by the diagonal ε-transitions in Figure 2. An implementation of *match* transition is simple: if the state on diagonal $j-1$ and level $d_{j-1,i-1}$ is active and horizontal transition leading from this state is labeled by symbol t_i, then the state on diagonal j and level $d_{j-1,i-1}$ becomes active. For an implementation of *delete* transition we have to search for the state on diagonal $j-1$ and level l, $d_{j-1,i-1} \leq l \leq k$, such that there is a *match* transition labeled by input symbol t_i leading from this state. In order to find such state in constant time we have to use auxiliary matrix G, in which there is for each position r in pattern P and input symbol t_i the number r', $0 \leq r'$, such that at $p_{r+r'} = t_i$, where r' is the lowest possible. If there is no such position, then $r' = k+1$. Since value of $d_{j-1,i-1}$ can be $k+1$ and the maximum number of diagonal, into which there lead match transitions, is $m-k$, the maximum position, for which a value of matrix G is required, is $m-k+k+1 = m+1$. Therefore the matrix has to be of size $(m+2) \times |\Sigma'|$, where $\Sigma' \subseteq \Sigma$ is the alphabet used in pattern P. The formula for computation of matrix G is as follows:

$$\begin{array}{lcll} g_{j,a} & := & \mathbf{min}(\{k+1\} \cup \{(l \mid p_{j+l} = a, 0 \leq l) \text{ or} & \\ & & \quad (k+1 \mid \text{if there is no such } l)\}), & 0 < j \leq m, a \in \Sigma \\ g_{m+1,a} & := & k+1, & a \in \Sigma \end{array} \tag{2}$$

Number $d_{j-1,i-1}+j$ gives the position of symbol $p_{d_{j-1,i-1}+j}$ in the pattern, which is used as a label of the *match* transition leading from the highest active state on diagonal $j-1$ to a state on diagonal j. Therefore $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ gives the level of the highest active state on diagonal j that has arisen by using *match* transition to each active state on diagonal $j-1$.

Term $d_{j,i-1}+1$ represents *replace* transition. In Figure 2, edit operation *replace* is represented by diagonal transition labeled by symbol $\overline{p}_{d_{j,i-1}+j+1}$ mismatching symbol $p_{d_{j,i-1}+j+1}$. To implement *replace* transition it is only needed to move the highest active state on diagonal j to the next lower position on the same diagonal. Since $d_{j,i-1}$ can reach $k+1$, the value of expression $d_{j,i-1}+j+1$ can be greater than m and in that case $p_{d_{j,i-1}+j+1}$ would give undefined value. To solve this problem we can add some **if** statements but it increases the time of the computation. A better solution is to put some symbols, that are not in input alphabet Σ, at positions $m+1$ and $m+2$ of the pattern – for example symbol $\langle$*end of string*$\rangle$.

Term $d_{j+1,i-1}+1$ represents *insert* transition. In Figure 2, edit operation *insert* is represented by vertical transition also labeled by mismatching symbol $\overline{p}_{d_{j+1,i-1}+j+2}$. The active state on diagonal $j+1$ and on level $d_{j+1,i-1}$ moves to level $d_{j+1,i-1}+1$ on diagonal j.

From these transitions we get minimum in order to obtain the highest active state on each diagonal. An example of matrix G for pattern $P = adbbca$ is shown

in Table 1 and the process of searching for pattern $P = adbbca$ with at most $k = 3$ errors in text $T = adcabcaabadbbca$ is shown in Table 2.

Table 1. Matrix G for pattern $P = adbbca$ and $k = 3$.

G	a	b	c	d	$\Sigma - \{a, b, c, d\}$
1	0	2	4	1	4
2	4	1	3	0	4
3	3	0	2	4	4
4	2	0	1	4	4
5	1	4	0	4	4
6	0	4	4	4	4
7	4	4	4	4	4

Table 2. Matrix D for pattern $P = adbbca$, text $T = adcabcaabadbbca$, and $k = 3$.

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	0
1	4	0	1	1	0	1	2	0	0	1	0	1	2	2	3	0
2	4	4	0	1	2	1	2	3	4	1	2	0	1	2	3	4
3	4	4	4	2	3	4	2	3	3	4	3	4	0	1	2	3

Below we also present an algorithm that uses the dynamic programming for the reduced *NFA* for the approximate string matching using the Levenshtein distance. While in Formula 1 there were evaluated the transitions incoming to the diagonals, in this algorithm there are evaluated the outgoing transitions. It simplifies the computation because there is only one test whether input symbol t_i is matching symbol. This test is necessary for deciding whether to use only *match* transition or to use *replace*, *insert*, and *delete* transitions.

At the beginning we perform the initial setting – only 0th diagonal contains active state. Then in each step i of computation we take the highest active state in each diagonal, perform transitions leading from this state with respect to the current input symbol, and according to the transitions we set the highest active states of the adjacent diagonals.

In the 0th diagonal we evaluate only transition match. Transition *replace* has no effect since the initial state is always active. In the first diagonal we evaluate transitions match, *delete*, and *replace*. Transition *insert* has no effect since it leads into the 0th diagonal. Then in each following diagonal all transitions are evaluated except the last diagonal, where there is no match transition.

Algorithm 1. DP for the reduced *NFA* for the approximate string matching using the Levenshtein distance
Input: Pattern $P = p_1 p_2 \ldots p_m$, text $T = t_1 t_2 \ldots t_n$, maximum number of differences allowed k.
Output: Matrix D of size $(m - k + 1) \times (n + 1)$.
Method:

$d_{0,0} := 0$ /* the initial settings of 0th diagonal */
for $j := 1, 2, \ldots, m - k$ **do**
  $d_{j,0} := k + 1$ /* the initial settings of other diagonals */
endfor
for $i := 1, 2, \ldots, n$ **do**
  $d_{0,i} := 0$ /* 0th diagonal ($j = 0$) */
  $d_{1,i} := g_{1,t_i}$ /* *delete & match* from the initial state *** */
  if $p_{d_{1,i-1}+2} = t_i$ **then** /* 1st diagonal ($j = 1$) */
    $d_{2,i} := d_{1,i-1}$ /* *match* */
  else
    $d_{2,i} := \mathbf{min}(g_{d_{1,i-1}+2,t_i} + d_{1,i-1}, k + 1)$ /* *delete & match* */
    $d_{1,i} := \mathbf{min}(d_{1,i-1} + 1, d_{1,i})$ /* *replace* */
  endif /* *** */
  for $j := 2, 3, \ldots, m - k - 1$ **do** /* the following diagonals */
    if $p_{d_{j,i-1}+j+1} = t_i$ **then**
      $d_{j+1,i} := d_{j,i-1}$ /* *match* */
    else
      $d_{j+1,i} := \mathbf{min}(g_{d_{j,i-1}+j+1,t_i} + d_{j,i-1}, k + 1)$ /* *delete & match* */
      $d_{j,i} := \mathbf{min}(d_{j,i-1} + 1, d_{j,i})$ /* *replace* */
      $d_{j-1,i} := \mathbf{min}(d_{j,i-1} + 1, d_{j-1,i})$ /* *insert* */
    endif /* *** */
  endfor
  $j := m - k$ /* the last diagonal */
  if $p_{d_{j,i-1}+j+1} \neq t_i$ **then**
    $d_{j,i} := \mathbf{min}(d_{j,i-1} + 1, d_{j,i})$ /* *replace* */
    $d_{j-1,i} := \mathbf{min}(d_{j,i-1} + 1, d_{j-1,i})$ /* *insert* */
  endif
  if $d_{m-k,i} < k + 1$ **then**
    write("pattern found at position i")
  endif
endfor

The first command of the second **for** cycle in the algorithm ($d_{0,i} := 0$) represents the self-loop of the initial state – the highest active state on 0th diagonal is always on level 0 and this is the initial state.

The second command ($d_{1,i} := g_{1,t_i}$) represents the only transition that leads from 0th diagonal, which is *match* transition. g_{1,t_i} gives the position l of the pattern, on which symbol t_i is located, or $k+1$, if t_i is not in the pattern. If $l < k+1$, then this position l is equal to the level of the 1st diagonal, in which there is the active state that arose by using *match* transition for t_i going from 0th diagonal.

The first **if** statement represents transitions leading from the highest active state of the first diagonal. In this case we do not evaluate *insert* transitions because they always lead to 0th diagonal, where the initial state is always active. If input symbol t_i is the same as the symbol $p_{d_{j,i-1}+2}$ used as a label of *match* transition leading from the highest active state in 1st diagonal, then we evaluate only this *match* transition ($d_{2,i} := d_{1,i-1}$). If the symbols are different, then we evaluate *delete* and *replace* transitions. For *delete* transition we search for the next occurrence of input symbol t_i in the pattern behind position $d_{j,i-1} + j + 1$ (the number of the diagonal plus the number of the level gives the position in the pattern corresponding to the state on that level of that diagonal). At first we perform *delete* transition (we move the highest active state down in the diagonal) and then we perform *match* transition for input symbol t_i. For *replace* transition we move the highest active state of the diagonal to the next lower position of the same diagonal.

In the next **for** cycle the transition leading from the highest active state of the next diagonals except the last one are evaluated. It is done in the same way as described in the previous paragraph but in addition *insert* transition is evaluated. For this *insert* transition we put the level of diagonal j increased by one to the previous diagonal $j - 1$.

In the last diagonal we evaluate only *replace* and *insert* transitions because *match* transition has no diagonal, into which it could lead.

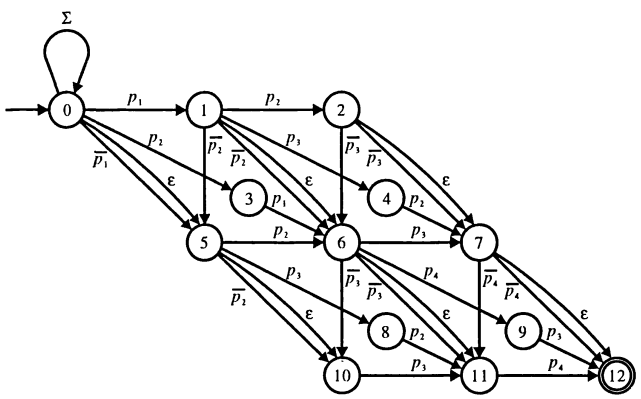

Fig. 3. Reduced *NFA* for the approximate string matching using the generalized Levenshtein distance ($m = 4$, $k = 2$).

This method can also be used for the simulation of the run of the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance. An example of such reduced *NFA* for $m = 4$ and $k = 2$ is shown in Figure 3. We have only to add term representing edit operation *transpose*. In Formula (1), the added term is as follows:

$$\begin{array}{ll} \textbf{if } p_{d_{j-1,i-2}+j+1} = t_{i-1} \textbf{ and } p_{d_{j-1,i-2}+j} = t_i & \\ \textbf{then } d_{j-1,i-2} + 1 & \textit{transpose} \\ \textbf{else } k + 1, & 0 < j \leq m - k, 1 < i \leq n \end{array} \qquad (3)$$

And in Algorithm 1, the added command is as follows:

if $p_{d_{j,i-2}+j+2} = t_{i-1}$ **and** $p_{d_{j,i-2}+j+1} = t_i$ **then**
  $d_{j+1,i} := \min(d_{j,i-2} + 1, d_{j+1,i})$ /* *transpose* */
endif

This command should be inserted into each part of Algorithm 1 where $0 \leq j < m-k$ and $1 < i \leq n$. Such places are behind the lines marked by '***'.

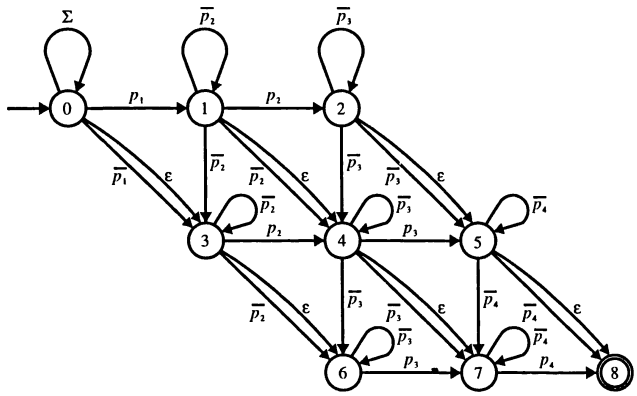

Fig. 4. Reduced *NFA* for the approximate sequence matching using the Levenshtein distance ($m = 4$, $k = 2$).

This type of simulation of the reduced *NFA*s can also be used for the reduced *NFA*s for the approximate sequence matching using the Levenshtein and generalized Levenshtein distances [4]. An example of the reduced *NFA* for the approximate sequence matching using the Levenshtein distance for $m = 4$ and $k = 2$ is shown in Figure 4.

To modify the presented algorithm so that it could simulate this reduced *NFA* we have to implement the self-loops in each non-final and non-initial state. It can be performed by inserting the following term into Formulae (1) and (1+3) for the approximate string matching.

$$\begin{array}{ll}
\textbf{if } p_{d_{j,i-1}+j+1} \neq t_i & \\
\textbf{then } d_{j,i-1} & \textit{self-loop} \\
\textbf{else } k+1, & 0 < j < m-k, 0 < i \leq n \\
\textbf{if } p_{d_{j,i-1}+j+1} \neq t_i \textbf{ and } d_{j,i-1} < k & \\
\textbf{then } d_{j,i-1} & \textit{self-loop} \\
\textbf{else } k+1, & j = m-k, 0 < i \leq n
\end{array} \tag{4}$$

The presented formulae and algorithm compute whole matrix D but in the practice only two (three for the generalized Levenshtein distance) columns from this matrix are used in each step of computation.

4. CONCLUSION

The resulting simulation runs in time $\mathcal{O}((m-k)n + m\mu)$ and needs space $\mathcal{O}(m\mu)$, where μ is the number of different symbols used in the pattern. We can decrease the space complexity by using another implementation of auxiliary matrix G but it increases the time complexity. Our algorithm also uses only one input symbol in each step of computation in case of the Levenshtein distance and two input symbols in case of the generalized Levenshtein distance.

The resulting algorithm has the time bound better than [6, 7], which runs in time $\mathcal{O}(mn)$ and for $k > \frac{m}{2}$ it has also the time bound better (not considering the preprocessing time) than [2], which runs in time $\mathcal{O}(kn + m \log \tilde{m})$, where $\tilde{m} = \min(m, |\Sigma|)$.

(Received May 12, 2000.)

REFERENCES

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

[2] Z. Galil and K. Park: An improved algorithm for approximate string matching. In: Proceedings of the 16th International Colloquium on Automata, Languages and Programming (G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., Lecture Notes in Computer Science 372), Springer–Verlag, Berlin, Stresa 1989, pp. 394–404.

[3] J. Holub: Reduced nondeterministic finite automata for approximate string matching. In: Proceedings of the Prague Stringologic Club Workshop'96 (J. Holub, ed.), Czech Technical University, Prague 1996, pp. 19–27. Collaborative Report DC–96–10.

[4] J. Holub: Simulation of NFA in approximate string and sequence matching. In: Proceedings of the Prague Stringology Club Workshop'97 (J. Holub, ed.), Czech Technical University, Prague 1997, pp. 39–46. Collaborative Report DC–97–03.

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

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

[7] E. Ukkonen: Finding approximate patterns in strings. J. Algorithms *6* (1985), 1–3, 132–137.

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

Ing. Jan Holub, Ph.D., Department of Computer Science and Engineering, Czech Technical University, Karlovo nám. 13, 121 35 Praha 2. Czech Republic.
e-mail: holub@fel.cvut.cz