Simulation of *NFA* in Approximate String and Sequence Matching[1]

Jan Holub

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

e-mail: holub@cs.felk.cvut.cz

Abstract. We present detailed description of simulation of nondeterministic finite automata (*NFA*) for approximate string matching. This simulation uses bit parallelism and used algorithm is called Shift-Or algorithm. Using knowledge of simulation of *NFA* by Shift-Or algorithm we design modification of Shift-Or algorithm for approximate string matching using generalized Levenshtein distance and modification for exact and approximate sequence matching.

Key words: finite automata, approximate string matching, simulation of nondeterministic finite automata, bitwise parallelism

1 Introduction

Approximate string matching is defined as a searching for all occurrences of pattern $P = p_1 p_2 \dots p_m$ in text $T = t_1 t_2 \dots t_n$ with at most k errors allowed. The number of errors allowed in a found substring is determined by a distance which is defined as a minimal number of edit operations needed to convert pattern P to the found substring. In the Hamming distance, the allowed edit operation is *replace* (replacing a character by another character). In the Levenshtein distance, the allowed edit operations are *replace*, *delete* (deletion of a character from the pattern) and *insert* (insertion of a character into the pattern). In the generalized Levenshtein distance, there is, besides edit operations *replace*, *delete* and *insert*, a new operation *transpose* (two adjacent characters are exchanged). This new edit operation represents situation when one types two characters in reversed order.

Sequence matching is defined as a searching for all occurrences of pattern $P = p_1 p_2 \dots p_m$ in text $T = t_1 t_2 \dots t_n$ such that between symbols p_i and p_{i+1}, $0 < i < m$, in text T can be located any number of input symbols. For approximate sequence matching we can also use Hamming, Levenshtein and generalized Levenshtein distance.

Nondeterministic finite automaton (*NFA*) is a quintuple (Q, A, δ, q_0, F), where Q is a set of states, A is a set of input symbols, δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto$ subsets of Q, q_0 is an initial state and F is a set of final states.

[1]This work was supported by grant FRVŠ 0892/97.

2 Exact String Matching

NFA for exact string matching is shown in Figure 1, where $m = 4$. Shift-Or algorithm [BG92], [WM92] for exact string matching uses one m-bit vector R in which l^{th} bit, $1 \leq l \leq m$ corresponds to l^{th} state of *NFA*. If l^{th} state is active then l^{th} bit is set to `0` and if l^{th} state is not active then l^{th} bit is set to `1`.

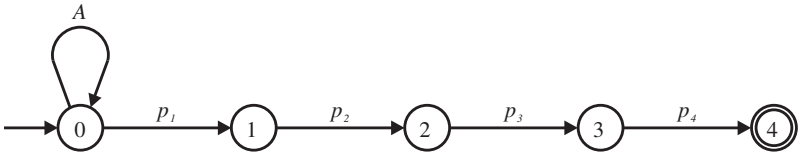

Figure 1: *NFA* for exact string matching.

In this *NFA* each nonfinal and noninitial state has transition to its right-hand neighbour. Hence transitions of all active states can be performed at once by bitwise operation `shift`. Now, it is necessary to select only those transitions that correspond to the input symbol t. It is handled by bitwise operation `or` with mask vector corresponding to the input symbol. These mask vectors are stored in mask table D. For each symbol a of input alphabet A there is one vector in which `0` is located in the same positions in which symbol a is located in the pattern P. The self-loop of the initial state is implemented by operation `shift` which inserts `0` at the beginning of the vector R. Before reading the first symbol of the input text the vector R is filled up by `1`. The formula for computing the vector when reading $(i+1)^{th}$ input symbol is (1)[2]. Shift-Or algorithm reports "pattern found" when m^{th} bit of the vector is set to `0`.

$$R_{i+1} \quad = \quad (shl(R_i)\, or D[t_{i+1}]) \tag{1}$$

Each version of Shift-Or algorithm described in this paper needs space $\mathcal{O}(\lceil \frac{m}{w} \rceil * \min(|A|, m+1))$ for mask table D, where $|A|$ denotes size of input alphabet A and w denotes length of computer word in bits. For exact string matching, space complexity of vector R is $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ and time complexity is $\mathcal{O}(\lceil \frac{m}{w} \rceil * n)$, where n is a length of the input text. Moreover at the beginning of searching we can use faster trivial searching for the first character of the pattern and when it is found then we start Shift-Or algorithm.

3 Approximate String Matching

3.1 Hamming Distance

NFA for approximate string matching using Hamming distance was shown in [Me95]. Such *NFA* for $m = 4$ and number of errors allowed $k = 3$ is shown in Figure 2 where symbol $\overline{p_i}$ represents any symbol of input alphabet A except symbol p_i. Each

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

level of states represents number of errors allowed. Operation *replace* is represented by transition that leads to the following state of the level of one more errors. This transition has the same direction for all states so we can use also operation `shift` applied to previous value of vector for one lower number of errors.

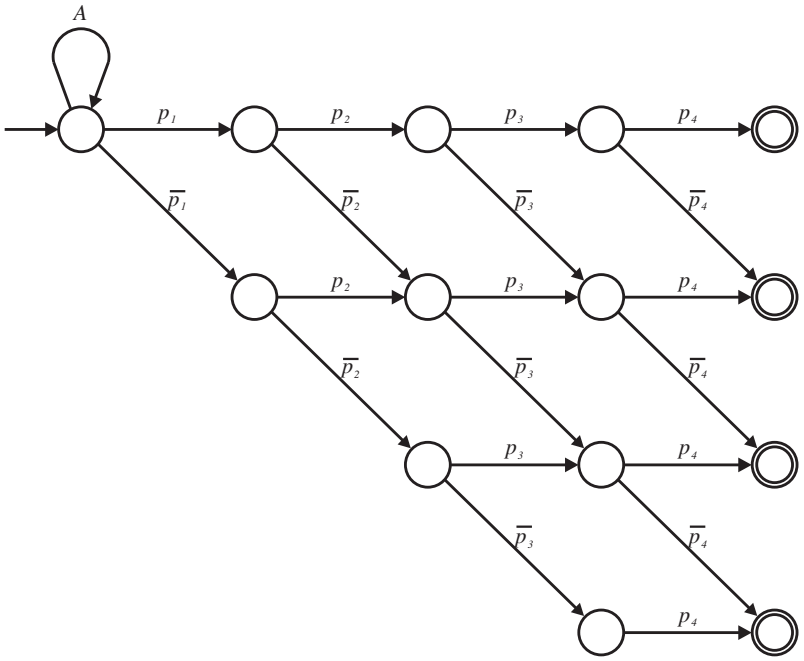

Figure 2: *NFA* for approximate string matching using Hamming distance.

Shift-Or algorithm uses for each level j, $0 \leq j \leq k$, of states one vector R^j. Vector R^0 is computed using formula (1). Vectors R^j, $j > 0$, allowing errors have to be computed with respect to transitions representing edit operation *replace*. They are computed using formula (2).

$$R^j_{i+1} \quad = \quad (shl(R^j_i)\, or\, D[t_{i+1}])\, and\, (shl(R^{j-1}_i) \tag{2}$$

This formula does not correspond exactly to the *NFA* because transition representing edit operation *replace* is performed for both unmatching and matching symbols. Vector of states in previous level is only shifted but it should be also masked by negation of $D[t_{i+1}]$ in order to select only the transitions for unmatching symbols. Since we always search for minimal number of errors this simplification does not influence result.

In this case vectors R^j need space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * (k+1))$.

3.2 Levenshtein Distance

NFA for approximate string matching using Levenshtein distance was shown in [Me96-1] and reduced in [Ho96]. Such *NFA* for $m = 4$ and number of errors allowed $k = 3$ is shown in Figure 3. There we can see edit operation *insert* which is represented by vertical transition — unmatching character inserted into the pattern. In Shift-Or

algorithm, this transition is implemented by adding previous value of the vector for one lower number of errors. It is located at the end of formula (3) for computing the vector. Like for edit operation *replace*, this transition is also made for both unmatching and matching symbols but it also does not influence the result.

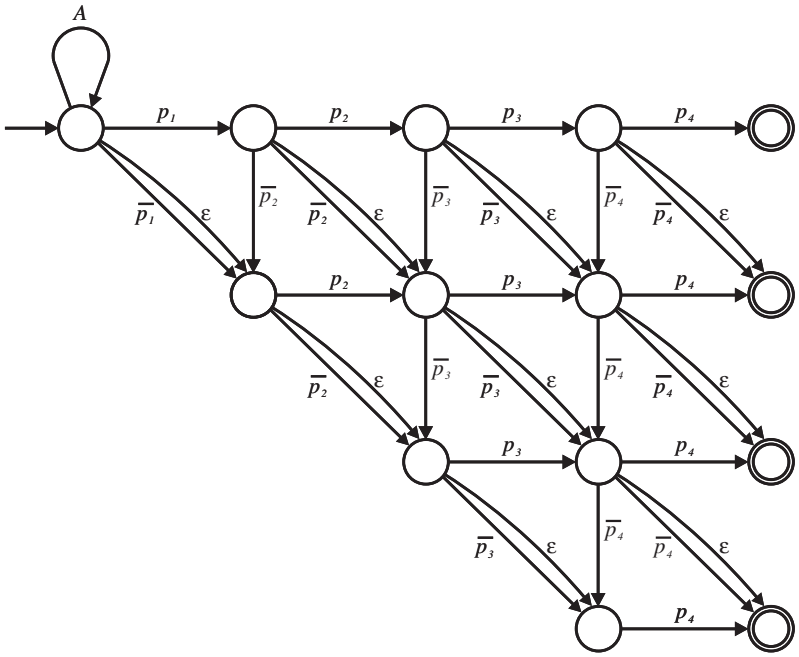

Figure 3: *NFA* for approximate string matching using Levenshtein distance.

$$R_{i+1}^{j} \quad = \quad (shl(R_i^j)\, or D[t_{i+1}])\, and\, (shl(R_i^{j-1}\, and\, R_{i+1}^{j-1}))\, and\, (R_i^{j-1}) \tag{3}$$

In the *NFA* the edit operation *delete* is represented by ε-transition — any symbol deleted from the pattern. In Shift-Or algorithm, matching transitions are made for all active states and then resulting active states are moved to their right neighbours in the level for one higher number of errors. In formula (3) it is implemented by $shl(R_{i+1}^{j-1})$.

In this case vectors R^j need space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * (k+1))$.

3.3 Generalized Levenshtein Distance

In approximate string matching using generalized Levenshtein distance we have new edit operation *transpose* that represents the situation when two adjacent symbols $p_i p_{i+1}$, $0 < i < m$, of the pattern P are placed in the found string in reversed order $p_{i+1} p_i$. In case of this edit operation we read two characters therefore this edit operation has to be represented by auxiliary state such that transition labeled by p_{i+1} leads to this state and transition labeled by p_i leads from this state. *NFA* for approximate string matching using generalized Levenshtein distance for $m = 4$ and $k = 3$ is shown in Fig. 4.

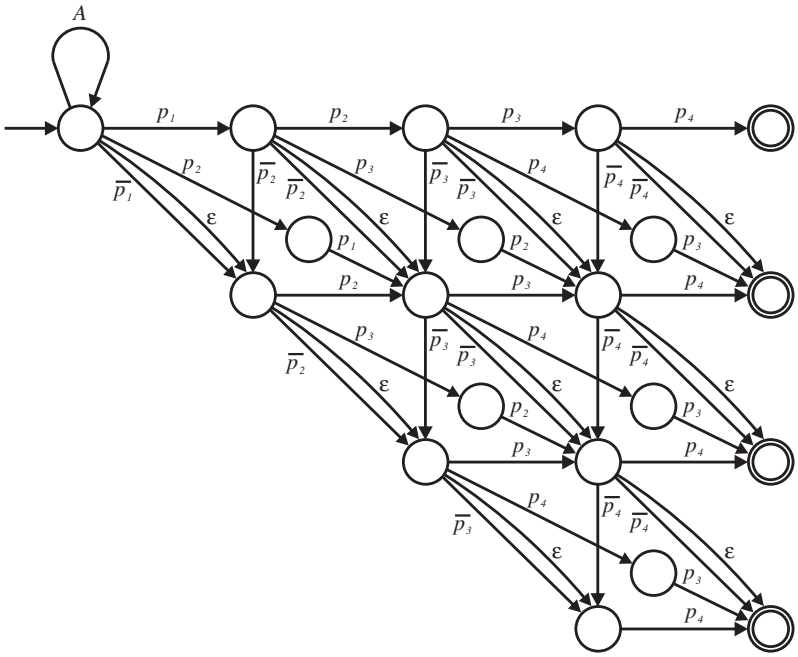

Figure 4: *NFA* for approximate string matching using generalized Levenshtein distance.

In Shift-Or algorithm we have to introduce new vectors S_j, $0 \leq j < k$, for auxiliary states. For each state q_i, whose right-hand neighbour q_{i+1} is not final state, holds that transition leading from this state to an auxiliary state q_{s_i} is labeled by matching symbol of state q_{i+1} and transition leading from the auxiliary state q_{s_i} is labeled by matching symbol of state q_i.

$$\begin{aligned} R_{i+1}^j &= (shl(R_i^j)\, or D[t_{i+1}])\, and\, (shl(R_i^{j-1}\, and\, R_{i+1}^{j-1}))\, and\, (R_i^{j-1}) \\ &\quad and\, (shl(S_i^{j-1}\, or D[t_{i+1}])) \qquad (4) \\ S_{i+1}^j &= shl(R_i^j)\, or (shr(D[t_{i+1}])) \qquad (5) \end{aligned}$$

Vectors R^j, $0 < j \leq k$, are computed by using formula (4) and vectors S^j, $0 \leq j < k$, are computed by using formula (5). At the beginning vectors S^j are filled up by `1`.

In this case vectors R^j and S^j both together need space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (2k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * (2k+1))$.

4 Sequence Matching

In previous sections we have shown simulation of *NFA*s for exact and approximate string matching. Since *NFA*s for exact and approximate sequence matching are very similar to corresponding *NFA*s for string matching we can use Shift-Or algorithm for sequence matching as well.

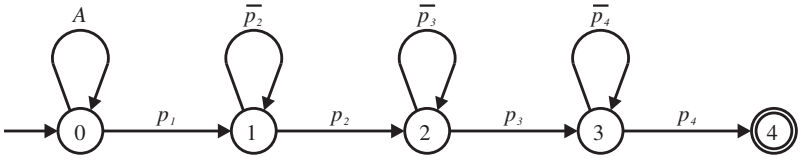

Figure 5: *NFA* for exact sequence matching.

NFA for sequence matching can be constructed from corresponding *NFA* for string matching by adding self-loop for mismatching symbols into all noninitial and nonfinal states. Such *NFA* for $m = 4$ is shown in Fig. 5. When constructing formula for computing vector R we use fromula for exact string matching (1) to which we have to add the part which represents self-loops for mismatching symbols. Self-loop expresses that a state is active even in the next step if mismatching symbol appears in the input. We implement it by adding previous value of vector R which is masked by negation of $D[t_{i+1}]$. The resulting formula is (6). This formula does not exactly correspond to *NFA* because it gives self-loop also into final state. Therefore when any final state reports "pattern found" the bit corresponding to this final state has to be set to `1`. It is faster than reseting the bit in formula (6) which is performed for each input symbol t_i.

$$R_{i+1} \quad = \quad (shl(R_i)\, or D[t_{i+1}])\, and\, (R_i\ or\ shr(not\ D[t_{i+1}])) \tag{6}$$

NFA for approximate sequence matching using Hamming distance is shown in Fig. 6 and formula for computing vector R is (7), for Levenshtein distance it is shown in Fig. 7 and formula is (9) and for generalized Levenshtein distance it is shown in Fig. 8 and formulae are (10) and (11).

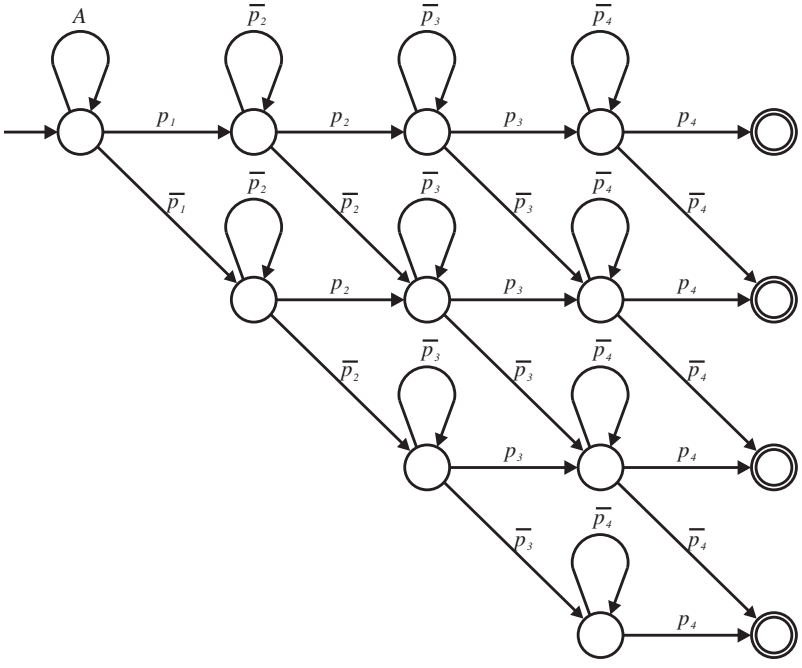

Figure 6: *NFA* for approximate sequence matching using Hamming distance.

$$R^j_{i+1} = (shl(R^j_i)\ or D[t_{i+1}])\ and\ (shl(R^{j-1}_i))\ and\ (R_i\ or\ shr(not\ D[t_{i+1}])) \quad (7)$$

$$R^j_{i+1} = (shl(R^j_i)\ or D[t_{i+1}])\ and\ (shl(R^{j-1}_i\ and\ R^{j-1}_{i+1}))\ and\ (R^{j-1}_i) \quad (8)$$

$$and\ (R_i\ or\ shr(not\ D[t_{i+1}])) \quad (9)$$

$$R^j_{i+1} = (shl(R^j_i)\ or D[t_{i+1}])\ and\ (shl(R^{j-1}_i\ and\ R^{j-1}_{i+1}))\ and\ (R^{j-1}_i)$$

$$and\ (shl(S^{j-1}_i\ or D[t_{i+1}]))\ and\ (R_i\ or\ shr(not\ D[t_{i+1}])) \quad (10)$$

$$S^j_{i+1} = shl(R^j_i)\ or (shr(D[t_{i+1}))\ and\ (S_i\ or\ shr(shr(not\ D[t_{i+1}]))) \quad (11)$$

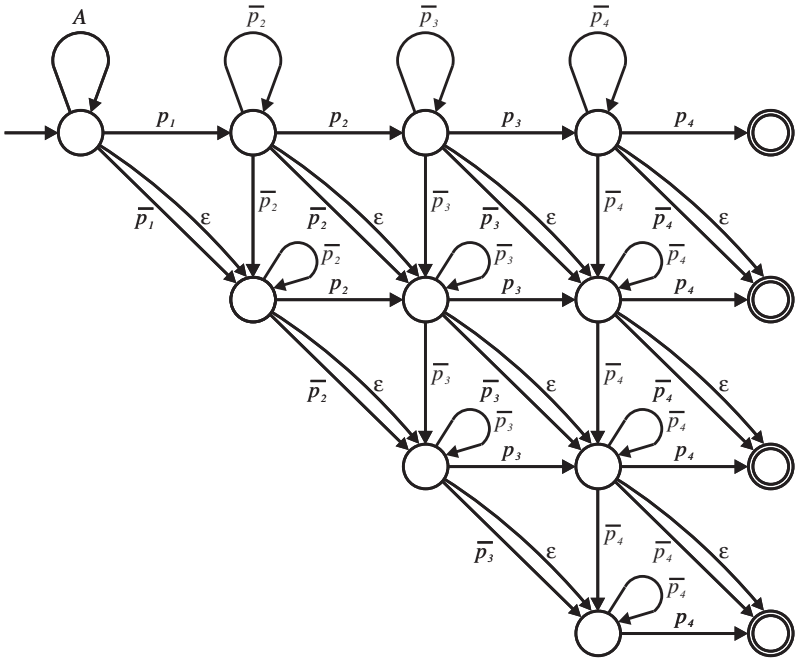

Figure 7: *NFA* for approximate sequence matching using Levenshtein distance.

Conclusions

We have presented detailed description of Shift-Or algorithm that runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * k)$ and needs space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (2k+1))$. Besides simulation of *NFA*s we can transform them to corresponding deterministic finite automata (*DFA*s) that run in time $\mathcal{O}(n)$. In case of exact string matching the number of states of *DFA* for pattern P is the same as the number of states of *NFA* for pattern P so except for very short text it is faster to use *DFA*.

In case of approximate string and sequence matching the number of states of *DFA* unfortunately seems to be exponential but exact bounds have not been determined. Some estimations were presented in [Me96-2] but they also seem to be pessimistic. In case that we do not want to know the number of errors in the found string we can reduce *NFA*, shorten vectors R^j and simplify formulae as shown in [Ho96].

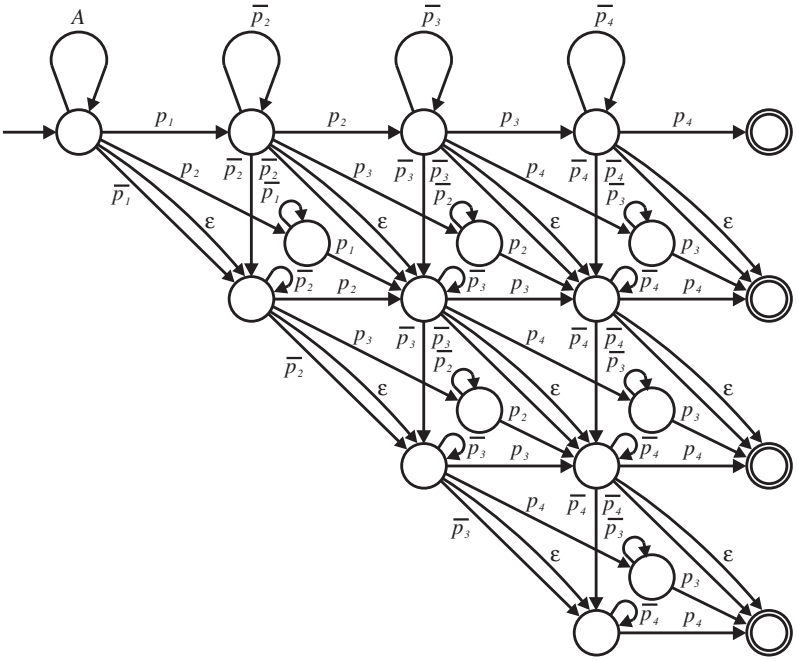

Figure 8: *NFA* for approximate sequence matching using generalized Levenshtein distance.

When searching it is necessary to consider length of the input text, length of the pattern and memory space available in order to choose optimal searching method — *DFA* or simulation of *NFA*.

References

[BG92] Baeza-Yates, R., Gonnet, G.H.: A New Approach to Text Searching. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 74–82.

[Ho96] Holub, J.: Reduced Nondeterministic Finite Automata for Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University, August 1996, pp. 19–27.

[Me95] Melichar, B.: Approximate String Matching by Finite Automata. Computer Analysis of Images and Patterns, LNCS 970, Springer-Verlag, Berlin 1995, pp. 342–349.

[Me96-1] Melichar, B.: String Matching with k Differences by Finite Automata. Proceedings of the 13^{th} ICPR, Vol. II, August 1996, pp. 256–260.

[Me96-2] Melichar, B.: Space Complexity of Linear Time Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University, August 1996, pp. 28–36.

[WM92] Wu, S., Manber, U.: Fast Text Searching Allowing Errors. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 83–91.