Two algorithms for approximate string matching in static texts *
(Extended Abstract)

Petteri Jokinen Esko Ukkonen

Department of Computer Science, University of Helsinki
Teollisuuskatu 23, SF-00510 Helsinki, Finland

Abstract. The problem of finding all approximate occurrences P' of a pattern string P in a text string T such that the edit distance between P and P' is $\leq k$ is considered. We concentrate on a scheme in which T is first preprocessed to make the subsequent searches with different P fast. Two preprocessing methods and the corresponding search algorithms are described. The first is based suffix automata and is applicable for edit distances with general edit operation costs. The second is a special design for unit cost edit distance and is based on q-gram lists. The preprocessing needs in both cases time and space $O(|T|)$. The search algorithms run in the worst case in time $O(|P||T|)$ or $O(k|T|)$, and in the best case in time $O(|P|)$.

Introduction

The *approximate string matching* problem is to find, given a pattern string P and a text string T, the approximate occurrences of P in T. Typically one wants to find all occurrences that are good enough in some measure of the approximation quality.

There are several situations where it is necessary to allow for approximate matches instead of exact ones. Some natural variation in the occurrences of P (e.g. due to morphological variation of the same base word in natural languages) sometimes takes place. In other cases, P or T or both may have been slightly distorted through noisy communication channels or through different types of errors (measurement error, typing error).

We concentrate on the important special case where T stays unchanged for searches with numerous different P, and we have the whole T available before the searches. Such a static T can first be preprocessed into a suitable form (an *index* for approximate searches) that makes the subsequent searches faster. Hence we want to find a preprocessing of T and the associated algorithm to search for approximate occurrences of P using the preprocessed T.

The edit distance will be used as the measure for the approximation quality.

Definition. Let P and P' be strings in alphabet Σ. The *edit distance* form P to P' is the minimum possible total cost of a sequence of editing steps that convert P' to P. Each

*Research supported by the Academy of Finland and by the Alexander von Humboldt Foundation (Germany). The work of the second author was in part carried out when visiting Institut fuer Informatik, University of Freiburg, Germany.

editing step is a rewriting step of the form $a \rightarrow \epsilon$ (a deletion), $\epsilon \rightarrow a$ (an insertion), or $a \rightarrow b$ (a change), where a, b in Σ are any symbols, $a \neq b$, and ϵ is the empty string. Each editing operation $x \rightarrow y$ has a cost $c(x \rightarrow y) > 0$. In the conversion from P to P' rewriting of each symbol is allowed only at most once; this makes it possible to use dynamic programming algorithms for edit distances. The special case where $c(x \rightarrow y) = 1$ for all edit operations $x \rightarrow y$ is called the *unit cost model* of the edit distance.

Definition (approximate string matching problem). Given two strings, *text* $T = t_1t_2 \ldots t_n$ and *pattern* $P = p_1p_2 \ldots p_m$ in alphabet Σ, and a threshold value $k \geq 0$, find the end locations j of all substrings P' of T such that the edit distance from P to P' is at most k. If the unit cost model is used, the problem is called the *k differences problem.*

The on-line version of the problem in which no preprocessing of T is allowed has recently received lot of attention [5, 6]. Standard solution is by dynamic programming in time $O(mn)$. For the k differences problem fast special methods are possible, including $O(kn)$ time algorithms [10, 7, 16, 14, 2].

In the case of *exact string matching* ($k = 0$) preprocessing of T leads to optimal time searches. If T is preprocessed into a suffix tree [17, 12] or into a suffix automaton [1, 3], the queries of P can be accomplished in time $O(m + \text{size of output})$. If the suffix array [11] is used, the search time becomes $O(m + \log n + \text{size of output})$.

In the case of the approximate matching ($k \geq 0$) we develop in this paper two data structures for representing a static T and give the corresponding search algorithms. The first solution combines suffix automata and dynamic programming, and is applicable for general edit operation costs c. Text T is represented as annotated suffix automaton. The search is performed by dynamic programming over P and the transition graph of the automaton. Based on certain properties of suffix automata we develop a search strategy that avoids entering the same state of the automaton repeatedly. This gives a time bound that is in the worst case the same as for the standard on-line dynamic programming but in the best case is essentially better.

The second data structure is a simple special design for the k differences problem. The structure is based on the so-called q-grams that are simply any strings of q symbols. The preprocessing phase creates for each q-gram of T a chain that links together all occurrences of the q-gram in T. This structure can be understood as an abridged version of the suffix array or—when the headers of the link chains are organized as a trie—as a suffix tree which has been cut to the depth q.

The search phase marks the areas of T that have a sufficient number of q-grams in common with P. The marked areas are then checked by dynamic programming for occurrences of P with at most k differences. The method has a predecessor in the work of Owolabi & McGregor [13], and related 'signature' methods have been used e.g. in spelling correction; see e.g. [9]. We show how the different parameters of the method should be chosen to solve the given k differences problem.

Annotated suffix automaton $SA(T)$

The *suffix automaton* [3, 4] (also known as *DAWG, directed acyclic word graph,* [1]) for a string $T = t_1t_2 \cdots t_n$ is the smallest DFA recognizing all the suffixes $T_i = t_i \cdots t_n$, $1 \leq i \leq n + 1$, of T. We let *root* denote its initial state and *goto* its transition function;

there is a transition from state s to state r on input symbol a if $r = goto(root, a)$. The suffix automaton can be constructed in time $O(n)$ by the methods given in [3, 4, 1]. The suffix automaton for T has at most $3n - 4$ *goto* transitions and at most $2n - 1$ states. It can be viewed as the *suffix tree* for T, with the identical subtrees merged. As a graph, it is a dag.

The *depth* of a state s of the automaton, denoted $depth(s)$, is the length of the (unique) longest string x such that there is a *goto* path from *root* to s, $goto(root, x) = s$; here we have extended the *goto* function for strings in the obvious way. Similarly, $mindepth(s)$ denotes the length of the shortest string y such that $goto(root, y) = s$.

The following property of a suffix automaton is an immediate consequence of the fact that the automaton accepts all the suffixes of a string and nothing more.

Lemma 1 *For a state s, let x be the longest string such that $goto(root, x) = s$. Then the set of strings y such that $goto(root, y) = s$ consists of all suffixes of x of length at least $mindepth(s)$.*

The important *fail* function on the states of the automaton has the following characterization.

Lemma 2 ([4]) *Let $s = goto(root, x)$ for some string x, and let w be the longest suffix of x such that $s \neq goto(root, w)$. Then, $goto(root, w) = fail(s)$ and $|w| = depth(fail(s))$.*

Corollary 1 $mindepth(s) = depth(fail(s)) + 1$.

The suffix automaton serves an an index giving the locations of different substrings of T. There are different ways to attach the location information to the automaton. For our purposes the following is suitable.

The states of the automaton are divided into two classes: the *primary states* and the *secondary states*. The primary states are the states $s_i = goto(root, t_1 \cdots t_i)$ for $0 \leq i \leq n$. These states are disjoint, and $depth(s_i) = i$. The other states are secondary.

A string x is said to *occur at location j* in T if $x = t_{j-|x|+1}t_{j-|x|+2} \cdots t_j$.

Lemma 3 *Let $goto(root, x) = s$, and let $L = \{depth(r) \mid r$ is primary and $s = fail^i(r)$ for some $i \geq 0\}$. Then L is the set of all locations at which x occurs in T.*

Hence the occurrences of a string leading to s can be found by finding the primary states from which there is a $fail$ transition path to s. Therefore we also need the inverse of $fail$: with each state r we attach a list of links, the *cofail* links, pointing to states r' such that $fail(r') = r$.

The *annotated suffix automaton* for T, denoted $SA(T)$, is the suffix automaton of T (i.e., the states and the *goto* function) with the states marked primary or secondary and with the $fail$ and *cofail* links and the *depth* value for each state. The annotations do not increase the construction time; in fact, both $fail$ and *depth* are needed in the construction, so the only extra work is to reverse $fail$ and mark states primary or secondary which clearly does not increase the asymptotic time requirement.

Proposition 1 *The annotated suffix automaton $SA(T)$ can be constructed in time and in space $O(n)$.*

Approximate string matching with $SA(T)$

The approximate string matching problem for text $T = t_1 t_2 \cdots t_n$ and pattern $P = p_1 p_2 \cdots p_m$ can be solved on-line, without preprocessing T, with the following well-known dynamic programming method.

Let D be a $m+1$ by $n+1$ table such that for $0 \le i \le m$, $0 \le j \le n$, $D(i,j)$ is the minimum edit distance from $p_1 \cdots p_i$ to the substrings of T ending at t_j. Clearly, there is an approximate occurrence of P in T, ending at t_j, with edit distance $\le k$ from P, if and only if $D(m,j) \le k$. Such indexes j can be found by evaluating D from

$$D(0,j) = 0, \quad 0 \le j \le n; \tag{1}$$

$$D(i,j) = \min \begin{cases} D(i-1,j) + c(p_i \to \epsilon) \\ D(i-1,j-1) + \text{ if } p_i = t_j \text{ then } 0 \text{ else } c(p_i \to t_j) \\ D(i,j-1) + c(\epsilon \to t_j) \end{cases} \tag{2}$$

for $1 \le i \le m$, $0 \le j \le n$.

As $D(i,j)$ depends only on entries $D(i-1,j)$, $D(i-1,j-1)$, and $D(i,j-1)$ of D, the evaluation conveniently proceeds column-by-column: Column $D(*,j)$ can be evaluated from column $D(*,j-1)$, proceeding in the order $D(0,j), \ldots, D(m,j)$. The total time is $O(mn)$.

The length $L(i,j)$ of the *shortest* suffix of $t_1 \cdots t_j$, whose edit distance from P is $D(i,j)$, can be computed together with $D(i,j)$ itself. Clearly, for $0 \le j \le n$ we have $L(0,j) = 0$, and for $1 \le i \le m$, $0 \le j \le n$:

$$\begin{aligned} L(i,j) = \; & \text{if } D(i,j) = D(i-1,j) + c(p_i \to \epsilon) \text{ then } L(i-1,j) \\ & \text{elsif } D(i,j) = D(i-1,j-1) + (\text{if } p_i = t_j \text{ then } 0 \text{ else } c(p_i \to t_j)) \\ & \qquad \text{then } L(i-1,j-1) + 1 \\ & \text{else } L(i,j-1) + 1. \end{aligned}$$

Then $D(i,j)$ equals the edit distance from $p_1 \cdots p_i$ to $t_{j'} \cdots t_j$ where $j' = j - L(i,j) + 1$.

Next we develop a method that performs a similar dynamic programming over P and $SA(T)$ to find the approximate occurrences of P in T. The method will attach with the states of $SA(T)$ similar columns of $m+1$ entries as are the columns of matrices D and L. The column representing edit distances at state r is denoted as $dcol(r)$, and the column representing the corresponding lengths is denoted as $lcol(r)$.

The method will work, roughly formulated, in the following steps.

1. Traverse the *useful subtree* $U(P,k)$ of $SA(T)$ starting from *root* and using a modified Dijkstra's shortest path algorithm to control the traversing order;

2. When the traversal enters state r along a transition $goto(s,a) = r$, evaluate $dcol(r)$ and $lcol(r)$ by dynamic programming from a, $dcol(s)$, and $lcol(s)$;

3. If $dcol(r)(m) \le k$, mark all states that can be reached from r along *cofail* links and are not already marked. Output $depth(q)$ for each primary state q that gets a mark.

Next we refine the above description of the algorithm, starting from step 2.

To understand the use of *dcol* and *lcol* some further notation is necessary. For any string x, we let $d(i,x)$ denote the minimum edit distance between $p_1 \cdots p_i$ and any suffix of x, and $l(i,x)$ denote the length of the shortest such a suffix. Then, for example, $D(i,j) = d(i, t_1 \cdots t_j)$ and $L(i,j) = l(i, t_1 \cdots t_j)$. It should be clear that d anf l can be evaluated in the same way as D and L from a recursion similar to (1) and (2); string x now takes the role of $t_1 \cdots t_j$.

The traversal over $SA(T)$ starts from *root*. Initially, $dcol(root) = d(*, \epsilon)$ and $lcol(root) = l(*, \epsilon)$, where $d(i,\epsilon) = \sum_{h=1}^{i} c(p_h \to \epsilon)$ and $l(i,\epsilon) = 0$, for $0 \le i \le m$. For other states s the columns $dcol(s)$ and $lcol(s)$ will be such that $dcol(s) = d(*, x)$ and $lcol(s) = l(*, x)$, where x is the string spelled out by the path from *root* to s in the traversed subtree $U(P,k)$. This property is preserved if, when the traversal takes transition $goto(s,a) = r$, the new columns $dcol(r) = d(*, xa)$ and $lcol(r) = l(*, xa)$ are evaluated by dynamic programming from $dcol(s)$, $lcol(s)$, and a. For example, for $dcol(r)$ this evaluation gets the form

$$dcol(r)(0) = 0 \tag{3}$$

$$dcol(r)(i) = \min \begin{cases} dcol(r)(i-1) + c(p_i \to \epsilon) \\ dcol(s)(i-1) + \text{ if } p_i = a \text{ then } 0 \text{ else } c(p_i \to a) \\ dcol(s)(i) + c(\epsilon \to a) \end{cases} \tag{4}$$

for $i = 1, 2, \ldots, m$.

Next consider step 1. Our goal is to develop a traversing order that guarantees that all approximate occurrences of P will be found but extra traversing is avoided. This should be done in such a way that each *goto* transition of $SA(T)$ is traversed at most once. As there are $O(n)$ transitions and taking a transition needs time $O(m)$ (for evaluating *dcol* and *lcol*), this would give an $O(mn)$ time bound for the whole method. It turns out that it suffices to traverse over subtree $U(P,k)$ which we shall define next.

We denote as $\lambda(x)$ the length of the longest suffix y of a string x such that the edit distance from some prefix of P to y is $\le k$. Obviously, $\lambda(x) = l(i,x)$ where $i \le m$ is the largest index such that $d(i,x) \le k$. We say that an entry $d(i,x)$ is *essential*, if $d(i,x) \le k$. Hence $\lambda(x)$ expresses the length of the part of x on which the essential part of $d(*,x)$ depends.

Let $goto(root, a_1 a_2 \cdots a_h) = s$ for some $a_i \in \Sigma$. The *goto* path $a_1 \cdots a_h$ is called *useful*, if $\lambda(a_1 \cdots a_i) \ge mindepth(goto(root, a_1 \cdots a_i))$ for all $1 \le i \le h$. State s is useful, if all *goto* paths from *root* to s are useful. In particular, *root* is useful.

Definition. The *useful subtree* $U(P,k)$ of $SA(T)$ is the subgraph of $SA(T)$ that contains all the useful states and for each such state s, it also contains the *goto* transitions on the longest useful *goto* path from *root* to s.

Useful subtree $U(P,k)$ is really a tree because every initial segment of a useful path is useful.

It is sufficient to restrict the traversal on $U(P,k)$. To prove this, we need first a lemma.

Lemma 4 *Let x be a string and y its suffix such that $|y| \ge \lambda(x)$. Then $d(*,x)$ and $d(*,y)$ are identical when restricted to the essential entries, and the correspondingly restricted $l(*,x)$ and $l(*,y)$ are identical.*

Let J be the set of all locations j that our algorithm will output (step 3) when performing dynamic programming over $U(P,k)$, and let J' be the correct set of locations we want to find, that is, $J' = \{j \mid D(m,j) \le k\}$.

Theorem 1 $J = J'$.

Finally we need an efficient way to isolate and traverse the useful subtree $U(P,k)$. This will be done by finding a slightly larger tree that consists of $U(P,k)$ and of some additional leaves.

A *goto* path $a_1 \cdots a_h$ is called a *bounding path*, if path $a_1 \cdots a_{h-1}$ is useful but path $a_1 \cdots a_h$ is not useful (that is, $\lambda(a_1 \cdots a_h) < \mathit{mindepth}(\mathit{goto}(\mathit{root}, a_1 \cdots a_h))$. A state s of $SA(T)$ is a *boundary state* if there is to s at least one bounding path but no useful path.

Definition. The *extended useful subtree* $U^+(P,k)$ of $SA(T)$ consists of $U(P,k)$ and of all boundary states of $SA(T)$ and of longest possible bounding paths to them.

Again, subgraph $U^+(P,k)$ is really a tree because the longest bounding path to each boundary state is unique, and its each initial segment is useful and longest possible and hence belongs to $U(P,k)$.

Assume for a moment that we know a priori the nodes of $U^+(P,k)$. Then its arcs can be found by Dijkstra's shortest path algorithm. We define the cost $w(s,r)$ of an arc (s,r) (i.e., $\mathit{goto}(s,a) = r$ for some a) as $w(s,r) = \mathit{depth}(r) - \mathit{depth}(s) - 1$, if s is a useful state. If s is a boundary state, then we set $w(s,r) = \infty$; hence, in effect, such arcs are removed from $SA(T)$.

Then find with Dijkstra's algorithm the minimum cost paths with respect to cost function w from *root* to all states in $U^+(P,k)$. Consider the path $s_0 = \mathit{root}, s_1, \ldots, s_{h-1}, s_h = s$ found in this way to some s.

Lemma 5 *The length h of the path to s is largest possible.*

The useful states and the boundary states are not known a priori, but we can recognize them easily during the execution of the Dijkstra's algorithm. The dynamic programming is performed at each state in the traversal order determined by the algorithm: When the algorithm reaches a new state r along transition $\mathit{goto}(s,a) = r$, columns $\mathit{dcol}(r)$ and $\mathit{lcol}(r)$ are computed from $\mathit{dcol}(s)$, $\mathit{lcol}(s)$, and a, as already explained.

Let x be the path from *root* along which r is found. Then $\lambda(x) = \mathit{lcol}(r)(i)$ where i is the largest index such that $\mathit{dcol}(r)(i)$ is essential. Hence $\lambda(x)$ can be evaluated locally at r, and we may write $\lambda(r) = \lambda(x)$.

Now the status of r can be decided.

Lemma 6 *If $\lambda(r) < \mathit{mindepth}(r)$, then r is a boundary state, otherwise r is a useful state.*

By Lemmas 5 and 6, our algorithm finds the boundary states and the useful states correctly along longest possible paths. Therefore $U^+(P,k)$ is found correctly which means, by Theorem 1, that the approximate occurrences of P are found correctly.

Theorem 2 *The described algorithm can be implemented such that it works in time $O(mn)$ in the worst case and, for the unit cost model of the edit distance, in time $O(m)$ in the best case.*

Proof. As there are $O(n)$ states in $U^+(P,k)$ and a dynamic programming step of time $O(m)$ is performed once at each, the time for dynamic programming is $O(mn)$. In Dijkstra's algorithm we use bucket sort instead of heap to get time $O(n)$, and hence total time $O(mn)$. For the best case bound consider P and T such that they do not have common symbols.

Remark 1. The algorithm of Theorem 2 satisfies the natural requirement that the worst case time $O(nm)$ is not larger than the time of the on-line solution, without preprocessing T. The best case time is $O(m)$ showing that we have achieved some progress with the preprocessing of T. Without it also the best case has to grow proportional to n.

When $k = 0$, the algoritm requires time $O(m^2)$. The time seems to grow very fast with k but we leave open a more complete analysis of this dependency.

Remark 2. The simplest way to find approximate P's from automaton $SA(T)$ would be to follow each *goto* path from the *root* until the corresponding string has an edit distance $> k$ from all prefixes of P. Such paths can have total length $O(mn)$. It can be shown that this leads to $O(mnk)$ time search.

The q-gram method

This section considers the k differences problem, that is, $c(x \to y) = 1$ for all editing operations $x \to y$.

A *q-gram* in Σ is any string in Σ^q. The usefulness of the q-grams is based on the following lemma.

Lemma 7 *Let an occurrence of P with at most k differences end at t_j in T. Then at least $m + 1 - (k+1)q$ of the $m - q + 1$ q-grams of P occur in $t_{j-m+1} \dots t_j$.*

Proof. Let P' be the approximate version of P that ends at t_j. Hence P' is a suffix of $t_{j-m+1} \cdots t_j$ or $t_{j-m+1} \cdots t_j$ is a suffix of P'. String P' is obtained from P with at most k insertions, deletions or changes. A deletion or a change at character p_i of P destroys at most q q-grams of P, namely those that contain p_i. An insertion between p_i and p_{i+1} destroys at most $q-1$ q-grams of P, namely those that contain both p_i and p_{i+1}. Hence at most $k_1 q + k_2(q-1)$ q-grams of P are missing in P', where k_1 is the total number of deletions and changes and k_2 is the total number of insertions. As $|P'| \leq m + k_2$, string $t_{j-m+1} \cdots t_j$ contains all q-grams of P' except for at most k_2. Hence at most $k_1 q + k_2(q-1) + k_2 = kq$ q-grams of P are not present in $t_{j-m+1} \cdots t_j$, which proves the lemma.

Using the lemma the areas of T that may contain a good enough approximate occurrence of P can be found fast. These are separately checked with dynamic programming.

Text T is preprocessed as follows: For each q-gram G in Σ^q we construct a list $L(G)$ consisting of all j such that T has an (exact) occurrence of G starting at t_j. The lists for all G can be created in one scan over T either by using a natural encoding of q-grams into integers to the base $|\Sigma|$ (c.f. [8]) or by using a modified suffix automaton with *fail*-transitions representing different q-grams of T [15]. We also create a search structure for finding fast the list $L(G)$, given G. A suitable stucture is an array indexed by the integer code of G, or a trie representing the different q-grams of T. The preprocessing time is $O(n + |\Sigma|^q)$ for the method based on integer codings; the size of the resulting structure is also $O(n + |\Sigma|^q)$ where n represents the total length of the lists and $|\Sigma|^q$ the search structure.

Assume then that we have to find the occurrences of P in T with $\leq k$ differences. The first phase of the search traverses all lists $L(G)$ where G occurs in P. The occurrences listed in $L(G)$'s are counted into initially zero buckets B_i, $0 \leq i \leq \lceil n/(m-1) \rceil + 1$. Bucket B_i is increased by 1 when the next element j of $L(G)$ satisfies $(i-1)(m-1)+1 \leq j \leq (i+1)(m-1)$. Hence the width of each bucket is $2(m-1)$ and two successive buckets have an overlap of length $m-1$; the overlap ensures that no occurrences of P are lost. (For simplicity, we assume that $m \geq 2$.) The rule for updating the buckets can be stated simply $B_{\lfloor \frac{j-1}{m-1} \rfloor} \leftarrow * + 1$; $B_{\lfloor \frac{j-1}{m-1} \rfloor + 1} \leftarrow * + 1$.

When B_i achieves value $m + 1 - (k+1)q$ we know by Lemma 7 that an approximate occurrence of P can end somewhere in $t_{i(m-1)} \cdots t_{(i+1)(m-1)}$. As an occurrence is of length $\leq m+k$, its leftmost possible starting character is t_j where $j = i(m-1)-m-k+1$. Hence we check by dynamic programming whether or not there is an approximate occurrence in $t_j \cdots t_{(i+1)(m-1)}$.

Because the total length of the q-gram lists $L(G)$ for G in P is $\leq n$, they can be traversed as described above in time $O(m+n)$. Under the random string assumption (each symbol in T is chosen uniformly and independently from Σ) the expected length of each list is $n/|\Sigma|^q$, hence the expected traversal time is $O(m + (m-q+1)n/|\Sigma|^q)$. In the best case each list is empty, hence time $O(m)$ suffices. Let r be the number of the buckets checked by dynamic programming. Using the $O(kn)$ version of dynamic programming [7, 16], the total time for the checking phase is $O(rkm)$ which in the worst case is $O(kn)$.

Theorem 3 *The q-gram lists for T can be constructed in time and in space $O(n + |\Sigma|^q)$. The search for occurrences of P with at most k differences can be done in time $O(m + n + rkm)$ where r is the number of buckets checked with dynamic programming. In the best case time $O(m)$ suffices for the search.*

The bound $m + 1 - (k+1)q$ for the number of q grams in Lemma 7 is non-trivial only if $q < (m+1)/(k+1)$. Hence it is possible that the q used in preprocessing T is too large for the present m and k. Fortunately, we can in this case use a smaller q' that is $< (m+1)/(k+1)$. The list $L(G)$ for a q'-gram G is the catenation of the q-gram lists $L(GX)$ where X is in $\Sigma^{q-q'}$.

Annotated suffix automaton $SA(T)$ is also a complete '$*$-gram' index for T containing q-gram lists for all $0 \leq q \leq n$. For a gram G of any length, $L(G)$ consists of all values $depth(s)$ such that s is primary and reachable from $goto(root, G)$ along *cofail*-links (Lemma 3). The q-gram method could be based on $SA(T)$ as well.

References

[1] Blumer,A., Blumer,J., Haussler, D., Ehrenfeucht, A., Chen, M.T. and Seiferas, J. (1985): The smallest automaton recognizing the subwords of a text. *Theor. Comp. Sci. 40,* 31-55.

[2] Chang,W. and Lawler,E (1990): Approximate string matching in sublinear expected time. *FOCS'90*, pp. 116-124.

[3] Crochemore, M. (1986): Transducers and repetitions. *Theor. Comp. Sci. 45*, 63-86.

[4] Crochemore, M. (1988): String matching with constraints. *Proc. MFCS'88. SLNCS 324*, pp. 44-58.

[5] Dowling, G. R. & Hall, P. (1980): Approximate string matching. *ACM Comput. Surv. 12*, 381-402.

[6] Galil, Z. & Giancarlo, R. (1988): Data structures and algorithms for approximate string matching. *J. Complexity 4*, 33-72.

[7] Galil, Z. & Park, K. (1989): An improved algorithm for approximate string matching. *ICALP'89. SLNCS 372*, pp. 394-404.

[8] Karp,R.M. and Rabin,M.O. (1987): Efficient randomized pattern matching. *IBM J. Res. Dev. 31*, 249-260.

[9] Kohonen,T. & Reuhkala,E. (1978): A very fast associative method for the recognition and correction of misspellt words, based on redundant hash-addressing. *Proc. 4th Int. Joint Conf. on Pattern Recognition*, 1978, Kyoto, Japan, pp. 807-809.

[10] Landau, G. & Vishkin, U. (1988): Fast string matching with k differences. *JCSS 37*, 63-78. (Also 26th *FOCS*, pp. 126-136).

[11] Manber, U. & Myers, G. (1990): Suffix arrays: a new method for on-line string searches. *SODA '90*, pp. 319-327.

[12] McCreight, E. M. (1976): A space economical suffix tree construction algorithm. *J. ACM* 23, 262-272.

[13] Owolabi, O. & McGregor, D. R.(1988): Fast approximate string matching. *Software - Practice and Experience* 18(4), 387-393.

[14] Tarhio, J. & Ukkonen, E. (1990): Boyer-Moore approach to approximate string matching. *2nd Scand. Workshop on Algorithm Theory (SWAT90), SLNCS 447*, pp. 348-359.

[15] Ukkonen, E. (1991): Approximate string matching with q-grams and maximal matches. *Theor. Comp. Sci.*, to appear.

[16] Ukkonen, E. & Wood, D. (1990): Approximate string matching with suffix automata. Report A-1990-4. Department of Computer Science, University of Helsinki.

[17] Weiner, P. (1973): Linear pattern matching algorithms. *Proc. 14th IEEE Symp. Switching and Automata Theory*, pp. 1-11.