Approximate String-Matching over Suffix Trees [⋆]

Esko Ukkonen

Department of Computer Science, University of Helsinki
P. O. Box 26, SF–00014 University of Helsinki, Finland
email: ukkonen@cs.Helsinki.FI

Abstract. The classical approximate string–matching problem of finding the locations of approximate occurrences P' of pattern string P in text string T such that the edit distance between P and P' is $\leq k$ is considered. We concentrate on the special case in which T is available for preprocessing before the searches with varying P and k. It is shown how the searches can be done fast using the suffix tree of T augmented with the suffix links as the preprocessed form of T and applying dynamic programming over the tree. Three variations of the search algorithm are developed with running times $O(mq+n)$, $O(mq\log q + \text{size of the output})$, and $O(m^2q + \text{size of the output})$. Here $n = |T|$, $m = |P|$, and q varies depending on the problem instance between 0 and n. In the case of the unit cost edit distance it is shown that $q = O(\min(n, m^{k+1}|\Sigma|^k))$ where Σ is the alphabet.

1 Introduction

The *approximate string–matching problem* is to find the approximate occurrences of a pattern in a text. We will consider the problem in the following form: Given *text* $T = t_1t_2\cdots t_n$ and *pattern* $P = p_1p_2\cdots p_m$ in alphabet Σ, and a number $k \geq 0$, find the end locations j of all substrings P' of T such that the edit distance between P and P' is $\leq k$.

The edit distance between P and P' is the minimum possible total cost of a sequence of editing steps that convert P to P'. Each editing step applies a rewriting rule of the forms $a \rightarrow \epsilon$ (deletion), $\epsilon \rightarrow b$ (insertion), or $a \rightarrow b$ (change) where $a, b \in \Sigma$, $a \neq b$.

The problem has the following four subcases:
1. $k = 0$, no preprocessing of T (*exact on–line string–matching*).
2. $k = 0$, with preprocessing of T (*exact off-line string–matching*).
3. $k > 0$, no preprocessing of T (*approximate on–line string-matching*).
4. $k > 0$, with preprocessing of T (*approximate off–line string–matching*).

Case 1 leads to the well–known Boyer–Moore and Knuth–Morris–Pratt algorithms. Case 2 has optimal solutions based on suffix trees [16, 25] or on suffix automata ('DAWG') [3, 6, 7]. Case 3 has recently received lot of attention [8, 9, 26]. The simplest solution is by dynamic programming in time $O(mn)$ where $m = |P|$ and $n = |T|$. For the *k–differences problem* (each edit operation has cost 1) fast special methods are possible, including $O(kn)$ time algorithms, see e.g. [14, 10, 23, 19, 5, 21].

⋆ This work was supported by the Academy of Finland and by the Alexander von Humboldt Foundation (Germany).

This paper deals with Case 4, which also could be called the problem of approximate string searches over indexed files. The problem is to find a suitable preprocessing for T and an associated search algorithm that finds the approximate occurrences of P using the preprocessed T for varying P and k. We show how this can be solved fast using the suffix tree (for simplicity, the algorithms will be formulated for the suffix–trie) of T augmented with the suffix links, and applying dynamic programming over the tree. Recall that a suffix tree for T is, basically, a trie representing all the suffixes of T. It can be constructed in time $O(n)$. Therefore the preprocessing phase of our algorithms will be linear.

Perhaps the most natural way of applying dynamic programming over a suffix tree is to make a depth–first traversal that finds all substrings P' of T at a distance $\leq k$ from P. (Note that this is not exactly our problem; we want only the end points of such strings P'.) The search is easy to organize because all possible substrings of T can be found along some path starting from the root of the tree. Each path is followed until the edit distance between the corresponding substring and all prefixes of P becomes $> k$. The backtracking point can be found using the column of edit distances that is evaluated at each node visited during the traversal. This type of method is described and analyzed by Baeza–Yates & Gonnet [2] (see also Remark 2 of [13]). The method is further applied in [2, 11] for finding significant alignments between all pairs of substrings of T.

In the worst case, the above method evaluates $\Theta(mn)$ columns of edit distances which is more than the n columns evaluated by the simple on–line algorithm with no preprocessing of T. In this paper we show how to apply dynamic programming over the suffix tree such that in the worst case the number of evaluated columns stays $\leq n$ and can in a good case be much smaller.

To explain the idea, let $T = \texttt{aaaaaaaabbbbbbbb}$, $P = \texttt{abbb}$, and $k = 1$. In this case there is lot of repetition in the on–line dynamic programming algorithm. It evaluates a table which has a column of $m + 1$ entries for each symbol t_j of T. We call an entry *essential* if its value is $\leq k$. The occurrences of P can be found using only the essential entries: if the last entry of a column is essential then there is an approximate occurrence whose edit distance from P is $\leq k$ ending at that column. A column and its essential part in particular can depend only on a substring of T of length $O(m)$. We call this substring a *viable k–approximate prefix* of P in T. If two columns have same viable prefix then their essential part must be identical. In our example, the eight columns corresponding to the eight a's at the beginning of T will have the same viable prefix and hence the same essential part of the column.

To avoid evaluating a column whose viable prefix has occurred earlier we store columns into the suffix tree. A column with viable prefix Q is stored with the state that can be reached along the Q–path from the root. The search algorithm performs a traversal over the tree that spells out string T. The traversal can follow both the normal trie transitions and the suffix transitions. During the traversal, new columns are evaluated for each t_j except if we can conclude that the viable prefix at t_j will be the same as some older prefix. In this case the evaluation can be skipped; we have already stored a column with the same essential part.

The number of columns evaluated by the method is $\leq n$ and proportional to q where q is the total number of different viable prefixes in T. For small k, q can be considerably smaller than n.

We elaborate the above idea into three algorithms of different degree of sophistication. The introductory Algorithm A (Section 4) runs in time $O(mq + n)$ and always needs time $\Omega(n)$. This undesirable dependency on n is eliminated by using more complicated data structures in Algorithm B (Section 5) which has running time $O(mq \log q)$ + size of the output). Algorithm C (Section 6) is finally an easy–to–implement simplification of Algorithms A and B. It can evaluate more than n columns and has running time $O(m^2 q +$ size of the output$)$. We also show that $q \leq \min(n, \frac{12}{5}(m+1)^{k+1}(|\Sigma| + 1)^k) = O(\min(n, m^{k+1}|\Sigma|^k))$.

The exponential growth of q as a function of k suggests that while our methods can be very fast for small k, their running time rapidly approaches the time of the on–line algorithm when k grows. In an interesting paper [17] (see also [1]), Myers points out that this inherent difficulty in our problem can be relieved by dividing P into smaller subpatterns and performing the search with a reduced error level for each subpattern. This filters out the interesting regions of T where one then attempts to expand the approximate occurrences of the subpatterns into k–approximate occurrences of the whole P. A simpler 'q–gram' method along similar lines is described in [13].

2 The approximate string matching problem

An *edit operation* is given by any rewriting rule of the form $a \rightarrow \epsilon$ (a *deletion*), $\epsilon \rightarrow a$ (an *insertion*), or $a \rightarrow b$ (a *change*), where a, b are any symbols in alphabet Σ, $a \neq b$, and ϵ is the empty string. Each operation $x \rightarrow y$ has a *cost* $c(x \rightarrow y) > 0$.

Operation $a \rightarrow a$ is called the *identity operation* for all $a \in \Sigma$. It has cost $c(a \rightarrow a) = 0$.

Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ be strings over Σ. A *trace* from A to B is any sequence $\tau = (x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots, x_h \rightarrow y_h)$ of edit operations and identity operations such that $A = x_1 x_2 \cdots x_h$ and $B = y_1 y_2 \cdots y_h$. The cost of a trace τ is $c(\tau) = \sum_{i=1}^{h} c(x_i \rightarrow y_i)$. The *edit distance* $E(A, B)$ between A and B is the minimum possible cost of a trace from A to B [24]. The *unit cost edit distance* which means that each edit operation has cost = 1 is denoted as $E_1(A, B)$.

The intuition behind this definition is that $E(A, B)$ will be the minimum possible total cost of a sequence of editing steps that convert A into B such that each symbol is rewritten at most once. Distance $E(A, B)$ can be evaluated in time $O(mn)$ by a very simple form of dynamic programming [24]. The method evaluates an $(m+1) \times (n+1)$ table e such that $e(i, j) = E(a_1 \cdots a_i, b_1 \cdots b_j)$. Hence $E(A, B) = e(m, n)$.

If $E(A, B) \leq k$ we say that B is a *k–approximation* of A.

Definition. Let $P = p_1 p_2 \cdots p_m$ be a *pattern string* and $T = t_1 t_2 \cdots t_n$ a *text string* over Σ, and let k be a number ≥ 0. The *approximate string matching problem with threshold* k is to find all j such that the edit distance $E(P, P')$ between P and some substring $P' = t_{j'} \cdots t_j$ of T ending at t_j is $\leq k$. Then P has a *k–approximate occurrence* P' at position j of T.

The approximate string matching problem can be solved on–line, without preprocessing of T, with a very slightly modified form of the dynamic programming for the the edit distance [18]: Let $D(i, j)$ be the minimum edit distance between the prefix $P_i = p_1 \cdots p_i$ of P and the substrings of T ending at t_j. The $(m+1) \times (n+1)$ table $D(i,j)$, $0 \le i \le m$, $0 \le j \le n$, of such values can be evaluated 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 $1 \le i \le m$, $0 \le j \le n$. It should be emphasized that all entries $D(0,j)$ on row 0 of this table have value 0 while in the corresponding table for the edit distance between P and T only the $(0,0)$–entry gets value 0.

The solution to the problem can be read from the last row of table D: there is a k–approximate occurrence of P in T at position j if and only if $D(m,j) \le k$.

In the sequel, an important technical tool will be the length $L(i,j)$ of the *shortest* substring of T ending at t_j whose edit distance from P_i equals $D(i,j)$. Value $L(i,j)$ obviously satisfies

$$L(0,j) = 0, \quad 0 \le j \le n; \tag{3}$$

$$\begin{aligned} L(i,j) = {} & \text{if } D(i,j) = D(i-1,j) + c(p_i \to \epsilon) \text{ then } L(i-1,j) \\ & \text{elsif } D(i,j) = D(i-1,j-1) + (\text{if } p_i = t_j \text{ then } 0 \text{ else } c(p_i \to t_j)) \\ & \qquad \text{then } L(i-1,j-1) + 1 \\ & \text{else } L(i,j-1) + 1 \end{aligned} \tag{4}$$

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

Tables D and L can be conveniently evaluated, column–by–column, in an on–line, left–to–right scan over T. Columns $D(*,j)$ and $L(*,j)$ can be produced from $D(*,j-1)$, $L(*,j-1)$, and symbol t_j of T. The evaluation can be organized as function *dp*, given below, which will return $(D(*,j), L(*,j))$ as $dp(D(*,j-1), L(*,j-1), t_j)$:

```
**function** $dp(d'(0 \ldots m), l'(0 \ldots m), t)$:
    $d(0) \leftarrow l(0) \leftarrow 0$;
    **for** $j \leftarrow 1$ **to** $m$ **do**
        $d(i) \leftarrow d(i-1) + c(p_i \to \epsilon)$
        $l(i) \leftarrow l(i-1)$
        **if** $d'(i-1) + ($**if** $p_i = t_j$ **then** $0$ **else** $c(p_i \to t_j)) < d(i)$ **then**
            $d(i) \leftarrow d'(i-1) + ($**if** $p_i = t_j$ **then** $0$ **else** $c(p_i \leftarrow t_j))$
            $l(i) \leftarrow l'(i-1) + 1$;
        **if** $d'(i) + c(\epsilon \to t_j) < d(i)$ **then**
            $d(i) \leftarrow d'(i) + c(\epsilon \to t_j)$
            $l(i) \leftarrow l'(i) + 1$
    **return**$(d, l)$.
```

This takes time $O(m)$ and the evaluation of D and L therefore takes total time of $O(mn)$. Other on–line algorithms running in $O(kn)$ expected time [20, 4] (these methods can easily be incorporated into procedure *dp*) or in $O(kn)$ worst–case time (for the unit cost edit distance) [10, 23] are also known.

In the next sections we develop algorithms that are off-line with respect to T. We assume that T has been preprocessed into a suffix tree and study how the evaluation of D can be organized in a more efficient way.

3 k–approximate prefixes of P

The on–line solution to our problem in Section 2 has the drawback that dynamic programming is explicitly repeated over identical repeated substrings of T. This may create unnecessary work because the content of each column $D(*,j)$ of D depends only on a relatively short substring of T. If such a substring occurs again in T, the dynamic programming would give a column that is equal to an old column. Our new algorithms avoid the repetition of such identical calculations.

To make this precise we first define the *essential entries* of D. The approximate string matching problem can be solved using only entries $D(i,j) \leq k$ of D. Therefore we call each entry $D(i,j) \leq k$ an *essential* entry. By (1), (2), an essential entry depends only on other essential entries in the sense that the inessential entries of D could be replaced by default value ∞ without affecting the content of the essential part.

Let $D(*,i)$ and $D(*,j)$ be any two columns of D and let $L(*,i)$ and $L(*,j)$ be the corresponding columns of L. Then pairs $(D(*,i), L(*,i))$ and $(D(*,j), L(*,j))$ are called *equivalent*, denoted $(D(*,i), L(*,i)) \equiv (D(*,j), L(*,j))$, if the essential entries of $D(*,i)$ and $D(*,j)$ have identical contents and the corresponding entries of $L(*,i)$ and $L(*,j)$ have identical contents. In other words, if $D(h,i) \leq k$ or $D(h,j) \leq k$ for some $0 \leq h \leq m$, then $D(h,i) = D(h,j)$ and $L(h,i) = L(h,j)$.

Next we define the substring Q_j of T that determines the essential part of $D(*,j)$. Recall here that the Knuth–Morris–Pratt algorithm of exact string matching has the property that it finds at each text location j the *longest* prefix $p_1 \cdots p_i$ of pattern P that occurs at j, i.e., $p_1 \cdots p_i = t_{j-i+1} \cdots t_j$ is a 0–approximation of $p_1 \cdots p_i$ that occurs at j. The use of Q_j can be seen as a generalization of this to the approximate case: Q_j will be a k–approximation of $p_1 \cdots p_i$ that occurs at j in T.

Let $T_j = t_1 \cdots t_j$ be the prefix of T ending at j, and let $\lambda(T_j) = L(i,j)$ where i is the largest index such that $D(i,j)$ is essential. Obviously, $P_i = p_1 \cdots p_i$ is the longest prefix of P that has a k–approximation at the end of T_j. String $t_{j-\lambda(T_j)+1} \cdots t_j$ is such an approximation, in fact, the shortest one.

Definition. String $Q_j = t_{j-\lambda(T_j)+1} \cdots t_j$ is called the *viable k–approximate prefix* of P at j (*viable prefix* at j, for short). If $\lambda(T_j) = 0$ then $Q_j = \epsilon$.

String Q_j is 'viable' in the sense that it can be a prefix of a k–approximate occurrence of the whole P.

Viable prefix Q_i determines the essential part of column $D(*,i)$:

Theorem 1. *If $Q_i = Q_j$ then $(D(*,i), L(*,i)) \equiv (D(*,j), L(*,j))$.*

Proof. It is helpful to consider table D as a solution to a shortest path problem in the *edit graph* associated with our pattern matching problem.

Such a graph consists of nodes $G(i,j)$, $0 \leq i \leq m$, $0 \leq j \leq n$, and of weighted directed arcs that form a regular grid as follows: There is an arc $(G(i-l,j), G(i,j))$ with weight $c(p_i \to \epsilon)$ for all $1 \leq i \leq m$, $0 \leq j \leq n$; an arc $G(i-1,j-1), G(i,j))$ with weight 0 if $p_i = t_j$ and with weight $c(p_i \to t_j)$ otherwise for all $1 \leq i \leq m$, $1 \leq j \leq n$; and an arc $(G(i,j-1), G(i,j))$ with weight $c(\epsilon \to t_j)$ for all $1 \leq i \leq m$, $1 \leq j \leq n$. Then $D(i,j)$ gives the length of a shortest path in this graph among all paths that lead from any node $G(0,j')$ on the row 0 to node $G(i,j)$. Value $L(i,j)$

indicates the start node of a steepest path: $L(i, j)$ is the smallest value such that a shortest path to $G(i, j)$ starts from $G(0, j - L(i, j))$.

Let now i and j be as in the theorem and let h be the largest index such that $D(h, i) \leq k$. Hence $|Q_i| = L(h, i)$. Then for each $r \leq h$, there is a shortest path to $G(r, i)$ that starts from some node $G(0, i - |Q_i|), G(0, i - |Q_i| + 1), \ldots, G(0, i)$. To evaluate the essential entries of column $D(*, i)$ correctly it therefore suffices to consider only subgraph G_i of the edit graph spanned by nodes $G(r, s)$, $0 \leq r \leq m$, $i - |Q_i| \leq s \leq i$. Similarly, to evaluate the essential entries of column $D(*, j)$ correctly it suffices to consider only subgraph G_j spanned by nodes $G(r, s)$, $0 \leq r \leq m$, $j - |Q_j| \leq s \leq j$. Graphs G_i and G_j have identical topology and weights because $Q_i = Q_j$. Hence their shortest path problems have identical solutions, in particular, the essential entries of $D(*, i)$ and $D(*, j)$ have to be identical as well as the corresponding entries of $L(*, i)$ and $L(*, j)$. □

Example. Let $T =$ `aaaaaaaabbbbbbbb`, $P =$ `abbb`, and $k = 1$. Assume the unit cost model of the edit distance (each edit operation has cost $= 1$). Then table D is

$$\begin{array}{c|ccccccccccccccccc}
 & & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathtt{a} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\mathtt{b} & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\mathtt{b} & 3 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\mathtt{b} & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 2 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}$$

and table L is

$$\begin{array}{|ccccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 4 & 3 & 3 & 3 & 3 & 3
\end{array}$$

The viable prefixes are $Q_1 =$ `a` (because $D(2, 1)$ is the last essential entry of $D(*, 1)$, and $L(2, 1) = 1$), $Q_2 = Q_3 = \ldots = Q_8 =$ `a`, $Q_9 =$ `ab`, $Q_{10} =$ `abb`, $Q_{11} =$ `abbb`, $Q_{12} = \ldots = Q_{16} =$ `bbb`. There are five different viable prefixes. □

For each j we let $Q'_j = Q_{j-1}t_j$. The following theorem says that viable prefix Q_j can not start properly before Q_{j-1}.

Theorem 2. *Q_j is a suffix of Q'_j.*

Proof. Using the interpretation of D as a solution to the shortest path problem (see the proof of Theorem 1), one first notices that values $L(h, j)$ are non–decreasing when h grows: If $h < h'$ then $L(h, j) \leq L(h', j)$. The rest of the proof is a simple case analysis of how $L(h, j)$ where h is the largest index such that $D(h, j) \leq k$ can depend on the entries of $L(*, j - 1)$. □

4 Dynamic programming over suffix trees

We will evaluate table D using T represented as a suffix tree. First we recall the alternative forms of such trees.

Suffix tree of T. The suffix tree of T is a data structure representing all the suffixes $T^i = t_i \cdots t_n$, $1 \leq i \leq n+1$, of T. We distinguish three versions of such a structure.

The uncompacted version of a suffix tree is called a *suffix trie* of T, denoted $\mathit{STrie}(T)$. It is the unique deterministic finite–state automaton that recognizes the suffixes T^i of T and nothing else, and has tree–shaped transition graph. The transition graph is the trie representing strings T^i.

Let *root* denote the initial state and g the transition function of $\mathit{STrie}(T)$. We say that there is a *goto–transition* from state r to state s on input $a \in \Sigma$ if $s = g(r, a)$. If there is a goto–transition path from r to s on input symbols whose catenation is string x we write $s = g(r, x)$.

We augment $\mathit{STrie}(T)$ with the *suffix function* f, defined for each state s, $s \neq \mathit{root}$, as follows: As $s \neq \mathit{root}$, there is a symbol a and a string x in Σ^* such that $g(\mathit{root}, ax) = s$. We set $f(s) = r$ where r is the state such that $g(\mathit{root}, x) = r$. We say that there is a *suffix transition* from s to r. A suffix transition does not consume any input.

The size of $\mathit{STrie}(T)$ is $O(|T|^2)$. $\mathit{STrie}(T)$ is easy to construct (see e.g. [23, 22]) but its quadratic size makes it impractical. Fortunately, $\mathit{STrie}(T)$ has linear size representations that can be constructed in linear time, namely the (compact) *suffix tree* [25, 16, 22] and the *suffix automaton* (DAWG) [3, 6, 7].

For simplicity, the suffix trie $\mathit{STrie}(T)$ consisting of functions g and f will be used in the description of our algorithms. However, the actual implementation will be done using the standard linear–size suffix tree or suffix automaton for T. This does not change the complexity bounds derived here for $\mathit{STrie}(T)$.

Algorithm A. The algorithm will traverse in $\mathit{STrie}(T)$ a path of goto and suffix transitions that starts from *root* and spells out in its goto–transitions string T. Combined with this the columns of D that correspond to different viable prefixes Q_i will be evaluated. Each such column $D(*, i)$ together with column $L(*, i)$ will be stored with state $r_i = g(\mathit{root}, Q_i)$ as $d(r_i) \leftarrow D(*, i)$, $l(r_i) \leftarrow L(*, i)$.

The traversal goes through states $r_0, s_1, \ldots, r_1, s_2, \ldots, r_{n-1}, s_n, \ldots, r_n$ where $r_0 = \mathit{root}$, $r_i = g(\mathit{root}, Q_i)$, and $s_i = g(\mathit{root}, Q'_i)$. The transition from r_{i-1} to s_i is a goto–transition for t_i because $s_i = g(\mathit{root}, Q'_i) = g(\mathit{root}, Q_{i-1}t_i) = g(r_{i-1}, t_i)$. The transition path from s_i to r_i consists of zero or more suffix transitions; such a path exists by Theorem 2.

Consider the subpath from r_{j-1} to r_j. The goto–transition $g(r_{j-1}, t_j) = s_j$ is taken first. After that there are two cases:

Case 1. If s_j has already been visited during the traversal, then follow the suffix transition path until the first state r is encountered such that $d(r)$ and $l(r)$ have non–empty values. Then $r = r_j$.

Case 2. If s_j has not been visited yet, then evaluate a pair (d, l) of columns as $(d, l) \leftarrow dp(d(r_{j-1}), l(r_{j-1}), t_j)$. Then (see Lemma 4 below) $(d, l) \equiv (D(*, j), L(*, j))$. This equivalence implies that $d(h) = D(h, j)$ and $l(h) = L(h, j) = |Q_j|$, where h is such that $d(h)$ is the last essential entry of d. The algorithm then follows the suffix link path from s_j to the state r whose depth (distance from *root*) is $|Q_j|$. Then $r = r_j$ and the algorithm saves columns (d, l) as $d(r) \leftarrow d$, $l(r) \leftarrow l$.

To make the whole traversal the above is repeated for $j = 1, \ldots, n$. As an initialization we set $d(\mathit{root}) \leftarrow D(*, 0)$, $l(\mathit{root}) \leftarrow L(*, 0)$. By (2), entry $D(h, 0)$ of $D(*, 0)$

is given as $D(h,0) = \sum_{i=1}^{h} c(p_i \to \epsilon)$, and by (4), entry $L(h,0)$ of $L(*,0)$ is given as $L(h,0) = 0$.

The algorithm has to output j whenever $D(m,j) \leq k$. This is implemented such that Algorithm A outputs j whenever $d(r_j)(m) \leq k$ during the traversal.

Consider then the correctness of Algorithm A. We need a notation: If x is a suffix of y, we write $y|x$, and if, moreover, y is a suffix of z, we write $z|y|x$.

The crucial point where Algorithm A saves compared to the on–line algorithm is Case 1. Assume that $s_j = g(root, Q'_j)$ has been visited earlier. This means that s_j has to belong to the suffix link path between s_i and r_i for some $i < j$, that is, $Q'_i|Q'_j|Q_i$. On the other hand we have:

Lemma 3. *If $Q'_i|Q'_j|Q_i$ for some $i < j$, then $Q_j = Q_i$.*

Proof. This is immediate when D is viewed as a solution to the shortest path problem (see the proof of Theorem 1). □

This implies, noting Theorem 1, that a pair of columns equivalent to $(D(*,j), L(*,j))$ has already been stored as $(d(r_i), l(r_i))$. The dynamic programming can be skipped; the algorithm just follows the suffix transition path from s_j to $r_i = r_j$. Hence Case 1 is correct.

It is correct to use in Algorithm A columns that are only equivalent to the actual columns of D and L. The essential entries of a new column of D are determined by the essential entries of the previous column. Therefore we have the following lemma.

Lemma 4. *If $(d', l') \equiv (D(*,j-1), L(*,j-1))$ and $(d,l) = dp(d', l', t_j)$, then $(d,l) \equiv (D(*,j), L(*,j))$.*

Hence Algorithm A correctly outputs all j such that $D(m,j) \leq k$.

Analysis. Let $\mathbf{Q} = \{Q_i \mid 1 \leq i \leq n\}$, and let $q = |\mathbf{Q}|$ be the size of $\mathbf{Q}$, i.e. the number of different viable prefixes. Moreover, let $\mathbf{Q}' = \{Q'_i \mid 1 \leq i \leq n\}$ and $q' = |\mathbf{Q}'|$.

Algorithm A evaluates $\leq q'$ pairs of columns of D and L, and stores q of them. As the evaluation of each pair of columns takes time and space $O(m)$, and the time consumption for the rest is proportional to n (note that the traversal takes n goto–transitions and at most n suffix transitions), we obtain:

Theorem 5. *Algorithm A runs in time $O(mq' + n)$ and needs working space $O(mq)$ for storing the columns of the tables.*

Next we analyze the growth of q in more detail in the special case of the unit cost edit distance. Let $U_k(P) = \{x \in \Sigma^* | E_1(P,x) \leq k\}$ be the set of strings whose unit cost edit distance from P is $\leq k$. The size of $U_k(P)$ has the following bound; c.f. Lemma 3 of [17].

Theorem 6. $|U_k(P)| \leq \frac{12}{5}(m+1)^k(|\Sigma|+1)^k$.

Proof. The size of $U_k(P)$ is $\leq$ the number of different traces (edit scripts) of length $\leq k$ that can be applied on P. Each trace consists of $\leq k$ actual editing steps and of zero or more identity steps $a \to a$. The number of traces equals the number of different possibilities to select the actual steps. This can be estimated by bounding the number of different ways of applying exactly k steps that can include both actual steps *and* identity steps.

The k steps are divided into two groups: The steps of the form $a \to x$ where $a \in \Sigma$, $x \in \Sigma \cup \{\epsilon\}$ (= group A; this contains the possible identity operations), and the steps of the form $\epsilon \to a$ where $a \in \Sigma$ (= group B).

In group A, each step $a \to x$ has a unique p_i such that $a = p_i$. Moreover, x can be selected in $|\Sigma| + 1$ different ways. Hence a group A consisting of t steps can be selected in $\leq \binom{m}{t}(|\Sigma| + 1)^t$ different ways.

In group B, each step $\epsilon \to a$ can be selected in $(m+1)|\Sigma|$ different ways because ϵ refers to any of the $m+1$ intervals between the m letters of P, and because a can be selected independently of ϵ in $|\Sigma|$ different ways. Each interval can be selected arbitrarily many times. Hence a group B consisting of t steps can be selected in $\leq (m+1)^t|\Sigma|^t$ different ways.

This gives

$$\begin{aligned}
|U_k(P)| &\leq \sum_{t=0}^{k}[\tbinom{m}{t}(|\Sigma|+1)^t + (m+1)^{k-t}|\Sigma|^{k-t}] \\
&= \sum_{t=0}^{k}[\tbinom{m}{t}(|\Sigma|+1)^t + (m+1)^t|\Sigma|^t|] \\
&\leq 2\sum_{t=0}^{k}(m+1)^t(|\Sigma|+1)^t \leq \frac{12}{5}(m+1)^k(|\Sigma|+1)^k
\end{aligned}$$

where we have assumed that $m \geq 1$ and $|\Sigma| \geq 2$. $\Box$

As $q \leq \sum_{i=k}^{m} U_k(p_1 \cdots p_i) \leq m \cdot |U_k(P)|$, we have by Theorem 6

$$q \leq \frac{12}{5}(m+1)^{k+1}(|\Sigma|+1)^k = O(m^{k+1}|\Sigma|^k). \tag{5}$$

As $q' \leq |\Sigma|q$, we further obtain

$$q' \leq \frac{12}{5}(m+1)^{k+1}(|\Sigma|+1)^{k+1} = O(m^{k+1}|\Sigma|^{k+1}). \tag{6}$$

Noting that $q \leq q' \leq n$, Theorem 5 with (5) and (6) gives:

Theorem 7. *Algorithm A runs for the k–differences problem in time* $O(m \cdot \min(n, m^{k+1}|\Sigma|^{k+1}) + n)$ *and needs working space of* $O(m \cdot \min(n, m^{k+1}|\Sigma|^k))$.

5 Finding the next viable prefix fast

The method of this section can be understood as an advanced implementation of Algorithm A. Algorithm A always needs time $\Omega(n)$ because it scans symbol by symbol over the whole text T. In Algorithm B to be developed next this dependency on n will be eliminated. Columns of D for different viable prefixes will be found using dictionary operations implemented with balanced search trees. The method is based on Lemma 3 and its implementation heavily depends on the special properties of $STrie(T)$.

Assume that Algorithm A has performed the dynamic programming at t_i, has obtained (d,l) equivalent to $(D(*,i),L(*,i))$, and has stored them as $d(r_i) \leftarrow d$, $l(r_i) \leftarrow l$ where $r_i = g(root, Q_i)$. Algorithm A will next examine the state $s_{i+1} = g(r_i, t_{i+1})$. If s_{i+1} has already been visited, Algorithm A knowns by Lemma 3 that dynamic programming can be skipped because Q_{i+1} has to be equal to Q_h for some $h \leq i$. State $r_{i+1} = g(root, Q_h) = g(root, Q_{i+1})$ is found by following the suffix link path from s_{i+1}. Then Algorithm A will examine $s_{i+2} = g(r_{i+1}, t_{i+2})$, and so on. Finally an unvisited state s_j will be found, and dynamic programming is resumed.

To find s_j *directly* after s_i, we first observe:

• the set of different viable prefixes can grow at s_i and again at s_j, but it remains unchanged between them;

• the set of the visited states remains unchanged between s_i and s_j;

• the string on the path from *root* to any state $s_{i+1}, \ldots, s_j$ is of the form $Q_h a$ for some $a \in \Sigma$, $h \leq i$.

Hence states $s_{i+1}, \ldots, s_j$ belong to the set

$$S_i = \{s \mid s = g(root, Q_h a) \text{ for some } h \leq i, a \in \Sigma\}$$

of states that are at the distance of one goto–transition from some state that can be reached from *root* along some viable prefix Q_h.

Algorithm B. For any state s of $STrie(T)$, let $Key(s)$ denote the string such that $g(root, Key(s)) = s$, and for a set S of states, let $Keys(S)$ be the set of strings $Key(s)$, where $s \in S$. We will associate with each state s in S_i value $loc(s)$ (to be defined precisely below) that gives the smallest index $h > i$ such that $Key(s)$ 'could be' equal to Q'_h. During Algorithm B the *uneliminated* states s in S_i will be kept in dictionary H. The records in the dictionary are of the form $(s, loc(s))$ where $loc(s)$ is used as the search–key for s. The dictionary has to support insertions, deletions, and minimum extractions. By extracting the minimum element from H we get the state s with the smallest $loc(s)$. This state s will be s_j and $j = loc(s_j)$. Then new columns have to be evaluated by dynamic programming from $d(r)$ and $l(r)$, where $r = father(s_j)$, and from symbol a such that $g(r,a) = s_j$.

For a precise definition of $loc(s)$ we need the concepts of *elimination* and *covering.* To introduce the latter, consider strings Q'_v, $i+1 \leq v \leq j$, in more detail. As already mentioned, each $Q'_v = Q_{v-1}t_v$ has to be equal to $Q_h a$ for some Q_h, $h \leq i$. Hence we have $Q_{v-1} = Q_h$. Moreover, viable prefix Q_{v-1} is the longest among *all* viable prefixes of T that are suffixes of $T_{v-1} = t_1 t_2 \cdots t_{v-1}$:

Lemma 8. *If* $T_{v-1}|Q_e$ *then* $Q_{v-1}|Q_e$.

Proof. Use the interpretation of D as a solution to the shortest path problem as presented in the proof of Theorem 1. □

This implies that each Q'_v, $i+1 \leq v \leq j$, has to be the *longest* string in $Keys(S_i)$ that is a suffix of T_v. If more than one string in $Keys(S_i)$ is a suffix of T_v, then these strings have to be suffixes of the longest one. With this in mind we make the following definition.

Definition. String X *covers* an occurrence of string Y at v if $T_v|X|Y$.

String $Key(s)$ is the longest element of $Keys(S_i)$ at v if and only if $T_v|Key(s)$ and no other string in $Keys(S_i)$ covers $Key(s)$ at v.

We still need the concept of elimination. Its purpose is to incorporate Lemma 3 into our algorithm.

Definition. Strings Q'_h and Q_h *eliminate* a state s and string $Key(s)$ if $Q'_h|Key(s)|Q_h$.

Note that the states visited by Algorithm A and the eliminated states defined here are same. By Lemma 3, dynamic programming need not be performed when entering an eliminated state.

We now define

$$loc(s) = \begin{cases} \infty, & \text{if } Key(s) \text{ is eliminated by some } Q'_h,\ Q_h \text{ where } h \le i; \\ v, & \text{otherwise,} \end{cases}$$

where $v > i$ is the first occurrence of $Key(s)$ after location i in T that is not covered by some other string in $Keys(S_i)$. Note that $loc(s)$ is defined for all states s, not only for members of S_i. The algorithm also maintains these values for all s.

The algorithm selects $j \leftarrow \min_{s \in S_i} loc(s)$ using dictionary H that contains $(s, loc(s))$ for states s in S_i. The dynamic programming is performed next at s_j such that $loc(s_j) = j$.

After this some *loc*–values have to be changed and H must be updated such that it represents S_j instead of S_i. The algorithm follows the suffix link path from s_j to $r_j = g(root, Q_j)$. All states s on this path become now eliminated if they are not eliminated earlier (this can be the case for all $s \neq s_j$). Hence $loc(s) \leftarrow \infty$; this is implemented simply by removing s from H.

We have still to add into H new elements corresponding to $S_j - S_i$ and to make the updates on *loc*–values due to covering. This happens only if r_j is a new state not visited earlier. Then $(s, loc(s))$ is inserted into H for all uneliminated s such that $s = g(r_j, a)$ for some symbol a. Moreover, the appropriate changes to $loc(w)$ have to be done for all w such that $Key(w)$ is covered by some $Key(s)$.

Here, again, the suffix transitions can be used. We call a state w *primary* if $Key(w) = t_1 \cdots t_h$ for some h. (Note that the suffix transitions constitute a tree, with primary states as the leaves and *root* as the root.) The next lemma follows from the definition of *loc* and gives a method for updating; recall that f denotes the suffix function.

Lemma 9. *If w is an eliminated state then $loc(w) = \infty$; if w is primary but not eliminated then $loc(w) = depth(w)$; otherwise*

$$loc(w) = \min loc(w') \tag{7}$$

where the minimum is over all w' such that $f(w') = w$ and w' is not in S_i.

This means, each $loc(w)$ that needs updating can be found by traversing the suffix link path from each new state $s \in S_j - S_i$. At each uneliminates state w, $w \neq s$, on such path the updated $loc(w)$ is evaluated from (7). As there are at most $|\Sigma|$ different w' such that $f(w') = w$, the minimization in (7) can be done in time $O(\log |\Sigma|)$. If $(w, loc(w))$ is in H, the update is performed in H, too.

In summary, Algorithm B starts by inserting $(root, loc(root) = 0)$ into an initially empty dictionary H. Then $(s_j, j) \leftarrow$ *extract-min*(H) is performed, H and the *loc*–values are updated, and this is repeated until H becomes empty. Whenever a column $d(r)$ is stored such that $d(r)(m) \leq k$, state r is marked for output. The final output phase lists all occurrences of $Key(r)$ in T, for all states r marked for output. These occurrences can be found from $STrie(T)$ by standard methods.

The preprocessing phase creates $STrie(T)$ and initializes values $loc(s)$ using the method of Lemma 9 with $S_i = \emptyset$.

Theorem 10. *Algorithm B runs in time $O(mq \log q +$ size of the output$)$ and needs working space of $O(mq)$ for dictionary H and the columns of dynamic programming tables.*

Proof. Algorithm B evaluates q' columns of D and L. Dictionary H is implemented as a balanced search tree which takes $O(\log |H|)$ time per dictionary operation. The algorithm performs the following q' times: selection of next s_j from H in time $O(\log |H|)$; evaluation of new columns in time $O(m)$; traversal from s_j to r_j, removal of the eliminated states from H in time $O(m \log |H|)$; insertion of states $s = g(r_j, a)$ into H in time $O(|\Sigma| \log |H|)$. Moreover, for each new state s inserted into H during the algorithm, $loc(w)$ has to be updated for states w on the suffix link path from s to *root* and the corresponding changes have to be done in H. The length of each such path is $O(m)$, hence the updates take total time of $O(|H| m (\log |\Sigma| + \log |H|))$.

This gives total time bound $O(q'(\log |H| + m + m \log |H|) + |H| m (\log |\Sigma| + \log |H|))$ which is $O(mq \log q)$ because $q' \leq |\Sigma| q$, $|H| \leq |\Sigma| q$, and $|\Sigma|$ is assumed constant.

The output time can be made linear in the size of the output if some care is devoted to the elimination of duplicated output.

The space requirement is $O(mq)$ for the columns and $O(|\Sigma| q)$ for H, hence $O(mq)$. □

Theorem 10 together with upper bound (6) of q shows that for small k and large n Algorithm B can be faster than Algorithm A.

6 Simple algorithm

Dictionary H and the other mechanisms of Algorithm B for maintaining values $loc(s)$ create relatively large overhead. We describe next Algorithm C, a simplified version of Algorithm B that uses only elimination of states but does not use *loc*–values. Algorithm C is easy to implement and has low overhead.

Algorithm C makes a depth–first–search over the uneliminated states. All states with a saved pair (d, l) of columns are now kept in a stack. When there is a transition $g(r, a) = s$ from the top state r of the stack to an uneliminated state s, new columns are evaluated as $(d, l) \leftarrow dp(d(r), l(r), a)$. Columns (d, l) and state r' are saved in the stack; state r' is the state on the suffix link path from s such that its distance from *root*, *depth*(r'), equals the length of the viable prefix associated with (d, l).

The resulting algorithm is given below. Function *viable–prefix–length*(d, l) gives the length of the viable prefix represented by columns (d, l), i.e., the value of $l(h)$ where h is the largest index such that $d(h) \leq k$. Function *output–mark*(r) adds state

r to the list of states that represent the locations of the k–approximate occurrences of P in T.

Algorithm C.

```
1.  $eliminated(root) \leftarrow$ true
2.  $search(root, D(*, 0), L(*, 0))$.
3.  procedure $search(r, d'(0 \ldots m), l'(0 \ldots m))$:
4.      for each state $s = g(r, a)$ for some $a \in \Sigma$ do
5.          if not($eliminated(s)$) then
6.              $(d, l) \leftarrow dp(d', l', a)$
7.              $length \leftarrow viable\text{–}prefix\text{–}length(d, l)$
8.              if $depth(s) > length$ do
9.                      $eliminated(s) \leftarrow$ true; $s \leftarrow f(s)$
10.                 until $depth(s) = length$ or $eliminated(s)$
11.             if $depth(s) = length$ and not($eliminated(s)$) then
12.                 if $d(m) \leq k$ then $output\text{–}mark(s)$
13.                 $eliminated(s) \leftarrow$ true
14.                 $w \leftarrow s$
15.                 while $f(w) \neq root$ and
                    $eliminated(f(w')) =$ true for all $w'$ such that $f(w') = f(w)$ do
16.                     $w \leftarrow f(w)$; $eliminated(w) \leftarrow$ true
17.                 $search(s, d, l)$.
```

In Algorithm C the selection order of the next state s is not based on $loc(s)$. Therefore Algorithm C can select a state s that would have never been selected by Algorithm B; the optimal selection order implemented in Algorithm B can result into total covering of s and therefore into an elimination of s before it would come selected.

Fortunately, it is not a fatal error to select such an s. It only means that the algorithm first finds a too short viable prefix for some locations of T but will find the correct, long–enough prefix later. All different essential parts of columns of D will ultimately be evaluated.

Each viable prefix is of length $O(m)$. Before finding the correct prefix Algorithm C may find one or more of its proper suffixes. Therefore the total number of extra columns evaluated is $O(mq)$. In any case, the algorithm evaluates the same q' columns as Algorithm B. Thus the total number of columns is $O(mq + q') = O(mq)$ and we have the following theorem.

Theorem 11. *Algorithm C runs in time $O(m^2q$ + size of the output$)$ and needs working space of $O(m^2q)$.*

7 Concluding remarks

Several relevant questions concerning the new algorithms remained unanswered. Most notably, these include theoretical analysis of the expected running times and experimental comparison of these and related algorithms from [2, 13, 17].

For modestly long T it is feasible to implement our algorithms using the (compact) suffix tree of T. Adapting the methods for suffix automata seems simple, too.

However, for very long texts it is better to use the more space economical suffix array [15, 12] instead. The details and a practical fine–tuning of such an implementation are a subject for further study.

References

1. Altschul, S., Gish, W., Miller, W., Myers, E. & Lipman, D. (1990): A basic local alignment search tool. *J. of Molecular Biology 215*, 403–410.
2. Baeza–Yates, R. A. & Gonnet, G. H.: All–against–all sequence matching (Extended Abstract).
3. 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.
4. Chang, W. & Lampe, J. (1992): Theoretical and empirical comparisons of approximate string matching algorithms. *Proc. Combinatorial Pattern Matching 1992*, (Tucson, April 1992), Lect. Notes in Computer Science 644 (Springer–Verlag 1992), pp. 175–184.
5. Chang, W. & Lawler, E (1990): Approximate string matching in sublinear expected time. *Proc. IEEE 1990 Ann. Symp. on Foundations of Computer Science*, pp. 116-124.
6. Crochemore, M. (1986): Transducers and repetitions. *Theor. Comp. Sci. 45*, 63-86.
7. Crochemore, M. (1988): String matching with constraints. *Proc. MFCS'88 Symposium.* Lect. Notes in Computer Science 324 (Springer–Verlag 1988), pp. 44–58.
8. Dowling, G. R. & Hall, P. (1980): Approximate string matching. *ACM Comput. Surv. 12*, 381–402.
9. Galil, Z. & Giancarlo, R. (1988): Data structures and algorithms for approximate string matching. *J. Complexity 4*, 33–72.
10. Galil, Z. & Park, K. (1989): An improved algorithm for approximate string matching. *SIAM J. on Computing 19*, 989–999.
11. Gonnet, G. H. (1992): *A tutorial introduction to Computational Biochemistry using Darwin.* Informatik E. T. H. Zuerich, Switzerland.
12. Gonnet,G.H., Baeza-Yates,R.A. & Snider,T. (1991): Lexicographical indices for text: Inverted files vs. PAT trees. Report OED-91-01, UW Centre for the New Oxford English Dictionary and Text Research, 1991.
13. Jokinen, P. & Ukkonen, E. (1991): Two algorithms for approximate string matching in static texts. *Proc. MFCS'91*, Lect. Notes in Computer Science 520 (Springer–Verlag 1991), pp. 240-248.
14. Landau, G. & Vishkin, U. (1988): Fast string matching with k differences. *J. Comp. Syst. Sci. 37*, 63-78.
15. Manber, U. & Myers, G. (1990): Suffix arrays: A new method for on–line string searches. In: *SODA-90*, pp. 319–327.
16. McCreight, E. M. (1976): A space economical suffix tree construction algorithm. *J. ACM* 23, 262-272.
17. Myers, E. W.: A sublinear algorithm for approximate keyword searching. TR 90–25, Department of Computer Science, The Univ. of Arizona, Tucson (to appear in *Algorithmica*).
18. Sellers, P. H. (1980): The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms* 1, 359–373.
19. Tarhio, J. & Ukkonen, E. (1990): Boyer-Moore approach to approximate string matching. *2nd Scand. Workshop on Algorithm Theory*, Lect. Notes in Computer Science 447 (Springer–Verlag 1990), pp. 348-359. Full version is to appear in *SIAM J. Comput. 22.*

20. Ukkonen, E. (1985): Finding approximate patterns in strings. *J. Algorithms 6*, 132–137.
21. Ukkonen, E. (1992): Approximate string–matching with q–grams and maximal matches. *Theoretical Computer Science 92*, 191–211.
22. Ukkonen, E. (1992): Constructing suffix trees on–line in linear time. In: J. van Leeuwen (ed.), *Algorithms, Software, Architecture. Information Processing 92*, vol. I, pp. 484–492. Elsevier.
23. Ukkonen, E. & Wood, D.: Approximate string matching with suffix automata. *Algorithmica* (to appear in 1993).
24. Wagner, R. A. & Fischer, M. J. (1974): The string-to-string correction problem. *J. ACM* 21, 168-173.
25. Weiner, P. (1973): Linear pattern matching algorithms. *Proc. 14th IEEE Symp. Switching and Automata Theory*, pp. 1-11.
26. Wu, S. & Manber, U. (1992): Fast text searching allowing errors. *Comm. ACM 35*, 83–91.