Theoretical Computer Science 92 (1992) 191–211
Elsevier

Approximate string-matching with q-grams and maximal matches*

Esko Ukkonen

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

Abstract

Ukkonen, E., Approximate string-matching with q-grams and maximal matches, Theoretical Computer Science 92 (1992) 191–211.

We study approximate string-matching in connection with two string distance functions that are computable in linear time. The first function is based on the so-called q-grams. An algorithm is given for the associated string-matching problem that finds the locally best approximate occurrences of pattern P, $|P|=m$, in text T, $|T|=n$, in time $O(n \log(m-q))$. The occurrences with distance $\leqslant k$ can be found in time $O(n \log k)$. The other distance function is based on finding maximal common substrings and allows a form of approximate string-matching in time $O(n)$. Both distances give a lower bound for the edit distance (in the unit cost model), which leads to fast hybrid algorithms for the edit distance based string-matching.

1. Introduction

The *approximate string-matching* problem is to find the approximate occurrences of a pattern string P in a text string T [6, 8]. The approximation quality can be measured with different string distance functions. Recently, the version of the problem that is based on the edit distance has received lot of attention [2, 9, 10, 14, 15, 21, 22, 24].

In this paper we study the approximate string-matching in connection with two other distance measures. The first measure is based on the so-called q-grams and the second measure on the intuition that if two strings are close to each other then they must have long matching substrings. Our main motivation is to find alternatives to

* This work was supported by the Academy of Finland and by the Alexander von Humboldt Foundation (Germany). The work was done during the author's visit to the Institut für Informatik, University of Freiburg, Rheinstrasse 10–12, D-7800 Freiburg, Germany.

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

the edit distance because it leads to dynamic programming that is often relatively slow.

In fact, computing the edit distance between strings A and B needs time $\Theta(|A||B|)$ in the worst case while the two other distances are computable in linear time in $|A|+|B|$. This suggests that also the approximate string-matching problem could be solved faster for the two other measures than for the edit distance. It turns out that this is really the case.

The q-grams ("n-grams") are simply substrings of length q; the concept dates back to Shannon [20]. They have been applied, in many variations, e.g. in different spelling correction methods. Such systems typically preprocess the text (which represents a static dictionary) to make the subsequent searches for "correct" words faster [13, 18]. In Section 2 we study some properties of a q-gram based string distance measure. In Section 3 we give algorithms for different string-matching problems that are based on this measure. For finding the approximate occurrences of P in T that are locally the best ones according to the q-gram distance, we give a solution with running time $O(|T|\log|P|-q))$. For the threshold version of the problem, in which one wants to find the occurrences with a distance $\leqslant k$, an algorithm with running time $O(|T|\log k)$ is given. We only deal with the case *without* text preprocessing.

The second measure for string distance is defined as the minimum number of characters that have to be removed from one string such that the remaining fragments occur as substrings in the other string [7]. In Section 4 a corresponding approximate string-matching problem is defined and solution algorithms are studied.

The final Section 5 points out some important connections to the edit distance based string-matching. We show that the two string distance measures studied in Sections 2, 3, and 4 provide nontrivial lower bounds on the unit cost edit distance of approximate occurrences of P in T. This leads to the following scheme for solving the k differences problem (the problem of finding the substrings of T such that the unit cost edit distance between the substring and P is $\leqslant k$, where k is a given threshold value): Compute at each text location i the lower bound for the edit distance between P and the potential occurrence of P that ends at i. At the locations where the bound is $\leqslant k$, check by some dynamic programming method whether or not there really is an occurrence with at most k differences ending at that location. The dynamic programming can skip over the locations with bound $>k$.

Methods of this type are expected to be very fast in practice because the lower bounds can be computed in time $O(|T|)$, and because the dynamic programming can be largely avoided, at least when k is relatively small. Other algorithms with similar hybrid structure have recently been proposed in [2, 10, 21].

2. The q-gram distance

The first string-distance measure is based on counting the number of the occurrences of different q-grams in the two strings; the strings are the closer relatives the more they have q-grams in common.

Let Σ be a finite alphabet, and let Σ^* denote the set of all strings over Σ and Σ^q all strings of length q over Σ, for $q=1, 2, \ldots$ A *q-gram* is any string $v=a_1 a_2 \ldots a_q$ in Σ^q.

Definition. Let $x=a_1 a_2 \ldots a_n$ be a string in Σ^*, and let v in Σ^q be a q-gram. If $a_i a_{i+1} \ldots a_{i+q-1}=v$ for some i, then x has an *occurrence* of v. Let $G(x)[v]$ denote the total number of the occurrences of v in x. The *q-gram profile* of x is the vector $G_q(x)=(G(x)[v])$, $v\in\Sigma^q$.

The distance between two strings is defined as the L_1 norm of the difference of their q-gram profiles.

Definition. Let x, y be strings in Σ^*, and let $q>0$ be an integer. The *q-gram distance* between x and y is

$$D_q(x, y)=\sum_{v\in\Sigma^q} |G(x)[v]-G(y)[v]|. \tag{1}$$

Example. Let $x=01000$ and $y=001111$ be strings in the binary alphabet. Their 2-gram profiles are, listed in the lexicographical order of the 2-grams, (2, 1, 1, 0) and (1, 1, 0, 3), and their 2-gram distance is 5.

It is an easy exercise to prove the following properties of the q-gram distance. The length of a string x is denoted $|x|$.

Theorem 2.1. *For all x, y, z in Σ^*,*

(i) $D_q(x, y)=D_q(y, x)$;

(ii) $D_q(x, y)\leqslant D_q(x, z)+D_q(z, y)$;

(iii) $|(|x|\dot{-}(q-1))-(|y|\dot{-}(q-1))|\leqslant D_q(x, y)\leqslant(|x|\dot{-}(q-1))+(|y|\dot{-}(q-1))$,

where $r\dot{-}s=\max(0, r-s)$;

(iv) $D_q(x_1 x_2, y_1 y_2)\leqslant D_q(x_1, y_1)+D_q(x_2, y_2)+2(q-1)$;

(v) *If h is a non-length-increasing homomorphism on Σ^*, then $D_q(h(x), h(y))\leqslant D_q(x, y)$.*

By the properties (*i*) and (*ii*), the q-gram distance is a pseudometric. It is not a metric as $D_q(x, y)$ can be 0 even if $x\neq y$. This is the case if x and y have identical q-gram profiles.

Let x be a string and $q\leqslant|x|$. Let $Z_q(x)=\{y\in\Sigma^*: D_q(x, y)=0\}$ be the set of strings with the same q-gram profile as x. The next theorem gives a few observations on the structure of $Z_q(x)$; the proof is left to the reader.

Theorem 2.2. (i) *Let $x=a_1 a_2 \ldots a_n$ for some a_i in Σ. Then $Z_1(x)$ consists of all permutations of $a_1, a_2, \ldots, a_n$.*

(ii) *All strings in $Z_q(x)$ are of length $|x|$.*

(iii) *If x contains at most one occurrence of each $(q-1)$-gram then $|Z_q(x)|=1$.*

Given a string y in $Z_q(x)$, a new string in $Z_q(x)$ can be found by either of the following two transformations on y:

1. (transposition) If y can be written as $y=y_1z_1y_2z_2y_3z_1y_4z_2y_5$ for some $(q-1)$-grams z_1 and z_2 and for some strings $y_1,\ldots,y_5$, then the string $y_1z_1y_4z_2y_3z_1y_2z_2y_5$ where y_2 and y_4 have changed places, is also in $Z_q(x)$. If $y=y_1zy_2zy_3zy_4$, where z is a $(q-1)$-gram, then also $y_1zy_3zy_2zy_4$ is in $Z_q(x)$.
2. (rotation) If y can be written as $y=z_1y_1z_2y_2z_1$ for some $(q-1)$-grams z_1 and z_2 and for some strings y_1 and y_2, then also $z_2y_2z_1y_1z_2$ is in $Z_q(x)$.

A large part of $Z_q(x)$ can obviously be generated from x by repeatedly applying rules 1 and 2. Whether or not it is possible to generate the whole $Z_q(x)$ in this way remains open.

It is not difficult to see that $D_q(x, y)$ can be evaluated in time linear in $|x|$ and $|y|$. The main task is to compute fast the nonzero part of the q-gram profiles of x and y. We will present two alternative methods, both based on well-known techniques.

First method. In general we can not assume that a q-gram v as such could serve as an index. Rather, we need an encoding of v as an integer. A natural encoding is to interpret v directly as a c-ary integer, where $c=|\Sigma|$.

Let $v=b_1b_2\ldots b_q$, and let $\Sigma=\{A_0, A_1,\ldots, A_{c-1}\}$. Then the *integer code* of the q-gram v is

$$\tilde{v}=\tilde{b}_1c^{q-1}+\tilde{b}_2c^{q-2}+\cdots+\tilde{b}_qc^0,$$

where $\tilde{b}_i=j$ if $b_i=A_j$. Then, let $x=a_1\ldots a_n$, and let $v_i=a_i\ldots a_{i+q-1}$, $1\leqslant i\leqslant n-q+1$, be the q-grams of x. Obviously,

$$\tilde{v}_{i+1}=(\tilde{v}_i-\tilde{a}_i\cdot c^{q-1})\cdot c+\tilde{a}_{i+q}. \tag{2}$$

By setting $\tilde{v}_1=\sum_{i=1}^{q}\tilde{a}_ic^{q-i}$ and then applying (2) for $1\leqslant i\leqslant n-q$ we get the integer codes for all q-grams of x (cf. [12]). Simultaneously, we count the number of the occurrences of each q-gram in an array $G[0:c^q-1]$ by setting $G[\tilde{v}_i]\leftarrow G[\tilde{v}_i]+1$ for all i. At the end $G[\tilde{v}]=G(x)[v]$ for each q-gram v. Moreover, we create a list L of codes that really occur in x. Assuming that each application of (2) takes constant time (this holds true at least for small c and q), the total time for computing G and L is $O(|x|)$. Let us denote these G and L as G_1 and L_1.

Similarly, we get $G=G_2$ and $L=L_2$ for a string y in time $O(|y|)$.

Now (1) gets the form $D_q(x, y)=\sum_{\tilde{v}\in L_1\cup L_2}|G_1[\tilde{v}]-G_2[\tilde{v}]|$ which, obviously, can be evaluated in time $O(|x|+|y|)$.

Theorem 2.3. *The q-gram distance $D_q(x, y)$ can be evaluated in time $O(|x|+|y|)$ and in space $O(|\Sigma|^q+|x|+|y|)$.*

The $O(|\Sigma|^q)$ space requirement for tables G_1 and G_2 can, for large Σ and q, limit the applicability of this very simple algorithm. As only at most $|x|+|y|-2(q-1)$ elements of tables G_1 and G_2 are active and hence the tables are often sparse, their space requirement can be reduced by standard hashing techniques without increasing the (expected) running time of the method.

Second method. Next we represent a space-efficient but more complicated method that does not use hashing or integer arithmetic. The method codes only the relevant q-grams with small numbers, that are found using suffix automata [1, 4]; suffix trees [26, 17] could be used as well.

Let $Gr(x)$ be the set of different q-grams of x. Equation (1) can be written as

$$D_q(x, y) = \sum_{v \in Gr(x)} |G(x)[v] - G(y)[v]| + \sum_{v \in \Sigma^q - Gr(x)} G(y)[v]. \tag{3}$$

For evaluating $D_q(x, y)$ it therefore suffices to know the q-gram profile of x and y restricted to $Gr(x)$, and the total number of q-gram occurrences of y such that the corresponding q-gram does not occur in x.

Let $Gr(x) = \{v_1, \ldots, v_r\}$. For each q-gram v in Σ^q, the *code of v with respect to x* is $\bar{v} = i$, if $v = v_i$, and $\bar{v} = 0$, otherwise. Hence, there are $r + 1 \leqslant |x| - q + 2$ different codes.

The codes with respect to x for the q-grams of any string can be found by scanning the string with a modified suffix automaton for x. We first recall some properties of suffix automata. The suffix automaton $SA(x)$ for x is a minimal deterministic finite-state automaton recognizing all the suffixes of x, extended with some extra transitions. It consists of initial state (*Start*), other states, the *goto* function defining the *goto* transitions between the states on different symbols in Σ, and the *fail* function defining a *fail* transition for each state different from the start state.

The extra transitions include $goto(Start, a) = Start$ for each a in Σ that does not appear in x, and all the *fail* transitions. A *fail* transition is followed if the current state has no *goto* transition on the next input symbol; such a transition does not consume any input symbol. The extra transitions extend the language accepted such that $SA(x)$ accepts all strings yz, where z is a nonempty suffix of x. Hence, $SA(x)$ has the next property.

Lemma 2.4. *If $SA(x)$ is in state s after scanning a string u then $s = goto(Start, z)$, where string z is the longest suffix of u that is also a factor (substring) of x.*

For each state s, $depth(s)$ denotes the length of the longest acyclic *goto* path in $SA(x)$ from *Start* to s. The *fail* transitions have the following property.

Lemma 2.5 (Crochemore [5]). *Let y be a substring of x, and let s be a state of $SA(x)$, $s \neq Start$, such that $goto(Start, y) = s$. Let w be the longest suffix of y such that $s \neq goto(Start, w)$. Then, $goto(Start, w) = fail(s)$ and $|w| = depth(fail(s))$.*

Crochemore [5] (also [1]) gives an algorithm that constructs $SA(x)$ in time and in space $O(|x||\Sigma|)$.

Next we modify $SA(x)$ such that when it scans a string $b_1 b_2 \ldots b_m$, it outputs the codes $\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_{m-q+1}$ with respect to x for the q-grams $w_i = b_i b_{i+1} \ldots b_{i+q-1}$. The resulting automaton is denoted $SA_q(x)$.

Let s and z be as in Lemma 2.4. We call $d=|z|$ the *depth of the execution* at s. The depth depends on the string scanned and is not determined only by s as there can be several *goto* paths of different lengths from *Start* to s.

To get $SA_q(x)$, we change the execution of $SA(x)$ such that the depth d is as large as possible but stays $\leqslant q$. Assume that $d \leqslant q$ at state s. If $d=q$, we reduce d to $q-1$ before reading the next input symbol. This is done as follows: If $depth(fail(s))=q-1$, then d is reduced to $q-1$ by taking the *fail* transition from s. Otherwise, we know by Lemma 2.5 that both $goto(Start, z)=s$ and $goto(Start, z')=s$, where z is the q characters long suffix of the scanned string and z' is the $q-1$ characters long suffix of z. Hence, we can adopt depth $d=q-1$ at state s as well (in a way, we just ignore the first symbol of z). After this the next input symbol is processed normally (follow *fail* transitions until a state with a *goto* transition for the next input is entered, then take this *goto* transition), which increases d at most by one. Hence, the depth of the execution is $\leqslant q$ also in the new state entered by reading the next input.

Algorithm 2.6 gives the modified form of the execution.

Algorithm 2.6 The execution of $SA_q(x)$ for input $b_1 b_2 \ldots b_m$.

1. $s \leftarrow Start$; $d \leftarrow 0$;
2. for $i \leftarrow 1, \ldots, m$ do
3. if $depth(fail(s))=q-1$ then
   $s \leftarrow fail(s)$; $d \leftarrow q-1$
4. else if $d=q$ then $d \leftarrow q-1$
5. while $goto(s, b_i)$ undefined do
   $s \leftarrow fail(s)$; $d \leftarrow depth(s)$
6. $s \leftarrow goto(s, b_i)$
7. if $s=Start$ then $d \leftarrow 0$ else $d \leftarrow d+1$
8. if $i \geqslant q$ then $output(code(s))$
9. enddo

Automaton $SA_q(x)$ behaves as described by Algorithm 2.6. It satisfies (as can be proved by induction) the following counterpart of Lemma 2.4.

Lemma 2.7. *If $SA_q(x)$ is in a state s after scanning a string u then $s=goto(Start, z)$, where z is the longest suffix of length $\leqslant q$ of u that is also a factor of x. Moreover, the execution depth d equals $|z|$ at s.*

By the lemma, whenever $SA_q(x)$ is in a state s and d equals q, an occurrence of the unique q-gram z of x such that $goto(Start, z)=s$ has been recognized; note that the (unmodified) $SA(x)$ can be in different states after scanning z, depending on the left context of z. Hence, $SA_q(x)$ should output the code of z, stored in $code(s)$ (line 8 of Algorithm 2.6). The *code* fields (they are not present in $SA(x)$) are set by scanning string x itself by $SA_q(x)$. When a state s is entered with execution depth $d=q$, the next

free code number from 1, 2, ..., r is assigned to $code(s)$ if $code(s)$ has not been defined yet. All states s that do not get a *code* value in this way finally get $code(s)=0$.

Let v_i be the ith q-gram of x, in the left-to-right order of the first occurrence of the q-grams in x. Then, obviously, the above procedure gives to v_i the code $\bar{v}_i=i$, as $code(goto(Start, v_i))=i$.

Distance $D_q(x, y)$ can now be evaluated by Algorithm 2.8.

Algorithm 2.8 Evaluation of $D_q(x, y)$ with $SA_q(x)$.

1. Construct $SA_q(x)$.
2. Evaluate an array $G_1[1:r]$ representing the q-gram profile of x: scan x with SA_q and accumulate the distribution of the q-gram codes of x into G_1 such that finally for each $v \in Gr(x)$, $G_1[\bar{v}]=G(x)[v]$.
3. Evaluate an array $G_2[0:r]$ representing the q-gram profile of y, restricted to the q-grams of x: scan y with $SA_q(x)$ and accumulate the distribution of the q-gram codes of y into G_2 such that finally for each $v \in Gr(x)$, $G_2[\bar{v}]=G(y)[v]$, and $G_2[0]$ equals the total number of q-grams in y that do not occur in x.
4. Evaluate $D_q(x, y)$ from (3): $D_q(x, y)=\sum_{i=1}^{r}|G_1[i]-G_2[i]|+G_2[0]$.

Step 1 of Algorithm 2.8 takes time and space $O(|x||\Sigma|)$. Step 2 needs time $O(|x|)$ and step 3 time $O(|y|)$. Step 4 requires time $O(|x|)$ as $r=O(|x|)$.

Theorem 2.9. *Algorithm* 2.8 *evaluates the q-gram distance* $D_q(x, y)$ *in time* $O(|x||\Sigma|+|y|)$ *and in space* $O(|x||\Sigma|)$.

If the transitions of the automaton $SA_q(x)$ in Algorithm 2.8 have to be implemented as a balanced search tree at each state instead of direct indexing over Σ, then the time bound in Theorem 2.9 becomes $O((|x|+|y|)\log|\Sigma|)$ and the space bound $O(|x|\log|\Sigma|)$.

3. String-matching with the q-gram distance

In this section we consider the problem of finding the approximate occurrences of a pattern in a text when the approximation quality is measured with the q-gram distance.

Definition. Let $T=t_1t_2\ldots t_n$ be the *text* and $P=p_1p_2\ldots p_m$ the *pattern*, and let q be an integer, $0<q\leqslant m$. Both T and P are strings in the alphabet Σ. Let d_i be the minimum q-gram distance between P and the substrings of T ending at t_i, i.e. $d_i=\min_{1\leqslant j\leqslant i+1} d_i(j)$, where $d_i(j)=D_q(P, t_j\ldots t_i)$. Moreover, let s_i be the starting location of the *longest* substring of T that gives d_i, i.e. s_i is the smallest j such that

$d_i = d_i(j)$, $1 \leqslant j \leqslant i+1$. (The requirement of the longest string is only to make the problem well-defined; we could require the shortest string as well.) The *approximate string-matching problem with the q-gram distance* is to find (d_i, s_i) for $1 \leqslant i \leqslant n$.

The following simple properties of d_i and s_i are useful. First, we have

$$0 \leqslant d_i \leqslant m-q+1 \tag{4}$$

as the q-gram distance is nonnegative by definition, and $d_i(i-q+2) = m-q+1$ because $t_{i-q+2} \ldots t_i$ contains no q-grams but P contains $m-q+1$ of them. Moreover,

$$i-2m+q \leqslant s_i \leqslant i-q+2. \tag{5}$$

The upper bound holds true because always $d_i(j) = m-q+1$ for $i-q+2 \leqslant j \leqslant i+1$, and s_i is the *smallest* starting point of a substring giving d_i. For the lower bound, we note first that $t_j \ldots t_i$ contains more than $2m-2q+2$ q-grams when $j < i-2m+q$. More than $m-q+1$ of them can not occur in P because P has only $(m-q+1)$ q-grams. Therefore, $d_i(j) > m-q+1$ when $j < i-2m+q$. On the other hand, $d_i(i-q+2) = m-q+1$, and the lower bound follows.

To evaluate d_i it suffices by (5) to find the minimum of $d_i(j)$ for $i-2m+q \leqslant j \leqslant i-q+2$. These values $d_i(j)$ clearly satisfy the recursion.

$$d_i(j-1) = \begin{cases} d_i(j)-1 & \text{if the number of the occurrences of the } q\text{-gram } v = t_{j-1} \ldots t_{j+q-2} \text{ in } t_j \ldots t_i \\ & \text{is } < G(P)[v], \\ d_i(j)+1 & \text{otherwise,} \end{cases} \tag{6}$$

where the starting value is given by $d_i(i-q+2) = m-q+1$. This rule simply says that the q-gram v at t_{j-1} makes the distance $d_i(j-1)$ smaller (compared to $d_i(j)$) if the number of the occurrences of v in $t_j \ldots t_i$ is not as large as in P; otherwise, v makes the distance larger.

In (6) we need to know for every q-gram v of T whether or not v occurs in P. This information is provided by scanning T with the automaton $SA_q(P)$, whose construction was described in Section 2.

Let the different q-grams of P be $w_1, w_2, \ldots, w_M$, in the order of the first occurrence in P. When $SA_q(P)$ has scanned w_h, it outputs code h. A q-gram not in P gets code 0. We denote by τ_j the code of the q-gram $t_j t_{j+1} \ldots t_{j+q-1}$ (the jth q-gram of T), $1 \leqslant j \leqslant n-q+1$.

The q-gram profile $G_q(P)$ of P is represented as a table $G[0:M]$ such that $G[0] = 0$ and for $h > 0$, $G[h] = G(P)[w_h]$. The construction of $SA_q(P)$ also produces table G.

Algorithm 3.1 is a straightforward implementation of recursion (6) for computing (d_i, s_i). The algorithm uses table $C[0:M]$ for counting the occurrences of the q-grams of P in $t_j \ldots t_i$ for $j = i-q+1, \ldots, i-2m+q$. Function *Scan* makes $SA_q(P)$ to scan the

next text symbol t_i and to return the code of the q-gram ending at t_i; if $i<q$, then the automaton returns 0.

Algorithm 3.1 Evaluation of (d_i, s_i), $1 \leqslant i \leqslant n$, for text $T=t_1 \ldots t_n$ and pattern $P=p_1 \ldots p_m$.

```
 1. Construct automaton SA_q(P) and q-gram profile G[0:M] (=G_q(P))
 2. for i←1−q downto 1−2m+q do τ_i←0
 3. for i←1,...,n do
 4.    d←D←m−q+1
 5.    C[0:M]←0; S←0
 6.    τ_{i−q+1}←Scan(SA_q(P), t_i)
 7.    for j←i−q+1 downto i−2m+q do
 8.       if C[τ_j]<G[τ_j] then
 9.          d←d−1; C[τ_j]←C[τ_j]+1
10.       else d←d+1
11.       if d≤D then
12.          S←j; D←d
13.    enddo
14.    (d_i, s_i)←(D, S)
15. enddo
```

Theorem 3.2. *Given a pattern P of length m, a text T of length n, and an integer $q>0$, Algorithm* 3.1 *solves the approximate string-matching problem with the q-gram distance function in time* $O(m|\Sigma|)$ *for preprocessing P, and in time* $O((m-q)n)$ *for processing T. The algorithm needs working space* $O(m|\Sigma|)$.

Proof. The construction of $SA_q(P)$ and $G[0:M]$ (line 1) takes time $O(m|\Sigma|)$ by the methods of Section 2. The time for each repetition of the main loop (lines 3–15) is dominated by the initialization of array C, which takes time $O(m-q)$ because $M \leqslant m-q+1$, and by the loop (lines 7–13) which takes time $O(m-q)$, too. Hence, the total running time is $O((m-q)n)$ for the main loop; this includes the total time $O(n)$ for scanning T (line 6). Time $O(m|\Sigma|+(m-q)n)$ for the whole algorithm follows.

A working space of $O(m|\Sigma|)$ is needed for $SA_q(P)$ and $O(m)$ for tables G and C. Codes τ_j seemingly require $O(n)$ space. Fortunately, for each i the code τ_j is needed only for $i-2m+q \leqslant j \leqslant i-q+1$ (the $2(m-q+1)$ latest values). Hence, a buffer of size $O(m)$ suffices for storing the relevant values τ_j. □

Next we develop a method, based on balanced search trees, that implements the innermost minimization loop (lines 7–13) of Algorithm 3.1 in time $O(\log(m-q))$.

We write (6) as

$$d_i(j-1)=d_i(j)+h_i(j-1),$$

where $h_i(j-1)$ is $+1$ or -1, as explained in (6). Then solving (6) gives $d_i(j)=m-q+1+H_i(j)$, where $H_i(j)=\sum_{k=j}^{i-q+1} h_i(k)$. To find d_i we have to find the minimum of the values $H_i(j)$, $i-2m+q \leqslant j \leqslant i-q+1$, for $i=1, 2, \ldots, n$. We will do this by maintaining a balanced binary search tree.

At the moment when the minimal $H_i(j)$ can be read from the tree for some i, the tree has $2(m-q+1)$ leaves, that represent, from left-to-right, the numbers $h_i(j)$, $i-2m+q \leqslant j \leqslant i-q+1$. Hence, the leftmost leaf stores $h_i(i-2m+q)$, the rightmost leaf stores $h_i(i-q+1)$, and the leaves of the subtree rooted at some node v store an interval of values $h_i(j)$.

Each node v has the normal *llink*, *rlink* and *father* fields. An explicit search key is not needed; logically the key for a leaf λ storing $h_i(j)$ is j. A direct access to λ is given by *Leaf*(j).

Node v has also three special fields: *sum*, *min*, and *minindex*. They are defined as follows. Assume that the leaves of the subtree rooted at v are nodes *Leaf*(j), $j_1 \leqslant j \leqslant j_2$ for some $j_1 \leqslant j_2$. Then

$$\begin{cases} sum(v)=\sum_{j=j_1}^{j_2} h_i(j) \\ min(v)=\min_{j_1 \leqslant k \leqslant j_2+1} \sum_{j=k}^{j_2} h_i(j) \\ minindex(v)=\text{smallest } k,\ j_1 \leqslant k \leqslant j_2+1, \text{ that gives } min(v) \text{ above.} \end{cases} \tag{7}$$

Hence, *sum* stores the total sum of the values $h_i(j)$ in the leaves of the subtree, *min* stores the smallest partial sum of these values when summed up from right to left, and *minindex* stores the left end point of the longest interval giving the smallest sum.

Given the *sum*, *min*, and *minindex* values for nodes $v_r=rlink(v)$ and $v_l=llink(v)$ that satisfy (7), it is immediate that the following rules give these values for v such that (7) is again satisfied.

$$\begin{cases} sum(v)=sum(v_l)+sum(v_r) \\ min(v)=\min(sum(v_r)+min(v_l),\ min(v_r)) \\ minindex(v)=\text{if } sum(v_r)+min(v_l) \leqslant min(v_r) \text{ then} \\ \qquad\qquad minindex(v_l) \text{ else } minindex(v_r). \end{cases} \tag{8}$$

With rule (8) bottom-up building and updating of the tree is possible in constant time per node.

The root (*Root*) of the tree satisfying (7) gives the information we need, because $min(Root)=\min\{H_i(j):\ 1-2m+q \leqslant j \leqslant i-q+1\}$. Hence, $d_i=m-q+1+min(Root)$ and $s_i=minindex(Root)$.

Let B_i be the tree described above and $Root(B_i)$ its root; we call B_i the *tail sum tree* for sequence $(h_i(j))$, $i-2m+q \leqslant j \leqslant i-q+1$.

Tree B_{i+1} is obtained from B_i by a rather simple transformation. As the leaves of B_{i+1} represent sequence $(h_{i+1}(j))$, $i-2m+q+1 \leqslant j \leqslant i-q+2$, we have to find out how $h_{i+1}(j)$ differs from $h_i(j)$.

We call j the *change point* for $i+1$, denoted $j=cp(i+1)$, if the following three conditions are met (recall that τ_j is the code of the q-gram starting at t_j and $G[0:M]$ represents the profile $G_q(P)$).

1. $\tau_j > 0$;
2. there are exactly $G[\tau_j]$ occurrences of the q-gram $t_j \ldots t_{j+q-1}$ in $t_j \ldots t_i$;
3. $\tau_j = \tau_{i-q+2}$ (hence, there are $G[\tau_j]+1$ occurrences of $t_j \ldots t_{j+q-1}$ in $t_j \ldots t_{i+1}$).

Note that the change point is not always defined.

Numbers $h_{i+1}(j)$ and $h_i(j)$ differ only at the change point as we have

$$h_{i+1}(j) = \begin{cases} +1 & (=-h_i(j)) \text{ if } j=cp(i+1) \\ h_i(j) & \text{otherwise} \end{cases} \tag{9}$$

Rule (9) shows how the contribution $h_i(j)$ of τ_j to $d_i(j')$, $j' \leqslant j$, can differ from its contribution $h_{i+1}(j)$ to $d_{i+1}(j')$. If there are $G[\tau_j]$ occurrences of the q-gram $t_j \ldots t_{j+q-1}$ in $t_j \ldots t_i$ but $G[\tau_j]+1$ occurrences in $t_j \ldots t_{i+1}$, then τ_j contributes -1 to $d_i(j')$ but $+1$ to $d_{i+1}(j')$; only the $G[\tau_j]$ rightmost occurrences of $t_j \ldots t_{j+q-1}$ in $t_j \ldots t_{i+1}$ can decrease $d_{i+1}(j')$.

Change point $j=cp(i+1)$ can be found in constant time by using queues $L[1:M]$. Queue $L[\tau]$ contains indexes r, in increasing order, such that $\tau_r = \tau$, $h_i(r) = -1$, and tree B_i has a leaf representing $h_i(r)$. The size of $L[\tau]$ is given by $S[\tau]$. Obviously, if $\tau_{i-q+2} > 0$ and $S[\tau_{i-q+2}] = G[\tau_{i-q+2}]$ then $cp(i+1)$ is the head of $L[\tau_{i-q+2}]$ (i.e. the smallest index stored in this list); otherwise, $cp(i+1)$ is not defined. When $j=cp(i+1)$ is defined, we update the *sum* field of $Leaf(j)$ to $+1$ and remove j from $L[\tau_{i-q+2}]$.

The other changes to B_i include deleting the leftmost leaf $Leaf(i-2m+q)$, because it does not belong to B_{i+1}. The rightmost leaf of B_{i+1} should represent

$$h_{i+1}(i-q+2) = \begin{cases} -1 & \text{if } \tau_{i-q+2} > 0, \\ +1 & \text{otherwise.} \end{cases}$$

This has no predecessor in B_i. Therefore, we insert a new rightmost leaf, pointed by $Leaf(i-q+2)$.

Algorithm 3.3 summarizes the transformation of B_i into B_{i+1}.

Algorithm 3.3 Updating a tail sum tree B_i into B_{i+1}.

Delete(λ) removes leaf λ from B_i, rebalances the tree, and updates the nodes with (8);

Change(λ, h), where λ is a leaf, sets $sum(\lambda) \leftarrow h$, and updates the nodes on the path from λ to *Root* with (8);

Insert(j, h) creates a new rightmost leaf λ such that $Leaf(j) = \lambda$, sets $sum(\lambda) \leftarrow h$, rebalances the tree, and updates the nodes with (8);

Enqueue and *Dequeue* are the standard queue operations of adding a tail element and removing the head element.

1. *Delete*($Leaf(i-2m+q)$)
2. if $Head(L[\tau_{i-2m+q}])=i-2m+q$ then
3. $\quad Dequeue(L[\tau_{i-2m+q}])$
4. $\quad S[\tau_{i-2m+q}]\leftarrow *-1$
5. if $\tau_{i-q+2}>0$ and $S[\tau_{i-q+2}]=G[\tau_{i-q+2}]$ then
6. $\quad j\leftarrow Dequeue(L[\tau_{i-q+2}])$ $\quad$ % $j=cp(i+1)$
7. $\quad S[\tau_{i-q+2}]\leftarrow *-1$
8. $\quad Change(Leaf(j), 1)$
9. *Insert*($i-q+2$, if $\tau_{i-q+2}>0$ then -1 else 1)
10. *Enqueue*($L[\tau_{i-q+2}], i-q+2$)
11. $S[\tau_{i-q+2}]\leftarrow *+1$

The approximate string-matching problem can now be solved with Algorithm 3.4 that uses Algorithm 3.3 as a subroutine.

Algorithm 3.4 Evaluation of (d_i, s_i), $1\leqslant i\leqslant n$, for text $T=t_1\ldots t_n$ and pattern $P=p_1\ldots p_m$ using a tail sum tree.

1. Construct $SA_q(P)$ and $G[0:M]$
2. Construct the initial tail sum tree B_0, representing
   $h_0(j)=1$ for $-2m+q\leqslant j\leqslant -q+1$
3. for $i\leftarrow 0,\ldots,n-1$ do
4. $\quad \tau_{i-q+2}\leftarrow Scan(SA_q(P), t_{i+1})$
5. $\quad$ Update B_i to B_{i+1} with Algorithm 3.3
6. $\quad (d_{i+1}, s_{i+1})\leftarrow(min(Root(B_{i+1}))+m-q+1, minindex(Root(B_{i+1})))$
   enddo

Theorem 3.5. *Given a pattern P of length m, a text T of length n, and an integer $q>0$, Algorithm* 3.4 *solves the approximate string-matching problem with the q-gram distance function in time* $O(m|\Sigma|)$ *for preprocessing P and in time* $O(n\log(m-q))$ *for processing T. The algorithm needs working space* $O(m|\Sigma|)$.

Proof. The preprocessing phase is the same as for Algorithm 3.1 and, hence, needs time and space $O(m|\Sigma|)$. The initial tail sum tree (a balanced binary tree with $2m-2q+1$ leaves) can be constructed by standard methods in time and space $O(m-q)$.

The main loop makes n calls to Algorithm 3.3. Each call includes one deletion, at most one change, and one insertion to a balanced binary tree of height $O(\log(m-q))$. Each operation also includes re-establishing conditions (8). All this can be performed

in $O(\log(m-q))$ time if our tail sum tree is implemented by augmenting some standard balanced tree scheme such as the red-black trees (see e.g. [3, Theorem 15.1]). The operations on queues L in Algorithm 3.3 clearly take time $O(1)$. The total time for scanning T (line 4) is again $O(n)$. Hence, the main loop takes time $O(n \log(m-q))$.

A working space of $O(m)$ suffices for the tail sum tree, the lists L, and the relevant values τ_i. □

The balanced tree in Algorithm 3.4 creates a considerable overhead compared to the simple Algorithm 3.1. However, the tree is used in a very restricted fashion: all deletions remove the leftmost leaf, all insertions create a rightmost leaf, and the size of the tree remains unchanged.

Therefore, it is rather easy to design a tailor-made tree structure with smaller overhead for this particular application. In this structure, the tail sum tree B is represented as a subtree of a larger balanced binary tree C that has twice as many leaves as B has. Tree B circularly glides over C when new leaves are inserted and old ones are deleted. The shape of C can be kept unchanged which means that no time-consuming rebalancing operations are needed. We leave the details of the construction as an exercise to the interested reader.

Next we consider a natural variation of the approximate string-matching problem.

Definition. The *threshold problem* is to find, for a given integer $k \geqslant 0$, all text locations i such that the q-gram distance d_i is $\leqslant k$.

Lemma 3.6. *If* $d_i \leqslant k$ *then* $i-m+1-k \leqslant s_i \leqslant i-m+1+k$.

Proof. To derive a contradiction, assume first that $s_i > i-m+1+k$. Then the string $t_{s_i} \ldots t_i$ contains $<(m-q+1-k)$ q-grams. Hence, more than k of the $(m-q+1)$ q-grams of P are missing, therefore $d_i > k$, a contradiction. The case $s_i < i-m+1-k$ is similar. □

By the lemma, one can test whether or not $d_i \leqslant k$ by computing $d = \min_{a(i)-k \leqslant j \leqslant a(i)+k} d_i(j)$, where $a(i)$ denotes $i-m+1$. If $d \leqslant k$, then we know that $d_i \leqslant k$ because $d_i = d$. If $d > k$, then also $d_i > k$. As d is the minimum of only $2k+1$ elements, it turns out that it can be found for each i in time $O(\log k)$. Only a minor modification to Algorithm 3.4 is needed.

Theorem 3.7. *Given a pattern* P *of length* m *and a text* T *of length* n *in alphabet* Σ, *and* $k \geqslant 0$, *the threshold problem can be solved in time* $O(m|\Sigma|)$ *for preprocessing* P *and* $O(n \log k)$ *for processing* T. *The working space requirement is* $O(m|\Sigma|)$.

Proof. Let $a(i) = i-m+1$. Then by Lemma 3.6, if $d_i \leqslant k$, we can write $d_i = F_i + H_i + m - q + 1$, where

$$H_i = \sum_{j=a(i)+k+1}^{i-q+1} h_i(j)$$

and

$$F_i = \min_{a(i)-k \leqslant s \leqslant a(i)+k} \sum_{j=s}^{a(i)+k} h_i(j).$$

Values F_i can be found in $O(\log k)$ time for each i by using a tail sum tree that represents $h_i(j)$ for $a(i)-k \leqslant j \leqslant a(i)+k$. Each H_i can be found in $O(1)$ time for each i because

$$H_{i+1} = H_i - h_i(a(i)+k+1) + h_{i+1}(i-q+2) + \Delta,$$

where $\Delta = 2$ if $cp(i+1) > a(i+1)+k+1$ and $\Delta = 0$, otherwise. Change point $cp(i+1)$ can be found in $O(1)$ time using queues L, as in Algorithm 3.4. □

Another variation of interest is to find the distances between P and substrings of T of a fixed length, i.e. we want to evaluate values $d_i(i-\delta)$ for some fixed $\delta \geqslant 0$.

A linear-time method, based on straightforward bookkeeping, is easy to develop. We formulate it as Algorithm 3.8; a similar method in the special case $q=1$ is given in [10].

Algorithm 3.8 Evaluation of $D_i = d_i(i-\delta)$, $\delta+1 \leqslant i \leqslant n$, for text $T = t_1 \ldots t_n$, and pattern $P = p_1 \ldots p_m$, and integer $\delta \geqslant 0$. We assume that $\delta+1 \geqslant q$; otherwise the problem is trivial as then each $D_i = m-q+1$.

1. Construct $SA_q(P)$ and $G[0:M]$
2. $Scan(SA_q(P), t_1 \ldots t_{q-1})$
3. for $i \leftarrow q, \ldots, \delta$ do
4. $\quad \tau_{i-q+1} \leftarrow Scan(SA_q(P), t_i)$
5. $\quad G[\tau_{i-q+1}] \leftarrow * - 1$

   enddo
6. $D \leftarrow |G[0]| + \cdots + |G[M]|$
7. for $i \leftarrow \delta+1, \ldots, n$ do
8. $\quad \tau_{i-q+1} \leftarrow Scan(SA_q(P), t_i)$
9. $\quad G[\tau_{i-q+1}] \leftarrow * - 1$
10. $\quad$ if $G[\tau_{i-q+1}] \geqslant 0$ then $D \leftarrow D-1$ else $D \leftarrow D+1$
11. $\quad D_i \leftarrow D$ % $D_i = d_i(i-\delta)$
12. $\quad G[\tau_{i-\delta}] \leftarrow * + 1$
13. $\quad$ if $G[\tau_{i-\delta}] \leqslant 0$ then $D \leftarrow D-1$ else $D \leftarrow D+1$

    enddo

In Algorithm 3.8, array G is initialized (line 1), as before, to represent the q-gram profile of P. During the main loop at line 11 G satisfies the invariant $G[\bar{v}] = G_q(P)[v] - G_q(t_{i-\delta} \ldots t_i)[v]$ for the q-grams v of P, and $G[0]$ counts (in negative) the number of other q-grams in $t_{i-\delta} \ldots t_i$. Moreover, D satisfies the invariant $D = \sum_{j=0}^{M} |G[j]|$. Hence, $D = D_q(t_{i-\delta} \ldots t_i, P)$, i.e. the algorithm is correct. Clearly, it needs time $O(n)$ for processing T.

Theorem 3.9. *Algorithm* 3.8 *evaluates the q-gram distances* $d_i(i-\delta)$ *for* $\delta+1 \leqslant i \leqslant n$ *in time* $O(n)$ *for processing* T. *The preprocessing time and working space are as in Theorem* 3.7.

4. String-matching with maximal matches

The string-matching problem studied in this section uses as the distance measure the minimum number of characters in one string that have to be removed so that the remaining substrings, between the removed characters, are also substrings of the second string. This measure and a string distance metric based on it were introduced by Ehrenfeucht and Haussler [7]. Our interest is mainly motivated by the application, to be presented in Section 5, of the methods of this section in speeding up the edit distance based string-matching.

Following [7] we set the next definitions. A nonempty string $x=a_1a_2\ldots a_n$ in Σ^* of length n has n *places*, the places from 1 to n. A *marking* of x is a (possibly empty) subset M of the places of x. The set $\{a_i: i\in M\}$ is called the set of the (M-) *marked characters* of x. A substring of x is either the empty string or a string $a_ia_{i+1}\ldots a_j$ for some i, j such that $1 \leqslant i \leqslant j \leqslant n$. For any string y, marking M is said to make x *compatible with* y if any substring of x that does not contain any M-marked characters is also a substring of y.

For x, y in Σ^*, let $Dc(x, y)$ be the *minimal* number of places in any marking of x that makes x compatible with y. Function Dc is the distance measure we use in this section.

Clearly, $0 \leqslant Dc(x, y) \leqslant |x|$ and $Dc(x, y)=0$ if and only if x is a substring of y.

Let $M_l(x, y)$ be the unique marking S of x such that $x=x_1s_1\ldots x_rs_rx_{r+1}$, where symbols s_i are the S-marked characters of x, all strings x_i occur as substrings in y but strings x_is_i are not substrings of y, $1 \leqslant i \leqslant r$.

Proposition 4.1 (Ehrenfeucht and Haussler [7]). *For any* x, y *in* Σ^*, $Dc(x, y)=|M_l(x,y)|$.

This gives a fast method for computing $Dc(x,y)$. The method "greedily" finds the maximal matches between x and y, in a left-to-right scan of x. First construct the suffix automaton $SA(y)$ in time $O(|y||\Sigma|)$. Then scan x with $SA(y)$ until no *goto*-transitions can be applied on the next input, say a. Then the scanned part of x is the longest prefix of x that is a substring of y. Hence, this part is x_1 in the above definition of $M_l(x,y)$, and $a=s_1$. The process is repeated with the remaining part of x beyond a until the whole x is scanned. This gives $M_l(x,y)$ in total time $O(|x|+|y||\Sigma|)$.

To define our approximate string-matching problem that is based on Dc, let again $P=p_1\ldots p_m$ and $T=t_1\ldots t_n$ be the pattern and the text. We want to find the Dc distances between the substrings of T of some fixed length, say e, and P. Hence, the problem is to compute distances $dc_i(i-e+1)=Dc(t_{i-e+1}t_{i-e+2}\ldots t_i, P)$ for $i=e, e+1, \ldots, n$.

By directly applying Proposition 4.1, value $dc_i(i-e+1)$ can be found by scanning $t_{i-e+1}\dots t_i$ with $SA(P)$. This gives $dc_i(i-e+1)$ in time $O(e)$; hence, all such values are obtained in time $O(en)$.

The method has a variant with smaller overhead which avoids repeated scanning of overlapping parts of T with $SA(P)$. We first evaluate values f_i denoting the length of the longest prefix of $t_i t_{i+1}\dots t_n$ that is also a substring of P, $1\leqslant i\leqslant n$. All values f_i can be found by scanning T only once with $SA(P)$ in time $O(n)$, see [24]. Then the first place in the marking $M_l(t_{i-e+1}\dots t_i, P)$ is $i-e+1+f_{i-e+1}$. Generally, if j is in the marking then $j+1+f_{j+1}$ will be there, too, as long as $j+1+f_{j+1}\leqslant i$. Finding the marking still takes time $O(e)$, giving the total time $O(en)$.

This method additionally gives an efficient solution to the threshold version of the problem in which we are asked to evaluate $dc_i(i-e+1)$ accurately only if its value is $\leqslant k$ for some fixed $k\geqslant 0$. In that case the construction of the marking $M_l(t_{i-e+1}\dots t_i, P)$ can be finished immediately when its size exceeds k. Hence, the time requirement is $O(k)$ per marking and $O(kn)$ for the whole algorithm.

Next we develop a useful approximate method with running time $O(n)$. We use the global marking $M_l(T,P)$ of T. From it the values $dc_i(i-e+1)$ can be read with an error at most 1.

Let $M(i,j)=\{r\in M_l(T,P)\colon i\leqslant r\leqslant j\}$ be the restriction of $M_l(T,P)$ to interval $[i,j]$. Clearly, marking $M(i,j)$ makes $t_i\dots t_j$ compatible with P. Define $dc'_i(i-e+1)=|M(i-e+1,i)|$.

Theorem 4.2. $dc_i(i-e+1)\leqslant dc'_i(i-e+1)\leqslant dc_i(i-e+1)+1$.

Proof. Let the elements of $M_l(t_{i-e+1}\dots t_i, P)$ be $(j_1,\dots,j_r)$ in increasing order. Since the marking $M(i-e+1,i)$ makes $t_{i-e+1}\dots t_i$ compatible with P and it is a restriction of a minimal marking, it must contain for each $1\leqslant k\leqslant r$ exactly one element j such that $j_{k-1}<j\leqslant j_k$. Moreover, because of minimality there is at most one element of $M(i-e+1,i)$ larger than j_r. Hence, the theorem follows. □

The evaluation of $dc'_i(i-e+1)$ reduces to the evaluation of the size of $M(i-e+1,i)$. Let C_k denote the number of the elements j in $M_l(T,P)$ such that $j\leqslant k$. Then $dc'_i(i-e+1)=|M(i-e+1,i)|=C_i-C_{i-e}$. Numbers C_k can be evaluated together with the construction of $M_l(T,P)$; actually we need only the numbers C_k, not an explicit $M_l(T,P)$. Marking $M_l(T,P)$ starts with f_1+1, and generally, if it contains j it also contains $j+f_{j+1}+1$ provided that this value is $\leqslant n$.

Summarized, we get the following procedure for evaluating numbers C_k.

```
C←0; j←0
repeat
  J←j+f_{j+1}+1
  for k←j,...,J−1 do C_k←C
  C←C+1
  j←J
until j=n+1.
```

This algorithm clearly runs in $O(n)$ time because the values f_j can be provided by scanning T with $SA(P)$ in time $O(n)$.

Finally, we get each $dc'_i(i-e+1)=C_i-C_{i-e}$ in constant time. By a careful programming, a buffer of size $O(e)$ suffices for the relevant part of vectors C and f. Additional working space $O(m|\Sigma|)$ is needed for $SA(P)$.

We have obtained Theorem 4.3.

Theorem 4.3. *Given a pattern P of length m, a text T of length n, and an integer e, the distances $dc'_i(i-e+1)$ for $e\leqslant i\leqslant n$ can be evaluated in time $O(n)$ and in space $O(m|\Sigma|+e)$. The preprocessing time of P is $O(m|\Sigma|)$.*

So, we have a linear-time algorithm for finding a very good approximation for values $dc_i(i-e+1)$. Finding such an accurate algorithm remains open.

5. Speeding-up edit distance based string-matching

The edit *distance* $DE(x,y)$ between strings x and y is defined as the minimum possible number of editing steps that convert x into y [16, 25]. We restrict our consideration to the case where each editing step is a rewriting rule of the form $a\to\varepsilon$ (a deletion), $\varepsilon\to a$ (an insertion), or $a\to b$ (a change) where a, b in Σ are any symbols, $a\neq b$, and ε is the empty string. Each symbol of x is allowed to be rewritten by some editing operation at most once.

The associated approximate pattern-matching problem with threshold is, given pattern P and text T as before, and an integer $k\geqslant 0$, to find all i such that the minimum of the edit distances between P and the substrings of T ending at t_i is $\leqslant k$; the problem is also known as the *k differences problem*. Hence, if we let $de_i=\min_{1\leqslant j\leqslant i} DE(P,t_j\ldots t_i)$, the problem is to find all i such that $de_i\leqslant k$.

Now, let $d_i(j)=D_q(P,t_j\ldots t_i)$ be as in Section 3, for some $q>0$.

Theorem 5.1. *For $1\leqslant i\leqslant n$, $d_i(i-m+1)/(2q)\leqslant de_i$.*

Proof. Let $d=de_i$. The theorem follows if we show that at most dq of the q-grams of P are missing in $t_{i-m+1}\ldots t_i$ (here $t_j=\varepsilon$ if $j<1$), as then $d_i(i-m+1)\leqslant 2dq$.

Let P' be the substring of T ending at t_i such that $DE(P,P')=d$. String P' can be obtained from P with at most d insertions, deletions and 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 $d_1q+d_2(q-1)$ q-grams of P are missing in P', where d_1 is the total number of deletions and changes, and d_2 is the total number of insertions. As $|P'|\leqslant m+d_2$, string $t_{i-m+1}\ldots t_i$ contains all q-grams of P' except for at most d_2. Hence, at most $d_1q+d_2(q-1)+d_2=dq$ of the q-grams of P are not present in $t_{i-m+1}\ldots t_i$, which proves the theorem. □

A similar proof shows that we also have $d_i/(2q) \leqslant de_i$, where $d_i = \min_{j \leqslant i} d_i(j)$, and that for all strings x, y, we have $D_q(x,y)/(2q) \leqslant DE(x,y)$.

Values de_i can be found by evaluating table (D_{ji}), $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, where $D_{ji} = \min_{1 \leqslant i' \leqslant i} DE(t_{i'} \ldots t_i, p_1 \ldots p_j)$ from recursion $D_{ji} = \min\{D_{j-1,i}+1, D_{j,i-1}+1$, if $p_j = t_i$ then $D_{j-1,i-1}$ else $D_{j-1,i-1}+1\}$ with initial conditions $D_{0,i}=0$ for $0 \leqslant i \leqslant n$ and $D_{j,0}=j$ for $1 \leqslant j \leqslant m$. This dynamic programming method solves the problem as $de_i = D_{mi}$, see [19, 22]. In particular, the k differences problem can be solved in $O(kn)$ time by a "diagonal" modification of this method [9, 24]. Such algorithms can be speeded-up by the following hybrid method.

First, evaluate by Algorithm 3.8 the distances $d_i(i-m+1)$ for $1 \leqslant i \leqslant n$. Mark all i such that $d_i(i-m+1)/(2q) \leqslant k$. Then evaluate de_i only for the marked i. As only such de_i can be $\leqslant k$ by Theorem 5.1, we get the solution of the k differences problem. By Theorem 3.9, the marking phase takes time $O(n)$, with $O(m|\Sigma|)$ time for preprocessing P.

A *diagonal* e of (D_{ji}) consists of entries D_{ji} such that $i-j=e$. If $de_i \leqslant k$, it is easy to see that to find $de_i = D_{mi}$ correctly, it suffices to restrict the evaluation of (D_{ji}) to $2k+1$ successive diagonals with the diagonal of D_{mi} in the middle. The simplest way to do this is just to apply the above recursion for (D_{ji}), restricted to these diagonals. This takes time $O(km)$.

An asymptotically faster method is obtained by restricting the algorithm of [9] or [24] to the $2k+1$ diagonals. This gives a method requiring time $O(k^2)$ for the evaluation of the diagonals. It also needs time $O(m^2)$ for preprocessing P (this can be improved to $O(m)$ by using lowest common ancestor algorithms). Moreover, the method scans $t_{i-m-k+1} \ldots t_i$. However, we have to scan over this string anyway when marking i.

If the relevant diagonals for different marked i overlap, their evaluation can be combined such that each diagonal is evaluated at most once. As the method spends time $O(k)$ per diagonal, the total time for evaluating the marked values de_i is always $O(kn)$. The marking and dynamic programming phases can be combined so that repeated scanning of T is not necessary.

Summarizing, we have the following result.

Theorem 5.2. *The k differences problem can be solved in time $O(\min(n+rk^2, kn))$, where r is the number of indexes i such that $d_i(i-m+1)/(2q) \leqslant k$. The method needs time $O(m|\Sigma|+m^2)$ for the preprocessing of P.*

Grossi and Luccio [10] propose a similar method for the special case where $q=1$ and the only editing operation used in the definition of the edit distance is the change (the *k mismatches problem*).

The string distance measure of Section 4 can be used in the above scheme as well. Let $dc_i(j)$ and $dc'_i(j)$ be as in Theorem 4.2.

Theorem 5.3. *For $1 \leqslant i \leqslant n$, $dc'_i(i-m+1+de_i)-1 \leqslant de_i$.*

Proof. Let $d=de_i$, and let P' be the substring of T ending at t_i such that $DE(P,P')=d$. As P' can be obtained from P in d editing steps, P' can be written as $\pi_1 a_1 \pi_2 a_2 \ldots a_g \pi_{g+1}$, where each a_j is the result of an insertion or a change, or one or more symbols immediately to the left of (the original version of) a_i in P has been removed by a deletion. Each π_j, $1 \leqslant j \leqslant g+1$, is either empty or a substring of P because no editing operation has been applied on it. Hence, a marking of $a_1,\ldots,a_g$ makes P' compatible with P. Therefore $Dc(P',P) \leqslant g \leqslant d$. As $|P'| \geqslant m-d$, then also $dc_i(i-m+1+d) \leqslant d$. Then, by Theorem 4.2, $dc'_i(i-m+1+d)-1 \leqslant d$, as required. □

To evaluate values de_i that are $\leqslant k$, we first evaluate $dc'_i(i-m+1+k)$ for $1 \leqslant i \leqslant n$. By Theorem 4.3, this can be done in time $O(n)$, with $O(m|\Sigma|)$ preprocessing time for P. If the value is $\leqslant k+1$, index i is marked. We claim that then every i such that $de_i \leqslant k$ will get a mark. To see this, assume that $de_i \leqslant k$. Then by Theorem 5.3, $dc'_i(i-m+1+de_i)-1 \leqslant k$. As $dc'_i(i-m+1-de_i) \geqslant de'_i(i-m+1+k)$, it follows that $dc'_i(i-m+1+k) \leqslant k+1$, i.e. i will be marked.

For marked i, values de_i are then evaluated by dynamic programming as in the previous method. Hence, we have the following theorem.

Theorem 5.4. *The k differences problem can be solved in time $O(\min(n+r'k^2, kn))$, where r' is the number of indexes i such that $de'_i(i-m+1+k) \leqslant k+1$. The method needs time $O(m|\Sigma|+m^2)$ for the preprocessing of P.*

A similar method has recently been independently proposed and analyzed by Chang and Lawler [2].

In a separate paper [11] the performance of several algorithms for the k differences problem is experimentally compared. The hybrid methods of Theorems 5.2 and 5.4 were often found to be the fastest among the compared algorithms. Finally, note that the two phases of the hybrid methods (1. marking, 2. checking by dynamic programming) can be combined so that the resulting algorithm is on-line with respect to T. Only one scan over T is needed, using a window of size $O(m)$.

6. Conclusion

We presented fast approximate pattern-matching methods for strings. The approximation quality was measured with two string distance functions, one based on q-grams, the other on maximal matches. For finding the locally best approximate occurrences of pattern P in text T with respect to the q-gram distance we gave an $O(|T|\log(|P|-q))$ algorithm. Whether or not this is optimal, remains open.

Both distances were shown to have a simple relation to the unit cost edit distance. This leads to efficient hybrid methods for solving the so-called k differences problem of string-matching. A problem for further study is to generalize this approach to edit distances that are more general than the rather restricted unit cost distance of the k differences problem. Then one should allow a larger variety of editing operations and their costs.

References

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

[2] W.I. Chang and E.L. Lawler, Approximate string matching in sublinear expected time, in: *Proc. IEEE 1990 Ann. Symp. on Foundations of Computer Science* (1990) 116–124.

[3] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (The MIT Press, Cambridge, MA, 1990).

[4] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986) 63–89.

[5] M. Crochemore, String matching with constraints, in: *Proc. MFCS'88 Symp.* Lecture Notes in Computer Science, Vol. 324 (Springer, Berlin, 1988) 44–58.

[6] G.R. Dowling and P. Hall, Approximate string matching, *ACM Comput. Surveys* **12** (1980) 381–402.

[7] A. Ehrenfeucht and D. Haussler, A new distance metric on strings computable in linear time, *Discrete Appl. Math.* **20** (1988) 191–203.

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

[9] Z. Galil and K. Park, An improved algorithm for approximate string-matching, in: *Automata, Languages, and Programming (ICALP'89)*, Lecture Notes in Computer Science, Vol. 372 (Springer, Berlin, 1989) 394–404.

[10] R. Grossi and F. Luccio, Simple and efficient string matching with k mismatches, *Inform. Process. Lett.* **33** (1989) 113–120.

[11] P. Jokinen, J. Tarhio, and E. Ukkonen, A comparison of approximate string-matching algorithms, submitted.

[12] R.M. Karp and M.O. Rabin, Efficient randomized pattern matching, *IBM J. Res. Develop.* **31** (1987) 249–260.

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

[14] G. Landau and U. Vishkin, Fast string matching with k differences, *J. Comput. System Sci.* **37** (1988) 63–78.

[15] G. Landau and U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms* **10** (1989) 157–169.

[16] V.I Levenshtein, Binary codes of correcting deletions, insertions and reversals, *Soviet Phys. Dokl.* **10** (1966) 707–710.

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

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

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

[20] C.E. Shannon, A mathematical theory of communications, *The Bell Systems Tech. J.* **27** (1948) 379–423.

[21] J. Tarhio and E. Ukkonen, Boyer–Moore approach to approximate string matching, in: *Proc. 2nd Scand. Workshop on Algorithm Theory (SWAT'90)*, Lecture Notes in Computer Science, Vol. 447 (Springer, Berlin, 1990) 348–359.

[22] E. Ukkonen, Finding approximate patterns in strings, *J. Algorithms* **6** (1985) 132–137.
[23] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100–118.
[24] E. Ukkonen and D. Wood, Approximate string matching with suffix automata, Report A-1990-4, Department of Computer Science, University of Helsinki, 1990.
[25] R.E. Wagner and M.J. Fisher, The string-to-string correction problem, *J. ACM* **21** (1974) 168–173.
[26] P. Weiner, Linear pattern matching algorithms, in: *Proc. 14th IEEE Ann. Symp. on Switching and Automata Theory* (1973) 1–11.