In Proceedings of Fourth South American Workshop
on String Processing, Valparaiso, Chile, November 1997

A General Technique to Improve Filter Algorithms for Approximate String Matching

Robert Giegerich *Frank Hischke*
Stefan Kurtz *Enno Ohlebusch*

Technische Fakultät
Universität Bielefeld
Postfach 100 131
33501 Bielefeld
Germany
Email: {robert, kurtz, enno}@TechFak.Uni-Bielefeld.DE

Abstract. Approximate string matching searches for occurrences of a pattern in a text, where a certain number of character differences (errors) is allowed. Fast methods use filters: A fast preprocessing phase determines regions of the text where a match cannot occur; only the remaining text regions must be scrutinized by the slower approximate matching algorithm. Such filters can be very effective, but they (naturally) degrade at a critical error threshold.

We introduce a general technique to improve the efficiency of filters and hence to push out further this critical threshold value. Our technique intermittently reevaluates the possibility of a match in a given region. It combines *precise* information about the region already scanned with filtering information about the region yet to be searched. We apply this technique to four approximate string matching algorithms published by Chang & Lawler and Sutinen & Tarhio.

1 Introduction

The problem of approximate string matching is stated as follows: given a database string T, a query string P, a threshold k, find all approximate matches, i.e., all subwords v of T whose edit distance to P is at most k. The dynamic programming approach [8] provides the general solution, and is still unbeaten w.r.t. its versatility. Its running time, however, is $\mathcal{O}(mn)$, where $m = |P|$ and $n = |T|$. This is impractical for large-scale applications like biosequence analysis, where the size of the gene and protein databases (i.e. T) grows exponentially due to the advances in sequencing technology. Recent filter techniques for the unit edit distance (e.g. [14, 12, 2, 3, 11, 9, 10]) reduce the average case complexity to at least linear time. Filter algorithms are based on the observation that—given that k is small such that approximate matches are rare—the dynamic programming computation spends most of its time verifying that there is *no* approximate match in a given region of T. A filter rules out regions of T in which approximate matches *cannot* occur, and then a dynamic programming computation is applied only to the remaining regions.

By static filter techniques we refer to the approach as described before. There is a strict separation between the filtering phase and the subsequent checking phase via a dynamic programming computation. Dynamic filtering as introduced in this paper *improves* on this by merging the filtering and the checking phase. It evaluates the statically derived filter information during the checking phase, strengthening it by information determined dynamically. Rather than using static information once to decide whether a (complete) region must be checked, it is consulted intermittently, in order to immediately abandon a region as soon it becomes clear that, due to a poor start, an approximate match is no longer possible.

To understand how this works, recall that a dynamic programming computation gives us information about *actual* differences of the subwords of T and P under consideration. As opposed to this, the statically derived information tells us the rightmost positions where actual differences might occur. These positions are referred to as the positions of the *guaranteed* differences. By definition, the actual differences occur *before* or *at* the guaranteed differences. Now suppose we have a subword s passing the static filter, and we know that, after reading a prefix v_1 of s, d actual differences have occurred. Now, if the remaining suffix v_2 of s contains more than $k-d$ guaranteed differences, we know that s cannot contain an approximate match.

To further clarify the idea of dynamic filtering, let us sketch how it applies to Chang and Lawler's linear expected time algorithm (LET for short) [2]. This algorithm divides $T = w_1c_1 \dots w_rc_rw_{r+1}$ into subwords $w_1, \dots, w_r, w_{r+1}$ of P and characters $c_1, \dots, c_r$. The subwords are of maximal length (i.e. w_ic_i is not a subword of P), and so the characters $c_1, \dots, c_r$ mark the positions of the guaranteed differences. In particular, for each subword of T beginning within, say w_hc_h, the ith difference is guaranteed to occur with character c_{h+i}, or left to it. Therefore, an approximate match contains at most the first $k+1$ of the characters $c_h, c_{h+1}, \dots$, i.e., it is a subword of $s = w_hc_h \dots w_{h+k}c_{h+k}w_{h+k+1}$. Since an approximate match is of length $\geq m-k$, LET discards s if $|s| < m-k$. Otherwise, LET applies a dynamic programming computation to the entire subword s. This is where the dynamic version of LET behaves differently. It suspends the dynamic programming computation at c_{h+i}, whenever $d > i$ differences have occurred already, and an extension up to length $m-k$ adds more than $k-d$ guaranteed differences. This is the intuitive idea—its realization is much more complicated since partial matches may overlap.

One of the main characteristics of a filter algorithm is its critical threshold $k_{\max}$, that is, the maximal value of k such that the filter algorithm is still linear or sublinear, respectively. Strictly speaking, $k_{\max}$ is the value for which we can *prove* the (sub)linear expected running time. Usually, in an expected case analysis, $k_{\max}$ is obtained by roughly estimating the probability of approximate matches. In practice, however, one observes values for $k_{\max}$ which are larger than those theoretically obtained. In the sequel, $k_{\max}$ always refers to the practically obtained values.

In this paper, we develop the technique of dynamic filtering. Applying this

general technique to a given filter algorithm leads to an improved version of that algorithm with a larger critical threshold $k_{\max}$. In particular, we show how dynamic filtering can be applied to the well-known static filter algorithms LET and SET devised by Chang and Lawler [2] and the more sophisticated algorithms LEQ and LAQ recently presented by Sutinen and Tarhio in [9, 10]. For the versions of LET and SET employing dynamic filtering, we present experimental results verifying that an improved critical threshold value $k_{\max}$ is achieved for all alphabet sizes and pattern lengths. The first part of the paper extracts the core of a much wider report [5], where we clarify the proofs by Chang and Lawler, describe some further improvements also for static filters, give new complete proofs for expected time complexity, and present extensive experimental results.

2 Basic Notions and Definitions

We assume that the reader is familiar with the standard terminology on strings, as used, e.g., in [2]. Let u and v be strings. An *alignment* A of u and v is a list $[a_1 \to b_1, \ldots, a_p \to b_p]$ of edit operations (i.e., deletions $a \to \varepsilon$, replacements $a \to b$, and insertions $\varepsilon \to b$ of single characters) such that $u = a_1 \ldots a_p$ and $v = b_1 \ldots b_p$. If A is an alignment, then $\delta(A)$ denotes the number of edit operations $a \to b$, $a \neq b$, in A. $edist(u, v)$ denotes the *edit distance* between u and v and is defined by $edist(u, v) = \min\{\delta(A) \mid A \text{ is an alignment of } u \text{ and } v\}$. An alignment A of u and v is *optimal*, if $\delta(A) = edist(u, v)$.

Suppose we are given a *threshold value* $k \in \mathbb{N}_0$, a *pattern string* P of length m, and a *text string* T of length n. An *approximate match* is a subword v of T s.t. $edist(P, v) \leq k$. The *k-differences problem* is to enumerate all positions in T where an approximate match ends. A generalization of this problem is the *approximate string matching problem* which additionally asks for a corresponding approximate match. Sellers' algorithm [8] solves the k-differences problem in $\mathcal{O}(mn)$ time and $\mathcal{O}(m)$ space by evaluating an $(m+1) \times (n+1)$ table $D(i, j) = \min\{edist(P[1 \ldots i], s) \mid s \text{ is a suffix of } T[1 \ldots j]\}$ using dynamic programming. If $D(m, j) \leq k$, then there is an approximate match ending at position j.

The following definitions are motivated by Ehrenfeucht and Haussler's [4] notion of *compatible markings*. A *partition* of v w.r.t. u is a list $[w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$ of subwords $w_1, \ldots, w_r, w_{r+1}$ of u and characters $c_1, \ldots, c_r$ such that $v = w_1 c_1 \ldots w_r c_r w_{r+1}$. Let $\Psi = [w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$ be a partition of v w.r.t. u. The size of Ψ, denoted by $|\Psi|$, is r. $w_1, \ldots, w_r, w_{r+1}$ are the *submatches* in Ψ. $c_1, \ldots, c_r$ are the *marked characters* in Ψ. If for all $h, 1 \leq h \leq r$, $w_h c_h$ is not a subword of u, then Ψ is the *left-to-right partition* of v w.r.t. u, denoted by $\Psi_{lr}(u, v)$. $|\Psi_{lr}(u, v)|$ is the *maximal matches distance* of u and v, denoted by $mmdist(u, v)$.[1] It can be computed in $\mathcal{O}(|u| + |v|)$ time and $\mathcal{O}(|u|)$ space, using the suffix tree for u (see [7]). The following lemma states an important relation between alignments and partitions. Using this lemma, it is easy to show that $mmdist(u, v) \leq edist(u, v)$.

[1] Note that *mmdist* is not a distance in the usual mathematical sense, since it is not symmetric, see [4].

Lemma 1. *Let A be an alignment of u and v. There is an $r, 0 \leq r \leq \delta(A)$, and a partition $[w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$ of v w.r.t. u such that w_1 is a prefix and w_{r+1} is a suffix of u.*

3 Dynamic Filtering applied to LET

The *linear expected time algorithm* (LET for short) of Chang and Lawler [2] is based on the observation that an approximate match is at least of length $m-k$ and contains at most $k+1$ marked characters of the partition $\Psi_{lr}(P,T) = [w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$. Thus, a subword of T which is shorter than $m-k$ and which contains $k+1$ consecutive marked characters can be discarded since it does not contain an approximate match. The remaining subwords may contain approximate matches. Hence, they are considered as interesting subwords and processed by a dynamic programming computation.

Algorithm LET. Compute $\Psi_{lr}(P,T) = [w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$. For each $h, 1 \leq h \leq r+1$, let $s_{h,p} = w_h c_h \ldots w_{h+p-1} c_{h+p-1} w_{h+p}$. If $|s_{h,k+1}| \geq m-k$, then $s_{h,k+1}$ is an interesting subword. Merge all overlapping interesting subwords. For each interesting subword obtained in such a way, solve the k-differences problem using a dynamic programming computation.

An implementation of algorithm LET represents the partition $\Psi_{lr}(P,T)$ by some integers $mp_0, mp_1, \ldots, mp_r, mp_{r+1}$, where $mp_0 = 0$, $mp_{r+1} = n+1$ and mp_h is the position of the marked character c_h in T, $1 \leq h \leq r$.

In our terminology, LET is a static filter algorithm because the preprocessing and filtering phase is strictly separated from the checking phase. As described in the introduction, the basic idea of dynamic filtering is to merge these phases in order to obtain a more sensitive filter.

Besides table D, the dynamic version of LET (DLET for short) maintains parts of an $(m+1) \times (n+1)$ table W of strings defined as follows: $W(i,j)$ is the shortest suffix of $T[1 \ldots j]$ such that $edist(P[1 \ldots i], W(i,j)) = D(i,j)$. $W(i,j)$ is called the *shortest witness* for $D(i,j)$. Table W can be computed in $\mathcal{O}(mn)$ time according to the following recurrence:

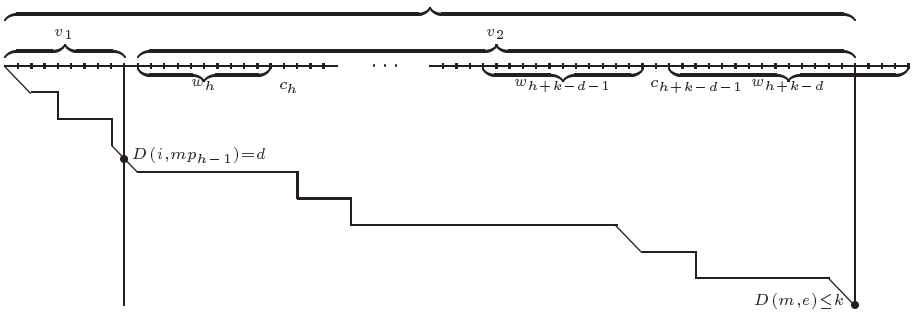
$$W(i,j) = \begin{cases} \varepsilon & \text{if } i = 0 \text{ or } j = 0 \\ W(i-1,j) & \text{else if } D(i,j) = D(i-1,j) + 1 \\ W(i-1,j-1)T[j] & \text{else if } D(i,j) = D(i-1,j-1) + \delta_{i,j} \\ W(i,j-1)T[j] & \text{else if } D(i,j) = D(i,j-1) + 1 \end{cases}$$

where $\delta_{i,j} = 0$ if $P[i] = T[j]$ and $\delta_{i,j} = 1$ otherwise. In practice, it suffices to compute the *length* of each $W(i,j)$. Each entry $|W(i,j)|$ can be computed together with $D(i,j)$ in a single dynamic programming step at virtually no cost. Note that if $D(m,j) \leq k$, then $W(m,j)$ is the shortest approximate match ending at position j. That is, DLET and the subsequent dynamic filter algorithms do not only solve the k-differences problem, but the more general approximate string matching problem. We point out that this is required in practical applications, anyhow.

Let us again consider the partition $\Psi_{lr}(P,T) = [w_1, c_1, \ldots, w_r, c_r, w_{r+1}]$ and define $s_{h,p} = w_h c_h \ldots w_{h+p-1} c_{h+p-1} w_{h+p}$ for each $h, 1 \leq h \leq r+1$. Like algorithm LET, DLET determines interesting subwords $s_{h_b,k+1}$ by checking the "static" condition $|s_{h_b,k+1}| \geq m - k$. If this holds, it means that an approximate match may begin within subword $w_{h_b} c_{h_b}$. However, instead of applying a dynamic programming computation to the whole interesting subword $s_{h_b,k+1}$, DLET sequentially checks whether an approximate match can be continued with $w_h c_h$ for $h > h_b$. Thereby it exploits the following observation: Suppose there are approximate matches ending at position e in T, and assume v is the shortest. If a prefix v_1 of v ends at position mp_{h-1} and contains d differences, then the remaining part v_2 of v contains at most $k - d$ marked characters of $\Psi_{lr}(P, T[mp_{h-1}+1 \ldots n]) = [w_h, c_h, \ldots, w_r, c_r, w_{r+1}]$, i.e., $|v_2| \leq |w_h c_h \ldots w_{h+k-d-1} c_{h+k-d-1} w_{h+k-d}|$.

This observation is illustrated in Figure 1, which shows an optimal alignment

Fig. 1. An approximate match with a prefix v_1 which ends at position mp_{h-1}



of P and v as a path in the distance table D, crossing entry $D(i, mp_{h-1})$. It can be shown that there must be an $i, 0 \leq i \leq m$, such that $D(i, mp_{h-1}) = d$ and $W(i, mp_{h-1}) = v_1$; see proof of Theorem 4. With the above observation this leads to the following "dynamic" condition for approximate matches:

$$\exists i, 0 \leq i \leq m\text{: } D(i, mp_{h-1}) \leq k \text{ and } |W(i, mp_{h-1})| + |s_{h,k-D(i,mp_{h-1})}| \geq m - k$$

If this condition is false, the checking phase for $s_{h_b,k+1}$ is stopped and the beginning of the next potential approximate match is searched for. Otherwise, a potential approximate match can be continued with $w_h c_h$ and the checking phase proceeds.

In order to avoid redundant computations, T is scanned from left to right. For each $h, 1 \leq h \leq r+1$, a dynamic programming computation is sequentially applied to $w_h c_h$, only if it contains the beginning or the continuation of a potential approximate match, according to the "static" or "dynamic" condition. For this reason, each $w_h c_h$ is checked at most once.

Algorithm DLET. Define two predicates *BPM* and *CPM* as follows:

$$\begin{aligned} &BPM(h) \text{ iff } |s_{h,k+1}| \geq m-k \\ &CPM(h) \text{ iff } \exists i, 0 \leq i \leq m : D(i, mp_{h-1}) \leq k \text{ and} \\ &\qquad\qquad |W(i, mp_{h-1})| + |s_{h,k-D(i,mp_{h-1})}| \geq m-k \end{aligned}$$

Perform the following computations:

$h := 1$
while $h \leq r+1$ **do**
  if not *BPM*(h)
  then
    $h := h+1$
  else
    for $i := 0$ **to** m **do** $D(i, mp_{h-1}) := i$; $W(i, mp_{h-1}) := \varepsilon$
    repeat
      for $j := mp_{h-1}+1$ **to** mp_h **do**
        for $i := 0$ **to** m **do** compute $D(i,j)$ and $W(i,j)$
        if $D(m,j) \leq k$ **then output** "$W(m,j)$ is approximate match at j"
      $h := h+1$
    until not *CPM*(h)

BPM(h) indicates that there is a beginning of a potential approximate match within $w_h c_h$. *CPM*(h) indicates that $w_h c_h$ is the continuation of a potential approximate match. In order to check *CPM*(h), it suffices to compute table W and D only for those (i,j), where $D(i,j) \leq k$. Thus Ukkonen's "cutoff" trick [13] is applicable in this context. It is easy to see that the left-to-right scan in DLET only needs $\mathcal{O}(m)$ space.

The following example illustrates how DLET works.

Example 1. Let $P = abbcccddddeeeee$, $T = eeeeedddcccfbbfa$, and $k = 3$. Then $\Psi_{lr}(P,T)$ is as follows:

$$\underbrace{eeeee}_{w_1}\ \underset{c_1}{d}\ \underbrace{ddd}_{w_2}\ \underset{c_2}{c}\ \underbrace{cc}_{w_3}\ \underset{c_3}{f}\ \underbrace{bb}_{w_4}\ \underset{c_4}{f}\ \underbrace{a}_{w_5}$$

Obviously, $s_{1,k+1} = w_1c_1w_2c_2w_3c_3w_4c_4w_5 = T$, which is of length 17. Since $m-k = 15-3 = 12$, *BPM*(1) is true, i.e. T is an interesting subword. DLET now computes the first 7 columns of tables D and W. Since $D(i,5) = |W(i,5)| = i$, and $i + |s_{2,k-i}| \leq 11$, *CPM*(2) is false. Hence a potential approximate match cannot be continued, and DLET terminates. Note that LET would have to compute table D completely (i.e. 18 columns) to find that there is no approximate match in T.

To prove correctness of DLET, we need the following lemma and a new notion which is given in Definition 3.

Lemma 2. *Let $p = p_1p_2$, $t = t_1t_2$, and A be an alignment of p_2 and t_1 with $\delta(A) \leq d$. Assume $|\Psi_{lr}(p,t)| \geq d$ and let $w_1, c_1, \ldots, w_d, c_d, w_{d+1}$ be the first $2d+1$ elements of $\Psi_{lr}(p,t)$. Then $|t_1| \leq |w_1c_1 \ldots w_dc_dw_{d+1}|$.*

Proof. By Lemma 1, there is a $d', 0 \leq d' \leq d$, and a partition $\Psi = [x_1, a_1, \ldots, x_{d'}, a_{d'}, x_{d'+1}]$ of t_1 w.r.t. p_2, hence w.r.t. p. We use induction on i to show

$$|x_1 a_1 \ldots x_i| \leq |w_1 c_1 \ldots w_i| \tag{1}$$

for all $i, 0 \leq i \leq d' + 1$. If $i = 0$, then claim (1) holds trivially. For a proof by contradiction, let us assume that $|x_1 a_1 \ldots x_i a_i x_{i+1}| > |w_1 c_1 \ldots w_i c_i w_{i+1}|$, i.e., x_{i+1} ends at c_{i+1} or right to it. By the induction hypothesis, we know that x_{i+1} starts at the beginning of w_{i+1} or left to it. We conclude that $w_{i+1}c_{i+1}$ is a subword of x_{i+1}, hence of p. This, however, contradicts the fact that w_{i+1} and c_{i+1} are elements of $\Psi_{lr}(p, t)$. Thus (1) holds. Consequently, we infer

$$|t_1| = |x_1 a_1 \ldots x_{d'} a_{d'} x_{d'+1}| \leq |w_1 c_1 \ldots w_{d'} c_{d'} w_{d'+1}| \leq |w_1 c_1 \ldots w_d c_d w_{d+1}|.$$

Definition 3. For every i and j where $0 \leq i \leq m$ and $0 \leq j \leq n$, let $A(i, j)$ be the alignment defined by the following recurrences:

$$A(i,j) = \begin{cases} [] & \text{if } i = 0 \text{ or } j = 0 \\ A(i-1,j) {+\!+} [P[i] \rightarrow \varepsilon] & \text{else if } D(i,j) = D(i-1,j) + 1 \\ A(i-1,j-1) {+\!+} [P[i] \rightarrow T[j]] & \text{else if } D(i,j) = D(i-1,j-1) + \delta_{i,j} \\ A(i,j-1) {+\!+} [\varepsilon \rightarrow T[j]] & \text{else if } D(i,j) = D(i,j-1) + 1 \end{cases}$$

Here the symbol [] denotes the empty list and $+\!+$ denotes list concatenation.

Note that $A(i, j)$ is defined in the same scheme as $W(i, j)$. It is clear, that $A(i, j)$ is an optimal alignment of $P[1 \ldots i]$ and $W(i, j)$, i.e., $\delta(A(i, j)) = D(i, j) = edist(P[1 \ldots i], W(i, j))$.

Theorem 4. *Algorithm DLET correctly solves the approximate string matching problem.*

Proof. Suppose there is an approximate match ending at position e in T, i.e., $D(m, e) \leq k$. Let v be the shortest approximate match ending at position e and assume that v begins at position b. Obviously, there are integers h_b and h_e, $1 \leq h_b \leq h_e \leq r + 1$, such that $mp_{h_b-1} < b \leq mp_{h_b}$ and $mp_{h_e-1} < e \leq mp_{h_e}$. v will be detected if DLET computes $D(i, j)$ for all i, j with $0 \leq i \leq m$, $b \leq j \leq e$. Therefore, we have to show:

(1) *BPM*(h_b) is true,

(2) *CPM*(h) is true, for all $h, h_b < h \leq h_e$.

Clearly, (1) holds, due to the correctness of algorithm LET. To prove (2), consider some $h, h_b < h \leq h_e$, arbitrary but fixed. Let $v_1 = T[b \ldots mp_{h-1}]$ and $v_2 = T[mp_{h-1} + 1 \ldots e]$, as illustrated in Figure 1. Furthermore, note that $A(m, e)$ is an optimal alignment of P and v. Hence, there is an $i, 0 \leq i \leq m$, such that $A(m, e)$ can be split into alignments A_1 and A_2 where A_1 is an alignment of $P[1 \ldots i]$ and v_1, and A_2 is an alignment of $P[i + 1 \ldots m]$ and v_2. By construction, we have $A_1 = A(i, mp_{h-1})$. According to the remark

after Definition 3 it follows that $v_1 = W(i, mp_{h-1})$ and $\delta(A_1) = D(i, mp_{h-1})$. Since $\delta(A(m,e)) = D(m,e) \leq k$, we conclude $D(i, mp_{h-1}) \leq k$. Now let $d = D(i, mp_{h-1})$. By definition, $s_{h,k-d} = w_h c_h \ldots w_{h+k-d-1} c_{h+k-d-1} w_{h+k-d}$. Note that $w_h, c_h, \ldots, w_{h+k-d-1}, c_{h+k-d-1}, w_{h+k-d}$ are the first $2(k-d)+1$ elements of the partition $\Psi_{lr}(P, T[mp_{h-1}+1 \ldots n])$. Moreover, we have $\delta(A_2) = \delta(A(m,e)) - \delta(A_1) \leq k - d$. Therefore, Lemma 2 is applicable with $p_2 = P[i+1 \ldots m]$ and $t_1 = v_2$ and we conclude $|s_{h,k-d}| \geq |v_2|$. This implies $|W(i, mp_{h-1})| + |s_{h,k-d}| \geq |v_1| + |v_2| = |v| \geq m - k$, i.e., *CPM*(*h*) is true. □

3.1 Further Improvements

The filter applied in algorithm LET can be improved by taking into consideration the arrangement of the subwords of P in an approximate match. That is, one can apply Lemma 1, according to which an approximate match has a partition of size $\leq k$ beginning with a *prefix* and ending with a *suffix* of P. This means that an approximate match is a subword of $y_h c_h w_{h+1} \ldots w_{h+k} c_{h+k} z_{h+k+1}$, where y_h is the longest suffix of w_h that is a prefix of P, and z_{h+k+1} is the longest prefix of w_{h+k+1} that is a suffix of P. An improved filter algorithm which exploits this property additionally has to compute for each $h, 1 \leq h \leq r+1$, the length of y_h and z_h. This works as follows: let \$ be a character not occuring in P and let $ST(P\$)$ denote the suffix tree of the string $P\$$ (cf. [7]). We assume that the nodes in $ST(P\$)$ are annotated such that for each subword u of P we can decide in constant time if u is a prefix of P or if u is a suffix of P. Such annotations can be computed by a single traversal of $ST(P\$)$ in $\mathcal{O}(m)$ space and time, as described in [6]. The length of the submatch w_h is determined by scanning $ST(P\$)$ from the *root*, driven by the characters in $T[mp_{h-1}+1 \ldots n]$. For each prefix u of w_h one checks in constant time if u is a suffix of P. The longest such u is z_h. Thus, $|z_h|$ can be computed at virtually no cost. The length of y_h can be obtained by checking for each suffix u of w_h in constant time if u is a prefix of P. The longest such u is y_h. Each suffix of w_h can be found in constant time (on the average), using the *suffix links* of $ST(P\$)$ (see [7]). However, experimental results show that the computation of y_h does not pay off. Fortunately, in algorithm DLET table W contains all information necessary to determine the length of the prefix of a potential approximate match. Therefore, the prefixes y_h of P are not needed for checking the dynamic condition.

3.2 Dynamic Filtering Applied to SET

Our concept of dynamic filtering also applies to algorithm SET, the sublinear expected time algorithm of Chang and Lawler[2]. The approach is to combine the basic idea of algorithm SET with the "dynamic component" of algorithm DLET. As in algorithm SET, T is divided into non-overlapping regions $R_h = T[q_h \ldots q_h + l - 1]$, where $l = \lfloor (m-k)/2 \rfloor$ and $q_h = hl + 1$. In analogy to DLET, a dynamic programming computation is applied only to those regions, which contain the beginning or the continuation of a potential approximate match. For more details, see [5].

3.3 Expected Case Analysis

We assume that T is a uniformly random string over the finite alphabet $\mathcal{A}$ of size b (i.e., each character of $\mathcal{A}$ occurs with probability $1/b$ at a position in T). Our results are consistent with those of [2]. In contrast to [2], however, our proofs in [5] apply to every alphabet size (not only to the case $b = 2$). In particular, we are able to explain the mysterious constants c_1 and c_2 of the Main Lemma in [2] in terms of b and m by simply applying a variant of the well-known Tchebychev inequality instead of referring to "the Chernoff bound technique" as done in [2]. This gives more general, self-contained, and simpler proofs. The Main Lemma in [2] reads as follows.

Lemma 5. *For suitably chosen constants c_1 and c_2, and $k = \frac{m}{\log_b m + c_1} - c_2$, we have $Pr[|s_{h,k+1}| \geq m - k] < 1/m^3$.*

In the string searching literature, this result is often cited but few people seem to know how these constants can actually be computed. In [5], it is shown how c_1 and c_2 can be obtained from the theorem below.

Theorem 6. *If $k \leq \frac{m - 6\log_b m + 3}{\log_b m + 2 + 2\log_b c} - 2$, where $c = \sum_{d=0}^{m-\log_b m} (\frac{1}{\sqrt{b}})^d$, then we have $Pr[|s_{h,k+1}| \geq m - k] < 1/m^3$.*

Using Theorem 6, it is possible to show that the expected running time of the checking phase of LET and DLET is $\mathcal{O}(n)$. Since the preprocessing and filtering phase requires $\mathcal{O}(n)$ time, it follows that the expected running time of LET is $\mathcal{O}(n)$. The same holds for the improved version DLET, but the bound for k can be weakened; see [5].

3.4 Experimental Results

In our experiments we verified that our dynamic filtering technique, when applied to the static filter algorithms LET and SET, leads to an improved critical threshold $k_{\max}$ (cf. the introduction) for all alphabet sizes and pattern lengths. In a first test series we used random text strings T of length $n = 500{,}000$ over alphabets of size $2, 4, 10$ and 40. We chose patterns of a fixed length $m = 64$ over the same alphabets. Figures 2 and 3 show the effect of a varying threshold value k on the *filtration efficiency* $f = (n - n_p)/n$, where n_p is the number of positions in T left for dynamic programming. It can be seen that in order to achieve a particular filtration efficiency, algorithms DLET and DSET allow for a larger value of k than algorithms LET and SET, respectively. The advantage is independent of the alphabet size. Figure 4 shows the effect of the alphabet size on $k_{\max}$. All algorithms achieve a larger value of $k_{\max}$ with growing alphabet size. The dynamic filter algorithms are always superior to the static filter algorithms.

Fig. 2. Filtration efficiency for fixed pattern length $m = 64$ (LET and DLET)

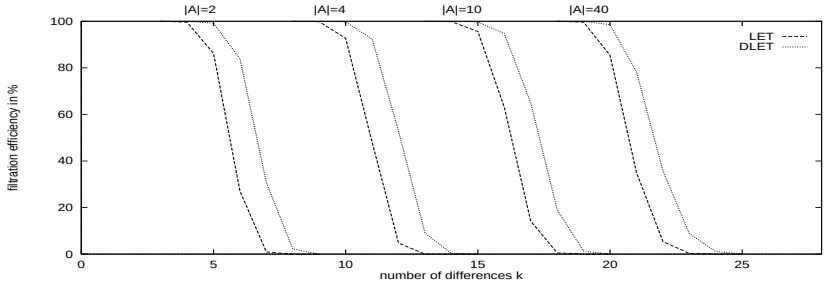


Fig. 3. Filtration efficiency for fixed pattern length $m = 64$ (SET and DSET)

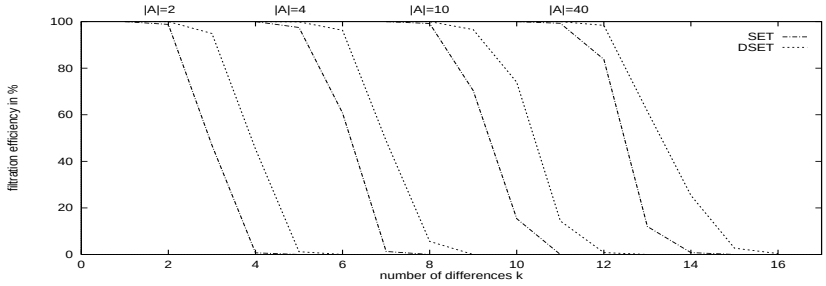
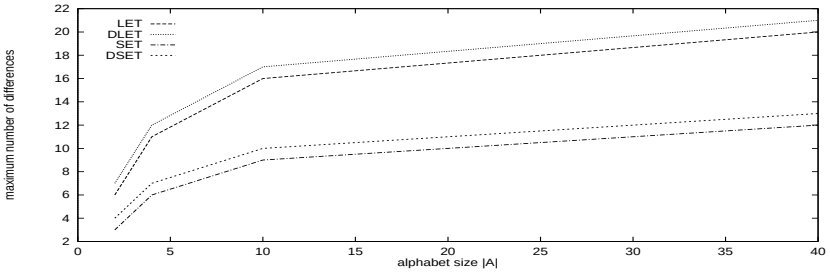


Fig. 4. $k_{\max}$ for fixed pattern length $m = 64$



In a second test series T was an English text of length n = 500,000. The alphabet size was $|\mathcal{A}| = 80$. We chose pattern of length $m \in \{16, 32, 64, 128\}$ over the same alphabet. Figures 5 and 6 show how a varying pattern length effects the filtration efficiency. It can be seen that in order to achieve a particular filtration efficiency, algorithms DLET and DSET allow for a larger value of k than algorithms LET and SET, respectively. The larger the pattern, the larger the improvement. Figure 7 shows the effect of the pattern length on $k_{\max}$. All algorithms achieve a larger value of $k_{\max}$ with m becoming larger. The dynamic filter algorithms are always superior to the static filter algorithms.

There is virtually no time penalty for the complex dynamic filter algorithms, if the filtration efficiency is almost 100%. When the static filter looses its effect, while the dynamic is still filtering, the latter is much faster than the former. Finally, if $k > k_{\max}$, then the dynamic filter has a considerable overhead. However, in this case, pure dynamic programming is preferable anyway.

Fig. 5. Filtration efficiency for fixed alphabet size $|A| = 80$ (LET and DLET)

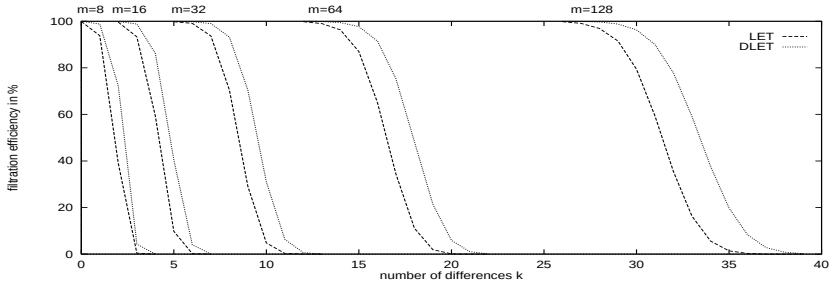


Fig. 6. Filtration efficiency for fixed alphabet size $|A| = 80$ (SET and DSET)

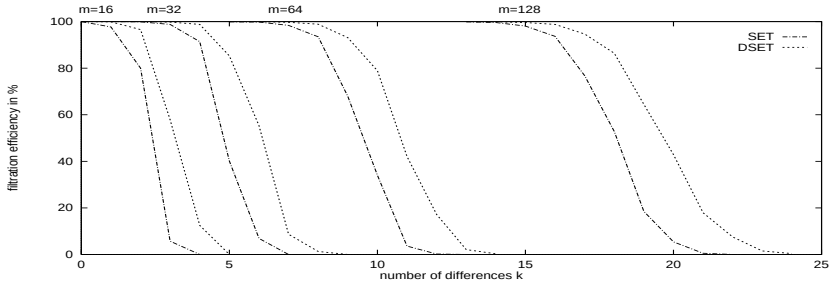
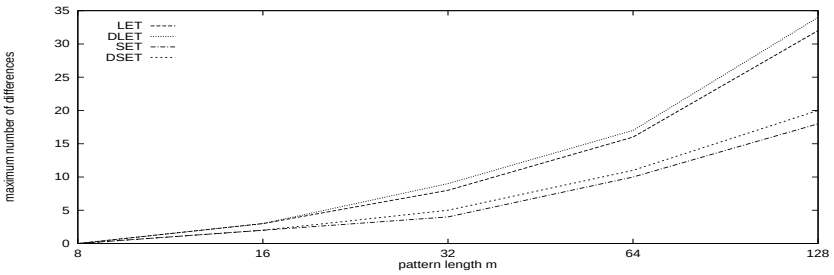


Fig. 7. $k_{\max}$ for fixed alphabet size $|A| = 80$



4 Dynamic Filtering applied to LEQ

Dynamic filtering is a general idea which can be applied to other known static filter techniques, e.g. [14, 12, 3, 11, 9, 10]. We exemplify this claim by showing how it can be applied to Sutinen and Tarhio's algorithms LEQ and LAQ [9, 10].[2] We first briefly recall the basic idea of LEQ and LAQ. Assume that q and s are positive integers and define $h = \left\lfloor \frac{m-k-q+1}{k+s} \right\rfloor$. LEQ and LAQ take q-samples

$$Tsam(j) = T[jh - q + 1 \ldots jh] \text{ for all } j \in \left\{1, 2, \ldots, \left\lfloor \frac{n}{h} \right\rfloor \right\}.$$

from the text T. Suppose $v = T[b \ldots e]$ is an approximate match, that is, $edist(P, v) \leq k$. If for the sampling step h the inequality $h \geq q$ holds, then v

[2] That is, it also applies to Takaoka's method with $s > 1$ instead of $s = 1$ q-samples taken from the text.

contains at least $k+s$ consecutive q-samples. Moreover, at least s of these must occur in P—in the same order as in v. This ordering is taken into account by dividing P into blocks $Q_1, \ldots, Q_{k+s}$, where

$$Q_i = P[(i-1)h+1 \ldots ih+q-1+k].$$

LEQ and LAQ are based on the following theorem which we cite from [10] (cf. also [9]): If $Tsam(j_a)$ is the leftmost q-sample of the approximate match $v = T[b \ldots e]$, then there is an integer t, $0 \leq t \leq k+1$, such that the $k+s$ consecutive q-samples $Tsam(j_a+1+t), \ldots, Tsam(j_a+k+s+t)$ are contained in v and $Tsam(j_a+l+t) \in Q_l$ holds for at least s of the samples. (We write $u \in v$, if string u is a subword of string v.) Defining $j_b = j_a + t$, LEQ can be formulated as follows:

Algorithm LEQ. If, for $k+s$ consecutive q-samples $Tsam(j_b+1), Tsam(j_b+2), \ldots, Tsam(j_b+k+s)$, we have $Tsam(j_b+l) \in Q_l$ for at least s indices l, $1 \leq l \leq k+s$, then apply a dynamic programming computation to the subword $T[j_b h - q + 2 - 2k \ldots (j_b+1)h - q + m + k]$.

This correctly solves the k differences problem because the range of the dynamic programming area is sufficiently large (see Theorem 3 in [9]). We next describe the dynamic version of LEQ which exploits the fact that whenever $Tsam(j+l) \notin Q_l$ holds, a guaranteed difference has been detected.

Algorithm DLEQ. Let the function $\varphi : bool \to \{0,1\}$ be defined by $\varphi(true) = 1$ and $\varphi(false) = 0$. For each j, $1 \leq j \leq \lfloor \frac{n}{h} \rfloor - (k+s) + 1$ define

- $GD(j,l,r) = \sum_{y=l}^{r} \varphi(Tsam(j+y) \notin Q_y)$,
- $BPM(j)$ *iff* $GD(j,1,k+s) \leq k$,
- $CPM(j)$ *iff* $\exists i, 0 \leq i \leq m : D(i,jh) + GD(j,1,k+s-l) \leq k$, where $l = \left\lfloor \frac{|W(i,jh)|-q}{h} + 1 \right\rfloor$.

Perform the following computations:

```
j := 1
while j ≤ ⌊n/h⌋ − (k + s) + 1 do
  if not BPM(j)
  then
    j := j + 1
  else
    left := jh − q + 2 − 2k
    for i := 0 to m do D(i, left − 1) := i; W(i, left − 1) := ε
    repeat
      for p := left to (j + 1)h do
        for i := 0 to m do compute D(i, p) and W(i, p)
        if D(m, p) ≤ k then output "W(i, p) is approximate match at" p
      left := (j + 1)h + 1
      j := j + 1
    until not CPM(j)
```

Notice that for $i = 0$ we have $D(i, jh) = 0$ and $l = \lfloor -\frac{q}{h} + 1 \rfloor = 0$. Hence $BPM(j)$ implies $CPM(j)$.

Theorem 7. *Algorithm DLEQ correctly solves the k-differences problem.*

Proof. Suppose there is an approximate match ending at position e in T. Let v be the shortest such approximate match and assume that it begins at position b, i.e., $v = T[b \dots e]$. v contains at least $k + s$ consecutive q-samples $Tsam(j_b + 1), \dots, Tsam(j_b + k + s)$ such that $Tsam(j_b + l) \in Q_l$ holds for at least s of the samples. Taking the maximal width of v into account, we obtain $j_b h - q + 2 - 2k \leq b \leq (j_b + 1)h - q + 1$ and $(j_b + k + s)h \leq e \leq (j_b + 1)h - q + m + k$. v will be detected if DLEQ computes $D(i, r)$ for all i, r with $0 \leq i \leq m$, $b \leq r \leq e$. Therefore we have to show:

(*i*) $BPM(j_b)$ is true,

(*ii*) $CPM(j)$ ist true for each $j, j_b + 1 \leq j \leq j_b + k + s$.

(*iii*) If $j > j_b + k + s$ and $CPM(j)$ is false then $e \leq jh$.

(*i*) For at least s indices $l, 1 \leq l \leq k + s$ we have $Tsam(j_b + l) \in Q_l$. Hence for at most k indices $l, 1 \leq l \leq k + s$, $Tsam(j_b + l) \notin Q_l$, i.e., $BPM(j_b)$ is true.
(*ii*) Consider an arbitrary but fixed $j, j_b + 1 \leq j \leq j_b + k + s$. By construction $A(m, e)$ (cf. Definition 3) is an optimal alignment of P and v, i.e., $\delta(A(m, e)) = D(m, e) \leq k$. Let $v_1 = T[b \dots jh]$ and $v_2 = T[jh + 1 \dots e]$. There is an $i, 0 \leq i \leq m$, such that $A(m, e)$ can be split into alignments A_1 and A_2 where A_1 is an alignment of $P[1 \dots i]$ and v_1, and A_2 is an alignment of $P[i + 1 \dots m]$ and v_2. By construction we have $A_1 = A(i, jh)$ and $\delta(A_1) = D(i, jh)$. Moreover, $v_1 = W(i, jh)$ which implies $b = jh - |W(i, jh)| + 1$. Let $l = \left\lfloor \frac{|W(i,jh)|-q}{h} + 1 \right\rfloor$. Since $b \leq (j_b+1)h-q+1$, we have $l = \left\lfloor \frac{jh-b+1-q}{h} + 1 \right\rfloor \geq \left\lfloor \frac{jh-((j_b+1)h-q+1)+1-q}{h} + 1 \right\rfloor = \left\lfloor \frac{jh-(j_b+1)h}{h} + 1 \right\rfloor = j - j_b$. Let $d = D(i, jh)$. Now $\delta(A_1) + \delta(A_2) = \delta(A(m, e))$ implies $\delta(A_2) = \delta(A(m, e)) - \delta(A_1) \leq k - d$, i.e., A_2 contains at most $k - d$ edit operations $a \to b$, $a \neq b$. Each of these can prevent at most one q-sample of $Tsam(j + 1), \dots, Tsam(j + k + s - (j - j_b))$ from occurring in the corresponding block of P. That is, for at most $k - d$ indices $y, 1 \leq y \leq k + s - (j - j_b)$ we have $Tsam(j + y) \notin Q_y$. In other words, $GD(j, 1, k + s - (j - j_b)) \leq k - d$. Since $l \geq j - j_b$, we have $GD(j, 1, k + s - (j - j_b)) \geq GD(j, 1, k + s - l)$. Thus $d + GD(j, 1, k + s - l) \leq d + k - d = k$, i.e., $CPM(j)$ holds.
(*iii*) Let $j > j_b + k + s$ and $CPM(j)$ be false. We assume $e > jh$ and derive a contradiction. There is an $i, 0 \leq i \leq m$ such that $D(i, jh) \leq k$ and $b = jh - |W(i, jh)| + 1$. Obviously, $W(i, jh)$ is a prefix of v, i.e., it contains the q-samples

$$Tsam(j_b + 1), \dots, Tsam(j_b + k + s), \dots, Tsam(j) \tag{2}$$

Since $j > j_b + k + s$, the sequence (2) is of length at least $k + s + 1$. Hence $|W(i, jh)| \geq (k + s)h + q$. Let $l = \left\lfloor \frac{|W(i,jh)|-q}{h} + 1 \right\rfloor$. Then $l \geq k + s + 1$ which

implies $GD(j, 1, k+s-l) = 0$. Thus we conclude $D(i, jh) + GD(j, 1, k+s-l) \leq k$, which means that $CPM(j)$ holds. This is a contradiction, i.e., $e \leq jh$ is true. □

Like LEQ (see [9]), an efficient implementation of DLEQ utilizes the shift-add technique of [1]: for each $j, 0 \leq j \leq \lfloor \frac{n}{h} \rfloor$ a vector M_j is computed, where

$$M_j(i) = \begin{cases} \sum_{l=0}^{i-1} \varphi(Tsam(j-l) \in Q_{i-l}) & \text{if } i \leq j \\ 0 & \text{otherwise} \end{cases}$$

One easily shows that $M_{j+1}(i+1) = M_j(i) + \varphi(Tsam(j+1) \in Q_{i+1})$ and $GD(j, 1, r) = r - M_{j+r}(r)$ hold. As a consequence (i) M_{j+1} can be obtained from M_j by some simple bit parallel operations (provided P is suitably preprocessed, see [9] for details), (ii) $BPM(j)$ can be decided in constant time, and (iii) $CPM(j)$ can be decided in $\mathcal{O}(m)$ time. Thus, the dynamic checking in DLEQ requires $\mathcal{O}(mn/h)$ time in the worst case.

4.1 Dynamic Filtering Applied to LAQ

A dynamic version of algorithm LAQ [10] is easily obtained from the above algorithm. Instead of counting one difference whenever $Tsam(j+l) \notin Q_l$ (like LEQ does), LAQ uses the asm distance introduced by Chang and Marr [3] in order to obtain a better lower bound for the guaranteed differences. Let $asm(u, B)$ denote the edit distance between string u and its best match with a subword of string B. LAQ is obtained from LEQ by simply replacing

$$\begin{aligned} GD(j, l, r) &= \textstyle\sum_{y=l}^{r} \varphi(Tsam(j+y) \notin Q_y) \\ \text{with } GD(j, l, r) &= \textstyle\sum_{y=l}^{r} asm(Tsam(j+y), Q_y). \end{aligned}$$

Since $Tsam(j+y) \notin Q_y$ implies $asm(Tsam(j+y), Q_y) \geq 1$, LAQ uses a stronger filter than LEQ. The price to be paid, however, is that either tables $asm(u, Q_i)$, $1 \leq i \leq k+s$, have to be precomputed for every string u of length q, or the required entries of the tables have to be computed on demand.

5 Conclusion

Although the technical details are different in each case, we have shown that our approach is a general technique for the improvement of filtering methods in approximate string matching. By analogy, we may plan a car route through a crowded city, based on advance information of traffic congestion at various points. There is always a chance to improve our routing decisions based on the traffic we have observed so far, still using advance information about the route ahead. This does not make much difference in very low traffic (practically no matches) or in times of an overfull traffic overload (matches almost everywhere). But there is always a certain level of traffic where the flexibility added by our method makes us reach the destination before our date has gone.

References

1. R.A. Baeza-Yates and G.H. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, **35**(10):74–82, 1992.

2. W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, **12**(4/5):327–344, 1994.

3. W.I. Chang and T.G. Marr. Approximate String Matching and Local Similarity. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'94), LNCS 807, pages 259–273, 1994.

4. A. Ehrenfeucht and D. Haussler. A New Distance Metric on Strings Computable in Linear Time. *Discrete Applied Mathematics*, **20**:191–203, 1988.

5. R. Giegerich, F. Hischke, S. Kurtz, and E. Ohlebusch. Static and Dynamic Filtering Methods for Approximate String Matching. Report 96-01, Technische Fakultät, Universität Bielefeld, 1996. `ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report96-01.ps.gz`.

6. S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, July 1995.

7. E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

8. P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**:359–373, 1980.

9. E. Sutinen and J. Tarhio. On Using q-Gram Locations in Approximate String Matching. In *Proceedings of the European Symposium on Algorithms*, pages 327–340. Lecture Notes in Computer Science **979**, Springer Verlag, 1995.

10. E. Sutinen and J. Tarhio. Filtration with q-Samples in Approximate Matching. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'96), LNCS 1075, pages 50–63, 1996.

11. T. Takaoka. Approximate Pattern Matching with Samples. In *Proceedings of ISAAC 1994*, pages 234–242. Lecture Notes in Computer Science **834**, Springer Verlag, 1994.

12. J. Tarhio and E. Ukkonen. Approximate Boyer-Moore String Matching. *SIAM Journal on Computing*, **22**(2):243–260, 1993.

13. E. Ukkonen. Finding Approximate Patterns in Strings. *Journal of Algorithms*, **6**:132–137, 1985.

14. E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, **92**(1):191–211, 1992.