Technical Report #2008-02
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada

A FILTERING ALGORITHM FOR APPROXIMATE PATTERN MATCHING WITH REDUCED VERIFICATION

by

Christoph J. Richter (1) and Wolfgang Banzhaf (2)

[1]Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany, Email: christoph.richter@cs.uni-dortmund.de

[2] Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, Canada A1C 5S7, Email: banzhaf@cs.mun.ca

Department of Computer Science
Memorial University of Newfoundland
St. John's, NF, Canada A1B 3X5

January 2008

A filtering algorithm for approximate pattern matching with reduced verification

Christoph J. Richter
University of Dortmund
Dept. of Computer Science
44221 Dortmund, Germany
christoph.richter@cs.uni-dortmund.de

Wolfgang Banzhaf
Memorial University of Newfoundland
Dept. of Computer Science
St. John's, NL, Canada A1C 5S7
banzhaf@cs.mun.ca

This paper describes an algorithm for approximate pattern matching based on the partitioning into exact search filtering approach. The Sequitur algorithm is mainly utilized to reduce the amount of checking needed, as checking is the expensive part in filtering algorithms. Thus, a better filtration efficiency could be achieved for higher error levels and also a good performance for shorter texts.

Keywords: approximate pattern matching, sequence comparison, filtering, partitioning into exact search, Sequitur

1 Introduction

Approximate string matching (or also called *approximate pattern matching* or *k differences problem*) describes the problem of finding a certain pattern in a text assuming that either the text or the pattern contains errors. As a result, all positions are accepted where a pattern is found in the text, which differs not more than a certain limited number of errors from the given pattern. Since this is a very general problem, there is a great variety of applications in different areas like computational biology, text retrieval, and others like listed in [23, 31, 43]. To solve the problem a lot of algorithms have been designed. Overviews are given in various papers and books whereas [17, 30, 31, 36] are the most recent ones.

One class of solutions is the class of filtering algorithms. The algorithms of this class are working in two phases, one for filtration and one for the checking (also called verification). Since the checking phase is more expensive, it is useful to reduce the proportion of this phase that the algorithm spends less time there.

Closely related to approximate string matching is the problem of *compressed approximate string matching*. It deals with compressed, i. e. redundancy reduced, texts instead of uncompressed texts. The pattern search is performed without uncompressing the text.

This problem was investigated only in the very recent years and there are only a few solutions to this problem up to now [22, 27, 28, 37].

The algorithm presented in this paper solves the problem of approximate string matching on the basis of a filtering approach, but uses the general idea of redundancy reduction to spend less time in the checking phase. Before performing the pattern search on the text, a preprocessing step is used to calculate the redundancy information (this can be compared to the compression step in compressed approximate string matching). During the search, the redundancy information can be used to skip search and verification in some areas of the text or to repeat matchings.

This paper is organized as follows. In the next section we briefly discuss prevoius work. After the algorithm itself is presented in Section 3, in Section 4 the practical behavior of the algorithm is estimated. Finally the last section draws conclusions and gives suggestions for future work.

2 Related Work

In this section, a formal definition of the problem is given. Furthermore, the general context is outlined and the algorithm presented here is positioned in this context.

The problem of approximate pattern matching is defined as follows: Given a text $T = t_1 \ldots t_n$, and a pattern $P = p_1 \ldots p_m$ $(t_i, p_j \in \Sigma)$, find all positions in T, where P appears with at most k errors, i. e. return the set $\{|x\bar{P}|, T = x\bar{P}y \wedge d(P, \bar{P}) \leq k\}$. x and y are substrings of T, $|.|$ gives the length of a string and $d(P, \bar{P})$ gives the edit distance (also called Levenshtein distance [26]) between P and $\bar{P}$. The edit distance between two strings characterizes the number of transformation operations (insertion, deletion and replacement), that are necessary to transform one string into the other one.

The general solution principle utilizes dynamic programming and was first used only to calculate the edit distance (e. g. in [40, 43, 44]); though with minor changes a search variant is also possible [45, 47]. Using the unit cost error model (counting of errors = cost 1 per error), the general dynamic programming algorithm takes $O(nm)$. Based on this principle, a lot of other algorithms have been developed [31] which achieve $O(kn)$ in the worst case and $O(kn/\sqrt{\sigma})$ in the average case (like the algorithm of Chang and Lampe [9]), where σ is the size of the alphabet Σ.

To achieve a better average case behavior the concept of filtering was applied to approximate pattern matching (first by Tarhio and Ukkonen [53] followed by many others). The idea behind this concept is, that it is sometimes easier to decide for a text position that no approximate matching occurs than to ensure whether there is an approximate matching. Different filtering algorithms can be classified by the filtering approach and additionally by the online applicability. Unlike online algorithms, offline algorithms preprocess the text in advance by building an index to use it for a better performance during the search. Usual indexing data structures are suffix trees [12, 19, 46, 56], suffix arrays

[34], q-grams [8, 19, 29, 32] and q-samples [39, 51]. If the text is too large or a search is performed in the text very frequently, the preprocessing costs may pay off and offline algorithms can be used as alternative approaches to online algorithms.

There are different filtering approaches. Most approaches can be seen as applications of the following Lemma (see [36]):

Lemma Let A and B be two strings, such that $d(A, B) \leq k$. Let $A = A_1 x_1 A_2 \ldots x_{j-1} A_j$ for strings A_i and x_i and for any $j \geq 1$.

1. For $j < k+1$: Let k_i be any set of nonnegative numbers such that $\sum_{i=1}^{j} k_i \geq k-j+1$. Then, at least one string A_i appears with at most k_i errors in B.
2. For $j \geq k+1$, then
   a) at least $j-k$ strings $A_{i_1} \ldots, A_{i_{j-k}}$ appear in B.
   b) the relative distances from these $j-k$ strings inside B cannot differ from those in A by more than k.

Though not all filtering approaches can be categorized by this Lemma (e.g. [29, 38, 55]), it is very useful to classify filtering approaches. The first case of the lemma ($j < k+1$) characterizes a partitioning of the problem into smaller problem instances, while the second case ($j \geq k+1$) characterizes what is called *partitioning into exact search*. Furthermore it is a difference for an algorithm, whether A in the lemma is an occurrance $\bar{P}$ of P in T (i.e. the errors are assumed to be in the text) or A corresponds to P directly (i.e. the errors are assumed to be in the pattern). Figure 1 shows the classification of different filtering algorithms following the lemma.

	smaller instances	exact search
error in pattern	[5], [34]*	[6],[10],[18],[32]*,[46]*,[53],[58]
error in text	[11], [39]*	[19]*,[50]*,[51]*,[52]*

Figure 1: Classification of different filtering algorithms. * denotes an indexed algorithm in the referenced paper.

In this paper partitioning into exact search is used assuming that errors occur in the pattern. A brief overview on the algorithms of this class follows now. The algorithms of Jokinen et al. [18] and of Tarhio and Ukkonen [53] can be classified also as *counting filters*, as the number of characters fulfilling certain conditions in a text window is counted. In both algorithms j is chosen as m and thus every A_i of the lemma corresponds to a single character of the pattern. While in [18] the only condition is the number of exact matching characters between a text window and the pattern, in [53] the number of bad characters is counted, i.e. the number of characters that do neither match at the actual position nor in a distance of at most k. Whenever the counting condition is not fulfilled

in the text window, it is shifted further along the text using Boyer–Moore [7] techniques. Also Chang and Lawler [10] apply the lemma in the same way with $j = m$. Basically, they check whether more than $m - k$ text characters are needed to cover k strokes of consecutive character matchings with the pattern. To search the strokes of matching characters a suffix tree of the pattern is used.

The algorithm presented in Section 3 applies the lemma in the same way as Wu and Manber [58] or Baeza-Yates and Perleberg [6]. The pattern is split into $k+1$ pieces (the A_i in the lemma) and all of these pieces are searched exactly in the text. If one of these pieces is found, an area containing this exact matching is checked for an approximate matching with a non filtering algorithm. To search for the pattern pieces, in [58] an extension of shift-or [4] is used, while in [6] the algorithm of Aho and Corasick [2] (which is a multi pattern variant of the Knuth–Morris–Pratt [21] search algorithm) is applied.

Navarro and Baeza-Yates [32] implemented an indexed variant of the $k+1$ partitioning with searching the pattern pieces in a q-gram index. Shi [46] extended the principle of $k+1$ partitioning to $k+s, s \geq 1$ partitioning and performed the search of the pattern pieces with the help of a suffix tree index of the text.

Besides these algorithms with errors assumed in the pattern, there are also some algorithms (see Figure 1) considering the errors to be in the text while following the partitioning into exact search approach.

The algorithms in [50], [52] and [19] can be seen as earlier versions of the algorithm of Sutinen and Tarhio [51], where a q-sample index (samples taken with a distance h, $q \leq h < m$) of the text is used. All pattern q-grams are searched in the index and if at least s (depending on h) consecutive q-samples are matched a checking is triggered.

As with $O(m^2)$ the costs for checking are expensive compared to the linear time of the search algorithms, it is very important for the filtering algorithms that the checking time is not dominant, i. e. that the average checking costs are $O(1)$. This is heavily dependent on the error level $\alpha = k/m$, because the less errors are allowed, the less possible hits needs to be checked. The kind of filter considered in this paper (partitioning into exact search of $k+1$ pattern pieces) has been proven to be good for low error levels [31], but whenever there are too many possible hits to check, the time needed for checking is too high.

There are different ideas to reduce the overall time needed for checking. In a general improvement method for filtering algorithms, Giegerich et al. [16] mixed the checking phase with the search phase. With the information of the search phase about the maximal number of errors left, the checking phase can be stopped prematurely if in the progress of checking the actual number of errors shows that an approximate matching is not possible anymore. With *hierarchical verification* another idea was presented by Navarro and Baeza-Yates [30, 33, 35]. They applied the lemma mentioned above not only during the search phase of pattern pieces with $\lfloor k/j \rfloor$ errors, but also in the checking phase. Instead of checking the complete area at once, two neighboring pattern pieces are merged and checked for $\lfloor k/\frac{j}{2} \rfloor$ errors. This merging is successively continued until either the whole

pattern is found with at most k errors, or in one of the merging steps the checking failed. Another idea to reduce the overall checking time, is to adapt the checking algorithm for reusing the information already calculated, if the area to be checked partly overlaps with the last checked area (*patchwork verification*).

A very different idea for the same purpose is presented in this paper. To reduce the total amount of checking needed, the redundancy of the given text can be used.

In a problem closely related to approximate string matching the principle of redundancy reduction is also utilized. In compressed approximate string matching the text is considered to exist in a compressed (i. e. redundancy reduced) form. There are several different compression schemes and for various dictionary based methods like the Lempel-Ziv family [57, 59, 60], Sequitur [42], BPE (byte pair encoding) [13], Re-Pair (recursive pairing) [25] and run length encoding, Kida et al. [20] introduced a collage system as a unifying framework. They also introduced a general algorithm for exact compressed pattern matching within this framework, but the problem of compressed approximate string matching was not addressed. Kärkkäinen et al. [22] presented the first algorithm to solve this problem. Their algorithm is for LZ78 [60] and LZW [57] compressed texts and uses a dynamic programming approach to achieve $O(mk\bar{n} + r)$ time and $O(\bar{n}km + \bar{n}\log\bar{n})$ space, where r is the number of matches and $\bar{n}$ the compressed length of the text. Based on the same compression schemes Matsumoto et al. [27] presented an algorithm using bit-parrallel techniques and running in $O(k^2\bar{n} + km)$ time and $O(k^2\bar{n})$ space. !!!!Navarro et al. [37] presented an algorithm with a better practical behavior using a filtering approach. They perform a multi pattern search on pieces of the pattern followed by verification on a locally decompressed text if necessary. A different text compression scheme, run length encoding, is assumed by Mäkinen et al. [28]. Their algorithm can handle arbitrary costs of the basic edit operations and runs in $O(m\bar{n}\bar{m})$ time, where $\bar{m}$ is the compressed length of the pattern. For other compression methods the problem of compressed approximate string matching has not been solved yet.

The algorithm presented in this paper does not deal with the compressed variant of the approximate string matching problem, but uses the Sequitur compression scheme [42] in the preprocessing step. The basic idea of the Sequitur algorithm is to represent the input sequence in a way that no phrase appears twice or more often. To achieve this, every phrase appearing more than once is replaced with a nonterminal symbol representing a rule that exactly describes this phrase. The algorithm processes the input sequence linearly and takes care that the following two properties hold:

- no pair of adjacent symbols appears more than once in the grammar (guarantees uniqueness of rules)
- every rule is used more than once (guarantees that each rule is useful)

For example, processing the text `abcabababc` results in the grammar

$$S \quad \rightarrow \quad R_1 R_2 R_1$$

$$R_1 \rightarrow R_2\texttt{c}$$
$$R_2 \rightarrow \texttt{ab}$$

where S is the start symbol and R_1 and R_2 are nonterminals.

A Sequitur grammar can be calculated in linear time [42] and used as compression method, it has been shown [41] to be extremely effective in the compression of semi-structured text.

In the next section a filtering algorithm for approximate pattern matching will be described that uses the Sequitur grammar to gain redundancy information, which will be used to reduce the amount of checking needed. Thus, as a complete filtering algorithm, the algorithm is not an alternative to the other approaches to reduce overall checking amount described earlier. It can be seen as an extension of the partitioning into exact search filtering algorithm, that still could be combined with one of the methods to reduce checking amount.

3 The Algorithm

The algorithm described in this section solves the problem of approximate pattern matching as defined in Section 2. First, the general principle of the algorithm will be explained. Afterwards, a more formal description of the algorithm will reveal details. At the end of this section some variations of the algorithm are discussed.

3.1 General Principle

Generally the algorithm consists of two phases. The first phase is the preprocessing phase, where information about the text redundancy is gained. The second phase performs the actual search of the approximate occurrences of the pattern.

The preprocessing phase focuses on the text primarily, but a very few calculations on the pattern are necessary also. Using the Sequitur algorithm (Section 2) a grammar is inferred from the text T. With the grammar the text is represented by a starting rule referring to symbols and other rules, while the existence of every other rule means that the text component represented by this rule appears at least twice in T (Sequitur condition). Additionally, for every rule of the grammar the length of represented text component and a list of all positions where this rule occurs in the text is stored.

The pattern is divided into $k + 1$ pattern pieces or subpatterns as it is necessary for the kind of filtering approach (Section 2) used here.

The processing phase is basically the same as in any filtering algorithm based on the principle of partitioning into exact search. Though, besides a search and a verification step a third step is used to evaluate information gained in the other two steps. For the search step, here, a multi pattern search extension (like in [33]) of Sunday's algorithm [48]

is used. In the verification step the algorithm of Chang and Lampe [9] is applied. However, in principle any multi pattern search algorithm resp. approximate pattern matching algorithm could serve as search algorithm resp. verification algorithm.

Starting point for the algorithm is a list of search areas initialized with $[1, n]$. The first interval in the list of search areas is selected. If there is none, the loop of searching, verification and evaluation is stopped. Otherwise, a multi pattern search for all subpatterns is performed in this interval and stops either if one of the subpatterns was found or if none of the subpatterns could be found in the interval. In the second case, the actual interval is removed from the list of search areas and as long as there is another interval in the list, the search is started anew with the new first interval. If a subpattern was found, an interval including this position is determined. This interval needs to be wide enough to ensure that every approximate matching containing the matched subpattern is included. Using the checking algorithm, all approximate matchings are found in this interval. These resulting hits are stored in a list collecting all hits.

All information of the checking (matching subpattern, verified area, results), called *verify* from now on, are stored in a list of verifies for further use in the next step, where the preprocessed rule information is evaluated. This step begins with updating the first (and current) search interval of the list of search areas by setting the new interval start to the position right behind the position where the exact matchig was found. Then three important calculations are performed to gain benefits from the rule information. First, it is checked whether a rule (at the position of the first appearance of this rule) covers a verified area completely. If so, all the hits found for this verify are duplicated to every position of the rule (see Figure 2) and inserted into the list of hits. Second, if a verified area wasn't covered completely, it is checked whether a rule covers the subpattern that triggered the verification. If so, the verified area is duplicated to every position of the rule (see Figure 3) and is stored in a list containing intervals, that need further checking. And finally the third calculation takes the rules that are not affected by future verification areas and removes for every rule position the area from the list of search areas, that definitively can be excluded from further search (see Figure 4).

With the modified list of search areas the search is started anew as long as the list of search areas is not empty.

Finally, to complete the list of hits, every element in the list of intervals that needed further checking is verified.

In the following the different parts of the algorithm are described more formally to enable a better understanding of some details.

3.2 Preprocessing Text

Using the Sequitur algorithm [42], a grammar is constructed from the text. Based on the grammar, a few additional features that are essential for the algorithm are calculated:

PSfrag replacements

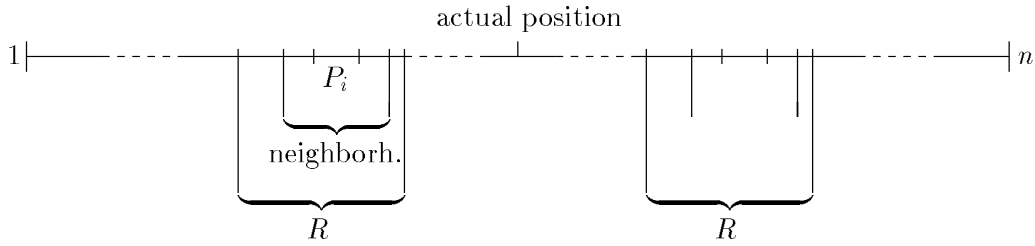


Figure 2: Duplication of checking results. In a former computational step a pattern piece P_i triggered a checking of a neighborhood. The neighborhood is covered by the rule R. All hits found in the neighborhood can be reproduced for every future occurrence of the rule R.

PSfrag replacements

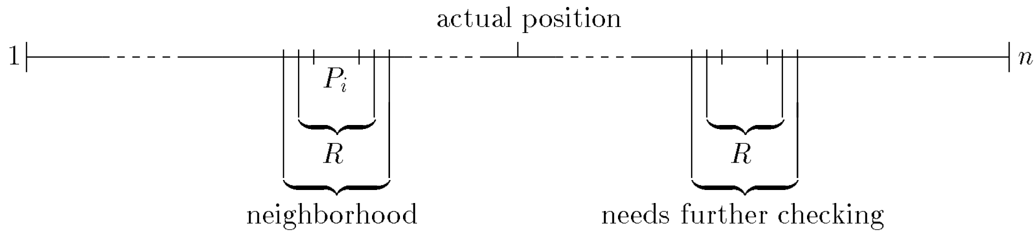


Figure 3: Duplication of checking information. The rule R covers only the pattern piece P_i completely, but not the neighborhood. For every future occurrence of the rule R, a neighborhood is marked to be checked later.

PSfrag replacements

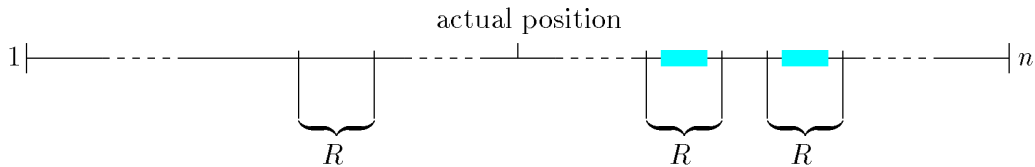


Figure 4: Preventing search on processed rules. For every future occurrence of the rule R a text aera (marked gray) can be excluded from further search.

- For each rule R the length $R.length$ of the rule, i.e. the number of text symbols covered by the rule, is determined.
- For each rule R, a sorted list $PositionList$ containing all positions of this rule in the text is build.
- An array SR containing all rules is constructed. In this array the rules are sorted regarding the position of the first appearance in the text (SR = sorted rules).
- For each rule R, the number $inclNumber$ of the rule a level above, i.e. the rule completely including this rule during the first appearance in the text, is determined. The number is set to -1 if there is no such rule.
- An array TR is build, containing for each text position the number of the last rule, that is completely included in the text up to this text position. For every rule only the end position of the first appearance is relevant. The number of the rules corresponds to the numbers in the SR-Array (TR = text-rule).
- An array TL is constructed, containing for each text position either the highest number of the rule, which covers this position or, if this position isn't covered by a rule, the number of the position before (TL = top level).

3.3 Preprocessing Pattern

The pattern is divided into $k+1$ subpatterns $P_1 \ldots P_{k+1}$. The length of the longest subpattern is P_{max}.

3.4 Processing

The processing can be divided into the three phases *initialization*, *search and verify* and *checking*, which will be explained in the following.

3.4.1 Initialization

In the initialization phase, a well-defined initial state for the other parts of the algorithm is created:

- List of search areas $SL = \{[1, n]\}$
- List of all final hits $Hitlist = \emptyset$
- List of *verifies* (verification areas) $VL = \emptyset$
  A verify contains the following information: the subpattern that matched exactly; the position in the text, where the subpattern matched; the start and the end position of the verification in the text; the number of true verified positions in this area; a pointer to the first of this true verified positions in the $Hitlist$.

- List of intervals, that need to be checked later $CL = \emptyset$
- Number of the last rule processed $R_l = -1$

3.4.2 Search and Verify

This phase contains the exact search filter and the evaluation of the search results. The algorithm *ProcessRules* (called in line 9) uses the rule information to duplicate search results if possible.

Algorithm *SearchAndVerify*
1. **while** $SL \neq \emptyset$ **do**
2. (Exact) linear multi pattern search of $P_1, \ldots, P_{k+1}$ in $[t_b, t_e]$
3. **if** P_i found at t_x **then**
4. $pos_b = t_x - (m + k - 1)$
5. $pos_e = t_x + (m + k - 1)$
6. Verify $[pos_b, pos_e]$
7. Insert the positions of the h resulting hits into the *Hitlist*.
8. Append the verify $(t_x, i, pos_b, pos_e, h, hPos)$ to VL (where $hPos$ is a pointer to the first element inserted into the *Hitlist*)
9. *ProcessRules*
10. **else**
11. Remove $[t_b, t_e]$ from SL

As the processing of rules in line 9 is a very crucial part of the algorithm, it's now described into more detail:

Algorithm *ProcessRules*
(* processes the rules: inferring hits and areas of no further interest *)
1. Substitute the first area $[t_b, t_e]$ in SL with $[t_x + 1, t_e]$
2. **for** $i = R_l + 1$ **to** $TR[t_x]$ (* only rules, which appeared at least once until t_x *)
3. $R = SR[i]$ (* get the rule *)
4. **if** ((R is long enough) **and** ($R.inclNumber > TR[t_x]$)) **then**
5. $CheckVL(R, i)$
6. $ActualizeSL(R)$
7. $R_l = TR[t_x]$ (* no rule needs to be considered twice *)

Algorithm *CheckVL*
Input: rule R, number i of R
(* tries to infer further hits from position of R and the elements of VL *)
1. **for** $(t_x, i, pos_b, pos_e, h, hPos) \in VL$
2. **if** $TL[pos_e] < i$ **then** (* this verify is not important anymore *)

3. Remove this verify from VL
4. **else**
5. **if** $R.Positionlist.first < t_x$ **then** (∗ this and none of the following verifies is covered by the R ∗)
6. **return**
7. **else**
8. **if** R covers $[pos_b, pos_e]$ completely **then**
9. Reproduce the hits of this verification to all positions in $R.Positionlist$
10. **elseif** R covers the subpattern P_i in this verify **then**
11. Reproduce this verify area to all positions in $R.Positionlist$ and insert these intervals in CL.

Algorithm *ActualizeSL*
Input: rule R
(∗ using positions of R to exclude intervals from SL, which are of no further interest ∗)
1. **for** $t_r \in R.PositionList$
2. Exclude $[t_r + P_{max} - 1, t_r + R.length - P_{max}]$ from SL

3.4.3 Checking

In the phases of final checking, all intervals collected in CL are verified. The positions of positive verifications are inserted into the $Hitlist$.

3.5 Improvements of the basic Algorithm

In the following some minor variations of the algorithms will be discussed.

3.5.1 Minimal Length of Rule

In line 2 of *ActualizeSL* the area is determined, which does not need to be examined further for exact machings of subpatterns. The size of this area depends on P_{max}, the size of the longest subpattern, and on $R.length$, the length of the rule. In order to exclude at least one symbol from further examination, the difference between both interval boundaries must be greater or equal to zero, i. e.

$$(t_r + R.length - P_{max}) - (t_r + P_{max} - 1) \geq 0.$$

Transforming this inequation results in

$$R.length \geq 2P_{max} - 1,$$

what is exactly one of the conditions inspected in line 4 of *ProcessRules*. It is obvious that it is better to skip larger areas and furthermore that it is worth to skip shorter areas

especially if positions are excluded, that trigger a verification (i. e. considering only the probabilities of the occurrence of symbols, it is easy to see, that the exclusion of shorter areas is more useful for smaller alphabets).

To be able to control the influence at this point, a parameter was introduced, that is also inspected in line 4 of *ProcessRules* and that defines the minimal length of rules to be processed at all.

3.5.2 Optimizing Checklist

It is possible, that in the list CL of intervals that need further verification (Section 3.4.3) some intervals are included, which are already verified in the end when it comes to the final verification. To avoid unnecessary checking, verifies can be stored instead of removing them in line 3 in *CheckVL* and thus it is possible to remove intervals that are already treated from CL right before the list is processed (Section 3.4.3).

3.5.3 Verification

Though the algorithm of Chang and Lampe [9] was implemented here, in general any approximate string matching algorithm (like [14, 24, 47, 54] or any other) could serve as verification algorithm.

Intuitively, it is better to select the area $[pos_b, pos_e]$, that is going to be verified (line 6 in *SearchAndVerify*), as small as possible. To determine pos_b and pos_e, first it is assumed, that the only knowledge is that P_i is a subpattern of P that matches at position t_x in T. With this, it could be possible that P_i is located at the very beginning of P and thus

$$pos_e = t_x + m + k - 1 \tag{3.1}$$

is obtained. On the other hand, P_i could be the last symbol of P and thus it is:

$$pos_b = t_x - m - k + 1 \tag{3.2}$$

Adding the additional information of the position p_x $(0 \leq p_x \leq m - 1)$ of P_i in P and the length $P_i.length$ of subpattern P_i, a smaller interval can be obtained. For that it is also important to know, whether the subpattern P_i is unique in P or not. In the first case, still all k errors could follow after t_x, but only a maximum of $m - p_x$ symbols of P_i could follow (including the position t_x). Also, all k errors could be before t_x and the first symbols of the pattern should be taken into account also. This results in:

$$pos_e = t_x + k + m - p_x - 1 \tag{3.3}$$

$$pos_b = t_x - k - p_x \tag{3.4}$$

In the second case (the subpattern P_i is not unique in P), nothing can be improved except considering that P_i is located at t_x and thus at least the symbols of P_i are not

before t_x. This case results in:

$$pos_e = t_x + k + m - 1 \quad (3.5)$$
$$pos_b = t_x - k - p_x \quad (3.6)$$

If a verify algorithm now completely processes the given interval $[pos_b, pos_e]$, it is for sure better to choose the shortest interval (i. e. equations 3.3-3.6).

4 Analytical Experiments

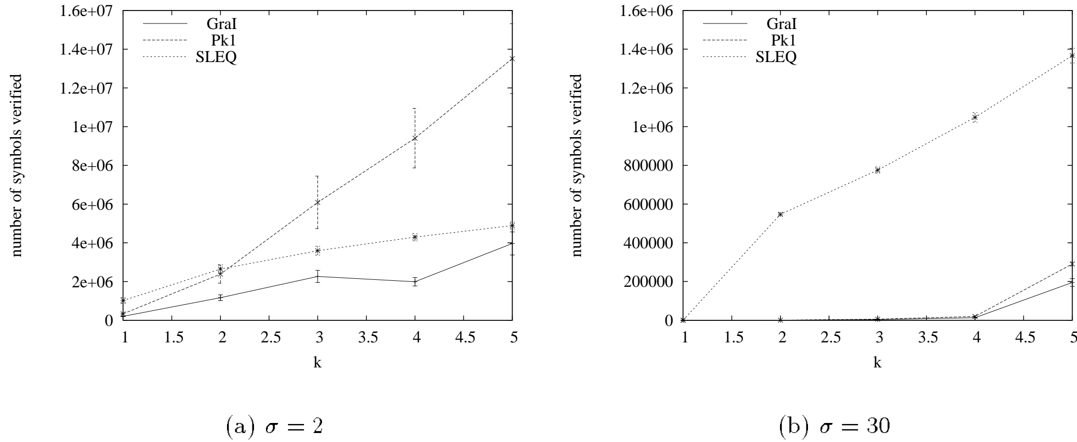
In this section, the algorithm of Section 3 is evaluated. For a better handling, this algorithm will be called *GraI* (Grammar based Index) here. As GraI was intended to reduce the amount of checking needed with a partitioning into exact search filter, the basic filtering algorithm of this kind (following the algorithm of Wu and Manber [58]) was also implemented. This algorithm will be called *Pk1* (Partitioning into k+1 pieces) here and uses the same algorithms for exact searching (multi pattern extension of Boyer–Moore–Sunday [48]) and for checking (Chang and Lampe [9]) like GraI. Furthermore, the *SLEQ* (static locations of exact q-grams) algorithm of Sutinen and Tarhio [51] serves as comparative indexed algorithm here, whereas the sample stepsize was chosen as $h = 3$ and the size of q-grams was $q = 3$. All algorithms were implemented in Java and all experiments were done on a 2.4 GHz Linux PC with 1 GB RAM. Within the experiments at least 20 repetitions were done, building a basis for the standard deviation bars in the figures.

During all experiments, the search pattern were randomly selected from the text. The text was either random with varying alphabet sizes σ or in English language (King James Bible converted to upper case, seperators except line breaks converted into space) with an alphabet size of 28.

In comparison to Pk1, GraI is intended to do less checking. As all algorithms share the same verification algorithm, it is sufficient to count the number of accesses to either text or pattern symbols in this algorithm. Figure 5 shows this number for a random text. It is obvious that for a very small alphabet (see Figure 5(a)) the effect of saving symbol accesses during checking with GraI is bigger than for larger alphabets (see Figure 5(b)). The reason for this is the grammar concept, which generates more rules (this is, where the algorithms starts working on), the more redundancy can be found in the text. Of course, a random text of a small alphabet contains more repeated areas than a random text of a larger alphabet.

Less usage of the verification algorithm has also a direct impact on the *filtration efficiency* f, which is a qualitative better measure. Basically, there are two possible ways to calculate the effective filtration efficiency.

- Filtration efficiency regarding the text length: $f_n = (n - n_p)/n$, where n_p denotes



(a) $\sigma = 2$

(b) $\sigma = 30$

Figure 5: Number of symbol accesses. $n = 100000$, $m = 10$

the number of text symbols considered during verification [51]. f_n describes the proportion of verified symbols.

- Filtration efficiency regarding the number of matches: $f = mat_r/mat_p$, where mat_r denotes the number of real matchings found and mat_p the number of potential matchings detected by the algorithm [49]. f describes the quota of real matchings per potential matching.

During the experiments, only f was calculated. A potential matching was counted every time when the verification algorithm was started. With this, f was calculated when the complete text was processed (and thus all matchings were found).

Note that even if f will be less then 1 mostly, it can be a greater than 1. This is the case, if the verification following after detecting a potential matching identifies more than one real matching and further this real maching does not trigger another potential matching later.

When comparing the algorithm to the basic partition into $k+1$ pieces filter, the conditions for achieving a better filtration efficiency can be calculated very roughly. As the filtering principle is exactly the same, it is only necessary to consider the cases, when a potential matching may lead to further potential or real matchings. This can only happen, if the condition

$$R.length \geq 2P_{max} - 1$$

holds (see Section 3.5.1). Not that but not every matching is necessarily part of a rule. For a noticeable effect it is necessary that this condition holds for the average rule.

Considering $P_{max} = m/(k+1)$ and the average rule length l_{av}, the condition is

$$l_{av} \geq 2\frac{m}{k+1} - 1.$$

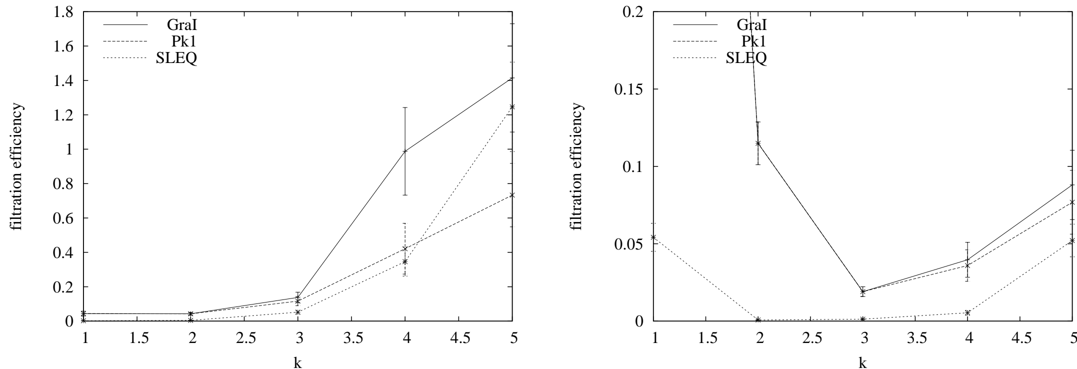
Sharpening this a little bit,

$$l_{av} \geq 2\frac{m}{k} - 1$$

is achieved. With the error level α defined as $\alpha = k/m$, this can be transformed to

$$\alpha \geq \frac{2}{l_{av}+1} \quad . \tag{4.1}$$

Figure 6(a) shows the filtration efficiency when searching random patterns of length $m = 10$ in a random text of length $n = 100000$ (alphabet size $\sigma = 4$). The average length of the rules is $l_{av} \doteq 6.63$. Using equation 4.1 results in error levels $\alpha \geq 0,262$, where the filtration efficiency is greater than that of Pk1. In Figure 6(a) the conformity of this calculated result and the measured value can be seen, as with an error level of $\alpha \geq k/m = 3/10$ the filtration efficiency is greater than that of Pk1 and also of the indexed algorithm SLEQ. Figure 6(b) illustrates the same effect in another case, but also that the analysis is really very roughly. Here the average length of rules is 3.67 and with equation 4.1 it is $\alpha \geq 0,428$. Thus, with $m = 10$ the effect of the better filtration efficiency should start around $k = 4$.

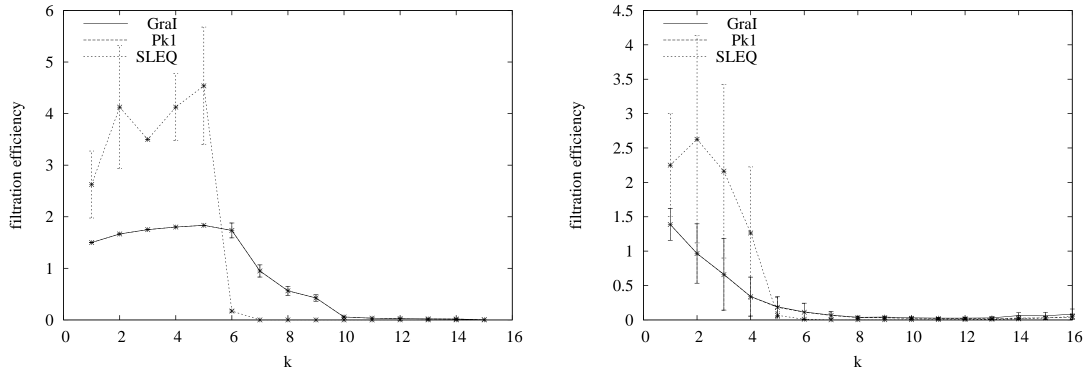


(a) $\sigma = 4, n = 100000, m = 10$

(b) $\sigma = 30, n = 20000, m = 10$

Figure 6: Filtration efficiency. A very crude analysis (equation 4.1) calculates roughly the point, where the filtration efficiency of GraI diverges from that of Pk1.

Considering equation 4.1 it is obvious that GraI performs better for higher error levels α. For low error levels and longer patterns SLEQ achieves a better filtering efficiency then GraI or Pk1. Figure 7 shows examples of this case.



(a) $\sigma = 30, n = 100000, m = 30$, random text (b) $\sigma = 28, n = 20000, m = 30$, English text

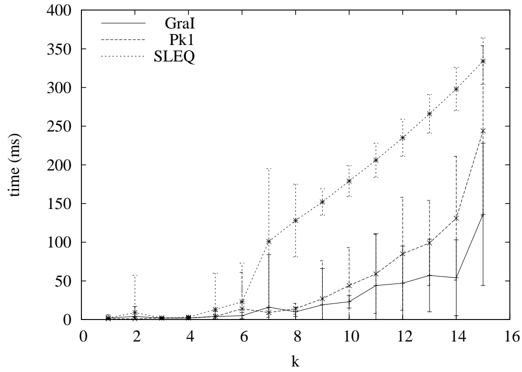
Figure 7: Filtration efficiency. GraI does not achieve a better filtration efficiency than Pk1 for low error levels and longer pattern. In fact, both algorithms can not compete with SLEQ in this case.

Not only the filtration efficiency is of interest, but also the search time. Figure 8 shows that GraI performs comparatively good for higher error levels (a higher number of errors k for a fixed m).

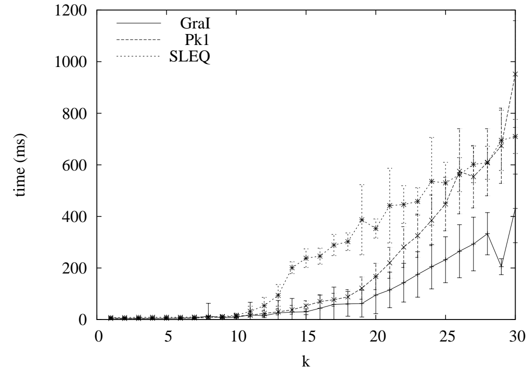
A better filtration efficiency does not always mean a better search performance. Usually, the price for this better filtration efficiency are higher expenses in other parts of the algorithm. Of course, in GraI the text is preprocessed, but this takes only linear time and is done once before performing several searches. Other expenses result from the management of additional information.

If there is a lot of additional information to manage, the search process is slowed down. This is the main drawback with GraI that for longer texts this effect of slowing down occurs. Figure 9 shows the same searches in the same English text, but restricted to different text lengths. For the longer text (Figure 9(b)) GraI performs worse than for the shorter text.

This leads to the benefits of the parameter introduced in Section 3.5.1 to allow skipping of shorter rules during the process of evaluation. The minimal length needed for a rule to be used for duplicating information (see Section 3.1) also controls directly the amount of additional information that is managed. Figure 10 illustrated the effect when the minimal length of rules is varied. Of course, the performance depends on the average length of rules in the grammar. If the minimal length of rules selected is a lot greater than the average length of rules in the grammar, not much can be gained, as only a very few rules will influence the calculation. In Figure 10 it can be seen that the overall best performance can be reached with a minimal rule length close to the average length of

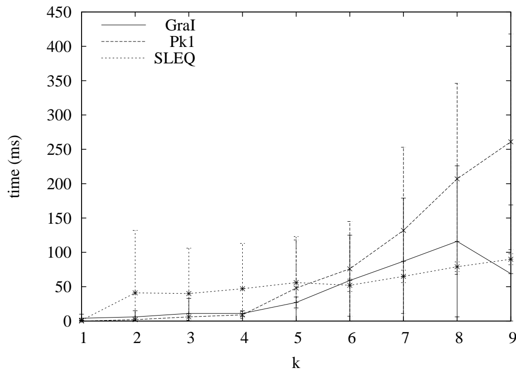


(a) $m = 30$

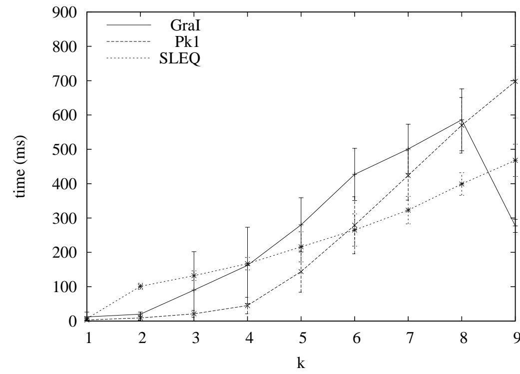


(b) $m = 60$

Figure 8: Search time. Time needed for searching pattern of different lengths in a part of the King James Bible of length $n = 20000$.



(a) $n = 20000$



(b) $n = 100000$

Figure 9: Expenses for management of additional information. In the longer English text, searching the same pattern ($m = 10$) takes more time.

rules in the grammar.

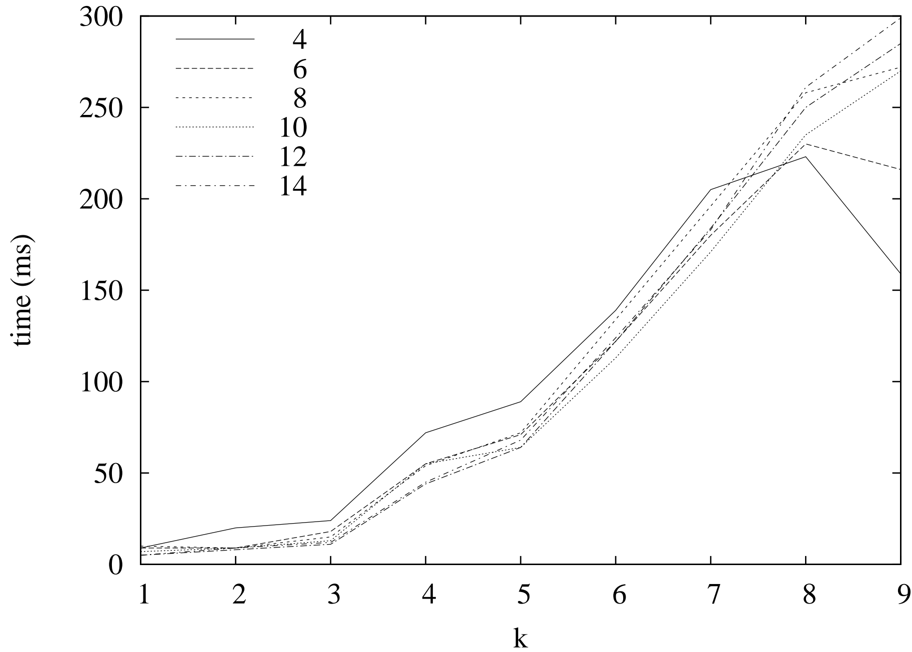


Figure 10: Effect of different minimal rule lengths. Approximate matching of short pattern ($m = 10$) in English text of length $n = 50000$. The average length of rules in the grammar of the text is 7.37.

5 Conclusions and Outlook

In this paper a filtering algorithm for approximate string matching is presented. The algorithms is based on the principle of partitioning into exact exact search with the intention to reduce the overall amount of checking. Using the Sequitur grammar, this aim was achieved for the filtering approach with partitioning the pattern into $k + 1$ pieces. It was also shown that compared to the basic approach a better filtration efficiency is reached for higher error levels. The algorithms perform comparatively good for a higher number of errors, but for long texts the expenses for the management of additional information dominate. Within small limits, this influence can be controlled with a parameter of the algorithm.

To remove the influence of management costs in longer text, the text could be devided in blocks that are processed seperately. Then, rule information should be duplicated only when processing of a block is finished.

A Sequitur grammar can be extended online for a growing text. Here, the additional features needed by the algorithm (Section 3.2) are calculated separately, but the calculation could be integrated into the Sequitur algorithm to achieve a better applicability.

Combining this algorithm with other methods to reduce the overall checking amount like hierarchical verification or patchwork verificaction (Section 2) is still possible and may lead to an enhanced overall performance.

Acknowledgements

Thanks to André Leier for numerous discussions and productive notes.

References

[1] *Proceedings of the 11th IEEE Data Compression Conference (DCC '01)*, IEEE Computer Society Press, March 2001.

[2] A. V. Aho and M. J. Corasick, *Efficient string matching: An aid to bibliographic search*, Communications of the ACM, 16 (1975), pp. 333–340.

[3] A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM'92)*, vol. 644 of LNCS, Springer, April/May 1992.

[4] R. A. Baeza-Yates and G. H. Gonnet, *A new apporach to text searching*, Communications of the ACM, 35 (1992), pp. 74–82. Preliminary version in ACM SIGR'89.

[5] R. A. Baeza-Yates and G. Navarro, *Faster approximate string matching*, Algorithmica, 23 (1999), pp. 127–158. Preliminary versions in Proceedings of CPM'96 (LNCS, vol. 1075, 1996) and in Proceedings of WSP'96, Carleton Uni. Press, 1996.

[6] R. A. Baeza-Yates and C. H. Perleberg, *Fast and practical approximate string matching*, in Apostolico et al. [3], pp. 185–192.

[7] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Communications of the ACM, 20 (1977), pp. 762–772.

[8] S. Burkhardt and J. Kärkkäinen, *One-gapped q-gram filters for Levenshtein distance*, in Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02), A. Apostolico and M. Takeda, eds., vol. 2373 of LNCS, Fukuoka, Japan, July 2002, Springer, pp. 225–234.

[9] W. I. Chang and J. Lampe, *Theoretical and empirical comparisons of approximate string matching algorithms*, in Apostolico et al. [3], pp. 175–184.

[10] W. I. Chang and E. L. Lawler, *Sublinear approximate string matching and biological applications*, Algorithmica, 12 (1994), pp. 327–344. Preliminary version in FOCS'90.

[11] W. I. Chang and T. G. Marr, *Approximate string matching and local similarity*, in Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94), M. Crochemore and D. Gusfield, eds., vol. 807 of LNCS, Asilomar, California, USA, June 1994, Springer, pp. 259–273.

[12] A. L. Cobbs, *Fast approximate matching using suffix trees*, in Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95), Z. Galil and E. Ukkonen, eds., vol. 937 of LNCS, Espoo, Finland, July 1995, Springer, pp. 41–54.

[13] P. Gage, *A new algorithm for data compression*, The C Users Journal, 12 (1994), pp. 23–38.

[14] Z. Galil and K. Park, *An improved algorithm for approximate string matching*, SIAM Journal on Computing, 19 (1990), pp. 989–999. Preliminary version in ICALP'89 (LNCS, vol. 372, 1989).

[15] R. Giancarlo and D. Sankoff, eds., *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, vol. 1848 of LNCS, Springer, June 2000.

[16] R. Giegerich, F. Hischke, S. Kurtz, and E. Ohlebusch, *A general technique to improve filter algorithms for approximate string matching*, in Proceedings of the Fourth South American Workshop on String Processing (WSP'97), R. Baeza-Yates, ed., Valparaiso, Chile, November 1997, Carleton University Press, pp. 38–52.

[17] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[18] P. Jokinen, J. Tarhio, and E. Ukkonen, *A comparison of approximate string matching algorithms*, Software—Practice and Experience, 26 (1996), pp. 1439–1458.

[19] P. Jokinen and E. Ukkonen, *Two algorithms for approximate string matching in static texts*, in Proceedings of 16th International Symposium on Mathematical Foundations of Computer Science (MFCS'91), A. Tarlecki, ed., vol. 520 of LNCS, Kazimierz Dolny, Poland, September 1991, Springer-Verlag, Berlin, pp. 240–248.

[20] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, *A unifying framework for compressed pattern matching*, in Proceedings of the 6th international

Symposium on String Processing and Information Retrieval (SPIRE'99), Cancun, Mexico, September 1999, IEEE CS Press, pp. 89–96.

[21] D. E. Knuth, J. H. Morris, jr., and V. R. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing, 6 (1977), pp. 323–350.

[22] J. Kärkkäinen, G. Navarro, and E. Ukkonen, *Approximate string matching over ziv-lempel compressed text*, in Giancarlo and Sankoff [15], pp. 195–209.

[23] K. Kukich, *Techniques for automatically correcting words in text*, ACM Computing Surveys, 24 (1992), pp. 377–439.

[24] G. M. Landau and U. Vishkin, *Fast parallel and serial approximate string matching*, Journal of Algorithms, 10 (1989), pp. 157–169. Preliminary version in ACM STOC'85.

[25] N. J. Larsson and A. Moffat, *Offline dictionary-based compression*, in Proceedings of the IEEE Data Compression Conference (DCC '99), Snowbird, Utah, USA, March 1999, IEEE Computer Society Press, pp. 296–305.

[26] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Soviet Physics - Doklady, 10 (1966), pp. 707–710. Original in Russian in *Doklady Akademii Nauk SSSR*, 163, 4, 845–848, 1965.

[27] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa, *Bit-parallel approach to approximate string matching in compressed texts*, in Proceedings of the 7th international Symposium on String Processing and Information Retrieval (SPIRE'00), A Coruña, Spain, September 2000, IEEE CS Press, pp. 221–228. Preliminary version as Tech. Rep. DOI-TR-174, Dept. of Informatics, Kyushu University, 2000.

[28] V. Mäkinen, G. Navarro, and E. Ukkonen, *Approximate matching of run-length compressed strings*, Algorithmica, 35 (2003), pp. 347–369.

[29] E. W. Myers, *A sublinear algorithm for approximate keyword searching*, Algorithmica, 12 (1994), pp. 345–374. Earlier version in Tech. Rep. TR-90-25, Dept. of Computer Science, Univ. of Arizona, 1990.

[30] G. Navarro, *Approximate Text Searching*, PhD thesis, Department of Computer Science, University of Chile, Santiago, Chile, December 1998.

[31] G. Navarro, *A guided tour to approximate string matching*, ACM Computing Surveys, 33 (2001), pp. 31–88.

[32] G. Navarro and R. Baeza-Yates, *A practical q-gram index for text retrieval allowing errors*, CLEI Electronic Journal, 1 (1998). Previous version in Proceedings of the XXII Latin American Conference on Informatics (CLEI'97).

[33] G. Navarro and R. Baeza-Yates, *Very fast and simple approximate string matching*, Information Processing Letters (IPL), (1999), pp. 65–70.

[34] G. Navarro and R. Baeza-Yates, *A hybrid indexing method for approximate string matching*, Journal of Discrete Algorithms, 1 (2000), pp. 205–239. Previous version in CPM'99.

[35] G. Navarro and R. Baeza-Yates, *Improving an algorithm for approximate pattern matching*, Algorithmica, 30 (2001), pp. 473–502. Previous version: Tech. Rep. TR/DCC-98-5, Dept. of Computer Science, University of Chile.

[36] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, *Indexing methods for approximate string matching*, IEEE Data Engineering Bulletin, 24 (2001), pp. 19–27. Special issue on Managing Text Natively and in DBMSs. Invited paper.

[37] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa, *Faster approximate string matching over compressed text*, in Proceedings of the 11th IEEE Data Compression Conference (DCC '01) [1], pp. 459–468.

[38] G. Navarro and M. Raffinot, *Fast and flexible string matching by combining bit-parallelism and suffix automata*, ACM Journal of Experimental Algorithmics (JEA), 5 (2000). Previous version in Proceeedings of CPM'98. LNCS, Springer Verlag, New York.

[39] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio, *Indexing text with approximate q-grams*, in Giancarlo and Sankoff [15], pp. 350–363.

[40] S. B. Needleman and C. D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology, 48 (1970), pp. 443–453.

[41] C. G. Nevill-Manning and I. H. Witten, *Compression and explanation using hierarchical grammars*, Computer Journal, 40 (1997), pp. 103–116.

[42] C. G. Nevill-Manning and I. H. Witten, *Identifying hierarchial structures in sequences: A linear-time algorithm*, Journal of Artificial Intelligence Research, 7 (1997), pp. 67–82.

[43] D. Sankoff and J. B. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Massachusetts, 1983.

[44] P. H. Sellers, *On the theory and computation of evolutionary distances*, SIAM Journal on Applied Mathematics, 26 (1974), pp. 787–793.

[45] P. H. Sellers, *The theory and computation of evolutionary distances: Pattern recognition*, Journal of Algorithms, 1 (1980), pp. 359–373.

[46] F. Shi, *Fast approximate string matching with q-blocks sequences*, in Proceedings of the Third South American Workshop on String Processing (WSP'96), N. Ziviani, R. Baeza-Yates, and K. S. Guimarães, eds., Carleton University Press, pp. 257–271.

[47] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences*, Journal of Molecular Biology, 147 (1981), pp. 195–197.

[48] D. M. Sunday, *A very fast substring search algorithm*, Communications of the ACM, 33 (1990), pp. 132–142.

[49] E. Sutinen, *Approximate Pattern Matching with the q-gram Family*, PhD thesis, Department of Computer Science, University of Helsinki, Finland, 1998. Tech. Rep. TR A-1998-3.

[50] E. Sutinen and J. Tarhio, *On using q-gram locations in approximate string matching*, in Proceedings of Third Annual European Symposium on Algorithms (ESA'95), P. Spirakis, ed., vol. 979 of LNCS, Corfu, Greece, September 1995, Springer-Verlag, Berlin, pp. 327–340.

[51] E. Sutinen and J. Tarhio, *Filtration with q-samples in approximate string matching*, in Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96), D. S. Hirschberg, ed., vol. 1075 of LNCS, Laguna Beach, California, USA, June 1996, Springer, pp. 50–63.

[52] T. Takaoka, *Approximate pattern matching with samples*, in Proceedings of ISAAC'94, D.-Z. Du and X.-S. Zhang, eds., vol. 834 of LNCS, Springer-Verlag, Berlin, 1994, pp. 234–242.

[53] J. Tarhio and E. Ukkonen, *Approximate boyer-moore string matching*, SIAM Journal on Computing, 22 (1993), pp. 243–260. Preliminary version in SWAT'90 (LNCS, vol. 447, 1990).

[54] E. Ukkonen, *Algorithms for approximate string matching*, Information and Control, 64 (1985), pp. 100–118. Preliminary version presented at the International Conference on "Foundations of Computation Theory", Sweden, Aug. 1983.

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

[56] E. Ukkonen, *Approximate string-matching over suffix trees*, in Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93), A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., vol. 684 of LNCS, Padova, Italy, June 1993, Springer, pp. 228–242.

[57] T. A. Welch, *A technique for high performance data compression*, IEEE Computer Magazine, 17 (1984), pp. 8–19.

[58] S. Wu and U. Manber, *Fast text searching allowing errors*, Communications of the ACM, 35 (1992), pp. 83–91.

[59] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory, 23 (1977), pp. 337–343.

[60] J. Ziv and A. Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Transactions on Information Theory, 24 (1978), pp. 530–536.