New and Faster Filters for Multiple Approximate String Matching

Ricardo Baeza-Yates, Gonzalo Navarro
Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile

Received 24 November 1998; revised 18 October 1999; accepted 8 December 2000

ABSTRACT: We present three new algorithms for on-line multiple string matching allowing errors. These are extensions of previous algorithms that search for a single pattern. The average running time achieved is in all cases linear in the text size for moderate error level, pattern length, and number of patterns. They adapt (with higher costs) to the other cases. However, the algorithms differ in speed and thresholds of usefulness. We theoretically analyze when each algorithm should be used, and show their performance experimentally. The only previous solution for this problem allows only one error. Our algorithms are the first to allow more errors, and are faster than previous work for a moderate number of patterns (e.g. less than 50–100 on English text, depending on the pattern length). © 2001 John Wiley & Sons, Inc. Random Struct. Alg., 20, 23–49, 2001

Key Words: string matching; multipattern search; search allowing errors

1. INTRODUCTION

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Given a text $T_{1,\ldots,n}$ of length n and a pattern $P_{1,\ldots,m}$ of length m

Correspondence to: Gonzalo Navarro; e-mail: gnavarro@dcc.uchile.cl.
Contract grant sponsor: FONDECYT.
Contract grant number: 1990627.

DOI 10.1002/rsa.10014

(both sequences over an alphabet Σ of size σ), and a maximal number of errors allowed, $0 < k < m$, we want to find all text positions where the pattern matches the text with up to k errors. Errors can be substituting, deleting or inserting a character. We use the term "error level" to refer to $\alpha = k/m$.

In this article, we are interested in the on-line problem (i.e., the text is not known in advance), where the classical solution for a single pattern is based on dynamic programming and has a running time of $O(mn)$ [26].

In recent years, several algorithms have improved the classical one [22]. Some improve the worst or average case by using the properties of the dynamic programming matrix [9, 11, 16, 30, 31]. Others filter the text to quickly eliminate uninteresting parts [10, 14, 24, 28, 29], some of them being "sublinear" on average for moderate α (i.e., they do not inspect all the text characters). Yet other approaches use bit-parallelism [3] in a computer word of w bits to reduce the number of operations [6, 19, 33–35].

The problem of approximately searching a set of r patterns (i.e., the occurrences of anyone of them) has been considered only recently. This problem has many applications, for instance

- Spelling: many incorrect words can be searched in the dictionary at a time, to find their most likely variants. Moreover, we may even search the dictionary of correct words in the "text" of misspelled words, hopefully at a much lesser cost.
- Information retrieval: when synonym or thesaurus expansion is done on a keyword and the text is error-prone, we may want to search all the variants allowing errors.
- Batched queries: if a system receives a number of queries to process, it may improve efficiency by searching all of them in a single pass.
- Single-pattern queries: some algorithms for a single pattern allowing errors (e.g., pattern partitioning [6]) reduce the problem to the search of many subpatterns allowing less errors, and they benefit from multipattern search algorithms.

A trivial solution to the multipattern search problem is to perform r searches. As far as we know, the only previous attempt to improve the trivial solution is due to Muth and Manber [17], who use hashing to search many patterns with *one* error, being efficient even for 1000 patterns.

In this work, we present three new algorithms that are extensions of previous ones to the case of multiple search. In Section 2, we explain some basic concepts necessary to understand the algorithms. Then we present the three new techniques. In Section 3, we present "automaton superimposition," which extends a bit-parallel simulation of a nondeterministic finite automaton (NFA) [6]. In Section 4, we present "exact partitioning," that extends a filter based on exact searching of pattern pieces [6, 7, 24]. In Section 5, we present "counting," based on counting pattern letters in a text window [14]. In Section 6, we analyze our algorithms and in Section 7, we compare them experimentally. Finally, in Section 8, we give our conclusions. Some detailed analyses are left for Appendices A and B.

Although [17] allows searching for many patterns, it is limited to only one error. Ours are the first algorithms for multipattern matching allowing more than one

error. Moreover, even for one error, we improve [17] when the number of patterns is not very large (say, less than 50–100 on English text, depending on the pattern length). Our multipattern extensions improve over their sequential counterparts (i.e., one separate search per pattern using the base algorithm) when the error level is not very high (about $\alpha \leq 0.4$ on English text). The filter based on exact searching is the fastest for small error levels, while the bit-parallel simulation of the NFA adapts better to more errors on relatively short patterns.

Previous partial and preliminary versions of this work appeared in [5, 20, 21].

2. BASIC CONCEPTS

We review in this section some basic concepts that are used in all the algorithms that follow. In this article S_i denotes the ith character of string S (S_1 being the first character), and $S_{i,\ldots,j}$ stands for the substring $S_i, S_{i+1}, \ldots, S_j$. In particular, if $i > j$, $S_{i,\ldots,j} = \epsilon$, the empty string.

2.1. Filtering Techiques

All the multipattern search algorithms that we consider in this work are based on the concept of *filtering*, and therefore it is useful to define it here.

Filtering is based on the fact that it is normally easier to tell that a text position does not match than to ensure that it matches. Therefore, a *filter* is a fast algorithm that checks for a simple necessary (though not sufficient) condition for an approximate match to occur. The text areas that do not satisfy the necessary condition can be safely discarded, and a more expensive algorithm has to be run on the text areas that passed the filter.

Since the filters can be much faster than approximate searching algorithms, filtering algorithms can be very competitive (in fact, they dominate on a large range of parameters). The performance of filtering algorithms, however, is very sensitive to the error level α. Most filters work very well on low-error levels and very badly with more errors. This is related with the amount of text that the filter is able to discard. When evaluating filtering algorithms, it is important not only to consider their time efficiency but also their tolerance to errors.

A term normally used when referring to filters is "sublinearity." It is said that a filter is sublinear when it does not inspect all the characters of the text (like the Boyer–Moore [8] algorithms for exact searching, which can be at best $O(n/m)$).

Throughout this work, we make use of the following two lemmas to derive filtering conditions.

Lemma 1 [6]. *If $S = T_{a..b}$ matches P with at most k errors, and $P = p_1, \ldots, p_j$ (a concatenation of subpatterns), then some substring of S matches at least one of the p_is, with at most $\lfloor k/j \rfloor$ errors.*

Proof. Otherwise, the best match of each p_i inside S has at least $\lfloor k/j \rfloor + 1 > k/j$ errors. An occurrence of P involves the occurrence of each of the p_is, and the total number of errors in the occurrences is at least the sum of the errors of the pieces. But here, just summing-up the errors of all the pieces we have more than $jk/j = k$

errors and therefore a complete match is not possible. Notice that this does not even consider that the matches of the p_i must be in the proper order, be disjoint, and that some deletions in S may be needed to connect them. ■

In general, one can filter the search for a pattern of length m with k errors by the search of j subpatterns of length m/j with k/j errors. Only the text areas surrounding occurrences of pieces must be checked for complete matches.

An important particular case of Lemma 1 arises when one considers $j = k + 1$, since in this case some pattern piece appears unaltered (zero errors).

Lemma 2 [32]. *If $T_{i,\ldots,j}$ matches P with k errors or less, then $T_{j-m+1,\ldots,j}$ includes at least $m - k$ characters of P.*

Proof. Suppose the opposite. If $j - i < m$, then we observe that there are less than $m - k$ characters of P in $T_{i,\ldots,j}$. Hence, more than k characters must be deleted from P to match the text. If $j - i \geq m$, we observe that there are more than k characters in $T_{i,\ldots,j}$ that are not in P, and hence we must insert more than k characters in P to match the text. A contradiction in both the cases. ■

Note that in case of repeated characters in the pattern, they must be counted as different occurrences. For example, if we search "`aaaa`" with one error in the text, the last four letters of each occurrence must include at least three `a`'s.

Lemma 2 (a simplification of that in [32]) says essentially that we can design a filter for approximate searching based on finding enough characters of the pattern in a text window (without regarding their ordering). For instance, the pattern "`survey`" cannot appear with one error in the text window "`surger`" because there are not five letters of the pattern in the text. However, the filter cannot discard the possibility that the pattern appears in the text window "`yevrus`".

2.2. Bit-Parallelism

Bit-parallelism is a technique of common use in string matching [3]. It was first proposed in [2, 4]. The technique consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By cleverly using this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in the computer word.

Since in current architectures w is 32 or 64, the speedup is very significant in practice (and improves with technological progress). To relate the behavior of bit-parallel algorithms to other works, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model of computation. However, we prefer to keep w as an independent value.

Some notation we use for bit-parallel algorithms is in order. We denote as $b_\ell, \ldots, b_1$ the bits of a mask of length ℓ, which is stored somewhere inside the computer word. We use C-like syntax for operations on the bits of computer words, e.g., "|" is the bitwise-or and "<<" moves the bits to the left and enters zeros from the right, e.g., $b_m b_{m-1}, \ldots, b_2 b_1 << 3 = b_{m-3}, \ldots, b_2 b_1 000$. We can also perform arithmetic operations on the bits, such as addition and

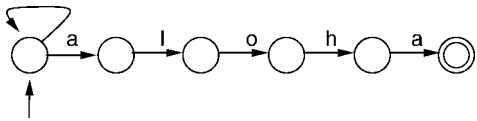

Fig. 1. Nondeterministic automaton that searches "aloha" exactly.

subtraction, which operate the bits as if they formed a number. For instance, $b_\ell, \ldots, b_x 10000 - 1 = b_\ell, \ldots, b_x 01111$.

We now explain the first bit-parallel algorithm, since it is the basis of much of which follows in this work. The algorithm searches a pattern in a text (without errors) by parallelizing the operation of a non-deterministic finite automaton that looks for the pattern. Figure 1 illustrates this automaton.

This automaton has $m + 1$ states, and can be simulated in its nondeterministic form in $O(mn)$ time. The Shift-Or algorithm achieves $O(mn/w)$ worst-case time (i.e., optimal speedup). Notice that if we convert the nondeterministic automaton to a deterministic one to have $O(n)$ search time, we get an improved version of the KMP algorithm [15]. However, KMP is twice as slow for $m \leq w$.

The algorithm first builds a table $B[\,]$ which for each character c stores a bit mask $B[c] = b_m, \ldots, b_1$. The mask $B[c]$ has the bit b_i in zero if and only if $P_i = c$. The state of the search is kept in a machine word $D = d_m, \ldots, d_1$, where d_i is zero whenever $P_{1,\ldots,i}$ matches the end of the text read up to now (i.e., the state numbered i in Fig. 1 is active). Therefore, a match is reported whenever d_m is zero.

D is set to all ones originally, and for each new text character T_j, D is updated using the formula

$$D' \leftarrow (D \ll 1) \mid B[T_j].$$

The formula is correct because the ith bit is zero if and only if the $(i-1)$th bit was zero for the previous text character and the new text character matches the pattern at position i. In other words, $T_{j-i+1,\ldots,j} = P_{1,\ldots,i}$ if and only if $T_{j-i+1,\ldots,j-1} = P_{1,\ldots,i-1}$ and $T_j = P_i$. It is possible to relate this formula to the movement that occurs in the nondeterministic automaton for each new text character: each state gets the value of the previous state, but this happens only if the text character matches the corresponding arrow.

For patterns longer than the computer word (i.e., $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time). The algorithm is $O(mn/w)$ worst case time, and the preprocessing is $O(m + \sigma)$ time and $O(\sigma)$ space. On average, the algorithm is $O(n)$ even when $m > w$, since only the first $O(1)$ states of the automaton have active states on average (and hence the first $O(1)$ computer words need to be updated on average).

It is easy to extend Shift-Or to handle *classes of characters*. In this extension, each position in the pattern matches with a *set* of characters rather than with a single character. The classical string matching algorithms are not so easily extended. In Shift-Or, it is enough to set the ith bit of $B[c]$ for every $c \in P_i$ (P_i is a set now). For instance, to search for "survey" in case-insensitive form, we just set the first bit of B["s"] and of B["S"] to "match" (zero), and the same with the rest. Shift-Or can also search for multiple patterns (where the complexity is $O(mn/w)$ if we consider that m is the total length of all the patterns) by arranging many masks B and D in the same machine word. Shift-Or was later enhanced [34] to support a larger set

of extended patterns and even regular expressions. Recently, in [25], Shift-Or was combined with a sublinear string matching algorithm, obtaining the same flexibility and an efficiency competitive with the best classical algorithms.

Many on-line text algorithms can be seen as implementations of clever automata (classically, in their deterministic form). Since its invention, bit-parallelism became a general way to simulate simple nondeterministic automata instead of converting them to deterministic. It has the advantage of being much simpler, in many cases faster (since it makes better usage of the registers of the computer word), and easier to extend to handle complex patterns than its classical counterparts. Its main disadvantage is the limitations it imposes with regard to the size of the computer word. In many cases, its adaptations to cope with longer patterns are not so efficient.

2.3. Bit-parallelism for Approximate Pattern Matching

We now present an application of bit-parallelism to approximate pattern matching, which is especially relevant for the present work.

Consider the NFA for searching "patt" with at most $k = 2$ errors shown in Figure 2. Every row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. At each iteration, a new text character is considered and the automaton changes its states. Horizontal arrows represent matching a character (they can only be followed if the corresponding match occurs). All the others represent errors, as they move to the next row. Vertical arrows represent inserting a character in the pattern (since they advance in the text and not in the pattern), solid diagonal arrows represent substituting a character (since they advance in the text and the pattern), and dashed diagonal arrows represent deleting a character of the pattern (since, as ϵ-transitions, they advance in the pattern but not in the text). The loop at the initial state allows considering any character as a potential starting point of a match. The automaton accepts a character (as the end of a match) whenever a rightmost state

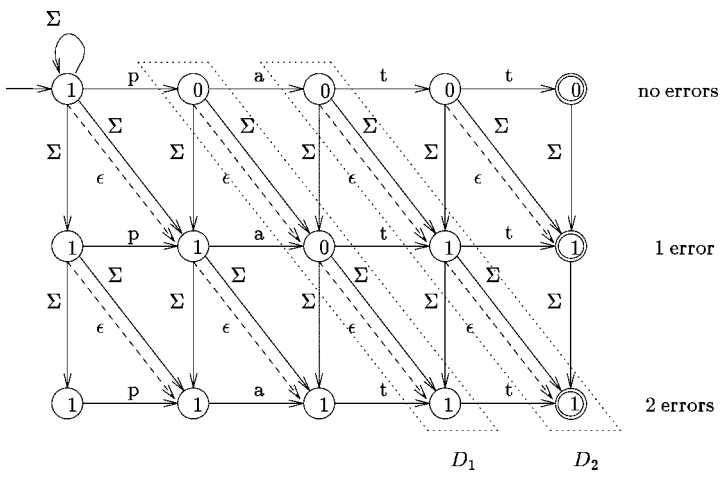

Fig. 2. An NFA for approximate string matching. We show the active states after reading the text "pait".

is active. Initially, the active states at row i ($i \in 0, \ldots, k$) are those at the columns from 0 to i, to represent the deletion of the first i characters of the pattern $P_{1,\ldots,m}$.

An interesting application of bit-parallelism is to simulate this automaton in its nondeterministic form. A first approach [34] obtained $O(k\lceil m/w \rceil n)$ time, by packing each automaton row in a machine word and extending the Shift-Or algorithm to account for the vertical and diagonal arrows. Note that, even if all the states fit in a single machine word, the $k+1$ rows have to be sequentially updated because of the ϵ-transitions. The same happens in the classical dynamic programming algorithm [26], which can be regarded as a columnwise simulation of this NFA.

In this article, we are interested in a more recent simulation technique [6], where we show that by packing *diagonals* of the automaton instead of rows or columns all the new values can be computed in one step if they fit in a computer word. We give a brief description of the idea.

Because of the ϵ-transitions, once a state in a diagonal is active, all the subsequent states in that diagonal become active too, so we can define the minimal active row of each diagonal, D_i (diagonals are numbered by looking the column they start at, e.g., D_1 and D_2 are enclosed in dotted lines in Figure 2). The new values for D_i ($i \in 1, \ldots, m-k$) after we read a new text character c can be computed by

$$D'_i = \min(D_i + 1, D_{i+1} + 1, g(D_{i-1}, c)),$$

where

$$g(D_i, c) = \min(\{k+1\} \cup \{j/j \geq D_i \wedge P_{i+j+1} = c\}),$$

where it always holds $D_0 = D'_0 = 0$ and we report a match whenever $D_{m-k} \leq k$. The formula for D'_i accounts for replacements, insertions, and matches, respectively. Deletions are accounted for by keeping the minimum active row. All the interesting matches are caught by considering only the diagonals $D_1, \ldots, D_{m-k}$.

We use bit-parallelism to represent the D_is in unary. Each one is hold in $k+1$ bits (plus an overflow bit) and stored sequentially inside a bit mask D. Interestingly, the effect is the same if we read the diagonals bottom-up and exchange $0 \leftrightarrow 1$, with each bit representing a state of the NFA. The update formula can be seen either as an arithmetic implementation of the previous formula in unary or as a logical simulation of the flow of bits across the arrows of the NFA.

As in Shift-Or, a table of (m bits long) masks $b[\,]$ is built representing match or mismatch against the pattern. A table $B[c]$ is built by mapping the bits of $b[\,]$ to their appropriate positions inside D. Figure 3 shows how the states are represented inside the masks D and B.

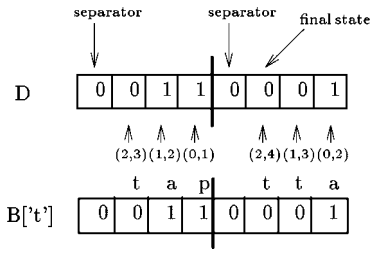

Fig. 3. Bit-parallel representation of the NFA of Figure 2.

This representation requires $k+2$ bits per diagonal, so the total number of bits is $(m-k)(k+2)$. If this number of bits does not exceed the computer word size w, the update can be done in $O(1)$ operations. The resulting algorithm is linear and very fast in practice.

For our purposes, it is important to realize that the only connection between the pattern and the algorithm is given by the $b[\,]$ table, and that the pattern can use classes of characters just as in the Shift-Or algorithm. We use this property next to search for multiple patterns.

3. SUPERIMPOSED AUTOMATA

In this section, we describe an approach based on the bit-parallel simulation of the NFA just described.

Suppose we have to search r patterns $P^1, \ldots, P^r$. We are interested in the occurrences of any one of them, with at most k errors. We can extend the previous bit-parallelism approach by building the automaton for each one, and then "superimpose" all the automata.

Assume that all the patterns have the same length (otherwise, truncate them to the shortest one). Hence, all the automata have the same structure, differing only in the labels of the horizontal arrows.

The superimposition is defined as follows: we build the $b[\,]$ table for each pattern, and then take the bitwise-*and* of all the tables (recall that 0 means match and 1 means mismatch). The resulting $b[\,]$ table matches at position i with the ith character of *any* of the patterns. We then build the automaton as before using this table.

The resulting automaton accepts a text position if it ends an occurrence of a much more relaxed pattern with classes of characters, namely

$$\{P_1^1, \ldots, P_1^r\}\{P_2^1, \ldots, P_2^r\} \cdots \{P_m^1, \ldots, P_m^r\},$$

for example, if the search is for "`patt`" and "`wait`", as shown in Figure 4, the string "`pait`" is accepted with *zero* errors.

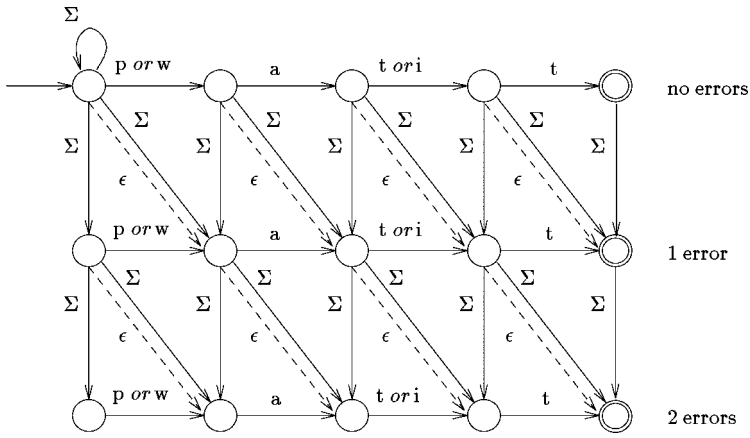

Fig. 4. An NFA to filter the search for "`patt`" and "`wait`".

For a moderate number of patterns, the filter is strict enough at the same cost of a single search. Each occurrence reported by the automaton has to be verified for all the involved patterns (we use the single-pattern automaton for this step). That is, we have to retraverse the last $m + k = O(m)$ characters to determine if there is actually an occurrence of some of the patterns.

If the number of patterns is too large, then the filter will be too relaxed and will trigger too many verifications. In that case, we partition the set of patterns into groups of r' patterns each, build the automaton of each group, and perform $\lceil r/r' \rceil$ independent searches. The cost of this search is $O(r/r'n)$, where r' is small enough to make the cost of verifications negligible. This r' always exists, since for $r' = 1$ we have a single pattern per automaton and no verification is needed.

When grouping, we use the heuristic of sorting the patterns and packing neighbors in the same group, trying to have the same first characters.

3.1. Hierarchical Verification

The simplest verification alternative (which we call "plain") is that, once a superimposed automaton reports a match, we try the individual patterns one by one in the candidate area. However, a smarter verification technique (which we call *hierarchical*) is possible.

First, assume that r is a power of two. Then, when the automaton reports a match, run two new automata over the candidate area: one which superimposes the first half of the patterns and another with the second half. Repeat the process recursively with each of the two automata that again finds a match. At the end, the automata will represent single patterns and if they find a match we know that their patterns have been really found (see Fig. 5). Of course the automata for the required subsets of patterns are all preprocessed. Since they correspond to the internal nodes of a binary tree of r leaves, they are $2r - 1 = O(r)$, so the space and preprocessing cost does not change. If r is not a power of two, then one of the halves may have one more pattern than the other.

The advantage of hierarchical verification is that it can remove a number of candidates from consideration in a single test. Moreover, it can even find that no pattern has really matched before actually checking any specific pattern (i.e., it may happen that none of the two halves match in a spurious match of the whole group). The worst-case overhead over plain verification is just a constant factor, that is, twice as many tests over the candidate area ($2r - 1$ instead of r). On average, as we

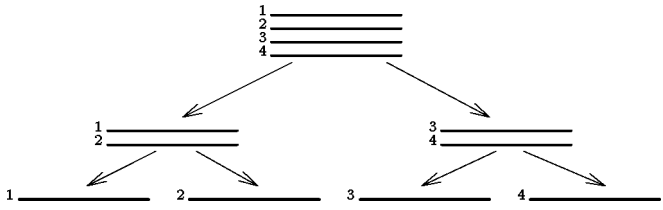

Fig. 5. The hierarchical verification method for four patterns. Each node of the tree represents a check (the root represents in fact the global filter). If a node passes the check, then its two children are tested. If a leaf passes the check, then its pattern has been found.

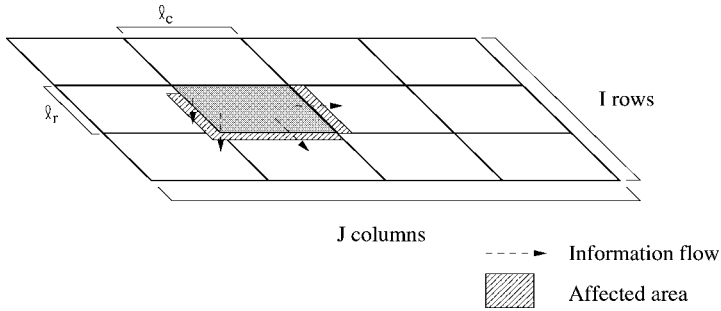

Fig. 6. A large NFA partitioned into a matrix of $I \times J$ computer words, satisfying $(\ell_r + 1)\ell_c \leq w$.

later show analytically and experimentally, hierarchical verification is by far superior to plain verification.

3.2. Automaton Partitioning

We have considered short patterns up to now, whose NFA fit into a computer word. If this is not the case (i.e., $(m - k)(k + 2) > w$), then we partition the problem. In this section and the next, we adapt the two partitioning techniques described in [6].

The simplest technique to cope with a large automaton is to use a number of machine words for the simulation. The idea is as follows: once the (large) automata have been superimposed, we partition the superimposed automaton into a matrix of subautomata, each one fitting in a computer word. Those subautomata behave slightly differently than the simple one, since they must propagate bits to their neighbors (see Fig. 6).

Once the automaton is partitioned, we run it over the text updating its subautomata. Each step takes time proportional to the number of cells to update, i.e., $O(k(m - k)/w)$. However, observe that it is not necessary to update all the subautomata, since those on the right may not have any active state. Following [31], we keep track of up to where we need to update the matrix of subautomata, working only on the "active" cells.

3.3. Pattern Partitioning

This technique is based on Lemma 1 of Section 2.1. We can reduce the size of the problem if we divide the pattern in j parts, provided we search all the subpatterns with $\lfloor k/j \rfloor$ errors. Each match of a subpattern must be verified to determine if it is in fact a complete match.

To perform the partition, we pick the smallest j such that the problem fits in a single computer word (i.e., $(\lceil m/j \rceil - \lfloor k/j \rfloor)(\lfloor k/j \rfloor + 2) \leq w$). The limit of this method is reached for $j = k + 1$, since in that case we search with zero errors. The algorithm for this case is qualitatively different and is described in Section 4.

We divide each pattern in j subpatterns as evenly as possible. Once we partition all the r patterns, we are left with $j \times r$ subpatterns to be searched with $\lfloor k/j \rfloor$ errors. We simply group them as if they were independent patterns to search with

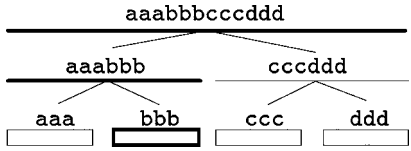

Fig. 7. The hierarchical piece verification method for a pattern split in four parts. The boxes (leaves) are the elements which are actually searched, and the root represents the whole pattern. At least one pattern at each level must match in any occurrence of the complete pattern. If the bold box is found, then all the bold lines may be verified.

the general method. The only difference is that, after determining that a subpattern has appeared, we have to verify its complete pattern.

Another kind of hierarchical verification, which we call "hierarchical piece verification," is applied in this case too. As shown in [23, 24], the single-pattern algorithm can verify hierarchically whether the complete pattern matches given that a piece matches (see Fig. 7). That is, instead of checking the complete pattern, we check the concatenation of two pieces containing the one that matched, and if it matches, then we check the concatenation of four pieces, and so on. This works because Lemma 1 applies at each level of the tree of Figure 7. The method is orthogonal to our hierarchical verification idea because hierarchical piece verification works bottom-up instead of top-down and operates on pieces of the pattern rather than on sets of patterns.

As we are using our hierarchical verification on the sets of pattern pieces to determine which piece matched given that a superimposition of them matched, we are coupling two different hierarchical verification techniques in this case: we first use our new mechanism to determine which piece matched from the superimposed group and then use hierarchical piece verification to determine the occurrence of the complete pattern the piece belongs to. Figure 8 illustrates the whole process.

4. PARTITIONING INTO EXACT SEARCHING

This technique (called "exact partitioning" for short) is based on a single-pattern filter which reduces the problem of approximate searching to a problem of

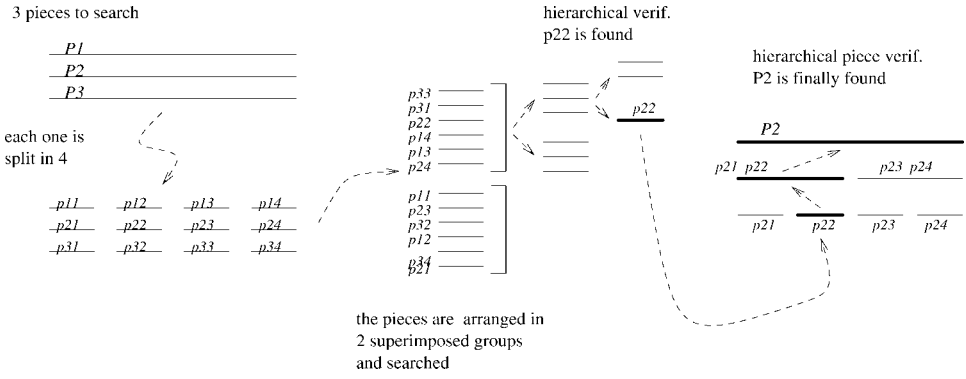

Fig. 8. The whole process of pattern partitioning with hierarchical verifications.

multipattern exact searching. The algorithm first appeared in [34], and was later improved in [6, 7, 24]. We first present the single-pattern version and then our extension to multiple patterns.

4.1. A Filter Based on Exact Searching

A particular case of Lemma 1 shows that if a pattern matches a text position with k errors, and we split the pattern in $k+1$ pieces, then at least one of the pieces must be present with no errors in each occurrence (this is a folklore property which has been used several times [12, 18, 34]). Searching with zero errors leads to a completely different technique.

Since there are efficient algorithms to search for a set of patterns exactly, we partition the pattern in $k+1$ pieces (of similar length), and apply a multipattern exact search for the pieces. Each occurrence of a piece is verified to check if it is surrounded by a complete match. If there are not too many verifications, then this algorithm is extremely fast.

From the many algorithms for multipattern search, an extension of Sunday's algorithm [27] gave us the best results. We build a trie with the subpatterns. From each text position we search the text characters into the trie, until a leaf is found (match) or there is no path to follow (mismatch). The jump to the next text position is precomputed as the minimum of the jumps allowed in each subpattern by the Sunday algorithm.

As in [24], we use the same technique for hierarchical piece verification of a single pattern presented in Section 3.3.

4.2. Searching Multiple Patterns

Observe that we can easily add more patterns to this scheme. Suppose we have to search for r patterns $P^1, \ldots, P^r$. We cut each one into $k+1$ pieces and search in parallel for *all* the $r(k+1)$ pieces. When a piece is found in the text, we use a classical algorithm to verify its pattern in the candidate area.

Note an important difference with the superimposed automata. In this multipattern search, we know which piece has matched. This is not the case in superimposed automata, where not only we do not know which piece matched, but it is even possible that no piece has really matched. The work to determine which is the matching piece (carried out by hierarchical verification in superimposed automata) is not necessary here. Moreover, we only detect real matches, so there are no more matches in the union of the patterns than the sum of the individual matches.

Therefore, there is no point in separating the search for the $r(k+1)$ pieces in groups. The only reason to superimpose less patterns is that the shifts of the Sunday algorithm are reduced as the number of patterns grow, but as we show in the experiments, this never justifies in practice splitting one search into two.

5. A COUNTING FILTER

We present now a filter based on counting letters in common between the pattern and a text window. This filter was first presented in [14] (a simple variant of [13]),

but we use a slightly different version here. Our variant uses a fixed-size instead of variable-size text window (a possibility already noted in [32]), which makes it better suited for parallelization. We first explain the single-pattern filter and then extend it to handle many patterns using bit-parallelism.

5.1. A Simple Counter

This filter is based in Lemma 2 of Section 2.1. It passes over the text examining an m-letters long window. It keeps track of how many characters of P are present in the current text window (accounting for multiplicities too). If, at a given text position j, $m-k$ or more characters of P are in the window $T_{j-m+1,\ldots,j}$, the window area is verified with a classical algorithm.

We implement the filtering algorithm as follows. We keep a counter *count* of pattern characters appearing in the text window. We also keep a table $A[\,]$ where, initially, the number of times that each character c appears in P is kept in $A[c]$. Throughout the algorithm, each entry $A[c]$ indicates how many occurrences of c can still be taken as belonging to P. For example, if 'h' appears once in P, then we count only one of the 'h's of the text window as belonging to P. When $A[c]$ is negative, it means that c must exit the text window $-A[c]$ times before we take it again as belonging to P. For example, if we run the pattern "`aloha`" over the text "`aaaaaaaa`", then it will hold $A[\text{"a"}] = -3$, and the value of the counter will be 2. This is independent on k.

To advance the window, we must include the new character T_{j+1} and exclude the last character, T_{j-m+1}. To include the new character, we subtract one from $A[T_{j+1}]$. If it is greater than zero before being decremented, it is because the new character T_{j+1} is in P, so we increment *count*. To exclude the old character T_{j-m+1}, we add one to $A[T_{j-m+1}]$. If it is greater than zero after being incremented, it is because T_{j-m+1} was considered to be in P, so we decrement *count*. Whenever *count* reaches $m-k$, we verify the preceding area.

As can be seen, the algorithm is not only linear (excluding verifications), but the number of operations per character is very small.

5.2. Keeping Many Counters in Parallel

To search r patterns in the same text, we use bit-parallelism to keep all the counters in a single machine word. We must do that for the $A[\,]$ table and for *count*.

The values of the entries of $A[\,]$ lie in the range $[-m, \ldots, m]$, so we need exactly $\ell + 1 = 1 + \lceil \log_2(m+1) \rceil$ bits to store them. This is also enough for *count*, since it is in the range $[0, \ldots, m]$. Hence, we can pack $\lfloor w/(1 + \lceil \log_2(m+1) \rceil) \rfloor \approx w/\log_2 m$ patterns of length m in a single search (recall that w is the number of bits in the computer word). If the patterns have different lengths, then we can either truncate them to the shortest length or use a window size of the longest length. If we have more patterns, then we must divide the set in subsets of maximal size and search each subset separately. We focus our attention on a single subset now.

The algorithm simulates the simple one as follows. We have a table $MA[\,]$ that packs all the $A[\,]$ tables. Each entry of $MA[\,]$ is divided in bit areas of length $\ell + 1$. In the area of the machine word corresponding to each pattern, we store its normal $A[\,]$ value, set to 1 the most significant bit of the area, and subtract 1 (i.e., we

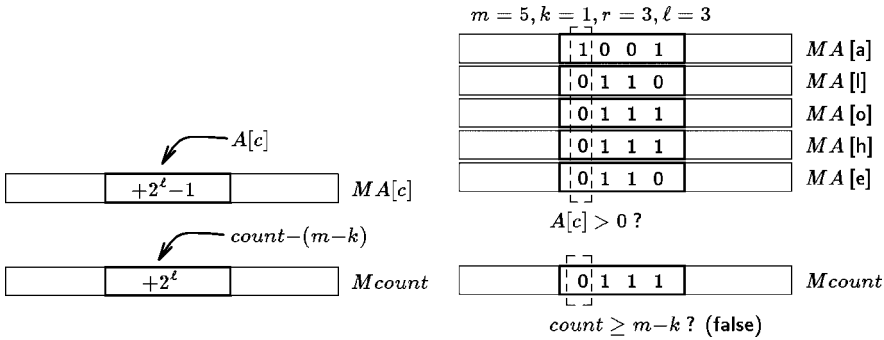

Fig. 9. The bit-parallel counters. The example corresponds to the pattern "aloha" searched with 1 error and the text window "hello". The A values are A['a'] = 2, A['l'] = A['e'] = −1, A['o'] = A['h'] = 0, and *count* = 3.

store $2^\ell - 1 + A[\,]$). When, in the algorithm, we have to add or subtract 1 to all $A[\,]$s, we can easily do it in parallel without causing overflow from an area to the next. Moreover, the corresponding $A[\,]$ value is not positive if and only if the most significant bit of the area is zero.

We have also a parallel counter *Mcount*, where the areas are aligned with $MA[\,]$. It is initialized by setting to 1 the most significant bit of each area and then subtracting $m - k$ at each one, i.e., we store $2^\ell - (m - k)$. Later, we can add or subtract 1 in parallel without causing overflow. Moreover, the window must be verified for a pattern whenever the most significant bit of its area reaches 1. The condition can be checked in parallel, but when some of the most significant bits reach 1, we need to sequentially check which one it was.

Finally, observe that the counters that we want to selectively increment or decrement correspond exactly to the $MA[\,]$ areas that have a 1 in their most significant bit (i.e., those whose $A[\,]$ value is positive). This allows an obvious bit mask-shift-add mechanism to perform this operation in parallel on all the counters. (See Fig. 9).

6. ANALYSIS

We are interested in the complexity of the presented algorithms, as well as in the restrictions that α and r must satisfy for each mechanism to be efficient in filtering most of the unrelevant part of the text.

To this effect, we define two concepts. First, we say that a multipattern search algorithm is *optimal* if it searches r patterns in the same time it takes to search one pattern. If we call $C_{n,r}$ the cost to search r patterns in a text of size n, then an algorithm is optimal if $C_{n,r} = C_{n,1}$. Second, we say that a multipattern search algorithm is *useful* if it searches r patterns in less than the time it takes to search them one by one with the corresponding sequential algorithm, i.e., $C_{n,r} < r\,C_{n,1}$. As we work with filters, we are interested in the average case analysis, since in the worst case none is useful.

We compare in Table 1 the complexities and limits of applicability of all the algorithms. Muth and Manber are included for completeness. The analysis leading to these results is presented later in this section.

TABLE 1 Complexity, Optimality, and Limit of Applicability for the Different Algorithms

Algorithm	Complexity	Optimality	Usefulness
Simple superimp.	$\frac{r}{\sigma(1-\alpha)^2}\ n$	$\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$	$\alpha < 1 - e/\sqrt{\sigma}$
Automaton part.	$\frac{\alpha m^2 r}{\sigma w(1-\alpha)} n$	$\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$	$\alpha < 1 - e/\sqrt{\sigma}$
Pattern part.	$\frac{mr}{\sigma\sqrt{w}(1-\alpha)}\ n$	$\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$	$\alpha < 1 - e/\sqrt{\sigma}$
Part. exact search	$\left(1 + \frac{rm}{\alpha\sigma^{1/\alpha}}\right)n$	$\alpha < \frac{1}{\log_\sigma(rm)+\Theta(\log_\sigma\log_\sigma(rm))}$	$\alpha < \frac{1}{\log_\sigma m+\Theta(\log_\sigma\log_\sigma m)}$
Counting	$\frac{r\log m}{w} n$	$\alpha < e^{-m/\sigma}$	$\alpha < e^{-m/\sigma}$
Muth and Manber	mn	$k = 1$	$k = 1$

We present in Figure 10 a schematical representation of the areas where each algorithm is the best in terms of complexity. We later show how the experiments match those figures.

- Exact partitioning is the fastest choice in most reasonable scenarios, for the error levels where it can be applied. First, it is faster than counting for $m/\log m < \alpha\sigma^{1/\alpha}/w$, which does not hold asymptotically but holds in practice for reasonable values of m. Second, it is faster than superimposing automata for $\min(\sqrt{w}, w/m) < \sigma^{1/\alpha-1}/(1/\alpha - 1)$, which is true in most practical cases.
- The only algorithm which can be faster than exact partitioning is that of Muth and Manber [17], namely for $r > \alpha\sigma^{1/\alpha}$. However, it is limited to $k = 1$.
- For increasing m, counting is asymptotically the fastest algorithm since its cost grows as $O(\log m)$ instead of $O(m)$ thanks to its optimal use of the bits of the computer word. However, its applicability is reduced as m grows, being useless at the point where it wins over exact partitioning.
- When the error level is too high for exact partitioning, superimposing automata is the only remaining alternative. Automaton partitioning is better for $m \leq \sqrt{w}$, while pattern partitioning is asymptotically better. Both the algorithms have the same limit of usefulness, and for higher error levels, no filter can improve over a sequential search.

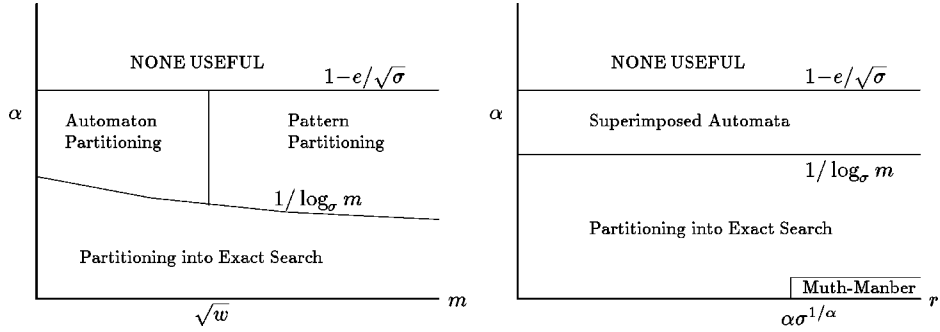

Fig. 10. The areas where each algorithm is better, in terms of α, m, and r. In the left plot (varying m), we have assumed a moderate r (i.e., less than 50).

6.1. Superimposed Automata

Suppose that we search r patterns. As explained before, we can partition the set in groups of r' patterns each, and search each group separately (with its r' automata superimposed). The size of the groups should be as large as possible, but small enough for the verifications to be not significant. We analyze which is the optimal value for r' and which is the complexity of the search.

In [6] we prove that the probability of a given text position matching a random pattern with error level α is $O(\gamma^m)$, where $\gamma = 1/(\sigma^{1-\alpha}\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)})$. It is also proved that $\gamma < 1$ whenever $\alpha < 1 - e/\sqrt{\sigma}$, and experimentally shown that this holds very precisely in practice if we replace e by 1.09. In fact, a very abrupt phenomenon occurs, since the matching probability is very low for $\alpha \leq 1 - 1.09/\sqrt{\sigma}$ and very high otherwise.

In this formula, $1/\sigma$ stands for the probability of a character crossing a horizontal edge of the automaton (i.e. the probability of two characters being equal). To extend this result, we note that we have r' characters on each edge now, so the above mentioned probability is $1 - (1 - 1/\sigma)^{r'}$, which is smaller than r'/σ. We use this upper bound as a pessimistic approximation (which stands for the case of all the r' characters being different, and is tight for $r' << \sigma$).

As the single-pattern algorithm is $O(n)$ time, the multipattern algorithm is optimal on average whenever the total cost of verifications is $O(1)$ per character. Since each verification costs $O(m)$ (because we use a linear-time algorithm on an area of length $m + k = O(m)$), we need that the total number of verifications performed is $O(1/m)$ per character, on average. If we used the plain verification scheme, this would mean that the probability that a superimposed automaton matches a text position should be $O(1/(mr))$, as we have to perform r verifications.

If hierarchical verification is not used, we have that, as r increases, matching becomes more probable (because it is easier to cross a horizontal edge of the automaton) *and* it costs more (because we have to check the r patterns one by one). This results in two different limits on the maximum allowable r, one for each of the two facts just stated. The limit due to the increased cost of each verification is more stringent than that of increased matching probability.

The resulting analysis without hierarchical verification is very complex and is omitted here because hierarchical verification yields considerably better results and a simpler analysis. As we show in Appendix A, the average cost to verify a match of the superimposed automaton is $O(m)$ when hierarchical verification is used, instead of the $O(rm)$ cost of plain verification. That is, the cost does *not* grow as the number of patterns increases.

Hence, the only limit that prevents us from superimposing all the r patterns is that the matching probability becomes higher. That is, if $\alpha > 1 - e\sqrt{r/\sigma}$, then the matching probability is too high and we will spend too much time verifying almost all text positions. On the other hand, we can superimpose as much as we like before that limit is reached. This tells that the best r (which we call r^*) is the maximum one not reaching the limit, i.e.,

$$r^* = \frac{\sigma(1-\alpha)^2}{e^2}. \tag{1}$$

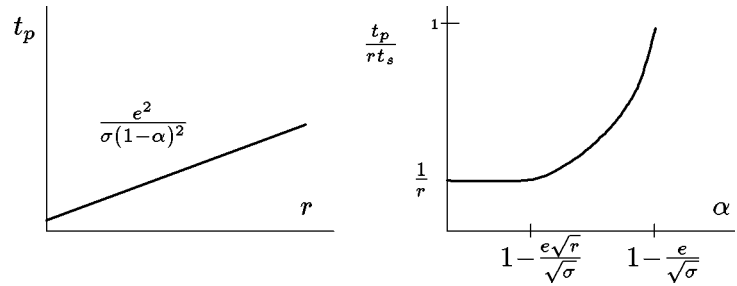

Fig. 11. Behavior of superimposed automata. On the left, the cost increases linearly with r, with slope depending on α. On the right, the cost of a parallel search (t_p) approaches r single searches (rt_s) when α grows.

Since we partition in sets small enough to make the verifications not significant, the cost is simply $O(r/r^*\ n) = O(rn/(\sigma(1-\alpha)^2))$.

This means that the algorithm is optimal for $r = O(\sigma)$ (taking the error level as a constant), or alternatively $\alpha \leq 1 - e\sqrt{r/\sigma}$. On the other hand, for $\alpha > 1 - e/\sqrt{\sigma}$, the cost is $O(rn)$, not better than the trivial solution (i.e., $r^* = 1$ and hence no superimposition occurs and the algorithm is not useful, see Fig. 11).

Automaton Partitioning. The analysis for this case is similar to the simple one, except that each step of the large automaton takes time proportional to the total number of subautomata, i.e., $O(k(m-k)/w)$. In fact, this is a worst case since on average not all cells are active, but we use the worst case because we superimpose all the patterns we can until the worst case of the search is almost reached. Therefore, the cost formula is

$$\frac{e^2}{(1-\alpha)^2\sigma}\frac{k(m-k)}{w}rn = O\left(\frac{\alpha m^2}{\sigma w(1-\alpha)}rn\right).$$

This is optimal for $r = O(\sigma w)$ (for constant α), or alternatively for $\alpha \leq 1 - e\sqrt{r/\sigma}$. It is useful for $\alpha \leq 1 - e/\sqrt{\sigma}$.

Pattern Partitioning. We now have jr patterns to search with $\lfloor k/j \rfloor$ errors. The error level is the same for subproblems (recall that the subpatterns are of length m/j).

To determine which piece matched from the superimposed group, we pay $O(m)$ independently of the number of pieces superimposed (thanks to the hierarchical verification). Hence the limit for our grouping is given by Eq. (1). In both the superimposed and in the single-pattern algorithm, we also pay to verify whether the match of the piece is part of a complete match. As we show in [23], this cost is negligible for $\alpha < 1 - e/\sqrt{\sigma}$, which is less strict than the limit given by Eq. (1).

As we have jr pieces to search, we need an analytical expression for j. Since j is just large enough so that the subpatterns fit in a computer word, $j = (m - k)d(w, \alpha)$, for

$$d(w, \alpha) = \frac{1 + \sqrt{1 + w\alpha/(1-\alpha)}}{w},$$

where $d(w, \alpha)$ can be shown to be $O(1/\sqrt{w})$ by maximizing it in terms of α (see [23]).

Therefore, the complexity is

$$\frac{jre^2}{\sigma(1-\alpha)^2}n = O\left(\frac{m}{\sigma\sqrt{w}(1-\alpha)}rn\right).$$

On the other hand, the search cost of the single-pattern algorithm is $O(jrn)$. With respect to the simple algorithm for short patterns, both costs have been multiplied by j, and therefore the limits for optimality and usefulness are the same.

If we compare the complexities of pattern versus automaton partitioning, then we have that pattern partitioning is better for $k > \sqrt{w}$. This means that for constant α and increasing m, pattern partitioning is asymptotically better.

6.2. Partitioning into Exact Searching

In [6], we analyze this algorithm as follows. Except for verifications, the search time can be made $O(n)$ in the worst case by using an Aho–Corasick machine [1], and $O(\alpha n)$ in the best case if we use a multipattern Boyer–Moore algorithm. This is because we search pieces of length $m/(k+1) \approx 1/\alpha$.

We are interested in analyzing the cost of verifications. Since we cut the pattern in $k+1$ pieces, they are of length $\lfloor m/(k+1) \rfloor$ or $\lceil m/(k+1) \rceil$. The probability of each piece matching is at most $1/\sigma^{\lfloor m/(k+1) \rfloor}$. Hence, the probability of any piece matching is at most $(k+1)/\sigma^{\lfloor m/(k+1) \rfloor}$.

We can easily extend that analysis to the case of multiple search, since we have now $r(k+1)$ pieces of the same length. Hence, the probability of verifying is $r(k+1)/\sigma^{\lfloor m/(k+1) \rfloor}$. We check the matches using a classical algorithm such as dynamic programming. Note that in this case we know which pattern to verify, since we know which piece matched. As we show in [23], the total verification cost if the pieces are of length ℓ is $O(\ell^2)$ (in our case, $\ell = m/(k+1)$). Hence, the search cost is

$$O\left(1 + \frac{rm}{\alpha\sigma^{1/\alpha}}\right)n,$$

where the "1" must be changed to "α", if we consider the best case.

We consider optimality and usefulness now. An optimal algorithm should pay $O(n)$ total search time, which holds for

$$\alpha < \frac{1}{\log_\sigma(rm) + \log_\sigma(1/\alpha)} = \frac{1}{\log_\sigma(rm) + \Theta(\log_\sigma \log_\sigma(rm))}.$$

The algorithm is always useful, since it searches at the same cost independently on the number of patterns, and the number of verifications triggered is exactly the same as if we searched each pattern separately. However, if $\alpha > 1/(\log_\sigma m + \Theta(\log_\sigma \log_\sigma m))$, then both algorithms (single and multipattern) work as much as dynamic programming and hence the multipattern search is not useful. The other case when the algorithm could not be useful is when the shifts of a Boyer–Moore search are shortened by having many patterns up to the point where it is better to perform separate searches. This never happens in practice.

6.3. Counting

If the number of verifications is negligible, each pass of the algorithms is $O(n)$. In the case of multiple patterns, only $O(w/\log m)$ patterns can be packed in a single search, so the cost to search r patterns is $O(rn\log(m)/w)$.

The difficult part of the analysis is the maximum error level α that the filtration scheme can tolerate while keeping the number of verifications low. We assume that we use dynamic programming to verify potential matches. We call λ the probability of verifying. If $\lambda \leq \log(m)/(wm^2)$, then the algorithm stays linear (i.e., optimal) on average. The algorithm is always useful since the number of verifications triggered with the multipattern search is the same as for the single-pattern version. However, if $\lambda \geq 1/m$ both algorithms work $O(rmn)$ as for dynamic programming and hence the filter is not useful.

We derive in Appendix B a pessimistic bound for the limit of optimality and usefulness, namely $\alpha < e^{-m/\sigma}$ (Eq. (B.3)). Hence, as m grows, we can tolerate smaller error levels. This limit holds for any condition of the type $\lambda = O(1/m^c)$, independently of the constant c. In our case, we need $c = 2$ for optimality and $c = 1$ for usefulness.

7. EXPERIMENTAL RESULTS

We experimentally study our algorithms and compare them against previous work. We tested with 10 megabytes of lower-case English text. The patterns were randomly selected from the same text. We use a Sun UltraSparc-1 running Solaris 2.5.1, with 64 megabytes of RAM, and $w = 32$. Each data point was obtained by averaging the Unix's user time over 10 trials. We present all the times in tenths of seconds per megabyte.

We do not present results on random text to avoid an excessively lengthly exposition. In general, all the filters improve as the alphabet size σ grows. Lowercase English text behaves approximately as random text with $\sigma = 15$, which is the inverse of the probability that two random letters are equal.

Figure 12 (left) compares the plain and hierarchical verification methods against a sequential application of the r searches, for the case of superimposed automata when the automaton fits in a computer word. We show the cases of increasing r and of increasing k. It is clear that hierarchical verification outperforms plain verification in all cases. Moreover, the analysis for hierarchical verification is confirmed since the maximum r up to where the cost of the parallel algorithm does not grow linearly is very close to $r^* = (1-\alpha)^2\sigma/1.09^2$. On the other hand, the algorithm with simple verification degrades sooner, since the verification cost grows with r.

The mentioned maximum r^* value is the point where the parallelism ratio is maximized. That is, if we have to search for $2r^*$ patterns, it is better to split them in two groups of size r^* and search each group sequentially. To stress this point, Figure 12 (right) shows the quotient between the parallel and the sequential algorithms, where the optimum is clear for superimposed automata. On the other hand, the parallelism ratio of exact partitioning keeps improving as r grows, as predicted by the analysis (there is an optimum for larger m, related to the Sunday shifts, but it still does not justify to split a search in two).

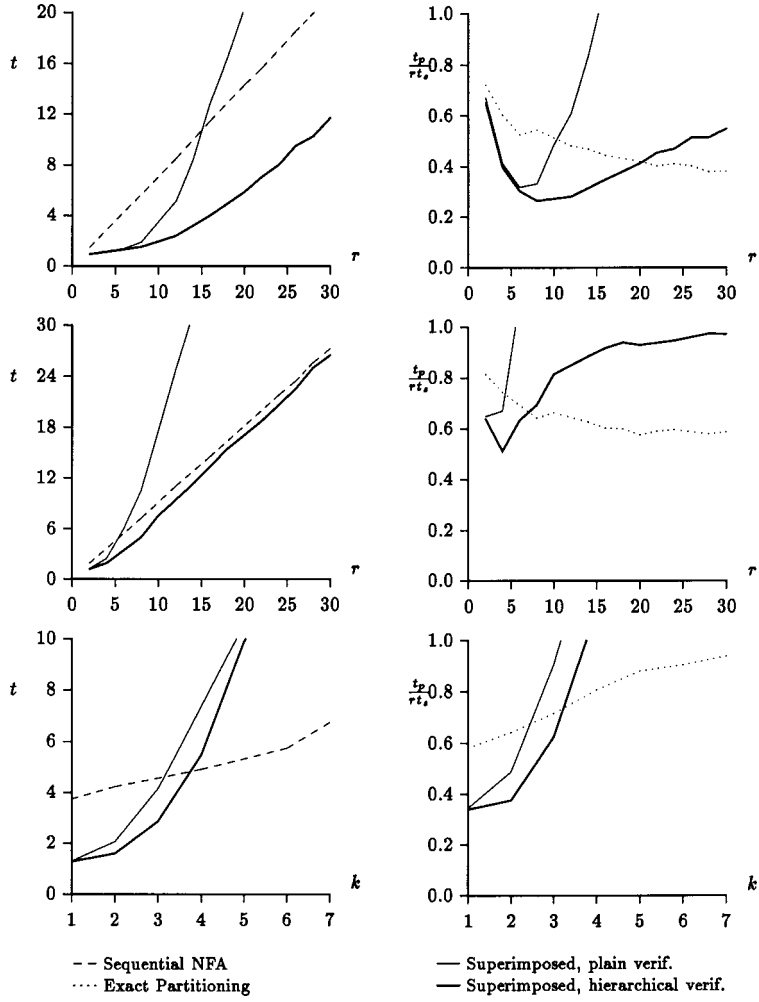

Fig. 12. Comparison of sequential and multipattern algorithms for $m = 9$. The rows correspond to $k = 1$, $k = 3$, and $r = 5$, respectively. The left plots show search time and the right plots show the ratio between the parallel (t_p) and the sequential time ($r \times t_s$).

When we compare our algorithms against the others, we consider only hierarchical verification and use this r^* value to obtain the optimal grouping for the superimposed automata algorithms. The exact partitioning, on the other hand, performs all the searches in a single pass. In counting, it is clear that the speedup is optimal and we pack as many patterns as we can in a single search.

Notice that the plots which depend on r show the point where r^* should be selected. Those which depend on k (for fixed r), on the other hand, just show how the parallelization works as the error level increases, which cannot be controlled by the algorithm.

We now compare our algorithms among them and against others. We begin with short patterns whose NFA fit in a computer word. Figure 13 shows the results for increasing r and for increasing k. For low and moderate error levels, exact

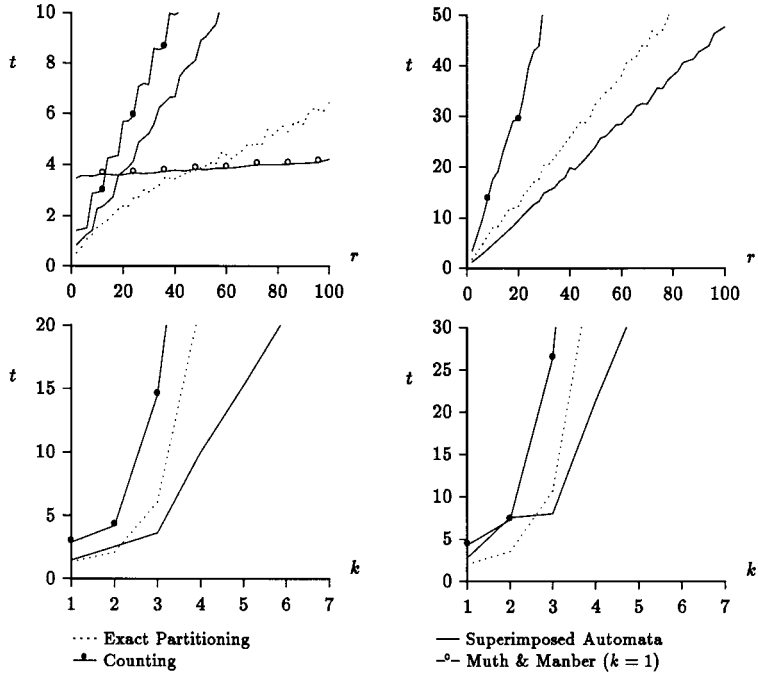

Fig. 13. Comparison among algorithms for $m = 9$. The top plots show increasing r for $k = 1$ and $k = 3$. The bottom plots show increasing k for $r = 8$ and $r = 16$.

partitioning is the fastest algorithm. In particular, it is faster than previous work [17] when the number of patterns is below 50 (for English text). When the error level increases, superimposed automata is the best choice. This agrees with the analysis.

We consider longer patterns now ($m = 30$). Figure 14 shows the results for increasing r and for increasing k. As before, exact partitioning is the best where it can be applied, and improves over previous work [17] for r up to 90–100. For these longer patterns, the superimposed automata technique also degrades, and only rarely is it able to improve over exact partitioning. In most cases, it only begins to be the best when it (and all the others) are no longer useful.

Figure 15 summarizes some of our experimental results, becoming a practical version of the theoretical Figure 10. The main differences are that exact partitioning is better in practice than what its complexity suggests, and that there is no clear winner between pattern and automaton partitioning.

8. CONCLUSIONS

We have presented a number of different filtering algorithms for multipattern approximate searching. These are the *only* algorithms that allow an arbitrary number of errors. On the other hand, the only previous work allows just one error and we have outperformed it when the number of patterns to search is below 50–100 on English text, depending on the pattern length.

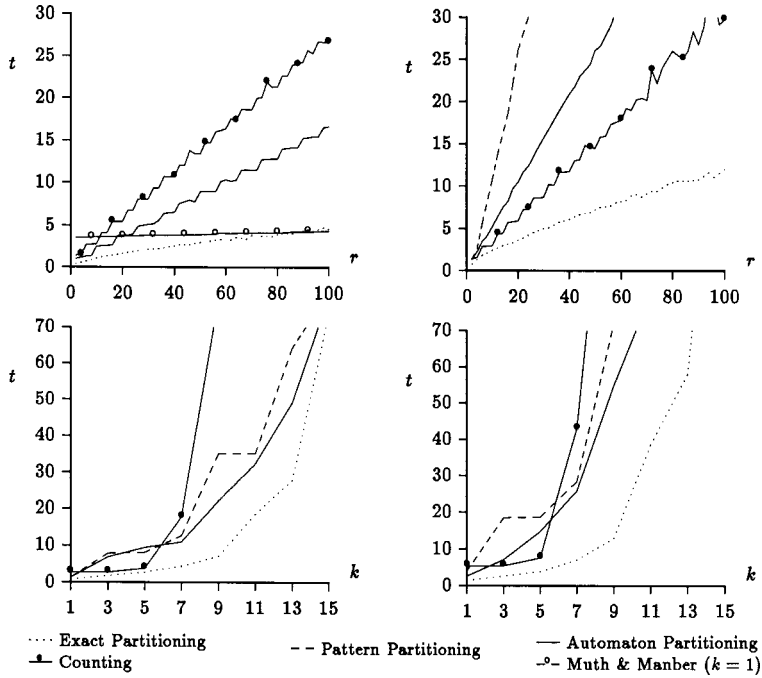

Fig. 14. Comparison among algorithms for $m = 30$. The top plots show, for increasing r, $k = 1$ and $k = 4$. The bottom plots show, for increasing k, $r = 8$ and $r = 16$. Pattern partitioning is not run for $k = 1$ because it would resort to exact partitioning.

We have explained, analyzed, and experimentally tested our algorithms. We have also presented a map of the best algorithms for each case. Many of the ideas we propose here can be used to adapt other single-pattern approximate searching algorithms to the case of multipattern searching. For instance, the idea of superimposing automata can be adapted to most bit-parallel algorithms, such as [19]. Another fruitful idea is that of exact partitioning, where a multipattern exact search is easily adapted to search the pieces of many patterns. There are many other filtering algorithms of the same type, e.g., [28]. On the other hand, other exact multipattern

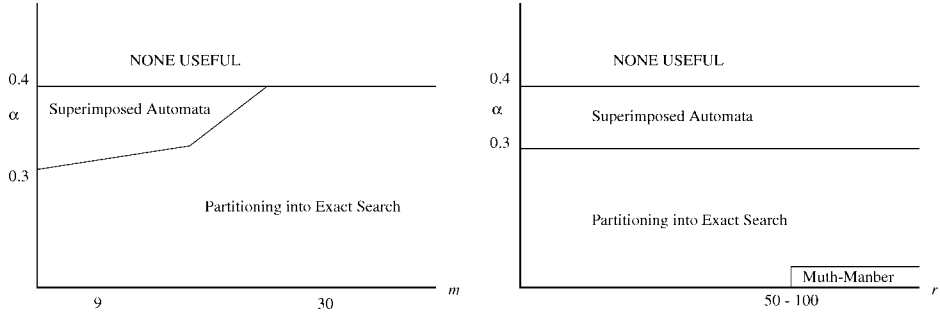

Fig. 15. The areas where each algorithm is better, in practice, on English text. In the right plot we assumed $m = 9$. Compare with Figure 10.

search algorithms may be better suited to other search parameters (e.g., working better on many patterns).

A number of practical optimizations to our algorithms are possible, for instance

- If the patterns have different lengths, we truncate them to the shortest one when superimposing automata. We can cleverly select the substrings to use, since having the same character at the same position in two patterns improves the filtering mechanism.
- We used simple heuristics to group subpatterns in superimposed automata. These can be improved to maximize common letters too. A more general technique could group patterns which are similar in terms of number of errors needed to convert one into the other (i.e., a clustering technique).
- We are free to partition each pattern in $k+1$ pieces as we like in exact partitioning. This is used in [24] to minimize the expected number of verifications when the letters of the alphabet do not have the same probability of occurrence (e.g., in English text). An $O(m^3)$ dynamic programming algorithm is presented there to select the best partition, and this could be applied to multipattern search.

APPENDIX A: ANALYSIS OF HIERARCHICAL VERIFICATION

We are interested in the expected cost to verify the presence of any of r patterns given that their superimposition has been found in a particular text region. The probability of such an automaton matching the text is the same as for a single pattern, except that the alphabet size is reduced from σ to σ/r. Moreover, when we find a match, we are not able to say which is the pattern that actually matched (in fact, it is possible that none matched). Hence, we must verify all the patterns in the area to determine which matched, if any.

We analyze the technique of "hierarchical verification": first divide the set of patterns in two halves and see whether each half matches. For each matching half, subdivide it and so on until the individual patterns that matched are detected.

Given an area to check for r patterns of length ℓ, the naive verification will take $O(rT(\ell))$ time, where $T(\ell)$ is the verification cost for a single pattern. We now analyze the effect of this hierarchical verification. In [23], it is proved that for a single pattern, the matching probability is $O(\gamma^\ell)$, where

$$\gamma = \left(\frac{e^2}{\sigma(1-\alpha)^2}\right)^{1-\alpha}.$$

When r patterns of length ℓ are superimposed, the value of γ changes because it is a function of σ. Instead, we use

$$\gamma_r = \left(\frac{e^2 r}{\sigma(1-\alpha)^2}\right)^{1-\alpha} = r^{1-\alpha}\gamma \tag{A.1}$$

(where we notice that the old γ corresponds now to γ_1). In the case of r superimposed patterns the matching probability is therefore $O(\gamma_r^\ell)$.

For each match, we have to verify an area of length $O(\ell)$ (at cost $T(\ell)$) for two sets of $r/2$ superimposed patterns each. Each set is found with probability $\gamma_{r/2}^{\ell}$ and so on. The probability that a group of size $r/2$ matches given that the larger set matched is (recall Fig. 7)

$$P(\text{parent node/child node}) = \frac{P(\text{parent} \wedge \text{child})}{P(\text{child})} \leq \frac{P(\text{parent})}{P(\text{child})} = \frac{\gamma_{r/2}^{\ell}}{\gamma_r^{\ell}} = \frac{1}{2^{\ell(1-\alpha)}}$$

where we have used Eq. (A.1) for the last step. In particular $\gamma_{r/2^i} = \gamma_r / 2^{i(1-\alpha)}$.

The total cost due to verifications is therefore

$$\gamma_r^{\ell}\left(2T(\ell) + 2\frac{\gamma_{r/2}^{\ell}}{\gamma_r^{\ell}}\left(2T(\ell) + 2\frac{\gamma_{r/4}^{\ell}}{\gamma_{r/2}^{\ell}}\cdots\right)\right) = 2T(\ell)\gamma_r^{\ell} + 4T(\ell)\gamma_{r/2}^{\ell} + 8T(\ell)\gamma_{r/4}^{\ell} + \cdots$$

or more formally

$$2T(\ell)\sum_{i=0}^{\log_2 r - 1} 2^i \gamma_{r/2^i}^{\ell} = 2T(\ell)\gamma_r^{\ell}\sum_{i=0}^{\log_2 r - 1} 2^{(1-\ell(1-\alpha))i}$$

and since the summation is $O(1)$, we have a total cost of $(\gamma_r^{\ell} T(\ell))$. Hence, we work $O(T(\ell))$ per verification instead of $O(rT(\ell))$ that we would work using a naive verification. This shows that we do not pay more per verification by superimposing more patterns (although the probability of verifying is increased).

Notice that we assumed that the summation is $O(1)$, which is not true for $\alpha \geq 1 - 1/\ell$. This happens only for very high error levels, which are of no interest in practice since the matching probability is already very high.

APPENDIX B: ANALYSIS OF THE COUNTING FILTER

We find an upper bound for the probability of triggering a verification, and use it to derive a safe limit for α to make verification costs negligible. We consider constant α and varying m (the results are therefore a limit on α). We then extend the results to the other cases.

The upper bound is obtained by using a pessimistic model which is simpler than reality. We assume that every time a letter in the text window matches the pattern, it is counted regardless of how many times it appeared in the window. Therefore, if we search "`aloha`" with 1 error in the text window "`aaaaa`", then the verification will be triggered because there are five letters in the pattern (where in fact our counter will not trigger a verification because it counts only 2 a's).

Consider a given letter in the text window. The probability of that letter being counted is that of appearing in the pattern. This is the same as being equal to some letter of the pattern. The probability of *not* being equal to a given letter is $(1 - 1/\sigma)$. The probability of not being in the pattern is therefore $p = (1 - 1/\sigma)^m$.

In our simplified model, each pattern letter is counted independently of the rest. Therefore, the number X of letters in the text window that did not match the pattern is the sum of m (window length) random variables that take the value 1 with probability p, and zero otherwise. This has a Binomial distribution $B(m, p)$.

Therefore, our question is when the probability $\Pr(X \leq k)$ is $O(1/m^2)$ (so that the algorithm is linear) or when it is $O(1/m)$ (so that it is useful). In the proof we use $O(1/m^2)$, since as we will see shortly, the result is the same for any polynomial in $1/m$.

We first analyze the case where the mean of the distribution is beyond k, i.e., $mp > k$. This is the same as the condition $\alpha < p$. As $\Pr(X = j)$ increases with j for $j < mp$, we have $\Pr(X \leq k) \leq k \Pr(X = k)$.

Therefore, it suffices to prove that $\Pr(X = k) = O(1/m^3)$ for optimality or that $\Pr(X = k) = O(1/m^2)$ for usefulness. By using the Stirling approximation to the factorial we have

$$\Pr(X = k) = \binom{m}{k} p^k (1-p)^{m-k} = \frac{m^m p^k (1-p)^{m-k}}{k^k (m-k)^{m-k}} O(\sqrt{m})$$

which can be rewritten as

$$\left(\frac{p^\alpha (1-p)^{1-\alpha}}{\alpha^\alpha (1-\alpha)^{1-\alpha}} \right)^m O(\sqrt{m}).$$

It is clear that the above formula is $o(1/\text{poly}(m))$ whenever the base of the exponential is < 1. This is

$$p^\alpha (1-p)^{1-\alpha} < \alpha^\alpha (1-\alpha)^{1-\alpha}. \tag{B.1}$$

To determine the cases where the above condition is valid, we define the function

$$f(x) = x^\alpha (1-x)^{1-\alpha}$$

which reaches its maximum at $x = \alpha$. This shows that Eq. (B.1) holds everywhere, and therefore the probability of matching is $O(1/m^2)$ in the area under consideration, i.e., whenever $\alpha < p$.

On the other hand, if the median of the distribution is less than k, then just the term of the summation corresponding to the median $r = mp$ is (using Stirling again)

$$\binom{m}{mp} p^{mp} (1-p)^{m(1-p)} = \left(\frac{p^p (1-p)^{1-p}}{p^p (1-p)^{1-p}} \right)^m \Omega(m^{-1/2}) = \Omega(m^{-1/2}) \tag{B.2}$$

which is not $O(1/m)$.

Therefore, we arrive at the conclusion that the filter is optimal and useful whenever

$$\alpha < p = \left(1 - \frac{1}{\sigma}\right)^m = e^{-m/\sigma}(1 + O(1/\sigma)) \tag{B.3}$$

and is not useful otherwise.

We have considered the case of constant $\alpha = k/m$. Obviously, the filter is linear for $k = o(m)$ and is not useful for $k = m - o(m)$. The unexplored area is $k = mp - o(m)$. It is easy to see that the filter is not useful in this case, by considering $\Pr(X = mp - \epsilon)$ with $\epsilon = o(m)$, and using Stirling as in Eq. (B.2). The resulting condition is $1 - \epsilon^2/(m^2 p(1-p)) = O(m^{-1/2})$, which does not hold for any $\epsilon = o(m)$.

ACKNOWLEDGMENTS

We thank Robert Muth and Udi Manber for their implementation of [17]. We also thank the anonymous referees for their detailed comments that improved this work.

REFERENCES

[1] A. Aho and M. Corasick, Efficient string matching: an aid to bibliographic search, CACM 18 (1975), 333–340.

[2] R. Baeza-Yates, Efficient text searching, Ph.D. thesis, Department of Computer Science, University of Waterloo, May 1989 (Also as Research Report CS-89-17).

[3] R. Baeza-Yates, Text retrieval: Theory and practice, Proc 12th IFIP World Computer Congress, Vol. I, Elsevier Science, Amsterdam, 1992, pp. 465–476.

[4] R. Baeza-Yates and G. Gonnet, A new approach to text searching, CACM 35 (1992), 74–82.

[5] R. Baeza-Yates and G. Navarro, Multiple approximate string matching, Proc WADS'97, LNCS 1272, 1997, pp. 174–184.

[6] R. Baeza-Yates and G. Navarro, Faster approximate string matching, Algorithmica 23 (1999), 127–158.

[7] R. Baeza-Yates and C. Perleberg, Fast and practical approximate pattern matching, Proc CPM'92, LNCS 644, 1992, pp. 185–192.

[8] R.S. Boyer and J.S. Moore, A fast string searching algorithm, CACM 20 (1977), 762–772.

[9] W. Chang and J. Lampe, Theoretical and empirical comparisons of approximate string matching algorithms, Proc CPM'92, LNCS 644, 1992, pp. 172–181.

[10] W. Chang and E. Lawler, Sublinear approximate string matching and biological applications, Algorithmica 12 (1994), 327–344.

[11] Z. Galil and K. Park, An improved algorithm for approximate string matching, SIAM J Comput 19 (1990), 989–999.

[12] D. Greene, M. Parnas, and F. Yao, Multi-index hashing for information retrieval, Proc FOCS'94, 1994, pp. 722–731.

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

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

[15] D.E. Knuth, J.H. Morris, Jr, and V.R. Pratt, Fast pattern matching in strings, SIAM J Comput 6 (1977), 323–350.

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

[17] R. Muth and U. Manber, Approximate multiple string search, Proc CPM'96, LNCS 1075, 1996, pp. 75–86.

[18] E. Myers, A sublinear algorithm for approximate keyword searching, Algorithmica 12 (1994), 345–374.

[19] G. Myers, A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming, Proc CPM'98, LNCS 1448, 1998, pp. 1–13.

[20] G. Navarro, Multiple approximate string matching by counting, Proc WSP'97, Carleton University Press, 1997, pp. 125–139.

[21] G. Navarro, Approximate text searching, Ph.D. thesis, Department of Computer Science, University of Chile, December 1988 (`ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/thesis98.ps.gz`. Also as Tech. Report TR/DCC-98-14).

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

[23] G. Navarro and R. Baeza-Yates, Improving an algorithm for approximate pattern matching, Algorithmica 30 (2001), 473–502.

[24] G. Navarro and R. Baeza-Yates, Very fast and simple approximate string matching, Inform Process Lett 72 (1999), 65–70.

[25] G. Navarro and M. Raffinot, A bit-parallel approach to suffix automata: Fast extended string matching, Proc CPM'98, LNCS 1448, 1998, pp. 14–33.

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

[27] D. Sunday, A very fast substring search algorithm, CACM 33 (1990), 132–142.

[28] E. Sutinen and J. Tarhio, On using q-gram locations in approximate string matching, Proc ESA'95, LNCS 979, 1995, 327–340.

[29] J. Tarhio and E. Ukkonen, Approximate Boyer-Moore string matching, SIAM J Comput 22 (1993), 243–260.

[30] E. Ukkonen, Algorithms for approximate string matching, Inform Control 64 (1985), 100–118.

[31] E. Ukkonen, Finding approximate patterns in strings, J Alg 6 (1985), 132–137.

[32] E. Ukkonen, Approximate string matching with q-grams and maximal matches, Theor Comput Sci 1 (1992), 191–211.

[33] A. Wright, Approximate string matching using within-word parallelism, Software Practice and Experience 24 (1994), 337–362.

[34] S. Wu and U. Manber, Fast text searching allowing errors, CACM 35 (1992), 83–91.

[35] S. Wu, U. Manber, and E. Myers, A sub-quadratic algorithm for approximate limited expression matching, Algorithmica 15 (1996), 50–67.