A Faster Algorithm for Approximate String Matching [⋆]

Ricardo Baeza-Yates Gonzalo Navarro

Department of Computer Science
University of Chile
Blanco Encalada 2120 - Santiago - Chile
{rbaeza,gnavarro}@dcc.uchile.cl

Abstract. We present a new algorithm for on-line approximate string matching. The algorithm is based on the simulation of a non-deterministic finite automaton built from the pattern and using the text as input. This simulation uses bit operations on a RAM machine with word length $O(\log n)$, being n the maximum size of the text. The running time achieved is $O(n)$ for small patterns (i.e. of length $m = O(\sqrt{\log n})$), independently of the maximum number of errors allowed, k. This algorithm is then used to design two general algorithms. One of them partitions the problem into subproblems, while the other partitions the automaton into sub-automata. These algorithms are combined to obtain a hybrid algorithm which on average is $O(n)$ for moderate k/m ratios, $O(\sqrt{mk/\log n}\; n)$ for medium ratios, and $O((m-k)kn/\log n)$ for large ratios. We show experimentally that this hybrid algorithm is faster than previous ones for moderate size of patterns and error ratios, which is the case in text searching.

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 of length n, a pattern of length m, and a maximal number of errors allowed, k, we want to find all text positions where the pattern matches the text up to k errors. Errors can be substituting, deleting or inserting a character. The solutions to this problem differ if the algorithm has to be on-line (that is, the text is not known in advance) or off-line (the text can be preprocessed). In this paper we are interested in the first case, where the classical dynamic programming solution is $O(mn)$ running time [13, 14].

In the last years several algorithms have been presented that achieve $O(kn)$ comparisons in the worst-case [20, 9, 10, 11] or in the average case [21, 9], by taking advantage of the properties of the dynamic programming matrix. In the same trend is [6], with average time complexity $O(kn/\sqrt{\sigma})$ (σ is the alphabet size). The algorithms which are $O(kn)$ in the worst case tend to involve too much overhead, and are not competitive in practice.

[⋆] This work has been supported in part by FONDECYT grant 1950622.

Other approaches attempt to filter the text, reducing the area in which dynamic programming needs to be used [18, 19, 17, 16, 7, 8]. These algorithms achieve sublinear expected time in many cases ($O(kn \log_\sigma m/m)$ is a typical figure) for moderate k/m ratios, but the filtration is not effective for larger ratios. A simple and fast filtering technique is shown in [5], which yields an $O(n)$ algorithm for moderate k/m ratios.

Yet other approaches use bit-parallelism [2, 25] in a RAM machine of word length $O(\log n)$ to reduce the number of operations. [24] achieves $O(kmn/\log n)$ time, which is competitive for patterns of length $O(\log n)$. [22] packs the cells differently to achieve $O(mn \log \sigma / \log n)$ time complexity. [26] uses a Four Russians approach and packs the table in machine words, achieving $O(kn/\log n)$ time on average.

We present a new algorithm which combines the ideas of taking advantage of the properties of the matrix, filtering the text and using bit-parallelism, being faster than previous work for moderate size patterns and error ratios, as we are interested in text searching.

We model the search with a non-deterministic finite automaton (NFA) built from the pattern and using the text as input. This automaton is simulated by an algorithm based on bit operations on a RAM machine of word length $O(\log n)$. The algorithm achieves running time $O(n)$, independently of k, for small patterns (i.e. $mk = O(\log n)$). This restricted algorithm is used to design two general algorithms.

A first one partitions the problem into subproblems, and has average time cost $O(mn/\log n)$ for small $\alpha = k/m$ (i.e. $\alpha < 1/\log n$), otherwise it is $O(\sqrt{mk/\log n}\; n)$ (i.e. $O(\sqrt{k}\; n)$ for $m = O(\log n)$, else $O(kn)$). It involves also a cost to verify potential matches, which is shown to be not significant for $\alpha < \alpha_1 \approx 1 - m^{1/\sqrt{\log n}}/\sqrt{\sigma}$. This algorithm is a generalization of an earlier heuristic [23, 5], that reduces the problem to subproblems of exact matching and is shown to be $O(n)$ for $\alpha < \alpha_0 = 1/(3 \log_\sigma m)$.

The second one partitions the automaton in sub-automata, being $O(k^2 n/(\sqrt{\sigma} \log n))$ on average. For $\alpha > 1 - 1/\sqrt{\sigma}$ its worst case, $O((m-k)kn/\log n)$, dominates. This algorithm is shown to be better than dynamic programming for $k \leq \log(n)/(1-\alpha)$.

We analyze the optimal way to combine the algorithms. We show experimentally that this hybrid algorithm is faster than previous ones, for moderate m and α. Table 1 shows the combined complexity.

As a corollary of our analysis, we give tight bounds for the probability of finding an occurrence of a pattern of length m with k errors starting at a fixed position in random text. We also show that the heuristic of [21] works $O(kn)$ time on average, with a constant tighter than that of [6].

Condition	Complexity	Method used
$mk = O(\log n)$	$O(n)$	the simple algorithm
$\alpha < \alpha_0$	$O(n)$	reducing to exact match
$\alpha_0 < \alpha < \alpha_1$	$O\left(\sqrt{mk/\log n}\ n\right)$	problem partitioning
$\alpha > \alpha_1 \wedge k < \log n/(1-\alpha)$	$O((m-k)kn/\log n)$	automaton partitioning
$\alpha > \alpha_1 \wedge k > \log n/(1-\alpha)$	$O(mn)$	plain dynamic programming

Table 1. Complexity of our hybrid algorithm.

2 Preliminaries

The problem of approximate string matching can be stated as follows: given a (long) *Text* of length n, and a (short) pattern *pat* of length m, both being sequences of characters from an alphabet Σ, find all segments (called "occurrences" or "matches") of *Text* whose *edit distance* to *pat* is at most k, the number of allowed errors. We use $\sigma = |\Sigma|$.

The *edit distance* between two strings a and b is the minimum number of *edit operations* needed to transform a into b. The allowed edit operations are deleting, inserting and replacing a character. Therefore, the problem is non-trivial for $k < m$.

Stated that way, we should report a number of segments that contain others. Because of that, it is common to report only minimal or maximal segments. It is also common to report not the matching segments but only their start or end point. In this work we focus on returning end points of minimal segments (i.e. those not containing others).

We use a C-like notation for the operations (e.g. $\&, |, ==, !=, \wedge, >>$). We use *text* to denote the current character of *Text* and, unlike C, $str[j]$ to denote the j-th character of *str*. Except when otherwise indicated, the log function denotes logarithm in base 2.

Consider the NFA for searching `text` with at most $k = 2$ errors shown in Figure 1. 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), vertical arrows represent inserting a character in the pattern, solid diagonal arrows represent replacing a character, and dashed diagonal arrows represent deleting a character of the pattern (they are empty transitions, since we delete the character from the pattern without advancing in the text). Finally, the empty transition at the initial state allows to consider any character as a potential starting point of a match, and the automaton accepts a character (as the end of a match) whenever a rightmost state is active. If we do not care about the number of errors, we can consider final states those of the last full diagonal. Because of the empty transitions, this makes acceptance equivalent to the lower-right state being active.

This NFA has $(m+1)(k+1)$ states. We assign number (i,j) to the state at row i and column j, where $i \in 0..k, j \in 0..m$. Initially, the active states at row i are at the columns from 0 to i, to represent the deletion of the first i characters of the pattern.

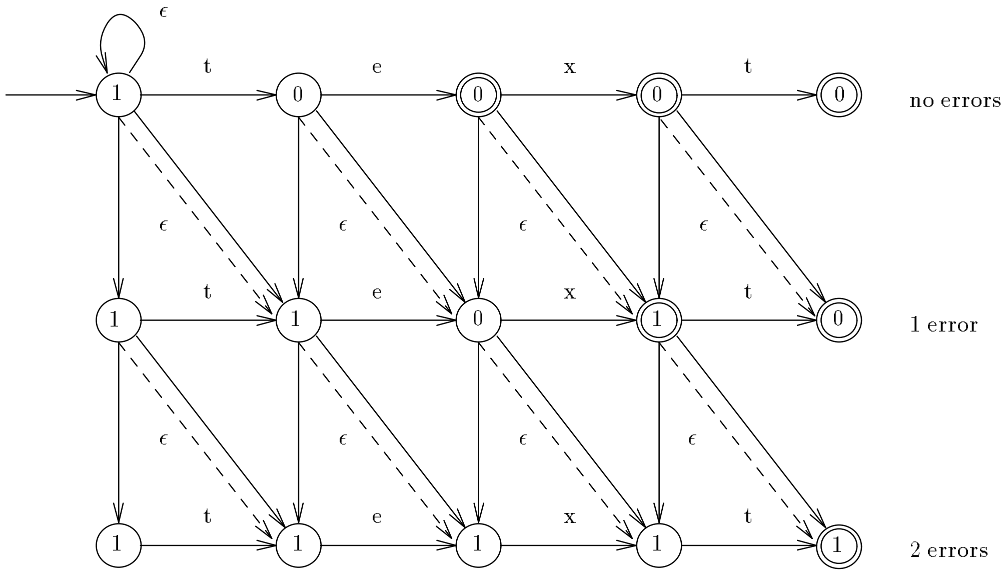

Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character. Active states are $(2,3)$, $(2,4)$ and $(1,3)$, besides those always active of the lower-left triangle.

Consider the boolean matrix A corresponding to this automaton. $A_{i,j}$ is 1 if state (i,j) is active and 0 otherwise. The matrix changes as each character of the text is read. The new values $A'_{i,j}$ can be computed from the current ones by the following rule

$$A'_{i,j} \; = \; (A_{i,j-1} \;\; \& \;\; (\; text == pat[j] \;)) \mid A_{i-1,j} \mid A_{i-1,j-1} \mid A'_{i-1,j-1} \qquad (1)$$

which is used for $i \in 0..k, j \in 1..m$. If $i = 0$ only the first term is used. Note that the empty transition is represented by immediately propagating a 1 at any position to all the elements following it in its diagonal, all in a single iteration (thus computing the ϵ-closure). The empty transition at the initial state is represented by the fact that column $j = 0$ is never updated.

The main comparison-based algorithms for approximate string matching consist fundamentally in simulating this matrix by columns, while many bit-parallelism approaches simulate it by rows or by diagonals (in this last case, of the dynamic programming matrix). In all cases, the dependencies introduced by the diagonal empty transitions prevent the parallel computation of the new values. We present in the next section an approach that avoids this dependence, by simulating the automaton using diagonals, such that each diagonal captures the ϵ-closure [3]. This idea leads to a new and fast algorithm.

3 A New Algorithm

Suppose we use just the full diagonals of the automaton (i.e. those of length $k+1$). This presents no problem, since those (shorter) diagonals below the full ones have always value 1, while those past the full ones do not influence state (m,k). The last statement may not be obvious, since the vertical transitions allow to carry 1's from the last diagonals to state (m,k). But each 1 present at the last diagonals must have crossed the last full diagonal, where the empty transition (deletion) immediately would have copied it to the state (m,k). That is, any 1 that goes again to state (m,k) corresponds to a segment containing one that has already been reported.

Another observation is that, if state (i,j) is active at any time, then states $(i+d, j+d)$ are also active for all $d>0$ (due to the empty deletion transition). Thus, if we number diagonals regarding the column at which they begin, the state of each diagonal i can be represented by a number D_i, the smallest row value active in that diagonal (i.e. the smallest error). Then, the state of this simulation consists of $m-k+1$ values in the range $0..k+1$. Note that D_0 is always 0, hence, there is no need to store it.

The new values for D_i ($i \in 1..m-k$) after we read a new text character are derived from Eq. (1)

$$D'_i \ = \ \min(\ D_i + 1, \ \ D_{i+1} + 1, \ \ g(D_{i-1}, text)\ ) \qquad (2)$$

where $g(D_i, c)$ is defined as

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

The first term of the D' update formula represents a substitution, which follows the same diagonal. The second term represents the insertion of a character (coming from the next diagonal above). Finally, the last term represents matching a character: we select the minimum active state (hence the min of the g formula) of the previous diagonal that matches the text and thus can move to the current one.

The deletion transitions are represented precisely by the fact that once a state in a diagonal is active, we consider active all subsequent states on that diagonal (so we keep the minimum). The empty initial transition corresponds to $D_0 = 0$. Finally, we find a match in the text whenever $D_{m-k} \leq k$.

This simulation has the advantage that can be computed in parallel for all i. We use this property to design a fast algorithm that exploits bit-parallelism for small patterns, and then extend it to handle the general case by partitioning either the problem or the automaton.

3.1 A Linear Algorithm for Small Patterns

We show in this section how to simulate the automaton by diagonals using bit-parallelism, assuming that our problem fits in a single word (i.e. $(m-k)(k+2) \leq w$, where w is the length of the computer word). We first select a suitable representation for our problem and then describe the algorithm.

Since we have $m-k$ non-trivial diagonals, and each one takes values in the range $0..k+1$, we need at least $(m-k)\lceil \log_2(k+2) \rceil$ bits.

However, the g function cannot be computed in parallel for all i with this optimal representation. It can be precomputed and stored, but it takes $O(\sigma(k+1)^{m-k})$ space if it has to be accessed in parallel for all i. At this exponential space cost, the automaton approach of [21] is preferable.

Therefore, we use unary encoding of the D_i values, since in this case g can be computed in parallel. Thus, we need $(m-k)(k+2)$ bits to encode the problem, where each of the $m-k$ blocks of $k+2$ bits stores the value of a D_i.

Each value of D_i is stored as 1's aligned to the right of its $(k+2)$-wide block (thus there is a separator at the highest bit having always 0). The blocks are stored contiguously, the last one ($i = m-k$) aligned to the right of the computer word. Thus, our bit representation of state $D_1, ..., D_{m-k}$ is

$$D \;=\; 0\; 0^{k+1-D_1}\; 1^{D_1} \;\; 0\; 0^{k+1-D_2}\; 1^{D_2} \;\; ... \;\; 0\; 0^{k+1-D_{m-k}}\; 1^{D_{m-k}}$$

where we use exponentiation to mean digit repetition. Observe that, in this way, what our word contains is a rearrangement of the 0's and 1's of (the relevant part of) the automaton. The rearrangement exchanges 0's and 1's and reads the diagonals left-to-right and bottom-to-top (see Figure 2).

With this representation, taking minimum is equivalent to *and*ing, adding 1 is equivalent to shifting one position to the left and *or*ing with a 1 at the rightmost position, and accessing the next or previous diagonal means shifting a block ($k+2$ positions) to the left or right, respectively.

The computation of the g function is carried out by defining, for each character c, an m bits long mask $t[c]$, representing match (0) or mismatch (1) against the pattern, and then computing a mask $T[c]$ having at each block the $(k+1)$ bits long segment of $t[c]$ that is relevant to that block (see Figure 2). That is,

$$t[c] \;=\; (c\;!=\;pat[m])\;(c\;!=\;pat[m-1])\;\ldots\;(c\;!=\;pat[1])$$

where each condition stands for a bit and they are aligned to the right. So we precompute

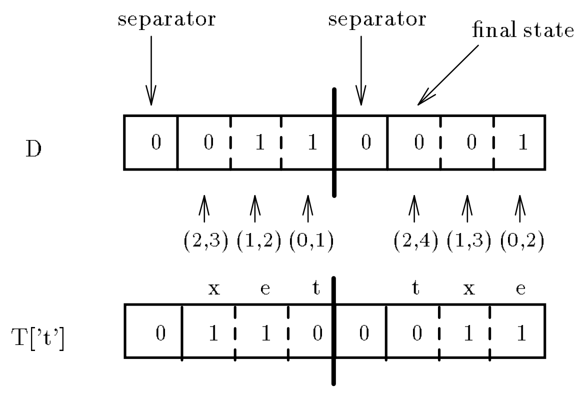

Fig. 2. Encoding of the example NFA.

$$T[c] \;=\; 0\; s_{k+1}(t[c],0)\;\; 0\; s_{k+1}(t[c],1)\;\; \ldots\;\; 0\; s_{k+1}(t[c],m-k-1)$$

for each c, where $s_j(x,i)$ shifts x to the right in i bits and takes the last j bits of the result (the bits that "fall" are discarded). Note that $T[c]$ fits in a word if the problem does.

We have now all the elements to implement the algorithm. We represent the current state by a computer word D. The value of all D_i's is initially $k+1$, so the initial value of D is $D_{in} = (0\ 1^{k+1})^{m-k}$. The formula to update D is derived from Eq. (2)

$$\begin{aligned} D' = \quad & (D << 1) \mid (0^{k+1}1)^{m-k} \\ & \&\ (D << (k+3)) \mid (0^{k+1}1)^{m-k-1}0\ 1^{k+1} \\ & \&\ (((x + (0^{k+1}1)^{m-k}) \ \wedge\ x) >> 1) \\ & \&\ D_{in} \qquad (3) \\ \text{where} \quad & x = (D >> (k+2)) \mid T[text] \end{aligned}$$

The update formula is a sequence of *and*'s, corresponding to the *min* of Eq. (2). The first line corresponds to $D_i + 1$, the second line to $D_{i+1} + 1$, the third line is the g function applied to the previous diagonal, and the fourth line ensures the invariant of having zeros in the separators (needed to limit the propagation of "+").

Note that we are assuming that the shifts get zeros from both borders of the word (i.e. unsigned semantics). If this is not so, additional masking is necessary.

We detect that the state (m,k) is active by checking whether $D \,\&\, (1 << k)$ is 0. When we find a match, we clear the last diagonal. This ensures that the reported occurrences always end with a match.

A further improvement is not to run the automaton through all the text, but to scan the text looking for any of the $k+1$ initial characters of the pattern, and only then starting the automaton. When the automaton returns to the initial configuration, we restart the scanning. This is much faster and correct, since one of the $k+1$ initial characters of the pattern must be present in any result with at most k errors. We precompute a boolean table $S[c]$, that says for each character c whether it is one of the first $k+1$ of the pattern. Observe that this table alone solves the problem for the case $k = m-1$ (since each positive answer of S is an occurrence).

Figure 3 presents the complete algorithm. To avoid complications, we do not refine the preprocessing, which can be done more efficiently than what the code suggests.

```
search ($Text, n, pat, m, k$)
 {         /* preprocessing */
   for each $c \in \Sigma$
     { $t[c] \;=\; (c\;! = \;pat[m])\;(c\;! = \;pat[m-1])\;\;...\;\;(c\;! = \;pat[1])$
       $T[c] \;=\; 0\;s_{k+1}(t[c],0)\;\;0\;s_{k+1}(t[c],1)\;\;...\;\;0\;s_{k+1}(t[c],m-k-1)$
       $S[c] \;=\; (c \in pat[1..k+1])$
     }
   $D_{in} = (0\;1^{k+1})^{m-k}$
   $M1 = (0^{k+1}1)^{m-k}$
   $M2 = (0^{k+1}1)^{m-k-1}\;0\;1^{k+1}$
   $M3 = 0^{(m-k-1)(k+2)}\;0\;1^{k+1}$
   $G = 1 << k$
           /* searching */
   $D = D_{in}$
   $i = 0$
   while $(++\,i <= n)$
      if $(S[Text[i]])$       /* is one of the first $k+1$ characters? */
         do { $x = (D >> (k+2))\;|\;T[Text[i]]$
              $D = ((D << 1)\;|\;M1)\;\&\;((D << (k+3))\;|\;M2)$
                   $\&\;(((x+M1)\;\wedge\;x) >> 1)\;\&\;D_{in}$
              if $(D\;\&\;G == 0)$
                 { report a match ending at $i$
                   $D = D\;|\;M3$        /* clear last diagonal */
                 }
            }
         while $(D\;! = \;D_{in}\;\&\&\;\;++i <= n)$
 }
```

Fig. 3. Algorithm to search for a short pattern.

3.2 Partitioning Large Automata

If the automaton does not fit in a single word, we can partition it using a number of machine words for the simulation.

First suppose that k is small and m is large. Then, the automaton can be "horizontally" split in as many sub-automata as necessary, each one holding a number of diagonals. We call "columns" those sets of diagonals packed in a single machine word. Those sub-automata behave differently than the simple one, since they must communicate their first and last diagonals with their neighbors.

Thus, if $(m-k)(k+2) > w$, we partition the automaton horizontally in J columns, where $J = \lceil (m-k)(k+2)/w \rceil$. Note that for this idea to work we need that at least one diagonal fits in a single machine word, i.e. $k+2 \leq w$.

Suppose now that k is large (close to m, so that the width $m-k$ is small). In this case, the automaton is not wide but tall, and a vertical partitioning becomes necessary. The sub-automata behave differently than the previous ones, since we must propagate the ϵ-transitions down to all subsequent sub-automata.

In this case, if $(m-k)(k+2) > w$, we partition the automaton vertically in I rows, where I has the same formula as J. The difference is that, in this case, we need that at least one row fits in a machine word, i.e. $2(m-k) \leq w$ (the 2 is because we need an overflow bit for each diagonal of each cell).

When none of the two applicability conditions hold, we need a generalized partition in rows and columns. We use I rows and J columns, so that each cell contains ℓ_r bits of each one of ℓ_c diagonals. It must hold $(\ell_r+1)\ell_c \leq w$.

There are many options to pick (I, J) for a given problem. We show later that they are roughly equivalent in cost (except for integer round-offs that are noticeable in practice). We use I as small as possible and then determine J. That is, $I = \lceil (k+1)/(w-1) \rceil$, $\ell_r = \lceil (k+1)/I \rceil$, $\ell_c = \lfloor w/(\ell_r+1) \rfloor$ and $J = \lceil (m-k)/\ell_c \rceil$. The cells of the last column and the last row may be smaller, since they have the residues.

The simulation of the automaton is now more complex, but follows the same principle of the update formula (3). We have a matrix of automata $D_{i,j}$ ($i \in 0..I-1, j \in 0..J-1$), and a matrix of masks $T_{i,j}$ coming from splitting the original T. The new update formula is

$$\begin{aligned} D'_{i,j} = \quad & (D_{i,j} << 1) \mid ((D_{i-1,j} >> (\ell_r - 1)) \;\&\; (0^{\ell_r}1)^{\ell_c}) \\ & \&\; ((D_{i,j} << (\ell_r+2)) \mid \\ & \quad ((D_{i-1,j} << 2) \;\&\; (0^{\ell_r}1)^{\ell_c}) \mid \\ & \quad (D_{i-1,j+1} >> ((\ell_r+1)(\ell_c-1) + \ell_r - 1)) \mid \\ & \quad (D_{i,j+1} >> ((\ell_r+1)(\ell_c-1) - 1))) \\ & \&\; (((x + (0^{\ell_r}1)^{\ell_c}) \;\wedge\; x) >> 1) \\ & \&\; D_{in} \end{aligned}$$

where $\quad x = ((D_{i,j} >> (\ell_r+1)) \mid (D_{i,j-1} << (\ell_r+1)(\ell_c-1)) \mid T_{i,j}[text])$
$\quad\quad \& \; ((D'_{i-1,j} >> (\ell_r-1)) \mid (1^{\ell_r}0)^{\ell_c})$

and it is assumed $D_{-1,j} = D_{i,J} = 1^{(\ell_r+1)\ell_c}$ and $D_{i,-1} = 0^{(\ell_r+1)\ell_c}$.

We find a match whenever $D_{I-1,J-1}$ has a 0 in its last position, i.e. at $(k - \ell_r(I-1)) + (\ell_r+1)(\ell_c J - (m-k))$, counting from the right. In that case, we must clear the last diagonal, i.e. that of $D_{i,J-1}$ for all i.

The fact that we select the minimal I and that we solve the case $k = m-1$ with a simpler algorithm (the S table) causes this general schema to fall into three simpler cases: (a) the automaton is horizontal, (b) the automaton is horizontal and only one diagonal fits in each word, (c) the automaton spreads horizontally and vertically but only one diagonal fits in each word. Those cases can be solved with a simpler (two or three times faster) update formula.

3.3 Partitioning Large Problems

The following lemma, which is a generalization of the idea presented in [24], suggests a way to partition a large problem into smaller ones.

Lemma: If $segm = Text[a..b]$ matches pat with k errors, and $pat = p_1...p_j$ (a concatenation of sub-patterns), then $segm$ includes a segment that matches at least one of the p_i's, with $\lfloor k/j \rfloor$ errors.

Proof: Suppose the opposite. Then, in order to transform pat into $segm$, we need to transform each p_i into an s_i, such that $segm = s_1...s_j$. But since no p_i is present in $segm$ with less than $\lfloor k/j \rfloor$ errors, each p_i needs at least $1 + \lfloor k/j \rfloor$ edit operations to be transformed into any segment of $segm$ (s_i in particular). Thus the whole transformation needs at least $j(1 + \lfloor k/j \rfloor) > j(k/j) = k$ operations. A contradiction. $\Box$

Thus, we can reduce the number of errors if we divide the pattern, provided we search all the sub-patterns. Each match of a sub-pattern must be checked to determine if it is in fact a complete match. Suppose we find at position i in $Text$ the end of a match for the sub-pattern ending at position j in pat. Then, the potential match must be searched in the area $Text[i-j+1-k, i-j+1+m+k]$, an $(m+2k)$-wide area. This checking must be done with a classical algorithm needing little preprocessing, e.g. the variation of [21] to standard dynamic programming. A related idea was used in [12] to search indexed text.

To perform the partition, we pick an integer j, and split the pattern in j sub-patterns of length m/j (more precisely, if $m = qj + r$, with $0 \le r < j$, r sub-patterns of length $\lceil m/j \rceil$ and $j-r$ of length $\lfloor m/j \rfloor$). Because of the lemma, it is enough to check if any of the sub-patterns is present in the text with at most $\lfloor k/j \rfloor$ errors. Thus, we select j as small as possible such that the subproblems fit in a computer word, that is

$$j \;=\; \min \left\{ r \;/\; \left(\left\lceil \frac{m}{r} \right\rceil - \left\lfloor \frac{k}{r} \right\rfloor \right) \left(\left\lfloor \frac{k}{r} \right\rfloor + 2 \right) \le w \;\wedge\; \left\lfloor \frac{m}{r} \right\rfloor > \left\lfloor \frac{k}{r} \right\rfloor \right\} \qquad (4)$$

where the second condition avoids searching a sub-pattern of length m' with $k' = m'$ errors (those of length $\lceil m/j \rceil$ are guaranteed to be longer than $\lfloor k/j \rfloor$ if $m > k$). Such a j always exists, since $j = k+1$ implies searching with 0 errors.

In case of 0 errors, we can use an Aho-Corasick machine [1] to guarantee $O(n)$ total search time, or use a faster heuristic such as extending the Boyer-Moore-Horspool-Sunday algorithm [15] to multipattern search.

Figure 4 shows the general algorithm, which is written that way for clarity. In a practical implementation, it is better to run all sub-searches in synchronization, picking at any moment the candidate whose initial checking position is the leftmost in the set, checking its area and advancing that sub-search to its next candidate position. This allows to avoid re-verifying the same text because of different candidates that imply overlapping areas. This is done by remembering the last checked position and keeping the state of the checking algorithm.

```
ProblemPartition ($Text, n, pat, m, k$)
 { $j = \min \{ r \;/\; (\lceil m/r \rceil - \lfloor k/r \rfloor)(\lfloor k/r \rfloor + 2) \le w \wedge \lfloor m/r \rfloor > \lfloor k/r \rfloor \}$
   if ($j == 1$) search ($Text, n, pat, m, k$)
   else { $a = 0$
          for $r \in 0..j-1$
            { $len = (r < m\%j)\ ?\ \lceil m/j \rceil : \lfloor m/j \rfloor$
              $b = a + len - 1$
              for each pos. $i$ reported by search($Text, n, pat[a..b], len, \lfloor k/j \rfloor$)
                 check the area $Text[i - b + 1 - k, i - b + 1 + m + k]$
              $a = b + 1$
 }      }      }
```

Fig. 4. Algorithm for problem partitioning.

4 Analysis

In this section we analyze the different aspects of our algorithms, and obtain a general heuristic to combine them. Recall that $\alpha = k/m$.

4.1 The Simple Algorithm

The preprocessing phase of this algorithm can be optimized to take $O(\sigma + m \min(m, \sigma))$ time, and it requires $O(\sigma)$ space. The search phase needs $O(n)$ time.

However, this algorithm is limited to the case in which $(m-k)(k+2) \leq w$. In the RAM model it is assumed $\log n \leq w$, so a machine-independent bound is $(m-k)(k+2) \leq \log n$.

Since $(m-k)(k+2)$ takes its maximum value when $k = m/2 - 1$, we can assure that this algorithm can be applied whenever $m \leq 2(\sqrt{w} - 1)$, independently of k. That is, we have a linear algorithm for $m = O(\sqrt{\log n})$, for example, $m \leq 9$ for $w = 32$, or $m \leq 14$ for $w = 64$.

4.2 Partitioning the Automaton

If we divide the automaton in IJ sub-automata (I rows and J columns), we must update I cells at each column. However, we use a heuristic similar to [21] (i.e. not processing the m columns but only up to the last active one), so we do not work on all the diagonals, but only the active portion.

To compute the expected last active diagonal, we first compute the last active column in the heuristic of [21] (these are real columns of the matrix, not our packed diagonals), i.e. the largest r satisfying $C_r \leq k$ (being C_r the smallest-row active state of column r). We follow the proof of [6], but we find a tighter bound. If we call L the last active column, we have

$$E(L) \;\leq\; K + \sum_{r>K} r \; P[C_r \leq k]$$

for any K. We show in the next section that if $k/r \leq 1 - e/\sqrt{\sigma}$, then $P[C_r \leq k] = O(\gamma^r)$ with $\gamma < 1$, thus we take $K = k/(1-e/\sqrt{\sigma})$ to obtain

$$E(L) \;\leq\; \frac{k}{1-e/\sqrt{\sigma}} \;+ \sum_{r>k/(1-e/\sqrt{\sigma})} r \; O(\gamma^r) \;=\; \frac{k}{1-e/\sqrt{\sigma}} \;+\; O(1)$$

which shows that, on average, the last active column is $O(k)$.

This refines the proof of [6] of that the heuristic of [21] is $O(kn)$. Since this measures active columns and we work on active diagonals, we subtract k, to obtain that on average we work on $ke/(\sqrt{\sigma} - e) + 1$ diagonals (one beyond the last active one). Our experiments suggest that, although the formal proof needs e, it is more accurate to replace e by 1. We do so in the following.

Since we pack $(m-k)/J$ diagonals in a single machine word, we work on average on $(k/(\sqrt{\sigma}-1)+1)\; J/(m-k)$ words. But since there are only J columns, our total complexity is

$$I \; J \; \min\left(1, \frac{1}{m-k} \; \left(\frac{k}{\sqrt{\sigma}-1} + 1\right)\right) \; n$$

which shows that any choice for I and J is the same for a fixed IJ. Since $IJ \approx (m-k)(k+1)/(w-1)$, the final cost expression is independent (up to round-offs) of I and J:

$$\min\left(m-k\ ,\ \frac{k}{\sqrt{\sigma}-1}+1\right)\ \frac{k+1}{w-1}\ n \qquad (5)$$

which is $O(k^2n/(\sqrt{\sigma}\log n))$ time, and is better if $\alpha \geq 1-1/\sqrt{\sigma}$, namely $O((m-k)kn/\log n)$ time. This last complexity is also the worst case of this algorithm.

The preprocessing time and space complexity of this algorithm is $O(mk\sigma/w)$.

4.3 Partitioning the Problem

There are two main components in the cost of problem partitioning. One is the j simple searches that are carried out, and the other is the checks that must be done. The first part is $O(jn)$, while the second one costs $O(jm^2f(m/j,k/j)n)$, where $f(m,k)$ is the probability that an automaton of size $(k+1)\times(m+1)$ accepts a given text position (observe that α is the same for the subproblems). The complexity comes from considering that we perform j independent searches, and each verification costs $O(m^2)$. Clearly, in the worst case each text position must be verified, and since we avoid re-checking the text, we have $O((j+m)n) = O(mn)$ worst-case complexity, but we show that this does not happen on average.

To determine j, we consider the following equation, derived from Eq. (4)

$$\left(\frac{m}{j}-\frac{k}{j}\right)\left(\frac{k}{j}+2\right) = w$$

whose solution is

$$j = \frac{m-k+\sqrt{(m-k)^2+wk(m-k)}}{w} = O\left(m/w+\sqrt{mk/w}\right) \qquad (6)$$

That is, if $\alpha > 1/w$, we have $j = O(\sqrt{mk/w}) = O(k)$, otherwise we have $j = O(m/w)$. If we further assume the general condition $m = O(w)$, then $j = O(\sqrt{k})$. A machine-independent complexity is obtained by assuming $w = O(\log n)$.

Observe that we discarded the second condition of Eq. (4), namely $\lfloor m/j \rfloor > \lfloor k/j \rfloor$. This is because if $\lfloor m/j \rfloor = \lfloor k/j \rfloor$, then $j > m-k$, which implies $\alpha > 1-1/(w-1)$. As we show next, this value of α is outside our area of interest (i.e. it is larger than $1-1/\sqrt{\sigma}$), except for $\sigma > (w-1)^2 = \Omega(\log^2 n)$, that is, extremely large alphabets.

The preprocessing time and storage requirements for the general algorithm are j times those of the simple one.

In the Appendix we show that for $\alpha \leq 1-e/\left(\sqrt{\sigma}\gamma^{1/(2(1-\alpha))}\right)$, $f(m,k) = O(\gamma^m)$, for $\gamma < 1$. Thus, for α small enough, $f(m/j,k/j) = O(\gamma^{m/j})$, which does not affect the complexity provided $\gamma^{m/j} = O(1/m^2)$. This happens for $\gamma \leq 1/m^{2j/m}$. Hence, $1/\gamma^{1/(2(1-\alpha))} \geq m^{2j/(2m(1-\alpha))} = m^{j/(m-k)}$. Therefore, $f(m/j,k/j) = O(1/m^2)$ if $\alpha \leq \alpha_1$, where

$$\alpha_1 = 1 - \frac{e}{\sqrt{\sigma}}\, m^{\frac{j}{m-k}} \;=\; 1 - \frac{e}{\sqrt{\sigma}}\, m^{\frac{1+\sqrt{1+w\alpha_1/(1-\alpha_1)}}{w}}$$

up to where the cost of verifications is not significant. We repeat that experimental results suggest that it is better to replace e by 1 in this formula. A good approximation for α_1 (except for very large m) is $\alpha_1 \approx 0.92 - m^{1/\sqrt{w}}/\sqrt{\sigma}$.

4.4 The Heuristic of Reducing to Exact Search

When we partition the problem, we want to have as few subproblems as possible. However, the special case $j = k+1$ is different, since in that case we search with 0 errors, and a faster algorithm is possible (i.e. exact multipattern search). Thus, it is possible that in some cases we may prefer to increase j beyond its minimum value, setting it to $k+1$. This is the heuristic used in [23, 5].

To analyze this algorithm, we assume the use of an Aho-Corasick machine [1], which guarantees $O(n)$ search time. However, in practice we find that an extension of Boyer-Moore-Horspool-Sunday [15] to multipattern searching is faster. This extension consists of building a trie with the sub-patterns, and at each position searching the text into the trie. If we do not find a sub-pattern, we shift the window using the Sunday heuristic, taking the minimum shift among all patterns (this minimum shift is, of course, precomputed).

Any match of any of the sub-patterns is a candidate for a complete match. Since there are $k+1$ sub-patterns of length $m/(k+1)$, the number of verifications that must be carried out is $(k+1)/\sigma^{m/(k+1)}$. For the verification phase to be not significant, it is necessary that the total number of verifications be $O(1/m^2)$ (the inverse of the cost to verify a candidate). This happens for $\alpha < \alpha_0 \approx 1/(3\log_\sigma m)$.

For values of m and k that allow discarding the effect of verifications, this algorithm is linear.

4.5 A General Strategy

In this section we depict the hybrid algorithm, that takes the best among the described strategies.

We first discard trivial cases: $k = 0$ is solved with exact matching, e.g. [15], at $O(n \log m/m)$ cost on average; $k = m-1$ is solved with the heuristic of the first $k+1$ characters (i.e. each hit of the S table is an occurrence) at $O(n)$ cost; $k \geq m$ means that any text position is an occurrence.

If $\alpha < \alpha_0$, reduction to exact search (i.e. problem partitioning with $j = k+1$) is $O(n)$, and even faster in practice than the simple algorithm for $\alpha < 1/3$.

For larger α, if the problem fits in a single word (i.e. $(m-k)(k+2) \leq w$), the simple algorithm can be used, which is $O(n)$. This algorithm is faster if we use also the heuristic of the first $k+1$ characters (the S table). If the problem does not fit in a single word, we may use problem or automaton partitioning.

For $\alpha > \alpha_1$, problem partitioning is not advisable, because of the large number of verifications (it is better to use the verification code [21] alone). To see when automaton partitioning is better than plain dynamic programming, consider that, for large α, the first one works $O(IJ) = O((m-k)(k+1)/(w-1))$ per text position, while the second one works $O(m)$. That means that for $k < (w-1)/(1-\alpha)-1$ (which is moderately large), it is better to partition the automaton, while for larger k it is better to just use dynamic programming. In both cases we can use also the S table to reduce the average cost.

What is left is the case of medium α. The general strategy is to partition the problem in a number of subproblems, which are in turn solved by partitioning each automaton. However, we find that except for $\alpha < \sqrt{\sigma}/2(w+\sqrt{\sigma})$, the best strategy is to apply only problem partitioning. Where convenient, the combination of both partitions leads to an $O(kn/\log n)$ expected time algorithm.

However, the area in which problem partitioning combined with automaton partitioning is the best is outperformed by the heuristic of reducing to exact matching, which is $O(n)$ for $\alpha < \alpha_0$. This α_0 is larger than $\sqrt{\sigma}/2(w+\sqrt{\sigma})$ for $m < \sigma^{2/3(1+w/\sqrt{\sigma})}$, which is a huge limit to reach. So for intermediate α, problem partitioning is the best choice.

Thus, the combined algorithm is $O(n)$ for $\alpha < \alpha_0$ or $mk = O(\log n)$, from there to $\alpha < \alpha_1$ it is $O(\sqrt{mk/\log n}\ n)$, and $O((m-k)k/\log n)$ for larger α. Recall that, for the α in which it is applied, $O(\sqrt{mk/\log n}\ n) = O(kn)$, and it is $O(\sqrt{k}\ n)$ if $m = O(\log n)$. Figure 5 (left side) shows the combined complexity.

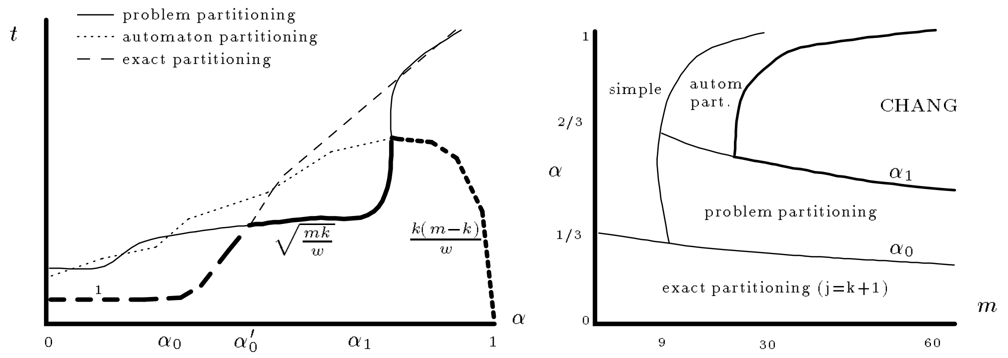

Fig. 5. The complexities of our algorithm (left side) and areas where each algorithm is the best (right side) for $\sigma = 32$ and $w = 32$.

Observe that this schema gets worse as m grows, since the area $\alpha > \alpha_1$ dominates, and automaton partition gets quickly worse than plain dynamic programming. Because of this, our algorithm is well suited for moderate patterns (or comparatively small α), which is the case in text searching.

5 Experimental Results

In this section we experimentally compare our algorithm against the fastest previous algorithms we are aware of. We use the experiments to confirm our analytical results about the optimal way to combine the heuristics, and to show that the resulting hybrid algorithm is faster than previous work on moderate patterns and error ratios.

Since we compare only the fastest algorithms, we leave aside [14, 20, 9, 10, 17], which were not competitive in our first experimental study. The algorithms included in this comparison are

Ukkonen [21] is the standard dynamic programming algorithm, modified to be $O(kn)$ on average. This is the algorithm we use to verify potential matches. The code is ours.

Chang [6] is the algorithm kn.clp, which computes only the places where the value of the dynamic programming matrix does not change along each column. The code is from the author.

Suntinen-Tarhio [16] is, to our knowledge, the best filtration algorithm. The method is limited to $\alpha < 1/2$, and the implementation to $k \leq w/2 - 3$. The code is from the authors. We use $s = 2$ (number of samples to match) and maximal q (length of the q-grams), as suggested in [16].

Baeza-Yates/Perleberg [5] is essentially the heuristic $j = k + 1$, that our hybrid algorithm uses in the appropriate case. The code is ours.

Wu-Manber [24] uses bit-parallelism to simulate the automaton by rows. Our implementation (taken from Wright's tests [22]) is limited to $m \leq 31$, and it would be slower if generalized.

Wright [22] uses bit-parallelism to pack the diagonals (perpendicular to ours) of the dynamic programming matrix (not the automaton). The code is from the author.

Wu-Manber-Myers [26] applies a Four Russians technique to the dynamic programming matrix, storing the states of the automaton in computer words. The code is from the authors, and is used with $r = 5$ as suggested in [26] (r is related with the size of the Four Russians tables).

Agrep [23] is a widely distributed approximate search software, that implements a hybrid algorithm. It is limited, although not inherently, to $m \leq 29$ and $k \leq 8$, so it is only included in the test for small patterns. Because of its match reporting policy and its options, it is hard to compare fairly with the other algorithms, but we include it as a reference point.

Ours are our algorithms.

We tested random patterns against 1 Mb of random text on a Sun SparcClassic, of approximately Specmark 26, running SunOS 4.1.3, with 16 Mb of RAM. We use $w = 32$ and $\sigma = 32$ (typical case in text searching). Each data point was obtained by averaging the Unix's user time over 10 trials.

Figure 5 (right side) summarizes the results, showing in which case should each algorithm be applied. As it can be seen, our heuristic is the best for moderate values of m and α, otherwise Chang is the best choice. We made three types of tests:

Small patterns: we tested the case $m = 9, k = 1..8$, using our simple algorithm (no partitions needed). The result is shown in Figure 6. Note that our algorithm is by far more efficient than any other when the problem fits in a single word. However, the heuristic of $j = k+1$ (i.e. [5]) is slightly faster for $\alpha \leq 1/3$.

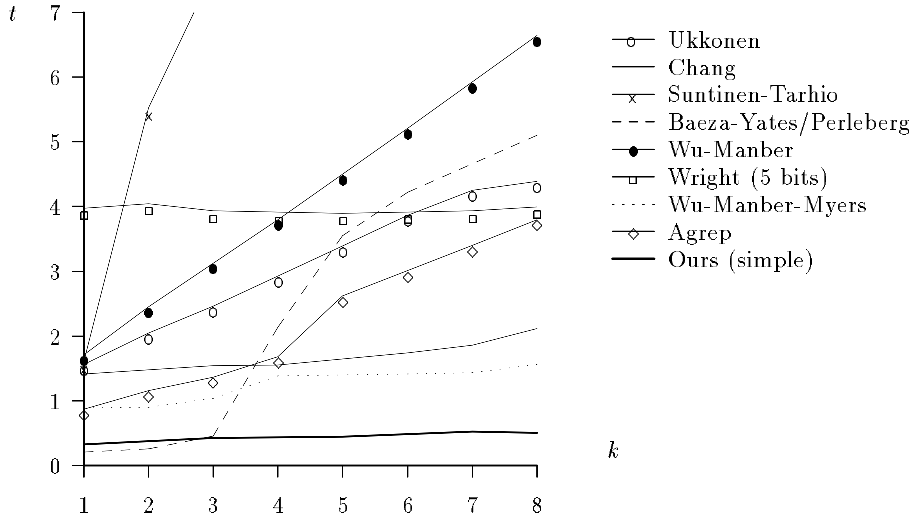

Fig. 6. Times compared for small $m = 9$ (in seconds).

Moderate patterns: we tested the cases $m = 31, k = 1..30$ and $m = 61, k = 1..60$, where the simple algorithm cannot be used. The result for $m = 31$ is shown in Figure 7 (left plot). We tested Agrep for $m = 29$ and $k = 1..8$, and found it worse than many others for $m = 31$.
For $m = 31$, we have $\alpha_0 = 0.34$, $\alpha_1 = 0.59$ ($k = 10$ and $k = 18$, respectively). Note that, as expected, [5] is the best choice for $\alpha < \alpha_0$, problem partitioning is the best for $\alpha_0 < \alpha < \alpha_1$, and automaton partitioning is the best for $\alpha > \alpha_1$. Since m is moderate, this combined heuristic is the fastest, except for Chang and Wu-Manber-Myers, which are better for α from α_1 to short before 1.
Note that for $\alpha < \alpha_0$, it is not clear which is the best among problem and automaton partitioning (since the best choice is to combine them). Note also

that problem partitioning behaves as $O(\sqrt{k}\ n)$ for fixed m and $\alpha < \alpha_1$.
For $m = 61$, we have $\alpha_0 = 0.28, \alpha_1 = 0.55$ ($k = 17$ and $k = 33$, respectively, what matches the simulation). In this case, automaton partition is noticeable worse, and outperformed by dynamic programming.

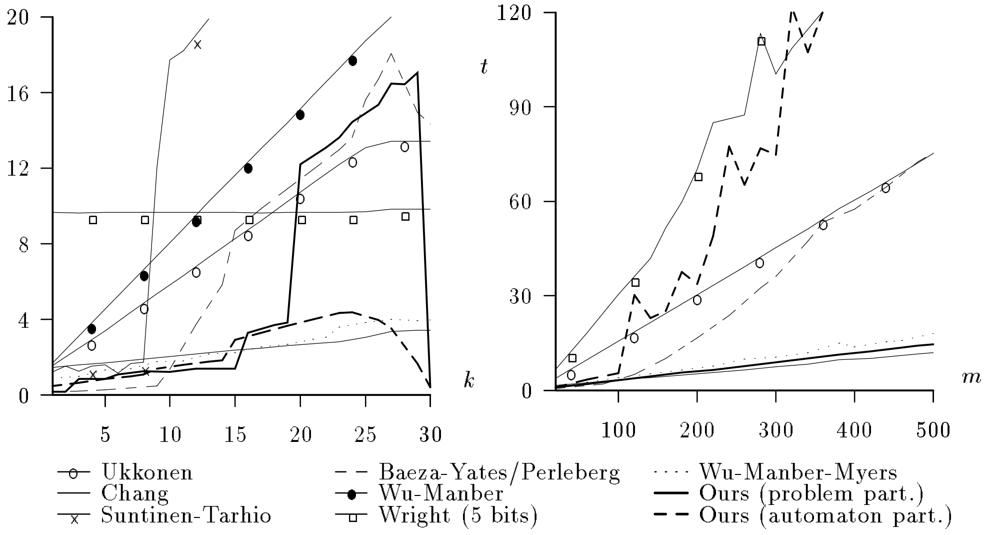

Fig. 7. Times in seconds for moderate $m = 31$ and $k = 0..30$ (left) and for large $m = 20..500$ and $\alpha = 0.3$ (right).

Very long patterns: we tested the case $\alpha = 0.3$ for m up to 500. Figure 7 (right plot) shows the results.
As it can be seen, reduction to exact matching is better up to $m = 100$ (we predict 105). Automaton partition is never the best choice. Our combined algorithm is the best except again for Chang, which is slightly better.
Note that automaton partition becomes worse than dynamic programming somewhere between $m = 100$ and $m = 160$. Our analysis predicts 152.

6 Concluding Remarks

We presented a new algorithm for approximate pattern matching, based on the simulation of an automaton by diagonals. This enables the possibility of computing the new values in parallel, using bit-parallelism. This is done if the problem fits in a single computer word (i.e. $(m - k)(k + 2) \leq w$). If it does not, we show a technique to partition the automaton into sub-automata. We show another technique to partition the problem into subproblems, that are searched with the simple algorithm and the candidate matches later verified. A special case of problem partitioning reduces the problem to exact search. The combined algorithm is $O(n)$ for small patterns or small α, $O(\sqrt{km/\log n}\ n)$ for medium α, and $O((m-k)kn/\log n)$ for large α.

We analyzed the optimum strategy to combine the algorithms and showed experimentally that our algorithm is the fastest for moderate patterns and error ratios. The longer the patterns, the smaller the error ratios for which our algorithm is the best. In the other cases, [6] is the best, except for k very close to m, where automaton partitioning becomes $O(mn/\log n)$ and outperforms the others.

As in the shift-or algorithm for exact matching [4], we can specify a set of characters at each position of the pattern instead of a single one (e.g. to search for "text" in case-insensitive, we search for $\{t,T\}\{e,E\}\{x,X\}\{t,T\}$). In fact, we can represent any "limited expression", as defined in [26]. This is achieved by modifying the t table, making any element of the set to match that position, with no running time overhead.

We can modify the algorithm to match whole words, by eliminating the initial empty transition (then the automaton computes edit distance), running the algorithm only from word beginnings (where we re-initialize $D = D_{in}$), and checking matches only at the end of words.

Although in this work we deal with finite alphabets, we can easily extend our algorithms for the infinite case, since the tables must only be filled for characters present in the pattern. In this case, a $\log m$ factor must be added to the complexities (to search into the tables), and the probability for two characters to be equal should no longer be $1/\sigma$ but a given p. This last modification allows also to analyze alphabets with non-uniform distribution.

Searching with different integral costs for insertion and substitution (including not allowing such operations) can be accomodated in our scheme, but deletion is built into the model in such a way that different costs for deletions cannot be accomodated.

The combination of problem and automaton partitioning may be useful in the area just past α_1, since the limit of low number of verifications can be moved to the right of α_1 by using a smaller j and partitioning the resulting automata. This provides a smooth transition between problem and automaton partitioning, since for $\alpha \geq 1 - m^{1/(m-k)}/\sqrt{\sigma}$ we have $j = 1$, i.e. pure automaton partitioning. This subject is currently under study.

More work, both analytical and experimental, is needed to refine the selection among heuristics. For example, we did not take into account the different constants involved in the shapes of the partitioned automata, or due to the heuristic of the first $k+1$ characters. The case of long patterns, small alphabets or high α deserves more study, since we concentrated mainly on text searching in this work.

Acknowledgments

We thank Gene Myers and Udi Manber for their helpful comments on this work. We also thank all the people who sent us working versions of their algorithms, what made the tests a lot easier and, in some sense, more fair: William Chang, Alden Wright (who also gave us his implementations of [24, 9, 20]), Gene Myers, Erkki Suntinen, and Tadao Takaoka.

References

1. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *CACM*, 18(6):333–340, June 1975.
2. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I: Algorithms, Software, Architecture, pages 465–476. Elsevier Science, September 1992.
3. R. Baeza-Yates. A unified view to pattern-matching problems. Dept. of Computer Science, Univ. of Chile. `ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/-unified.ps.gz`, 1995.
4. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *CACM*, 35(10):74–82, October 1992.
5. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192. Springer-Verlag, 1992. LNCS 644.
6. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, pages 172–181. Springer-Verlag, 1992. LNCS 644.
7. W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, Oct/Nov 1994.
8. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. of CPM'94*, pages 259–273. Springer-Verlag, 1994. LNCS 807.
9. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. of Computing*, 19(6):989–999, 1990.
10. G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer Systems Science*, 37:63–78, 1988.
11. G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10:157–169, 1989.
12. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
13. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.
14. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
15. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.
16. E. Suntinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. of ESA'95*. Springer-Verlag, 1995. LNCS 979.
17. T. Takaoka. Approximate pattern matching with samples. In *Proc. of ISAAC'94*, pages 234–242. Springer-Verlag, 1994. LNCS 834.

18. J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. In *Proc. of SWAT'90*, pages 348–359. Springer-Verlag, 1990. LNCS 447.
19. E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.
20. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
21. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
22. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.
23. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.
24. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
25. S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *J. of Algorithms*, 19:346–360, 1995.
26. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

Appendix. Upper bound for $f(m, k)$

To prove $f(m,k) = O(\gamma^m)$, we consider an upper bound to f: suppose a text area $Text[a..b]$ matches the pattern. Since we only report segments whose last character matches the pattern, we know that b is in pat. We consider a as the first character matching the pattern. Then, the length $s = b - a + 1$ is in the range $m-k..m+k$. Since there are up to k errors, at least $m-k$ characters of the pattern must be also in the text. Under a uniform model, the probability of that many matches is $1/\sigma^{m-k}$. Since these characters can be anyone in the pattern and in the text, we have

$$f(m,k) \le \sum_{s=m-k}^{m} \frac{1}{\sigma^{m-k}} \binom{m}{m-k} \binom{s-2}{m-k-2} + \sum_{s=m+1}^{m+k} \frac{1}{c^{s-k}} \binom{m}{s-k} \binom{s-2}{s-k-2}$$

where the two combinatorials count the ways to choose the $m-k$ (or $s-k$) matching characters from the pattern and from the text, respectively. The "-2" in the second combinatorials are because the first and last characters of the text must match the pattern. We divided the sum in two parts because if the area has length $s > m$, then more than $m-k$ characters must match, namely $s-k$. See Figure 8.

First assume constant α (we cover the other cases later). We begin with the first summation, which is easy to solve exactly to get $(1-\alpha)\binom{m}{k}^2/\sigma^{m-k}$. However, we prefer to analyze its largest term (the last one), since it is useful for the second summation too. The last term is

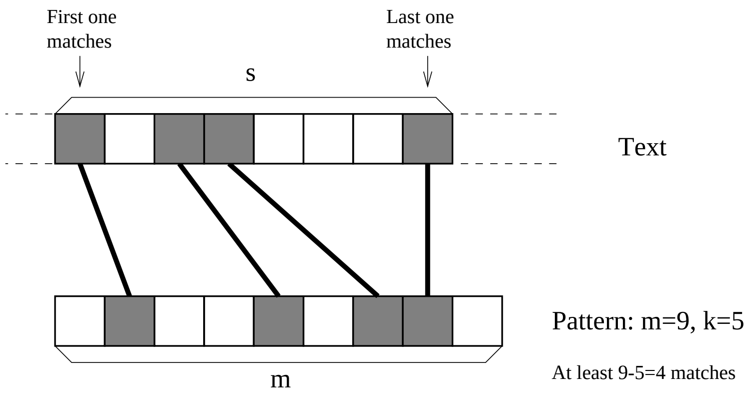

Fig. 8. Upper bound for $f(m,k)$.

$$\begin{aligned}\frac{1}{\sigma^{m-k}}\binom{m}{m-k}\binom{m-2}{m-k-2} &= \frac{(1-\alpha)^2}{\sigma^{m-k}}\binom{m}{k}^2\left(1+O\left(\frac{1}{m}\right)\right)\\ &= \left(\frac{1}{\sigma^{1-\alpha}\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)}}\right)^m m^{-1}\left(\frac{1-\alpha}{2\pi\alpha}+O\left(\frac{1}{m}\right)\right)\end{aligned}$$

where the last step is done using Stirling's approximation to the factorial.

Clearly, for the summation to be $O(\gamma^m)$ ($\gamma < 1$), this term must be of that order, and this happens if and only if the base of the exponential is $\leq \gamma$. On the other hand, the first summation is bounded by $k+1$ times the last term, so the first summation is $O(\gamma^m)$ if and only if this last term is. That is

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

It is easy to show analytically that $e^{-1} \leq \alpha^{\frac{\alpha}{1-\alpha}} \leq 1$ if $0 \leq \alpha \leq 1$, so for $\gamma = 1$ it suffices that $\sigma \geq e^2/(1-\alpha)^2$, or $\alpha \leq 1 - e/\sqrt{\sigma}$, while for arbitrary γ,

$$\alpha \leq 1 - \frac{e}{\sqrt{\sigma}\gamma^{\frac{1}{2(1-\alpha)}}} \qquad (7)$$

is a sufficient condition for the largest (last) term to be $O(\gamma^m)$, as well as the whole first summation.

We address now the second summation, which is more complicated. First, observe that

$$\sum_{s=m+1}^{m+k} \frac{1}{\sigma^{s-k}} \binom{m}{s-k} \binom{s-2}{s-k-2} \;\le\; \sum_{s=m}^{m+k} \frac{1}{\sigma^{s-k}} \binom{m}{s-k} \binom{s}{k}$$

a bound that we later find tight. In this case, it is not clear which is the larger term. We can see each term as

$$\frac{1}{\sigma^r} \binom{m}{r} \binom{k+r}{k}$$

where $m-k \le r \le m$. By considering $r = xm$ ($x \in [1-\alpha, 1]$) and applying again the Stirling's approximation, the problem is to maximize the base of the resulting exponential, that is

$$h(x) = \frac{(x+\alpha)^{x+\alpha}}{\sigma^x x^{2x} (1-x)^{1-x} \alpha^\alpha}$$

Elementary calculus leads to solve a second-grade equation that has roots in the interval $[1-\alpha, 1]$ only if $\sigma \le \alpha/(1-\alpha)^2$. Since due to Eq. (7) we are only interested in $\sigma \ge 1/(1-\alpha)^2$, $h'(x)$ does not have roots, and the maximum of $h(x)$ is at $x = 1-\alpha$. That means $r = m-k$, i.e. the first term of the second summation, which is the same larger term of the first summation.

We conclude that

$$f(m,k) \;\le\; \frac{2k+1}{m}\, \gamma^m \left(1 + O\left(\frac{1}{m}\right)\right) \;=\; O\left(\gamma^m\right)$$

Since this is an $O()$ result, it suffices for the condition to hold after a given m_0, so if $k = o(m)$ we always satisfy the condition.

We conducted experiments to determine the real limit for α (since this is an upper bound). The experimental curve fits almost exactly our formula, if we replace e by 1. Thus, we make that replacement in the heuristics.

We can prove, with a different model, that for $\alpha > 1 - 1/\sigma$ or $k = \Omega(m - o(m))$ the cost of verifications is significant.

This article was processed using the LaTeX macro package with LLNCS style