Algorithmica (1999) 23: 127–158

Algorithmica

Faster Approximate String Matching[1]

R. Baeza-Yates[2] and G. Navarro[2]

Abstract. We present a new algorithm for on-line approximate string matching. The algorithm is based on the simulation of a nondeterministic finite automaton built from the pattern and using the text as input. This simulation uses bit operations on a RAM machine with word length $w = \Omega(\log n)$ bits, where n is the text size. This is essentially similar to the model used in Wu and Manber's work, although we improve the search time by packing the automaton states differently. The running time achieved is $O(n)$ for small patterns (i.e., whenever $mk = O(\log n)$), where m is the pattern length and $k < m$ is the number of allowed errors. This is in contrast with the result of Wu and Manber, which is $O(kn)$ for $m = O(\log n)$. Longer patterns can be processed by partitioning the automaton into many machine words, at $O(mk/w\, n)$ search cost. We allow generalizations in the pattern, such as classes of characters, gaps, and others, at essentially the same search cost.

We then explore other novel techniques to cope with longer patterns. We show how to partition the pattern into short subpatterns which can be searched with less errors using the simple automaton, to obtain an average cost close to $O(\sqrt{mk/w}\, n)$. Moreover, we allow the superimposition of many subpatterns in a single automaton, obtaining near $O(\sqrt{mk/(\sigma w)}\, n)$ average complexity (σ is the alphabet size).

We perform a complete analysis of all the techniques and show how to combine them in an optimal form, also obtaining new tighter bounds for the probability of an approximate occurrence in random text. Finally, we show experimental results comparing our algorithms against previous work. These experiments show that our algorithms are among the fastest for typical text searching, being the fastest in some cases. Although we aim mainly at text searching, we believe that our ideas can be successfully applied to other areas such as computational biology.

Key Words. Text searching allowing errors, Bit-parallelism.

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. We call $\alpha = k/m$ the *error ratio*, and σ the alphabet size.

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 has $O(mn)$ running time [21], [22].

Recently several algorithms have been presented that achieve $O(kn)$ comparisons in the worst-case [28], [13], [16], [17] or in the average case [29], [13], by taking advantage

[1] This work has been supported in part by FONDECYT Grants 1950622 and 1960881.
[2] Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. {rbaeza,gnavarro}@dcc.uchile.cl.

Received November 22, 1996; revised October 13 and December 5, 1997. Communicated by M. Y. Kao.

of the properties of the dynamic programming matrix (e.g., values in neighbor cells differ at most by one). Following the same trend is [9], with average time complexity $O(kn/\sqrt{\sigma})$. All average-case figures assume random text and uniformly distributed alphabets.

Other approaches first filter the text, reducing the area where dynamic programming needs to be used [27], [30], [26], [25], [10], [11], [19], [8], [23]. These algorithms achieve "sublinear" expected time in many cases for low error ratios (i.e., not all text characters are inspected, $O(kn \log_\sigma m/m)$ is a typical figure), although filtration is not effective for more errors. Filtration is based on the fact that some portions of the pattern must appear with no errors even in an approximate occurrence.

In [29] the use of a deterministic finite automaton (DFA) which recognizes the approximate occurrences of the pattern in the text is proposed. Although the search phase is $O(n)$, the DFA can be huge. In [20] and [15] the automaton is computed in lazy form (i.e., only the states actually reached in the text are generated).

Yet other approaches use bit-parallelism [2], [4], [34], i.e., they simulate parallelism with bit operations in a RAM machine of word length $w = \Omega(\log n)$ bits, to reduce the number of operations. In [32] the cells of the dynamic programming matrix are packed in diagonals to achieve $O(mn \log(\sigma)/w)$ time complexity. In [35] a Four Russians approach is used to pack the matrix in machine words, achieving $O(kn/w)$ time on average (ending up in fact with a DFA where they can trade the number of states for their internal complexity).

Of special interest to this work is [34], which considers a nondeterministic finite automaton (NFA) functionally similar to the DFA but having only a few states. $O(kmn/w)$ time is achieved in [34] by parallelizing in bits the work of such an automaton.

The main contributions of the present work follow.

- We present a new algorithm for approximate string matching. It is based on bit-parallelism, where bit operations are used to simulate the behavior of an NFA built from the pattern and using the text as input. This automaton is similar to that of Wu and Manber [34]. However, we pack diagonals instead of rows into machine words. This leads to a worst-case running time of $O(n)$ (independently of k) for small patterns (i.e., $mk = O(w)$). This is the case in most single-word text patterns. This simple algorithm is experimentally shown to be the fastest one we are aware of for short patterns, except for very small k values where other filtration algorithms are even faster.
- We show how to partition a large automaton that does not fit in a computer word, to obtain an algorithm which performs near $\lceil mk/w \rceil n$ operations, against $\lceil m/w \rceil kn$ performed by Wu and Manber (their operations are simpler, though). The difference is especially noticeably in small patterns. This algorithm performs well for all error ratios if the pattern is not very long, being the fastest for high error ratios, an area largely unexplored in previous work.
- We show that some generalizations in the pattern (which we call *extended patterns*) can be handled at almost no additional search cost. We allow one to have classes of characters at each pattern position (instead of just one character per position), gaps inside the pattern, etc.
- We present a technique to partition a long pattern into subpatterns, such that the

subpatterns can be searched with less errors using the simple automaton. This search is in fact a filter, since the occurrences of the subpatterns must be verified for complete matches. The average time of this search is close to $O(\sqrt{mk/w}\, n)$, and can be used even for moderately high error levels. We show experimentally that this algorithm is competitive with the best known algorithms.

- We show that some of the subpatterns obtained in the previous algorithm can in fact be superimposed in a single automaton to speed up searching. We analytically find the maximum degree of superimposition allowed (which is limited by the error level), and obtain an algorithm with average cost close to $O(\sqrt{mk/(\sigma w)}\, m^{2/\sqrt{w}} n)$. We show experimentally that speedup factors of two and three can be obtained for low error ratios.
- We analyze the possibility of combining pattern and automaton partitioning, finding that the combination increases the tolerance to errors in pattern partitioning, and provide a smooth transition between pattern and automaton partitioning as the error level increases. This mixed partitioning has an average search cost of $O(k \log(m)/w\, n)$. We show experimentally how this mix works for an intermediate error level.
- We present a simple speedup technique that doubles in practice the performance of all our algorithms for low error levels, by allowing one to discard large parts of the text quickly at low cost per character. This technique is quite general and can be applied to most other algorithms as well (except perhaps the filtration algorithms).
- We thoroughly analyze all our algorithms and the areas where they are applicable, finding out the optimal heuristic that combines them. As a side effect, we give tighter bounds for the probability of an approximate occurrence on random text. We also give better bounds for the analysis of the Ukkonen cut-off algorithm [29].
- We present extensive experiments for the running times of our algorithms and previous work. They are aimed at showing the practical behavior of our algorithms, at verifying our analysis, and comparing our work against others. We show that our algorithms are among the fastest for typical text searching, especially for short patterns.

Approximate string matching is a particular case of sequence similarity, an important problem in computational biology. Although this work is specifically aimed at text searching, we believe that some ideas in this paper can be adapted to more generic cost functions associated to string editing operations or more generic patterns, for example, to RNA prediction [12] or to DNA sequence databases [18] (where, incidentally, the search is made on very short patterns).

Preliminary versions of parts of this work can be found in [6] and [5].

This paper is organized as follows. In Section 2 we introduce the problem and the associated NFA. In Section 3 we explain our automaton simulation and how to split it into many machine words. In Section 4 we present other problem partitioning techniques. In Section 5 we analyze the algorithms and find the optimal way to combine them in a general heuristic. In Section 6 we experimentally validate our analysis and compare our algorithm against others. Finally, in Section 7 we give our conclusions and future work directions.

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 of size σ, find all segments (called

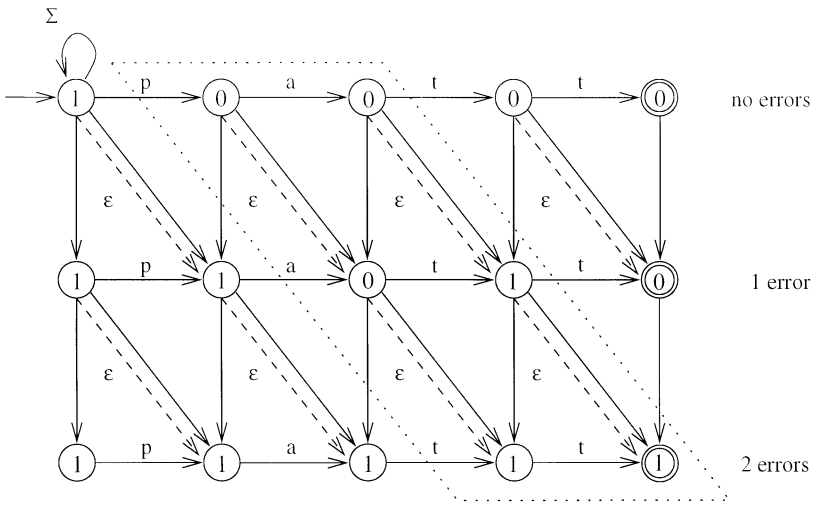

Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character. Active states are (2, 3), (2, 4), and (1, 3), besides those always active of the lower-left triangle. We enclose in dotted lines the states actually represented in our algorithm.

"occurrences" or "matches") of *Text* whose *edit distance* to *pat* is at most k, the number of allowed errors. It is common to report only the endpoints of occurrences, as well as to discard occurrences containing or contained in others. In this work we focus on returning endpoints of occurrences not containing others.

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 nontrivial for $0 < k < m$. The *error ratio* is defined as $\alpha = k/m$.

We use a C-like notation for the operations (e.g., $\&, |, ==, !=, \wedge, \gg$). We use *text* to denote the current character of *Text* and, unlike C, $str[j]$ to denote the jth character of *str* (i.e., the strings begin at position one, not zero).

Consider the NFA for searching `"patt"` 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 (since we advance in the pattern and in the text, and they can only be followed if the corresponding match occurs), vertical arrows represent inserting a character in the pattern (since we advance in the text but not in the pattern, increasing the number of errors), solid diagonal arrows represent replacing a character (since we advance in the text and pattern, increasing the number of errors), 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, and increase the number of errors). Finally, the self-loop at the initial state allows us 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 of the occurrences, we can consider as 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 $(m+1) \times (k+1)$ states. We assign number (i, j) to the state at row i and column j, where $i \in 0 \cdots k$, $j \in 0 \cdots m$. 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.

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:

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

which is used for $i \in 0 \cdots k$, $j \in 1 \cdots m$. If $i = 0$ only the first term is used. Note that the empty transitions are 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 self-loop 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 working on a dynamic programming matrix. Each column of the matrix holds the state of the search at a given text character. It is not hard to show that what the column stores at position j is the minimum row of this NFA which is active in column j of the automaton. In this sense, the dynamic programming matrix corresponds to simulating this automaton by columns (i.e., packing columns in machine words) [3].

On the other hand, the work of Wu and Manber consists fundamentally in simulating this automaton by rows (packing each row in a machine word). In both cases the dependencies introduced by the diagonal empty transitions prevent the parallel computation of the new values. In the next section we present 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 always have 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 us to carry 1's from the last diagonals to state (m, k). However, each 1 present at the last diagonals must have crossed the last full diagonal, where the empty transitions (deletions) would have immediately 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.

Notice also that, whenever state (i, j) is active, 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 $0 \cdots k + 1$. Note that D_0 is always 0, hence, there is no need to store it.

The new values for D_i $(i \in 1 \cdots m - k)$ after we read a new text character c are

derived from (1):

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

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

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

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 all subsequent states on that diagonal active (so we keep just 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 it 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.

3.1. *A Linear Algorithm for Small Patterns.* In this section we show 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 in bits of the computer word). We first select a suitable representation for our problem and then describe the algorithm.

Since we have $m-k$ nontrivial diagonals, and each one takes values in the range $0 \cdots k+1$, 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. We could precompute and store it, but it would take $O(\sigma(k+1)^{m-k})$ space if it had to be accessed in parallel for all i. At this exponential space cost, the automaton approach of [29] and [20] 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 always having 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 $D_1, \ldots, D_{m-k}$ is

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

where we use exponentiation to mean digit repetition. Observe that 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 D in 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.

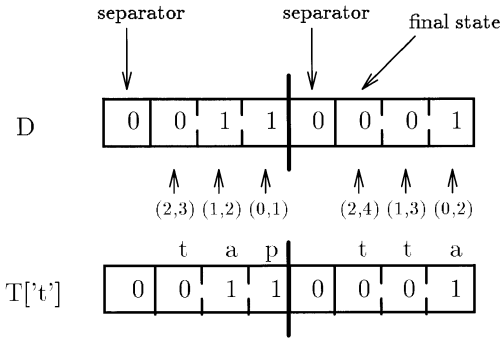

Fig. 2. Encoding of the example NFA. In this example, $t[\texttt{'t'}] = 0011$.

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,

$$t[c] = (c\,!=pat[m])\ (c\,!=pat[m-1])\ \cdots\ (c\,!=\ pat[1]), \tag{3}$$

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

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

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 computer word if the problem does.

We now have 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 $D_{\text{in}} = (0\ 1^{k+1})^{m-k}$. The formula to update D upon reading a text character c is derived from (2):

$$\begin{aligned} D' \ = \ & (D \ll 1) \mid (0^{k+1}1)^{m-k} \\ & \& \ (D \ll (k+3)) \mid (0^{k+1}1)^{m-k-1}0\ 1^{k+1} \\ & \& \ (((x + (0^{k+1}1)^{m-k}) \ \wedge \ x) \gg 1) \\ & \& \ D_{\text{in}}, \end{aligned} \tag{4}$$

where

$$x = (D \gg (k+2)) \mid T[c].$$

The update formula is a sequence of *and*'s, corresponding to the min of (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 0's in the separators (needed to limit the propagation of "+"). Note that we are assuming that the shifts get 0's from both borders of the word (i.e., unsigned semantics). If this is not the case in a particular architecture, additional masking is necessary.

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

3.2. *A Simple Filter.* We can improve the previous algorithm (and in fact most other algorithms as well) by noticing that any approximate occurrence of the pattern with k errors must begin with one of its $k + 1$ first characters. This allows us to discard large parts of the text quickly with very few operations per character.

We do not run the automaton through all text characters, but scan the text looking for any of the $k + 1$ initial characters of the pattern. Only then do we start the automaton. When the automaton returns to its initial configuration, we resume the scanning. The scanning is much cheaper than the operation of our automaton, and in fact it is cheaper than the work done per text character in most algorithms.

We precompute a boolean table $S[c]$, that stores for each character c whether it is one of the first $k + 1$ letters 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 (i.e., using the automaton plus the filter).

```
search (Text, n, pat, m, k)
 {        /* preprocessing */
   for each c ∈ Σ
     { 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 ∈ 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) ∧ 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. Strings are assumed to start at position 1.

For simplicity, we do not refine the preprocessing, which can be done more efficiently than the code suggests.

3.3. *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 subautomata as necessary, each one holding a number of diagonals. We call "d-columns" those sets of diagonals packed in a single machine word. Those subautomata 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 d-columns, where $J = \lceil (m-k)(k+2)/w \rceil$. Note that we need that at least one automaton 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 subautomata behave differently than the previous ones, since we must propagate the ε-transitions down to all subsequent subautomata. In this case, if $(m-k)(k+2) > w$, we partition the automaton vertically in I d-rows (each d-row holding some automaton rows of all diagonals), where I has the same formula as J. The difference is that, in this case, we need that at least one automaton 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 previous conditions hold, we need a generalized partition in d-rows and d-columns. We use I d-rows and J d-columns, so that each cell contains ℓ_r bits of each one of ℓ_c diagonals. It must hold that $(\ell_r+1)\ell_c \leq w$.

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 prefer to select 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 d-column and the last d-row may be smaller, since they have the residues.

Simulation of the automaton is now more complex, but follows the same principle of the update formula (5). We have a matrix of automata $D_{i,j}$ ($i \in 0 \cdots I-1$, $j \in 0 \cdots J-1$), and a matrix of masks $T_{i,j}$ coming from splitting the original T. The new update formula is

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

where

$$\begin{aligned} x \;=\; & ((D_{i,j} \gg (\ell_r+1)) \mid (D_{i,j-1} \ll (\ell_r+1)(\ell_c-1)) \mid T_{i,j}[\mathit{text}]) \\ & \&\; ((D'_{i-1,j} \gg (\ell_r - 1)) \mid (1^{\ell_r}0)^{\ell_c}) \end{aligned}$$

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 scheme to fall into three simpler cases: (a) the automaton is horizontal, (b) the automaton is horizontal and only one diagonal fits in each word, and (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. In practice, there are some cases where a purely vertical automaton is better [5]. In particular, Wu and Manber's automaton can be thought of as a vertical partitioning of the NFA of Figure 1 (although the details are different).

3.4. *Handling Extended Patterns*. We now show that some of the generalizations of the approximate string search problem considered in [4], [34] and [35] can be introduced into our algorithm at no additional search cost. We call those patterns involving some of these generalizations *extended patterns*.

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 (this is called "limited expressions" in [35]). For example, to find "patt" in case-insensitive, we search for {p, P}{a, A}{t, T}{t, T}; to find "patt" followed by a digit, we search for {p}{a}{t}{t}{0..9}. This is achieved by modifying the $t[]$ table (3), making any element of the set to match that position, with no running time overhead.

In addition to classes of characters, we can support the # operator as defined in [34]. That is, $x\#y$ allows zero or more arbitrary characters among the strings x and y in the occurrences. Those characters are not counted as errors. As shown in [33], in order to handle this operator we must enforce that whenever an automaton state in column $|x|$ is active, it keeps active from then on.

Hence, to search for $x_1\#x_2\#\cdots\#x_t$ we create a $D_\#$ word having all 1's except at all states of columns $|x_1|$, $|x_1| + |x_2|$, ..., $|x_1| + |x_2| + \cdots + |x_{t-1}|$. We now modify the computation of x in (2), which becomes

$$x = ((D \gg (k+2)) \mid T[text]) \ \& \ (D \mid D_\#)$$

(clearly this technique is orthogonal to the use of classes of characters). This technique is easily extended to partitioned automata.

We can modify the automaton to compute edit distance (more precisely, determine whether the edit distance is $\leq k$ or not). This is obtained by eliminating the initial self-loop and initializing the automaton at $D = D_{\text{in}}$. However, we need to represent the $k+1$ initial diagonals that we discarded. If we need the exact edit distance, we must also represent the last k diagonals that we discarded. If there is no a priori bound on the distance, we need to set $k = m$.

We can search for whole words, running the edit-distance algorithm only from word beginnings (where we re-initialize $D = D_{\text{in}}$), and checking matches only at the end of words.

Searching with different integral costs for insertion and substitution (including not allowing such operations) can be accommodated in our scheme, by changing the arrows.

Deletion is built into the model in such a way that in order to accommodate it we must change the meaning of our "diagonals," so that they are straight ε-transition chains.

Other generalizations are studied in [34]. We can handle them too, although not as easily and efficiently as the previous ones.

One such generalization is the combination in the pattern of parts that must match exactly with others that can match with errors. The adaptation to avoid propagation of ε-closures in our scheme is ad hoc and not as elegant as in [34]. However, we believe that the most effective way to handle these patterns is to search quickly for the parts that match exactly and then try to extend those matches to the complete pattern, using our automaton to compute edit distance.

Another such generalization is approximate search of regular expressions. In [34] the regularities among rows allow one to solve any regular expression of m letters using $\lceil m/8 \rceil$ or even $\lceil m/16 \rceil$ operations per text character, using $\lceil m/8 \rceil 2^8 \lceil m/w \rceil$ or $\lceil m/16 \rceil 2^{16} \lceil m/w \rceil$ machine words of memory, respectively. Our partitioned automata are not so regular, and we would need roughly $O(k^2)$ times Wu and Manber's space requirements and operations per text character. To be more precise, in our scheme their formulas are still valid provided we replace m by $(m-k)(k+2)$. For instance, at the cost they pay for $m \leq 32$, we can only solve for $m \leq 9$. However, our scheme is still reasonably applicable for short expressions.

4. New Partitioning Techniques. The following lemma, which is a generalization of the partitioning scheme presented in [34] and [18], suggests a way to partition a large problem into smaller ones.

LEMMA 1. *If segm* $=$ *Text*$[a..b]$ *matches pat with k errors, and pat* $= p_1 \cdots p_j$ (*a concatenation of subpatterns*), *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 $segm = s_1 \cdots s_j$. However, 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. □

The lemma is a generalization of that of [34] because Wu and Manber consider only the case $j = k+1$ (we return to this case later), and it is a (slight) generalization of that of [18] because Myers considers only cases of the form $j = 2^t$ (the pattern is partioned in a binary fashion until short enough subpatterns are obtained). Moreover, the previous authors always consider partitioning into pieces of almost the same size, while the lemma allows any split. We now explore different ways to use the lemma.

4.1. *Pattern Partitioning.* The lemma allows us to reduce the number of errors if we divide the pattern, provided we search all the subpatterns. Each match of a subpattern must be checked to determine if it is in fact a complete match (notice that the subpatterns

can be extended patterns themselves). Suppose we find at position i in *Text* the end of a match for the subpattern 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 an algorithm resistant to high error levels, such as our automaton partitioning technique. If the pattern is not extended, the Ukkonen cut-off algorithm [29] can also be used.

To perform the partition, we pick an integer j, and split the pattern into j subpatterns of length m/j (more precisely, if $m = qj + r$, with $0 \le r < j$, r subpatterns 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 subpatterns 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,

$$(5) \qquad j = \min\left\{ r \;\middle/\; \left(\left\lceil \frac{m}{r} \right\rceil - \left\lfloor \frac{k}{r} \right\rfloor\right)\left(\left\lfloor \frac{k}{r} \right\rfloor + 2\right) \le w \;\wedge\; \left\lfloor \frac{m}{r} \right\rfloor > \left\lfloor \frac{k}{r} \right\rfloor \right\},$$

where the second condition avoids searching a subpattern 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 zero errors.

In case of zero errors, we can use an Aho–Corasick machine [1] to guarantee $O(n)$ total search time. In [34] a variation of the Shift–Or algorithm [4] is used, while in [8] the use of an algorithm of the Boyer–Moore family is suggested. The advantage of the Wu and Manber approach is flexibility in the queries, while Boyer–Moore is faster (this is also supported by [31], since we typically have few subpatterns). In particular, if the pattern is extended, the Shift–Or algorithm is the correct choice.

For simple patterns, we preferred to use the Sunday algorithm [24] of the Boyer–Moore family, extended to multipattern search. This extension consists of building a trie with the subpatterns, and at each position searching the text into the trie. If we do not find a subpattern, we advance the window using the Sunday shift. This is the (precomputed) minimum shift among all patterns. However, this should be considered just an implementation of previous work.

Figure 4 shows the general algorithm, which is written in that way for clarity. In a practical implementation, it is better to run all subsearches in synchronization, picking at any moment the candidate whose initial checking position is the leftmost in the set,

```
PatternPartition (Text, n, pat, m, k)
 { j = min { r / (⌈m/r⌉ − ⌊k/r⌋)(⌊k/r⌋ + 2) ≤ w ∧ ⌊m/r⌋ > ⌊k/r⌋}
   if (j == 1) search (Text, n, pat, m, k)
   else { a = 0
          for r ∈ 0..j − 1
            { len = (r < m % j) ? ⌈m/j⌉ : ⌊m/j⌋
              b = a + len − 1
              for each position i reported by search(Text, n, pat[a..b], len, ⌊k/j⌋)
                  check the area Text[i − b + 1 − k, i − b + 1 + m + k]
              a = b + 1
 }      }      }
```

Fig. 4. Algorithm for pattern partitioning.

checking its area, and advancing that subsearch to its next candidate position. This allows us to avoid reverifying the same text because of different overlapping candidate areas. This is done by remembering the last checked position and keeping the state of the checking algorithm.

The effectiveness of this method is limited by the error level. If the subpatterns appear very often, we spend a lot of time verifying candidate text positions. In Section 5 we find out which is the error level that allows us to use this scheme.

4.2. *Superimposing the Subproblems.* When the search is divided into a number of subsearches for smaller patterns $P_1, \ldots, P_r$, it is possible not to search each one separately. We describe a technique, called *superimposition*, to collapse a number of searches into a single one.

In our scheme, all patterns have almost the same length. If they differ (at most by one), we truncate them to the shortest length. 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 $t[]$ table for each pattern (see (3)), and then take the bitwise-*and* of all the tables. The resulting $t[]$ table matches in position i with the ith character of *any* pattern. 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 (in fact it is an *extended pattern*), namely,

$$\{P_1[1], \ldots, P_r[1]\} \cdots \{P_1[m], \ldots, P_r[m]\},$$

for example, if the search is for `patt` and `wait`, the string `watt` is accepted with *zero* errors (see Figure 5). Each occurrence reported by the automaton has to be verified for all the patterns involved.

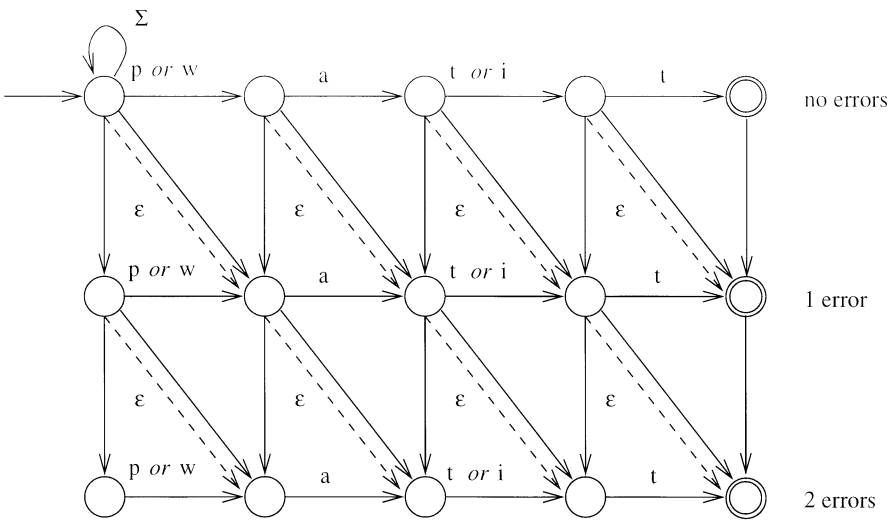

Fig. 5. An NFA to filter the parallel search of `patt` and `wait`.

For a moderate number of patterns, this still constitutes a good filtering mechanism, at the same cost of a single search. Clearly, the relaxed pattern triggers many more verifications than the simple ones. This severely limits the amount of possible superimposition. However, as we show later, in practice this can cut the search times by two or three. This idea has been applied to the problem of multiple approximate string matching, where similar speedup figures were obtained [7].

We analyze later how many subpatterns can be collapsed while keeping the number of verifications small. We must then form sets of patterns that can be searched together, and search each set separately. Notice that extended patterns containing #'s may not be superimposed because their $D_{\#}$ words are different.

Observe that having the same character at the same position for two patterns improves filtering efficiency. This fact can be used to select the best partition of the pattern.

4.3. *Mixed Partitioning.* Finally, it is possible to combine automaton and pattern partitioning. If we partition the pattern in subpatterns that are still too large to fit in a computer word, the automaton for each subpattern has to be further partitioned into subautomata. This is a generalization that includes both techniques as particular cases, and uses a smaller j value.

As we show in the analysis, however, this technique is no better in practice than pure pattern partitioning. However, we find that the larger j is, the less tolerant to the error level our scheme is. Therefore, using a smaller j can be necessary if α is high. For high error levels we use the largest j allowed. For α large enough, though, the only allowed value for j is one, which is equivalent to pure automaton partitioning. Therefore, mixed partitioning provides a smooth transition between pure pattern and pure automaton partitioning. Superimposition could also be applied on the resulting (large) automata.

5. Analysis. In this section we analyze the different aspects of our algorithms. We also study some statistics of the problem which are useful for the analysis. We make heavy use of the shorthand $\alpha = k/m$.

It is important to notice that our average-case analysis assumes that the pattern is not extended and that text and patterns are random strings over an independently generated and uniformly distributed alphabet of size σ. If the alphabet is not uniformly distributed we must replace the σ in the formulas by $1/p$, where p is the probability that two random letters match. For generalized patterns, the α_i values are different, but we are unable to compute them.

We summarize here the results of this section, which involve the optimal heuristic to combine our algorithms. Table 1 presents a summary of the important limiting α values found in the analysis.

We now sketch how the algorithm behaves among the areas delimited by m and $\alpha_0 < \alpha_1 < \alpha_2$. In all cases we can use the heuristic of the S table to reduce the average cost.

- If the problem fits in a single word (i.e., $(m-k)(k+2) \leq w$), the simple algorithm should be used, which is $O(n)$ time in the average and worst case. If the problem does not fit in a single word, we may use pattern, automaton, or mixed partitioning.

Table 1. Limiting α values in our analysis. The "simplified definition" is a valid approximation for moderate m values.

Name	Simplified definition	Meaning	Reference
α_0	$1-\sqrt{m/(\sigma\sqrt{w})}\,m^{1/\sqrt{w}}$	Up to where all the subpatterns in pattern partitioning can be safely superimposed	Equation (9), Section 5.4
α_1	$1-m^{1/\sqrt{w}}/\sqrt{\sigma}$	Up to where pure pattern partitioning can be used	Equation (13), Section 5.5
α_2	$1-1/\sqrt{\sigma}$	Up to where mixed partitioning can be used	Equation (17), Section 5.6

- For $\alpha < \alpha_1$ pattern partitioning (plus superimposition) is the best choice. If $\alpha < \alpha_0$, the effect of superimposition makes pattern partitioning use $O(n)$ time on average. In the range $\alpha_0 < \alpha < \alpha_1$ the average cost is close to $O(\sqrt{km/(\sigma w)}\,m^{2/\sqrt{w}}n)$ (see (12)). This grows as $\sqrt{k}$ for fixed m. For $w = 32$ the cost is approximately $O(\sqrt{k/(\sigma w)}\,m^{0.85}n)$.
- For $\alpha_1 < \alpha < \alpha_2$ the error level makes it necessary to use a smaller j (i.e. mixed partitioning). The average search cost becomes $O(k\log(m)/w\,n)$ (see (16)).
- For $\alpha > \alpha_2$, neither pattern nor mixed partitioning are advisable, because of the large number of verifications, and pure automaton partitioning becomes the only choice. The average and worst-case cost for this area becomes $O((m-k)k/w\,n)$ (see (7) for the case $\alpha > 1-e/\sqrt{\sigma}$).

Figure 6 illustrates the analysis. We show on the left the exact α values for $w = 32$, $\sigma = 32$, $m = 10\ldots60$, and e replaced by 1.09 (see later). As we see later, α_0' is the maximum error level up to which exact partitioning is linear (more precisely, it is the exact solution of (14)). On the right we show schematically the combined complexity.

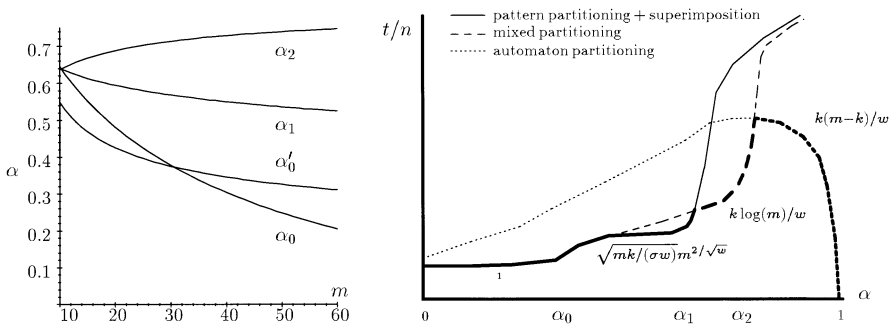

Fig. 6. On the left, exact α values for $w = 32$ and $\sigma = 32$. On the right, the (simplified) complexities of our algorithm.

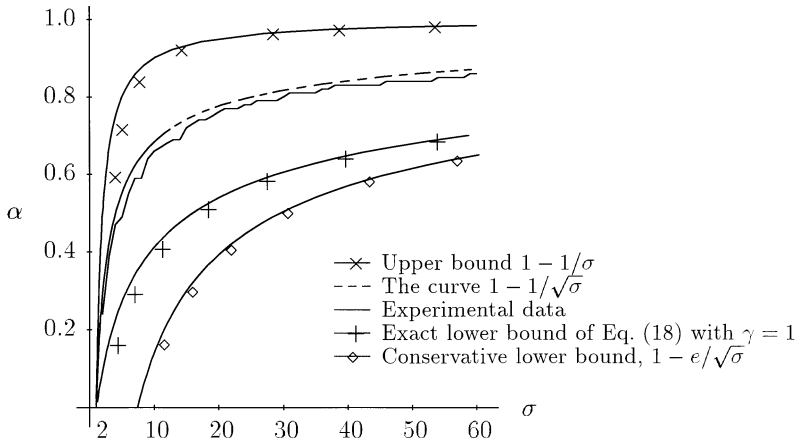

Fig. 7. Theoretical and practical bounds for α.

5.1. *The Statistics of the Problem.* Even natural questions about the distribution and statistical behavior of this problem are very hard to answer. Some of them are: what is the probability of an occurrence? How many occurrences are there on average? How many columns of the dynamic programming matrix are active? Some of these questions also arise in the analysis of our algorithms. We give our new results here.

A first concern is the probability of matching. Let $f(m, k)$ be the probability of a random pattern of length m matching a given text position with k errors or less. This probability is crucial to assure that the number of verifications of candidate matches in pattern partitioning is not too large. We show in the Appendix that for $\alpha \leq 1 - e/(\sqrt{\sigma}\,\gamma^{1/(2(1-\alpha))})$, $f(m, k)$ is $O(\gamma^m)$, with $\gamma < 1$. In particular, for $\alpha < 1 - e/\sqrt{\sigma}$, $f(m, k)$ is exponentially decreasing with m.

Figure 7 shows the experimental verification of this analysis for $m = 300$ (we obtained similar results with other m values). The curve $\alpha = 1 - 1/\sqrt{\sigma}$ is included to show its closeness to the experimental data. Least squares gives the approximation $\alpha = 1 - 1.09/\sqrt{\sigma}$, with a squared error smaller than 10^{-4}.

The experiment consists of generating a large random text and running the search of a random pattern on that text, with $k = m$ errors. At each text character, we record the minimum k for which that position would match the pattern. Finally, we analyze the histogram, and consider that k is safe up to where the histogram values become significant. The threshold is set to n/m^2, since m^2 is the cost of verifying a match. However, the selection of this threshold is not very important, since the histogram is extremely concentrated. For example, it has five or six significant values for m in the hundreds.

A second question we answer is: what is the number of columns we work on, on average, in the cut-off heuristic of Ukkonen [29]? That is, if we call C_r the smallest-row active state of column r in our NFA, what is the largest r satisfying $C_r \leq k$? The columns satisfying $C_r \leq k$ are called *active*, and columns past the last active one need not be computed. Ukkonen conjectured that the number of active columns was $O(k)$, but this

was proved later by Chang and Lampe [9]. We follow the proof of [9] to find a tighter bound. If we call L the last active column, we have

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

for any K. Since we know that if $k/r < 1 - e/\sqrt{\sigma}$, then $P[C_r \le k] = O(\gamma^r)$ with $\gamma < 1$, we take $K = k/(1 - e/\sqrt{\sigma})$ to obtain

$$E(L) \le \frac{k}{1 - e/\sqrt{\sigma}} + \sum_{r>k/(1-e/\sqrt{\sigma})} r\,O(\gamma^r) = \frac{k}{1 - e/\sqrt{\sigma}} + O(1), \tag{6}$$

which shows that, on average, the last active column is $O(k)$. This refines the proof of [9] that the heuristic of [29] is $O(kn)$. By using least squares on experimental data we find that a very accurate formula is $0.9k/(1 - 1.09/\sqrt{\sigma})$.

5.2. *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 $(m-k)(k+2) \le w$. In the RAM model it is assumed $\log_2 n \le w$, so a machine-independent bound is $(m-k)(k+2) \le \log_2 n$. Since $(m-k)(k+2)$ takes its maximum value when $k = m/2-1$, we can assure that this algorithm can be applied whenever $m \le 2(\sqrt{w} - 1)$, independently of k. That is, we have a linear algorithm for $m = O(\sqrt{\log n})$, for example, $m \le 9$ for $w = 32$ bits, or $m \le 14$ for $w = 64$ bits.

5.3. *Partitioning the Automaton.* If we divide the automaton in IJ subautomata (I d-rows and J d-columns), we must update I cells at each d-column. However, we use a heuristic similar to [29] (i.e., not processing the m columns but only up to the last active one), so we work only on active automaton diagonals.

To compute the expected last active diagonal, we use the result of (6). Since this measures active columns and we work on active diagonals, we subtract k to obtain that on average we work on $ke/(\sqrt{\sigma} - e)$ diagonals. Since we pack $(m - k)/J$ diagonals in a single cell, we work on average on $(ke/(\sqrt{\sigma} - e) + 1)J/(m - k)$ machine words. However, since there are only J d-columns, our total complexity is

$$IJ \min\left(1, \frac{ke}{(m - k)(\sqrt{\sigma} - e)}\right) n,$$

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:

$$\min\left(m - k\,,\ \frac{ke}{\sqrt{\sigma} - e}\right) \frac{k+1}{w-1}\, n, \tag{7}$$

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

To see where 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 $\alpha(1-\alpha) < w/m$, it is better to partition the automaton, otherwise it is better just to use dynamic programming. Since $\alpha(1-\alpha) \leq \frac{1}{4}$, automaton partitioning is always better than dynamic programming for $m \leq 4w$, i.e., for moderate-size patterns (this can be slightly different in practice because the operations do not cost the same).

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

5.4. *Partitioning the Pattern.* There are two main components in the cost of pattern 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 $O(jm^2 f(m/j, k/j)n)$, where $f(m,k)$ is defined in Section 5.1 (observe that the error level α is the same for the subpatterns). The complexity comes from considering that we perform j independent searches, and each verification costs $O(m^2)$ if dynamic programming is used for verification.[3] Clearly, in the worst case each text position must be verified, and since we avoid rechecking 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 (5),

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

whose solution is

$$j = \frac{m - k + \sqrt{(m-k)^2 + wk(m-k)}}{w}. \tag{8}$$

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

$$O(jn) = O(\sqrt{mk/w}\, n),$$

where the simplification is valid for $\alpha > 1/w$. However, we improve this complexity in the next section by using superimposition.

We now consider the limit error level of applicability. We know that for $\alpha \leq 1 - e/(\sqrt{\sigma}\,\gamma^{1/(2(1-\alpha))})$, $f(m,k) = O(\gamma^m)$, 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 $\gamma \leq 1/m^{2j/m}$. For that γ it holds that $1/\gamma^{1/(2(1-\alpha))} \geq m^{2j/(2m(1-\alpha))} = m^{j/(m-k)}$. Therefore, $f(m/j, k/j) = O(1/m^2)$ if $\alpha \leq \alpha_1$, where

$$\alpha_1 = 1 - \frac{e}{\sqrt{\sigma}}\, m^{j/(m-k)} = 1 - \frac{e}{\sqrt{\sigma}}\, m^{d(w,\alpha_1)} \tag{9}$$

up to where the cost of verifications is not significant. The function d is defined as $j/(m-k)$, which by replacing j according to (8) yields

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

where $d(w,\alpha) < 1$ for $\alpha < 1 - 1/(w-1)$. A good approximation for α_1 (for moderate m) is $\alpha_1 \approx 1 - m^{1/\sqrt{w}}/\sqrt{\sigma}$.

[3] Recall that we can use this algorithm only if the pattern is not extended, but this analysis holds only for simple patterns anyway.

5.5. *Superimposing the Subpatterns*. Suppose we decide to superimpose r patterns in a single search. We are limited in the amount of this superimposition because of the increase in the error level to tolerate. We now analyze how many patterns we can superimpose.

We prove in the Appendix that the probability of a given text position matching a random pattern is exponentially decreasing with m for $\alpha < 1 - e/\sqrt{\sigma}$, while if this condition does not hold the probability is very high.

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 $1 - (1 - 1/\sigma)^r$. Hence, the new limit for α is

$$\alpha < 1 - e\sqrt{1 - \left(1 - \frac{1}{\sigma}\right)^r} \approx 1 - e\sqrt{\frac{r}{\sigma}}, \tag{10}$$

where the (pessimistic) approximation is tight for $r \ll \sigma$.

If we use pattern partitioning, we must search j subpatterns (j is determined by m and k, it is not dependent on r). Using (9) again, we have that the number of verifications is negligible for

$$\alpha < 1 - e\sqrt{\frac{r}{\sigma}}\, m^{j/(m-k)} \tag{11}$$

where solving for r we get that the maximum allowed amount of superimposition r' is

$$r' = \frac{(1-\alpha)^2\sigma}{e^2 m^{2d(w,\alpha)}}$$

and therefore we must partition the set into $j' = j/r'$ subsets of size r' each. That implies that the cost of our algorithm is in general $O(j'n)$. Replacing $j = (m-k)\,d(w,\alpha)$ yields that the cost is

$$\frac{e^2\, d(w,\alpha) m^{1+2d(w,\alpha)}}{\sigma(1-\alpha)}\, n$$

and since the technique is limited to the case $\alpha < 1 - e/\sqrt{\sigma}$, we have a complexity of

$$O\left(\frac{d(w,\alpha)m^{1+2d(w,\alpha)}}{\sqrt{\sigma}}\, n\right) \approx O\left(\sqrt{\frac{km}{\sigma w}}\, m^{2/\sqrt{w}}\, n\right), \tag{12}$$

where the approximation is valid for moderate m values.

A natural question is for which error level can we superimpose all the j patterns to obtain a linear algorithm, i.e., where $r' \geq j$ holds. That is (using (11)),

$$\alpha < 1 - e\sqrt{\frac{j}{\sigma}}\, m^{j/(m-k)},$$

where the limit point is defined as

$$\alpha_0 = 1 - e\sqrt{\frac{m(1-\alpha_0)\, d(w,\alpha_0)}{\sigma}} m^{d(w,\alpha_0)}, \tag{13}$$

which is not easy to solve analytically. For moderate m a practical simplified solution is $\alpha_0 = 1 - \sqrt{m/(\sigma\sqrt{w})}\, m^{1/\sqrt{w}}$.

Observe that for $\alpha > 1 - e/\sqrt{\sigma}$, the cost is $O(jn)$, i.e., no better than pattern partitioning with no superimposition. However, that value also marks the limit of the usability of pattern partitioning.

It is interesting to compare this limit for the linearity of our algorithm with that of exact partitioning [8] (i.e., partitioning the pattern into $k+1$ pieces and searching them with no errors, verifying each subpattern occurrence for a complete match).

To analyze exact partitioning, we assume the use of an Aho–Corasick machine [1] to search the subpatterns in parallel in $O(n)$ guaranteed search time (as explained, there are better alternatives in practice). Since the search for the subpatterns is of linear time, we find out now when the total amount of work due to verifications is also linear.

We split the pattern into pieces of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$. In terms of probability of occurrence, the shorter pieces are σ times more probable than the others (where σ is the size of the alphabet). Since each occurrence of a subpattern is verified for a complete match at $O(m^2)$ cost, the total cost of verifications is no more than $(k+1)m^2/\sigma^{\lfloor m/(k+1) \rfloor} n$. This is linear approximately for $\alpha < \alpha'_0$, where α'_0 is defined as

$$\alpha'_0 = \frac{\sigma^{1/\alpha'_0}}{m^3}, \tag{14}$$

where a reasonable approximation is $\alpha'_0 = 1/(3\log_\sigma m)$. This limit is less restricting than our α_0 of (13) for sufficiently large m (a very rough approximation is $m > \sigma\sqrt{w}/2$).

For typical text searching the error level tolerated by our algorithm is higher (i.e., $\alpha_0 > \alpha'_0$). As we show in the experiments, however, exact partitioning is faster in practice, and therefore it should be preferred whenever $\alpha < \alpha'_0$.

5.6. *Mixed Partitioning.* We now analyze the general partitioning strategy: partition the pattern in a number of subpatterns, and then partition each automaton.

To obtain the optimal strategy, consider that if we partition in j subpatterns, we must perform j searches with $\lfloor k/j \rfloor$ errors. For $\alpha < 1 - e/\sqrt{\sigma}$, the cost of solving j subproblems by partitioning the automaton is (recall (7))

$$\frac{((ke/j)/(\sqrt{\sigma} - e))\,(k/j+1)}{w-1}\, jn, \tag{15}$$

which shows that the lowest cost is obtained with the largest j value, and therefore mixed partitioning should not be used if pure pattern partitioning is possible.

However, there is still place for this algorithm. The point is that the limit of usefulness for α is reduced when j grows (see (9)). Hence, for $\alpha > \alpha_1$ we may use a smaller j (and partition the automata) to keep the verifications negligible. We use the first part of (9) to obtain the maximum j value allowed by α, which is $\lfloor (m-k)\log_m(\sqrt{\sigma}(1-\alpha)/e) \rfloor$. By counting the cost of carrying out j searches with the resulting subautomata (using (15)), we have the complexity of this scheme.

It is possible to use superimposition with this scheme. We cannot, however, first select j and then superimpose subsets, because if we can superimpose two patterns, it means that the error level would allow us to use a larger j (and therefore our selection was not optimal). We try instead to optimize j and r simultaneously. Suppose

we superimpose groups of r subpatterns. Then, (11) must hold, from where we obtain $j = (m-k)\log_m(\sqrt{\sigma/r}(1-\alpha)/e)$. The cost is $1/r$ times that of pure automaton partitioning, which reaches its worst case because we are pushing j to its limit. Therefore, we optimize

$$\frac{(k/j)((m-k)/j)}{w-1}\,\frac{j}{r}\,n = \frac{k\log m}{w-1}\,\frac{1}{r\log(\sqrt{\sigma/r}(1-\alpha)/e)}\,n.$$

We minimize the above formula to find that the optimum r value is $\sigma(1-\alpha)^2/e^3$, which corresponds to $j = (m-k)/(2\ln m)$. Therefore, our optimal search cost is

$$O\left(\frac{e^3}{\sigma(1-\alpha)^2}\,\frac{k\,2\ln m}{m-k}\,\frac{k}{w\sqrt{\sigma}}\,n\right) = O\left(\frac{k\log m}{w}\,n\right). \tag{16}$$

Notice that when the maximum j allowed is 1, we have pure automaton partitioning. This happens for

$$\alpha_2 = 1 - \frac{e}{\sqrt{\sigma}}\,m^{1/m(1-\alpha_2)} \tag{17}$$

(observe that $\alpha_2 \longrightarrow 1 - e/\sqrt{\sigma}$ as m grows). Therefore, we have a smooth transition from pattern partitioning to automaton partitioning.

6. Experimental Results. In this section we experimentally compare the different variations of our algorithm, as well as the fastest previous algorithms we are aware of.

We tested random patterns against 1 Mbytes of random text on a Sun SparcStation 4 running Solaris 2.3, with 32 Mbytes of RAM.[4] This is a 32-bit machine, i.e., $w = 32$. We use $\sigma = 32$ (typical case in text searching). We also tested lowercase English text, selecting the patterns randomly from the same text, at the beginning of words of length at least 4, to mimic classical information retrieval queries. Each data point was obtained by averaging the Unix's user time over 20 trials.

The reader may be curious about the strange behavior of some of the curves, especially in our algorithms. Some curves are definitely nonmonotonical in k (besides the expected $k(m-k)$ behavior in automaton partitioning), and this does not depend on statistical deviations in the tests. Those "peaks" are due to integer round-offs, which are intrinsic to our algorithms and are more noticeable on small patterns. For instance, if we had to use pattern partitioning to split a search with $m = 30$ and $k = 17$, we would need to search four subpatterns, while for $k = 18$ we need just three. As another example, consider automaton partitioning for $m = 20$ and $k = 13$, 14, and 15. The number of cells to work on (IJ) change from four to three and then to five.

6.1. *Superimposition and Mixed Partitioning.* We show the effect of superimposition. In practice it is rare to be able to superimpose more than three patterns. Figure 8(a)

[4] Previous tests [5] run on a Sun SparcClassic with 16 Mbytes of RAM running SunOS 4.1.3 gave slightly worse results for the algorithm of [35]. This is probably due to the amount of main memory available, which some algorithms depend on.

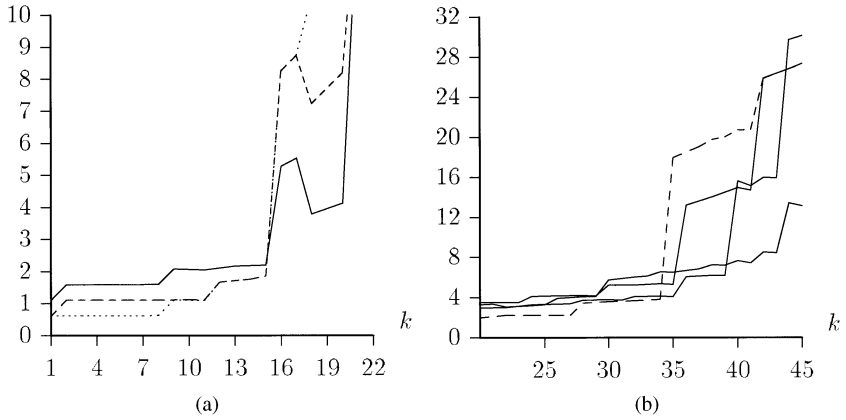

Fig. 8. Times in seconds for some combined heuristics. (a) Pattern partitioning with forced superimposition: $r = 1$ (solid line), $r = 2$ (dashed line), and $r = 3$ (dotted line). (b) Pattern partitioning with forced mixing: $j = 2, 4, 6$ (solid lines, the larger j jumps first) and maximal j (dashed line).

shows the times for pattern partitioning on random text and $m = 30$. The level of superimposition is forced at 1, 2, and 3. This shows that superimposition is indeed effective, and that we must reduce the level of superimposition as the error level increases.

We also show, in Figure 8(b), the effect of mixed partitioning. We use $m = 60$ (since the area is very narrow for shorter patterns) and random text. We force some different j values, and also show the maximal j as used by pure pattern partitioning. The figure shows that it is better to use larger j, but it is necessary to switch to smaller ones when the error level increases.

The peaks in Figure 8(a) correspond to the example of integer round-offs for pattern partitioning given before.

6.2. *A Comparison of Parallel-NFA Algorithms.* Our algorithm shares the NFA model and the idea of using bit-parallelism with Wu and Manber work [34]. However, the parallelization techniques are different. We compare both algorithms.

A general implementation of the Wu and Manber code needs to use an array of size $k + 1$. However, we implemented optimized versions for $k = 1$, 2, and 3. That is, a different code was developed for each k value, in order to avoid the use of arrays and enable the use of machine registers. We show both algorithms (optimized and general). We also show the effect of our simple speedup heuristic (the S table), running our algorithm with and without that filtration heuristic.

Figure 9 shows the results. We show the case $m = 9$ (where we can use the simple algorithm) and $m = 30$ (where we use automaton partitioning). Above $m = 32$ the times for the Wu and Manber algorithm should at least double since they need to use two machine words per row of the automaton.

As can be seen, our algorithms without the S heuristic outperform even the optimized versions of Wu and Manber in all the spectrum for short patterns, except for $k = 1$. For longer patterns, we outperform the general version of Wu and Manber, although their

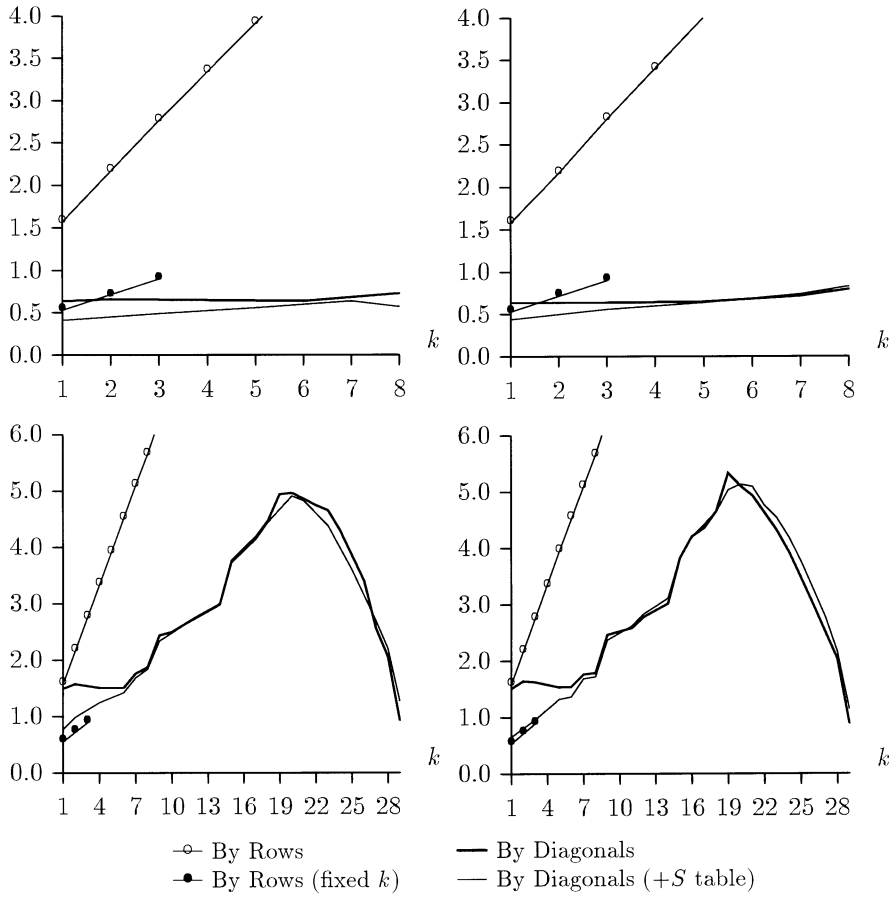

Fig. 9. Times in seconds for $m = 9$ (first row) and $m = 30$ (second row). The plots on the left are for random text ($\sigma = 32$), those on the right are for English text.

specialized versions for fixed k are faster, and would probably remain slightly faster for much larger k values (this also depends on the availability of machine registers, though). Of course, it is not practical to implement a different algorithm for every possible k value.

On the other hand, the use of the S table cuts down the running times of our algorithm from 40% to 65% if the error level is reasonably low (of course this heuristic can be applied to any other algorithm too).

6.3. *A Comparison of Hybrid Heuristics.* We developed a hybrid heuristic that automatically determines the best algorithm from our suite. The heuristic uses our analytical results to determine the best choice. This works very well on random text, although it still needs some tuning on English text. The fact that the best choice is always selected is an independent confirmation of the analytical results for random text.

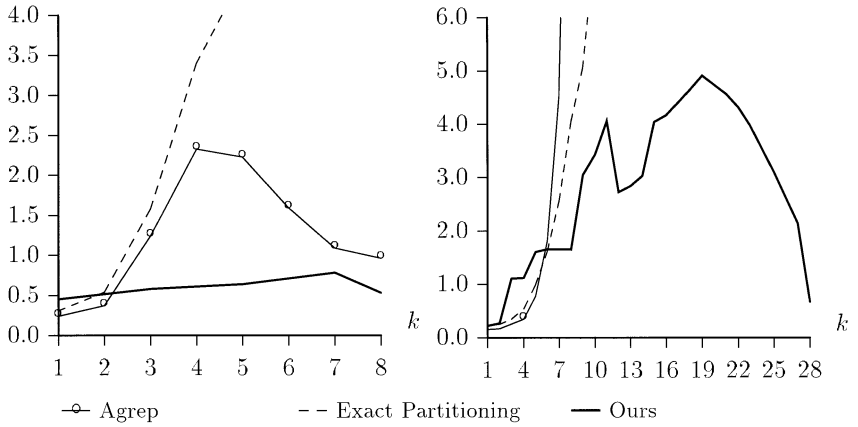

Fig. 10. Times in seconds for $m = 9$ (left) and $m = 29$ (right), on English text.

In this section we compare our complete heuristic against another hybrid algorithm, namely, Agrep [33]. Agrep is a widely distributed approximate string search software. In this case we include the filtering S table, since Agrep uses other speedup techniques altogether.

We also include our implementation of Baeza-Yates and Perleberg's algorithm [8], which we call "exact partitioning." This is because, whenever we determine that partitioning in $j = k + 1$ pieces is necessary, we fall naturally into this algorithm (although not developed as part of this work). In this sense, the algorithm could be considered as part of the heuristic we propose.

Figure 10 shows the results for $m = 9$ and $m = 29$ (maximum currently allowed in Agrep, as well as $k \leq 8$). As can be seen, exact partitioning is faster for low error ratios (roughly $\alpha \leq 0.2$). However, Agrep is especially optimized for small k values, being the fastest in this case. In the rest of the spectrum our algorithm is faster.

The reason why Agrep times drop past some point is that as soon as it finds a match in a line of text, it reports the line and stops searching in that line. Therefore, it skips lines faster for very high error ratios. The peak of our hybrid algorithm for $m = 30$ and English text, on the other hand, is due to imperfect tuning of our heuristic for nonrandom text.

6.4. *A Comparison of the Fastest Algorithms.* Finally, we present a comparison between our algorithms and the fastest previous algorithms we are aware of. Since we compare only the fastest algorithms, we leave aside [22], [28], [13], [16], [26] and [32], which were not competitive in the range of parameters we study here. We use our simple algorithm, as well as pure pattern and pure automaton partitioning. We do not make use of the S table speedup, since it could be applied to all other algorithms as well.

The algorithms included in this comparison are (in numerical order of Reference):

Pattern and Automaton Partitioning are our algorithms. For $m = 9$ we use the simple automaton, which is considered as "automaton partitioning" in the figures. Pattern partitioning includes superimposition.

Exact Partitioning [8] is essentially the case $j = k+1$. The code is ours. The algorithm was presented by Wu and Manber in [34], but the Boyer–Moore-like search we use is suggested in [8].

Column Partitioning [9] 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 by Chang and Lampe [9].

Counting [14] moves a window over the text, keeping how many letters in the window match the pattern. When the number is high enough, the area is verified. We use the variant implemented in [19] (window of fixed size m).

DFA [20] converts the NFA to a deterministic automaton which is computed in lazy form, i.e., a state is created only when it is first needed while processing the text. This idea was also presented in [15], although we use the implementation of [20].

q-grams [25] is a filtration algorithm based on finding portions (q-grams) of the pattern in the text, and verifying candidate areas. The method is limited to $\alpha < \frac{1}{2}$, and the implementation to $k \leq w/2-3$. The code is by Sutinen and Tarhio [25]. We use $s = 2$ (number of samples to match) and maximal q (length of the q-grams), as suggested in [25].

Cut-Off [29] is the standard dynamic programming algorithm, working only on active columns. The code is ours.

NFA by Rows [34] uses bit-parallelism to simulate the automaton by rows. The code is ours. Our code is limited to $m \leq 31$, and it would be slower if generalized. This is a general code for arbitrary k.

Four Russians [35] applies a Four Russians technique to pack many automaton transitions in computer words. The code is by Wu et al. [35], and is used with $r = 5$ as suggested in [35] (r is related with the size of the Four Russians tables).

Figure 11 shows the results. As can be seen, our algorithm is more efficient than any other when the problem fits in a single word, except for low error level, where Baeza-Yates and Perleberg's algorithm is unbeaten. For $m = 20$ and 30, our algorithms are not the fastest but are quite close to them. In particular, automaton partitioning is the fastest algorithm when k is close to m.

We show more detail for the case of small k for $m = 20$ and 30 in Figure 12.

The peaks of our pattern partitioning for $m = 30$ were explained at the beginning of this section. The same holds for automaton partitioning for $m = 20$ and $k = 14$. The reason for the peak of the same curve at $k = 2$ and 3 is more obscure, since the number of cells do not change, and, moreover, we have that for $k = 4$ the second cell is active more frequently (as expected from the analysis). What happens is that for $k = 2$ and 3 the second cell switches from inactive to active and back to inactive quite often (30% more times than for $k = 4$). The overhead to include and exclude the second cell explains the higher times.

On the other hand, our analysis is confirmed. On random text for $m = 20$ we predict (see (9) replacing e by 1.09) that pattern partitioning will be useful up to $\alpha_1 = 0.595$, which is $k = 11$, quite close to the real value $k = 13$. For $m = 30$ we predict $\alpha_1 = 0.569$ which is $k = 17$, close to the real $k = 20$. Such precision allows us to set up very fine-tuned heuristics. For English texts, however, that prediction is harder.

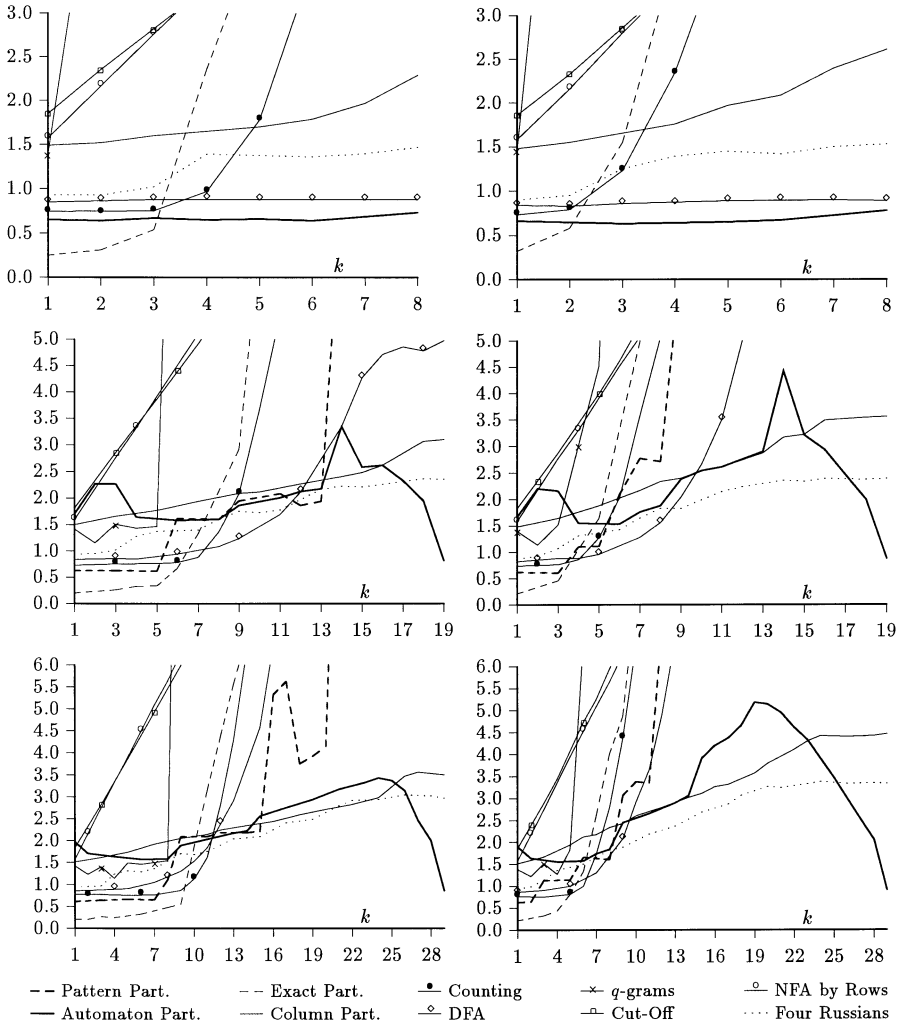

Fig. 11. Times in seconds for $m = 9$, 20, and 30 (first, second, and third row, respectively). The plots on the left are for random text ($\sigma = 32$), and those on the right are for English text.

7. Concluding Remarks. We presented a new algorithm for approximate pattern matching, based on the bit-parallel simulation of an automaton. We eliminate the dependencies introduced by the ε-transitions. This enables the possibility of computing the new values in $O(1)$ time per text character, provided the problem fits into 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 subautomata. We can handle "extended patterns" (which allow classes of characters, gaps, and other generalizations in the patterns). We show another technique to partition the pattern into subpatterns that are grouped into superimposed searches. Those searches are carried out with the simple algorithm and the candidate

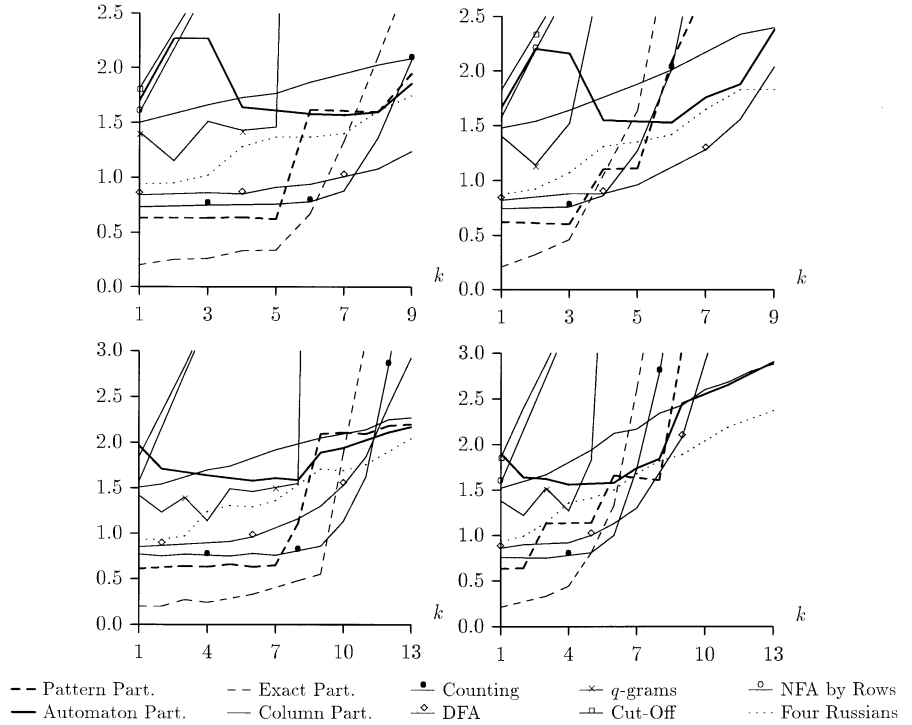

Fig. 12. Times in seconds for $m = 20$ and 30 (first and second row, respectively), and for small k. The plots on the left are for random text ($\sigma = 32$), and those on the right are for English text.

matches verified later. Finally, we show how to combine pattern and automaton partitioning. The combined algorithm is $O(n)$ for small patterns or moderate α, roughly $O(\sqrt{km/(\sigma \log n)}\, n)$ for moderately high 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 among the fastest for typical text searching. We have not studied other cases, such as very long patterns (where our algorithm does not perform well and filtration algorithms tend to be better) or small alphabets (where the algorithms of [32] and [35] are normally the best choices [5]).

Figure 13 illustrates the results on English text, showing in which case each algorithm should be applied for patterns of moderate size. We do not include Agrep because it is not a "pure" algorithm. Should Agrep be included, its area would replace those of Exact Partitioning and Counting. As can be seen, filtering algorithms are the best for low error ratios. On the other hand, different implementations of the automaton model (either deterministic or not) are the fastest choices for not very long patterns. In the remaining area, the Four Russians approach is the best choice (this can be seen as another version of a DFA). In particular, our algorithms are the fastest for short patterns (and a moderate error level) or a very high error level.

Although in this work we deal with finite alphabets, we can easily extend our

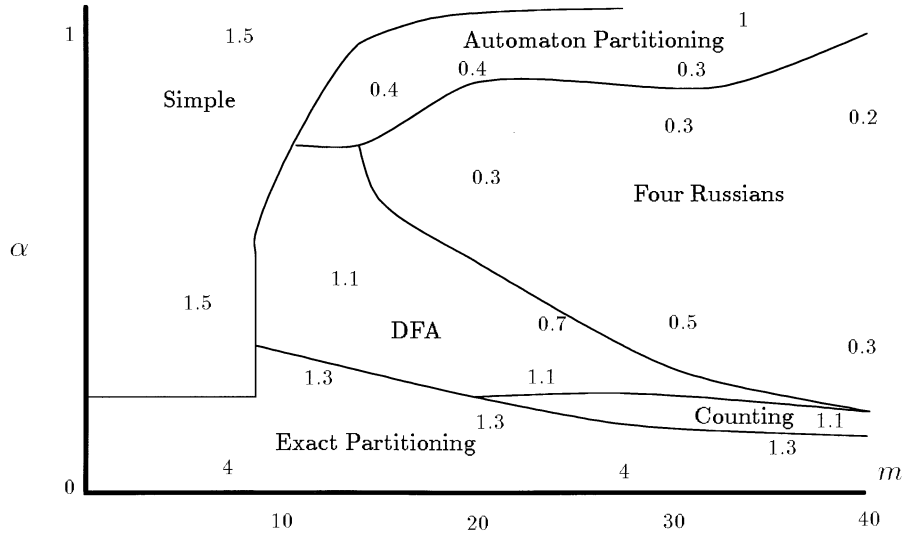

Fig. 13. The areas where each algorithm is the best for English text and $w = 32$. The numbers indicate megabytes per second on our machine.

algorithms for the unbounded 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.

Future work involves a detailed experimental study of how to combine the many heuristics presented here. This study can be guided by the theoretical analysis but must take into account practical complications such as round-offs, which are important in practice. For instance, if the text is not random the subpatterns in pattern partitioning could have different lengths so that their probabilities of occurrence are similar. Also, we could prefer to prune the pattern instead of splitting it, avoiding performing more searches if the probability of false matches is not too high. Other areas to pursue are optimal automaton partitioning, optimal amount of superimposition, and optimal use of mixed partitioning.

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, that made the tests a lot easier and, in some sense, more fair: William Chang, Alden Wright, Gene Myers, Erkki Sutinen, and Tadao Takaoka. Finally, we thank the anonymous referees for many suggestions to improve this paper.

Appendix. Upper Bound for $f(m, k)$. Let $f(m, k)$ be the probability that a pattern matches a given text position with at most k errors (i.e., that the text position is reported as the end of a match). We find an upper bound on the error level to make that probability $O(\gamma^m)$ for some $\gamma < 1$ (see (19)). If our only aim is to make that probability exponentially

small with m, we take the bound for $\gamma = 1$ and consider valid any error level *strictly* smaller than the bound. This is $\alpha < 1 - e/\sqrt{\sigma}$.

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 $m - k \cdots m + k$. 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 any in the pattern and in the text, we have

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

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 into two parts because if the area has length $s > m$, then more than $m - k$ characters must match, namely, $s - k$. See Figure 14.

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

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

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

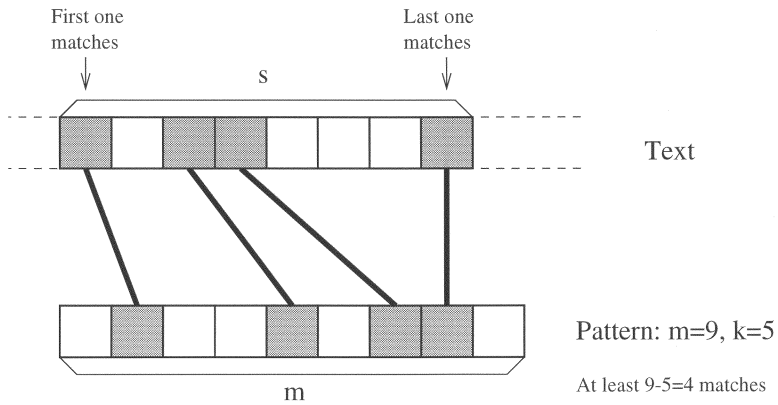

Fig. 14. Upper bound for $f(m,k)$.

Clearly, for the summation to be $O(\gamma^m)$ ($\gamma < 1$), this largest 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 (recall that our exponential is multiplied by m^{-1} and therefore we can safely multiply it by $k+1$). That is,

$$\sigma \geq \left(\frac{1}{\gamma\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)}}\right)^{1/(1-\alpha)} = \frac{1}{\gamma^{1/(1-\alpha)}\alpha^{2\alpha/(1-\alpha)}(1-\alpha)^2}. \tag{18}$$

It is easy to show analytically that $e^{-1} \leq \alpha^{\alpha/(1-\alpha)} \leq 1$ 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,

$$\alpha \leq 1 - \frac{e}{\sqrt{\sigma}\gamma^{1/(2(1-\alpha))}} \tag{19}$$

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

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

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

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

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

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

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

Elementary calculus leads to solving a second-degree equation that has roots in the interval $[1-\alpha, 1]$ only if $\sigma \leq \alpha/(1-\alpha)^2$. Since due to (19) we are only interested in $\sigma \geq 1/(1-\alpha)^2$, $h'(x)$ 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 largest term of the first summation.

We conclude that

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

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

References

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6): 333–340, June 1975.

[2] R. Baeza-Yates. Text retrieval: theory and practice. In J. van Leeuwen, editor, *Proc. 12th IFIP World Computer Congress*, volume I: Algorithms, Software, Architecture, pages 465–476. Elsevier Science, Amsterdam, September 1992.

[3] R. Baeza-Yates. A unified view of string matching algorithms. In K. Jeffery, J. Král, and M. Bartosek, editors, *SOFSEM '96: Theory and Practice of Informatics*, Milovy, Czech Republic, November 1996, LNCS 1175, pages 1–15. Springer-Verlag, Berlin, 1996. (Invited paper.)

[4] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10): 74–82, October 1992.

[5] R. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proc. WSP '96*, pages 47–63. Carleton University Press, Ottawa, 1996. `ftp//ftp.dcc.uchile.cl/pub/users/gnavarro/wsp96.2.ps.gz.`

[6] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. Hirschberg and G. Myers, editors, *Proc. CPM '96*, LNCS 1075, pages 1–23. Springer-Verlag, Berlin, 1996. `ftp//-ftp.dcc.uchile.cl/pub/users/gnavarro/cpm96.ps.gz.`

[7] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS '97*, LNCS 1272, pages 174–184. Springer-Verlag, Berlin, 1997. `ftp//ftp.dcc.uchile.cl/pub/users/-gnavarro/wads97.ps.gz.`

[8] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996.

[9] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM '92*, LNCS 644, pages 172–181. Springer-Verlag, Berlin, 1992.

[10] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, Oct./Nov. 1994.

[11] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM '94*, LNCS 807, pages 259–273. Springer-Verlag, Berlin, 1994.

[12] N. El-Mabrouk and F. Lisacek. Very fast identification of RNA motifs in genomic DNA. Application to tRNA search in yeast genome. *Journal of Molecular Biology*, 264, November 1996.

[13] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal of Computing*, 19(6):989–999, 1990.

[14] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

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

[16] G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer Systems Science*, 37:63–78, 1988.

[17] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[18] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct./Nov. 1994.

[19] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP '97*, pages 125–139. Carleton University Press, Ottawa, 1997. `ftp//ftp.dcc.uchile.cl/pub/users/gnavarro/wsp97.1.ps.gz.`

[20] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. WSP '97*, pages 112–124. Carleton University Press, Ottawa, 1997. `ftp//ftp.dcc.uchile.cl/pub/users/gnavarro/wsp97.2.ps.gz.`

[21] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[22] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[23] F. Shi. Fast approximate string matching with q-blocks sequences. In N. Ziviani, R. Baeza-Yates, and K. Guimarãs, editors, *Proc. WSP '96*, pages 257–271. Carleton University Press, Ottawa, 1996.

[24] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.
[25] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In P. Spirakis, editor, *Proc. ESA* '95, LNCS 979, pages 327–340. Springer-Verlag, Berlin, 1995.
[26] T. Takaoka. Approximate pattern matching with samples. In *Proc ISAAC* '94, LNCS 834, pages 234–242. Springer-Verlag, Berlin, 1994.
[27] J. Tarhio and E. Ukkonen. Boyer–Moore approach to approximate string matching. In J. Gilbert and R. Karlsson, editors, *SWAT* '90, LNCS 447, pages 348–359. Springer-Verlag, Berlin, 1990.
[28] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
[29] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
[30] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.
[31] B. Watson. The performance of single and multiple keyword pattern matching algorithms. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proc. WSP* '96, pages 280–294. Carleton University Press, Ottawa, 1996.
[32] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.
[33] S. Wu and U. Manber. Agrep—a fast approximate pattern-matching tool. In *Proc USENIX Technical Conference*, pages 153–162, 1992.
[34] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.
[35] S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.