A Fast Heuristic for Approximate String Matching[2]

Ricardo Baeza-Yates[1] Gonzalo Navarro[1]

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

Abstract. We study a fast algorithm for on-line approximate string matching. It is based on a non-deterministic finite automaton, which is simulated using bit-parallelism. If the automaton does not fit in a computer word, we partition the problem into subproblems. We show experimentally that this algorithm is the fastest for typical text search. We also show which algorithms are the best in other cases, and derive the fastest known heuristic for on-line approximate string matching, when the pattern is not very large. The focus of this work is mainly practical.

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.

The problem can be formally 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. It is common to report only minimal or maximal occurrences, as well as not the matching segments but only their end point.

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 $k < m$. We call $\alpha = k/m$ the "error ratio", and $\sigma = |\Sigma|$ 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 work we focus on on-line algorithms, where the classical solution, involving dynamic programming, is $O(mn)$ time [11, 12].

In the last years several algorithms have been presented that achieve $O(kn)$ comparisons in the worst-case [18, 8, 9, 10] or in the average case [19, 8], by taking advantage of the properties of the dynamic programming matrix. In the same trend is [5], with average time complexity $O(kn/\sqrt{\sigma})$.

Other approaches attempt to filter the text, reducing the area in which dynamic programming needs to be used [16, 17, 15, 14, 6, 7]. These algorithms

achieve sublinear expected time in many cases ($O(kn \log_\sigma m/m)$ is a typical figure) for moderate α, but the filtration is not effective for larger ratios. Moreover, they are not practical if m is not very large. An exception is a simple and fast filtering technique shown in [4], which yields an $O(n)$ algorithm for moderate α.

Yet other approaches use bit-parallelism [1, 23] in a RAM machine of word length $O(\log n)$ to reduce the number of operations. [22] achieves $O(kmn/\log n)$ time, which is competitive for patterns of length $O(\log n)$. [20] packs the cells differently to achieve $O(mn \log \sigma/\log n)$ time complexity. [24] uses a Four Russians approach and packs the table in machine words, achieving $O(kn/\log n)$ time on average.

In [3] we propose a new algorithm, which combines the ideas of taking advantage of the properties of the matrix, filtering the text and using bit-parallelism. In this work, we experimentally study this algorithm, and find the fastest known heuristic for on-line approximate string matching. This heuristic involves the use of [3] in the case of moderate pattern sizes, moderate error ratios, and not very small alphabet sizes. This is the normal case in text searching.

This paper is organized as follows. In section 2 we explain our algorithm. In section 3 we fine-tune it. In section 4 we compare it with others and derive the optimal heuristic. In section 5 we give our conclusions.

2 Our Algorithm

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 (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.

The main comparison-based algorithms for approximate string matching consist fundamentally in simulating this automaton by columns, while some bit-parallelism approaches simulate it by rows [2]. In all cases, the dependencies introduced by the diagonal empty transitions prevent the parallel computation of the new values. In [2, 3] we show how to avoid this dependence, by simulating the automaton using diagonals, such that each diagonal captures the ϵ-closure.

We arrange the states of the automaton diagonal-wise, and pack those diagonals in a computer word. We find a constant time formula to update the matrix

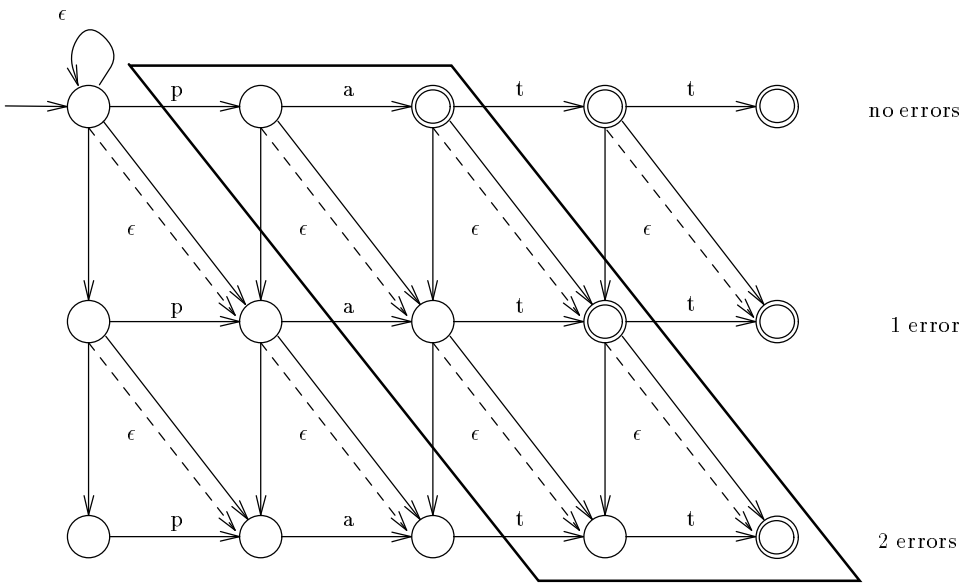

Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character.

according to each new character read, what leads to a linear time, very fast algorithm. Although this algorithm is $O(n)$ regardless of m and k, it is limited to the case $(m-k)(k+2) \leq w$, where w is the size in bits of the computer word. Figure 1 boxes the area of the automaton that is represented.

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 because every occurrence must include at least one of the $k+1$ initial characters. We use a boolean table S to store, for each character, whether it is one of the first $k+1$ of the pattern.

If the automaton is larger, we show how to partition it into smaller sub-automata, which are similar to the simple one and fit in a computer word.

We also show that it is possible to solve large problems is to partition the pattern in j subpatterns, and search each of them with $\lfloor k/j \rfloor$ errors, since for each occurrence of the complete pattern, at least one of the subpatterns must occur. The smaller patterns can be searched with the above automata. Later, we have to verify each sub-match to check whether it involves a complete match. This leads to a very fast algorithm provided the number of verifications is not too large.

Moreover, we can combine both techniques. We return to this later.

3 Tuning the Algorithm

The application of the different partition techniques require careful analysis and experimentation. In [3] we perform the theoretical analysis, while in this work we carry out a number of experiments to determine which is the best heuristic in practice. We keep our results independent of the machine architecture.

3.1 Probability of Matching

Let $f(m,k)$ be the probability of a pattern of size m matching a given text position with k errors. This probability is determinant to assure that the number of verifications of candidate matches in pattern partitioning is not too large. In [3] we find that for

$$\sigma > \frac{1}{\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2} \tag{1}$$

this probability is exponentially decreasing with m. An upper bound for the above expression is obtained, which is

$$\alpha < 1 - \frac{e}{\sqrt{\sigma}} \tag{2}$$

On the other hand, we can prove that the probability is not decreasing on m for $\alpha > 1 - 1/\sigma$.

Here, we experimentally find the maximum value of α for which the verifications are negligible. We tested different values of m and σ, and found that the results, as theoretically expected, do not depend on m.

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 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 to verify 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 significative values for m in the hundreds.

Figure 2 shows the results. The curve $\alpha = 1 - 1/\sqrt{\sigma}$ is included to show its closeness to the experimental data. We use a least squares technique to find that e must be replaced by 1.09. The squared error is smaller than 10^{-4}.

This value is also useful to compute the expected number of active columns in the heuristic of [19], which we also need later. We prove in [3] that the number of active columns is upper bounded by

$$\frac{k}{1 - e/\sqrt{\sigma}} + O(1)$$

where the e has the same source as above, and therefore can be replaced by 1.09. We made experiments with different alphabet sizes, and found that a very good

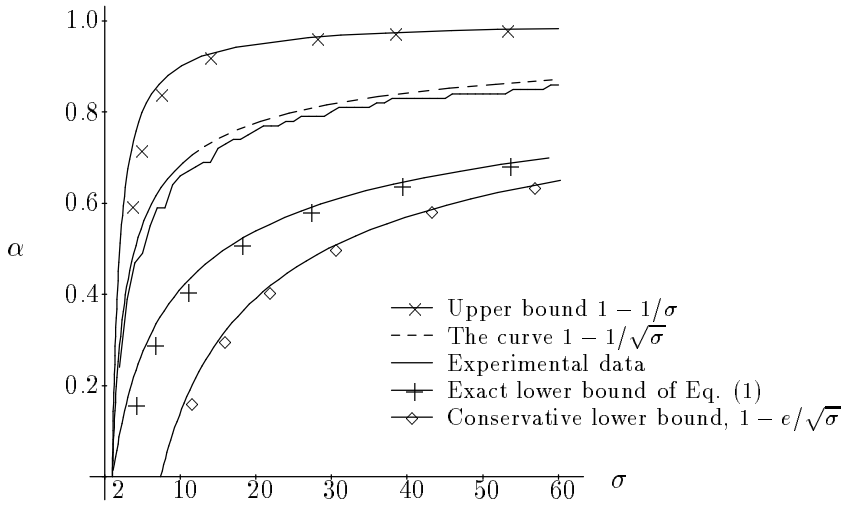

Fig. 2. Theoretical and practical bounds for α.

approximation to the exact number is

$$0.9\frac{k}{1-1.09/\sqrt{\sigma}} \tag{3}$$

with a squared error less than 0.03.

Figure 3 (left side) shows the last active column for random patterns of length 30 on random text, for different values of σ. Given the strong linearity, we take a fixed $k = 5$ and use least squares to find the slope of the curves. From that we obtain the 0.9 above. The right side of the figure shows the experimental data and the fitted curve. Our results are the same for any k less than $m(1-1.09/\sqrt{\sigma})$.

3.2 Automaton Partitioning

As explained earlier, we can partition a large automaton that does not fit in a computer word in a number of smaller sub-automata that do.

The automaton is partitioned into a matrix of I rows and J columns, each cell being a small sub-automaton, that stores ℓ_r rows of ℓ_c diagonals of the complete automaton. Because of the nature of the update formula, we need to store $(\ell_r + 1)\ell_c$ bits for each sub-automaton. Thus, the conditions to meet are

$$(\ell_r + 1)\ell_c \leq w \qquad I\ell_r \geq k + 1 \qquad J\ell_c \geq m - k \qquad IJ \text{ minimal}$$

Although there are many possible options for I and J, we show in [3] that the final cost is $O(IJn) = O(k(m-k)n/w)$, regardless of the selection. However, this does not account for the effect of round-offs (caused by not fully occupied cells), that are noticeable in practice.

We use a technique similar to [19] to compute only active diagonals. Given the results of Eqs. (2) and (3), we have now that for $\alpha < 1 - 1.09/\sqrt{\sigma}$, only

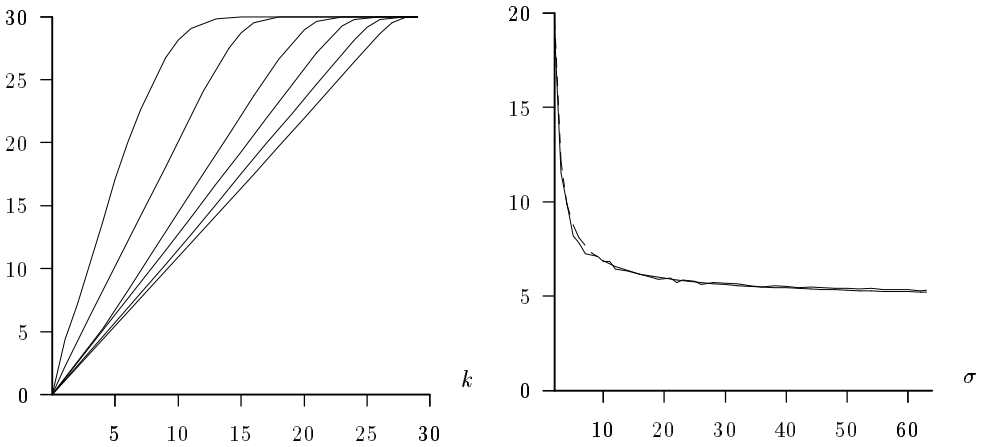

Fig. 3. On the left, last active column for $\sigma = 2$, 4, 8, 16, 32 and 64 (curves taken from left to right). On the right, last column for $k = 5$, experimental (full line) and theoretical (dashed line).

$0.9k/(1 - 1.09/\sqrt{\sigma})$ columns are active, while for larger α the worst case dominates. Since we work on diagonals instead of columns, and since we pack ℓ_c diagonals in a single computer word, we work on average on

$$I\left(0.5 + \frac{0.9\frac{k}{1-1.09/\sqrt{\sigma}} - k + 1}{\ell_c}\right) \tag{4}$$

cells. For larger α, we work on IJ cells.

We now focus on the problem of determining an optimal selection for I and J. One could, in fact, try every I and J and pick the configuration with less cells. However, it is shown in [3] that by selecting minimal I, the possible automata are: (a) horizontal $(I = 1)$, (b) horizontal and with only one diagonal per cell $(I = 1, \ell_c = 1)$, or (c) not horizontal nor vertical, and with only one diagonal per cell $(I > 1, J > 1, \ell_c = 1)$. Those cases can be solved with a simpler update formula (2 or 3 times faster than the general one). The special case $k = m - 1$ is solved with the S table alone (each hit ends an occurrence).

This much faster update formula is more important than the possible gains due to round-offs, since they cannot force a cell sub-occupancy smaller than 1/2 (this is because in that case we could pack two cells in one). Therefore, the gain with a better arrangement cannot be higher than a factor of 2, less than the gain for the better update formulas. Hence, we prefer to take minimal I, i.e. $I = \lceil (k+1)/(w-1) \rceil$, $\ell_r = \lceil (k+1)/I \rceil$, $\ell_c = \lfloor w/(\ell_r + 1) \rfloor$ and $J = \lceil (m-k)/\ell_c \rceil$.

However, it has not been studied whether a purely vertical partitioning could be advisable. We compare this with (c), and leave aside cases (a) and (b), since their update formulas are already faster than that of the vertical automaton, and they are not likely to need updating all the cells. Even (c) is faster than

vertical partitioning when not all the cells are updated, i.e. for $\alpha < 1 - 1.09/\sqrt{\sigma}$. We have experimentally verified this fact.

The selection for vertical partitioning (only applicable if $2(m-k) \leq w$) is $J_v = 1$, $\ell_{vc} = m - k$, $\ell_{vr} = \lfloor w/(m-k) \rfloor - 1$, $I_v = \lceil (k+1)/\ell_{vr} \rceil$.

We compare the number of cells of each technique multiplied by the number of operations carried out per cell (observe that, since this comparison only holds for large α, all cells are normally updated). That is, we select the vertical automaton whenever $2(m-k) \leq w$, $\alpha \geq 1 - 1.09/\sqrt{\sigma}$, and $I_v \leq 1.21 \times IJ$. Figure 4 (right plot) shows an example fitting our predictions.

3.3 Pattern Partitioning

As mentioned earlier, we can also partition the pattern into j sub-patterns, so that each one can be searched with a simple automaton. Each match is a candidate for a complete match. Hence, it has to be verified, at a cost of $O(m^2)$ time. This technique works if there are not too much verifications.

We analyze in [3] when this happens, and prove that it is safe to use this idea for

$$\alpha \leq 1 - \frac{e}{\sqrt{\sigma}} m^{\frac{j}{m-k}} \tag{5}$$

and by taking j large enough for the sub-patterns automata to fit in a computer word, we obtain an $O(\sqrt{mk/w}\ n)$ algorithm for $\alpha \leq \alpha_1$, where

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

although, as we found here, the e has to be replaced by 1.09 in practice.

What we only briefly mention in [3] is that it is useful to use a smaller j for $\alpha > \alpha_1$, since a smaller j allows to use larger α while keeping the verifications negligible. In this case, the smaller automata are still too large to fit in a computer word, and we have to use automaton partitioning.

We call this strategy "mixed partitioning": for $\alpha > \alpha_1$, we select the largest j that assures few verifications. We derive that j from Eq. (5)

$$j = \lfloor (m-k) \log_m(\sqrt{\sigma}(1-\alpha)/1.09) \rfloor = O(m/\log_\sigma m)$$

and by counting the cost of carrying out j searches with the resulting sub-automata, the complexity of this scheme is $O(kn \log_c(m)/w)$. This can be used up to where $j = 1$, which implies pure automaton partitioning. This happens for

$$\alpha_2 = 1 - \frac{1.09}{\sqrt{\sigma}}\ m^{\frac{1}{m(1-\alpha_2)}}$$

(observe that $\alpha_2 \longrightarrow 1 - 1.09/\sqrt{\sigma}$ when m grows).

Therefore, we have a smooth transition from pattern partitioning to automaton partitioning. The problem is how to select the best j for each case. Although the gross analysis of [3] suggests to use the largest admissible j, the

effect of round-offs in the underlying partitioned automata shows up here too, forcing the use of a more detailed heuristic. To determine which j is better, we consider Eq. (4) for the cost of each automaton (the worst case does not occur, since we have $\alpha < \alpha_2 < 1 - 1.09/\sqrt{\sigma}$). Thus, the cost for each j is j times formula 4, where m is replaced by $\lceil m/j \rceil$ and k is replaced by $\lfloor k/j \rfloor$.

Figure 4 (left plot) shows an example. As it can be seen, larger j tend to be better, but become useless sooner. Our heuristic makes always the better decision except that stops using each j short before it should. This is an artificially introduced guard to avoid catastrophic results in biased texts, since in special cases we could have much more verifications than in the random model.

Observe that we can take this idea as a generalization of plain pattern partitioning, using it in the range $\alpha_0 < \alpha < \alpha_2$.

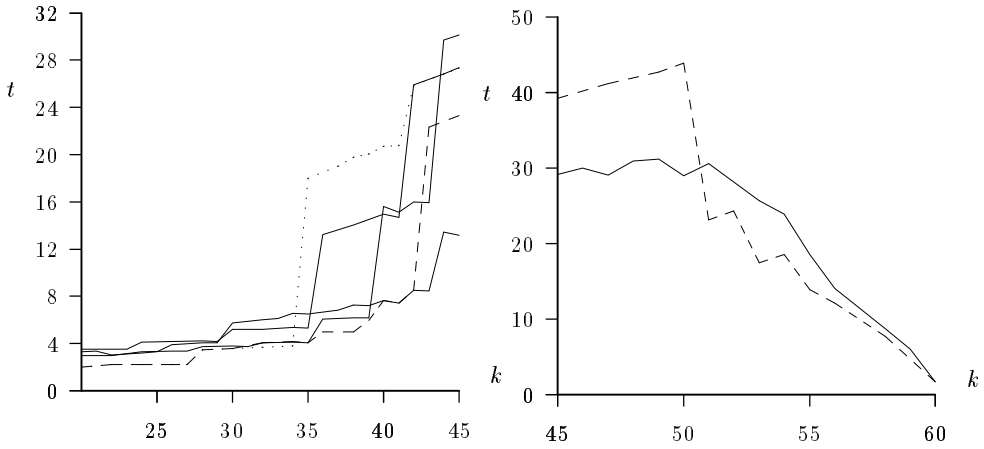

Fig. 4. On the left, pattern partitioning for $j = 2, 4, 6$ (full lines, the larger j jump first), for maximal j (dotted) and the heuristic (dashed). On the right, vertical partitioning (dashed) versus minimal rows partitioning (full). We use $m = 61$, $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

Finally, we observe that if $j = k + 1$, the sub-patterns are searched with zero errors, what can be accomplished by a fast multiple string searching algorithm. We use a variation of Boyer-Moore-Horspool-Sunday [13] to drive this search. Although this variant is linear and much faster than the others, it cannot be used past $\alpha_0 = 1/(3 \log_\sigma m)$, where the verifications become too expensive. This is essentially the idea of [4]. We find later a more exact replacement for the theoretical value 3 in the above expression.

3.4 The Combined Algorithm

Our final algorithm uses the simple automaton when it fits in a word, otherwise it uses $j = k + 1$ for $\alpha < \alpha_0$, pure pattern partitioning for $\alpha_0 < \alpha < \alpha_1$, mixed partitioning for $\alpha_1 < \alpha < \alpha_2$, and pure automaton partitioning for $\alpha > \alpha_2$.

This scheme, unlike the original one [3], does not degrade as m grows, since it is $O(kn \log_c(m)/w)$ up to α_2, which tends to a constant.

We have derived above exact practical values for α_1, α_2, the j of the mixed partition, where to use each kind of automaton partitioning, and the expected cost of the algorithm in each case. The only remaining practical value is α_0, which is obtained in the next section, by comparing the different algorithms.

4 Experimental Comparison

We run the fastest algorithms we are aware of, for a number of different values of σ, m and k. Later, we derive the fastest hybrid algorithm to solve this problem.

Since we compare only the fastest algorithms, we leave aside [12, 18, 8, 9, 15], which were not competitive in our experimental study. The algorithms included in this comparison are

Ukkonen [19] is the standard dynamic programming algorithm, except that it avoids to compute inactive columns. The code is ours.

Chang [5] 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 [14] 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 [14].

Baeza-Yates/Perleberg [4] is essentially the heuristic $j = k + 1$, that our hybrid algorithm uses for $\alpha < \alpha_0$. The code is ours.

Wu-Manber [22] uses bit-parallelism to simulate the automaton by rows. The code is taken from Wright's tests [20]. It is limited to $m \leq 31$, and it would be slower if generalized.

Wright [20] 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 [24] 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 [24] (r is related with the size of the Four Russians tables).

Agrep [21] is a widely distributed approximate search software (version 2.04), 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 (simple automaton, pattern partitioning and automaton partitioning). We show only the pure strategies (and exclude mixed partitioning) to simplify the exposition. As we see later, mixed partitioning is always beaten by other algorithms.

We tested random patterns against 1 Mb of random text on a Sun Sparc-Classic, of approximately Specmark 26, running SunOS 4.1.3, with 16 Mb of RAM. In our machine, $w = 32$. Each data point was obtained by averaging the Unix's user time over ten trials. We also tested on lowercase English text, for patterns randomly selected from the text, at word beginnings.

Figure 5 shows the results for $m = 9$ (where our simple automaton can be used for any k) and all values for k. Similarly, Figure 6 shows the results for $m = 31$, and Figure 7 for $m = 61$. Finally, Figure 8 shows the results for English text.

We use the experimental data to depict the fastest heuristic to solve this problem on-line. The behavior of the algorithms for English text is approximately the same as for random text with $\sigma = 1/p$, where p is the probability of two characters of the text being equal ($\sigma \approx 13$ in our English texts).

An exception to the above statement is Agrep, which performs significantly better for English than for random text, and Wright, which needs to represent all letters and digits, and hence was used with $\sigma = 64$.

If the problem is small enough for our simple automaton to be applied, that is the best choice, except for $\alpha < 1/(5.4 \log_\sigma m)$, where it is convenient to partition the pattern into exact matching ($j = k + 1$). Therefore, our algorithms are the best for small patterns, except for English text, where Agrep is better to search with one error.

If the simple automaton cannot be used, there are four clearly distinct areas (some of them can be empty for certain values of σ):

- For $\alpha < \alpha_0$, our partitioning into exact matching is the best choice (from α_0 its verifications become significant). Although we know from [3] that α_0 is close to $1/(3 \log_\sigma m)$, we refine here that value by using least squares. We find $\alpha_0 = 1/(2.9 \log_\sigma m)$.

- For $\alpha_0 < \alpha < \alpha_1$, our pattern partitioning is the best choice (from α_1 its verifications become significant). We know that

$$\alpha_1 = 1 - \frac{1.09}{\sqrt{\sigma}} \; m^{\frac{1+\sqrt{1+w\alpha_1/(1-\alpha_1)}}{w}}$$

  However, in practice it is better to terminate this area for k one or two before this value. The reason is that this algorithm is very sensitive to the input distribution, and starting the next area before the right point is not very harmful, while doing it after the right point can be catastrophic.

- For $\alpha_1 < \alpha < \alpha_3$, the best choice is between Chang [5] and Wright [20] (we define α_3 in the next item). To compare them, we proceed as follows.

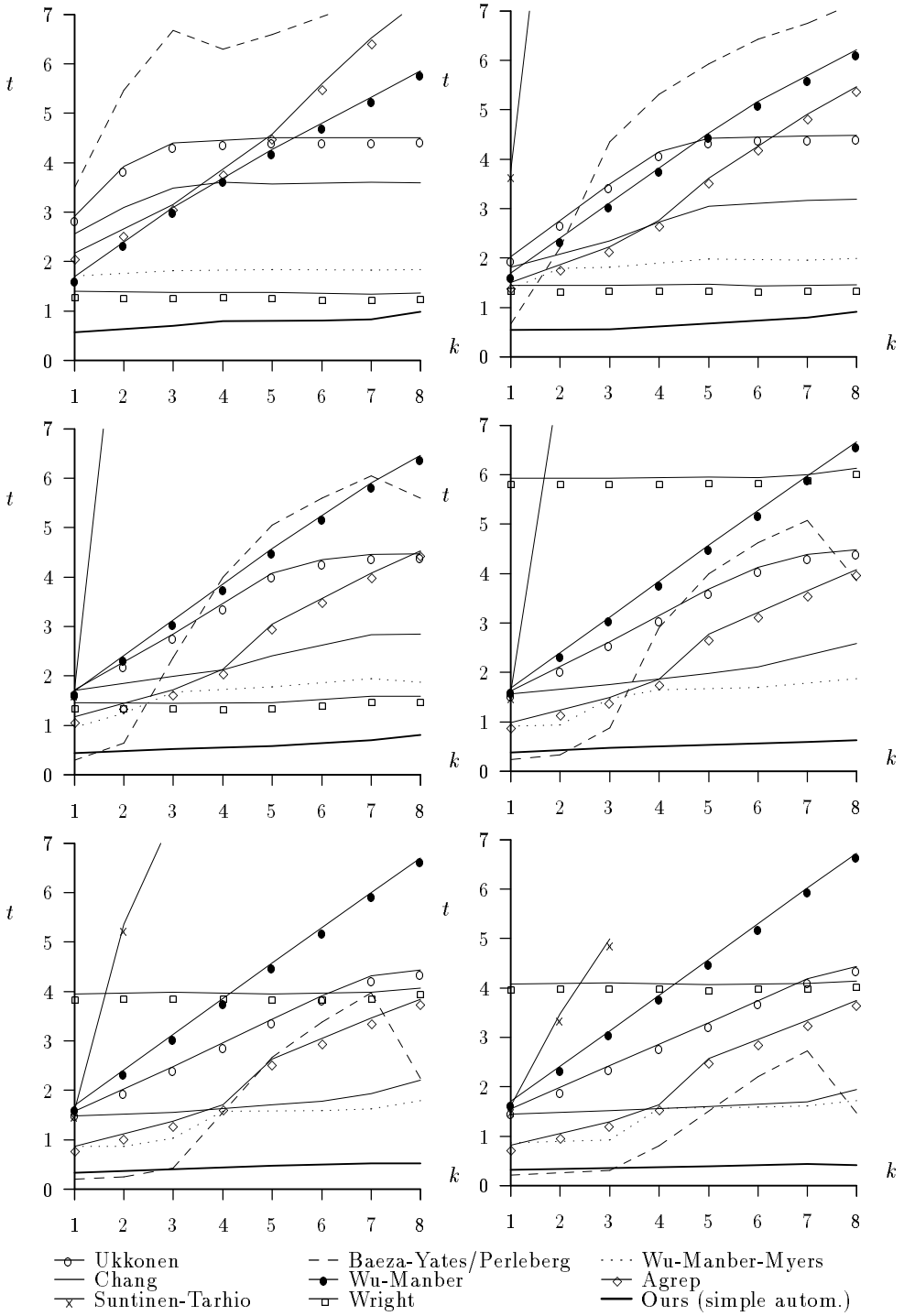

Fig. 5. Times in seconds for $m = 9$ and $k = 0..8$. From top to bottom and left to right, the plots are for $\sigma = 2$, 4, 8, 16, 32 and 64.

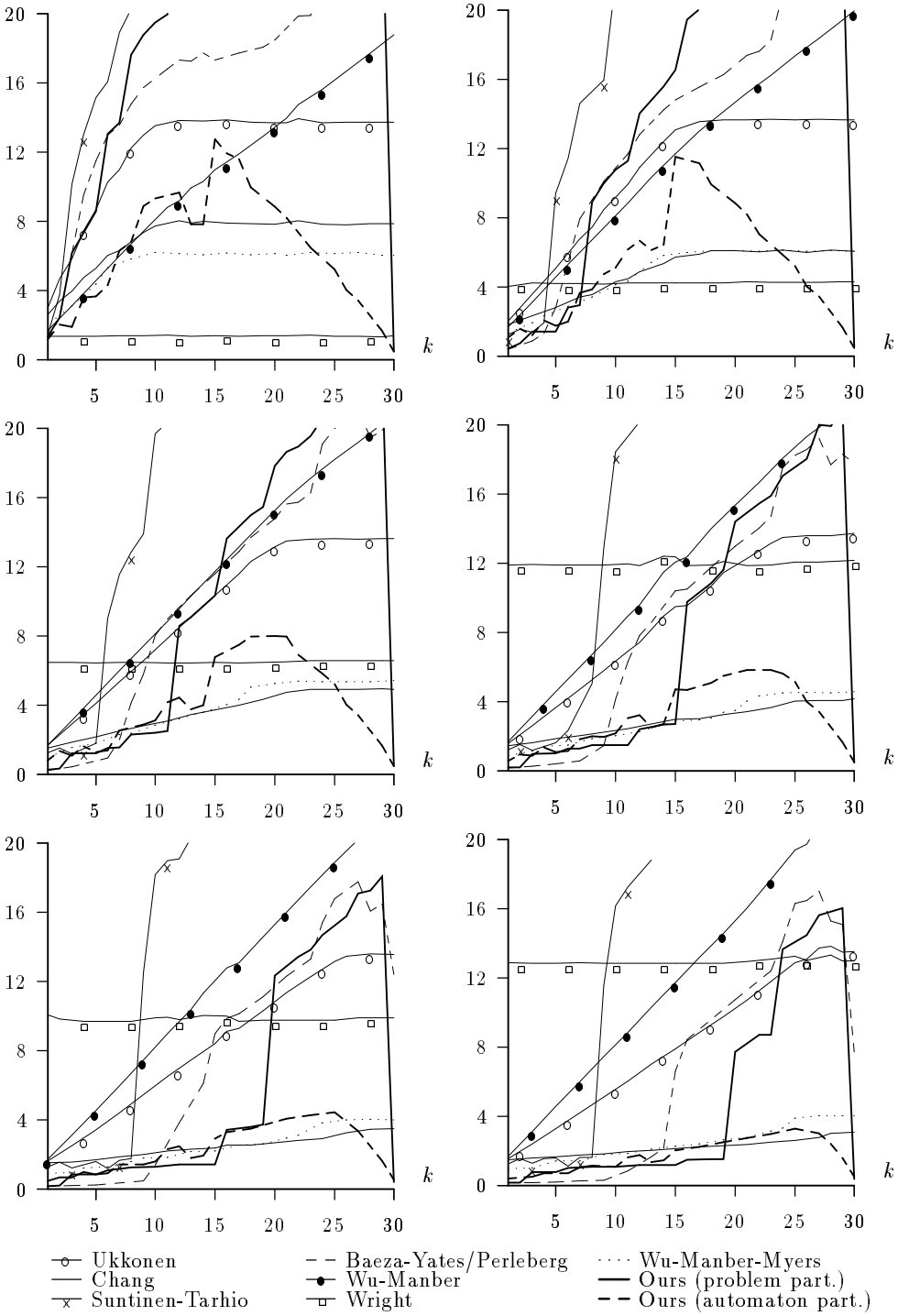

Fig. 6. Times in seconds for $m = 31$ and $k = 0..30$. From top to bottom and left to right, the plots are for $\sigma = 2$, 4, 8, 16, 32 and 64.

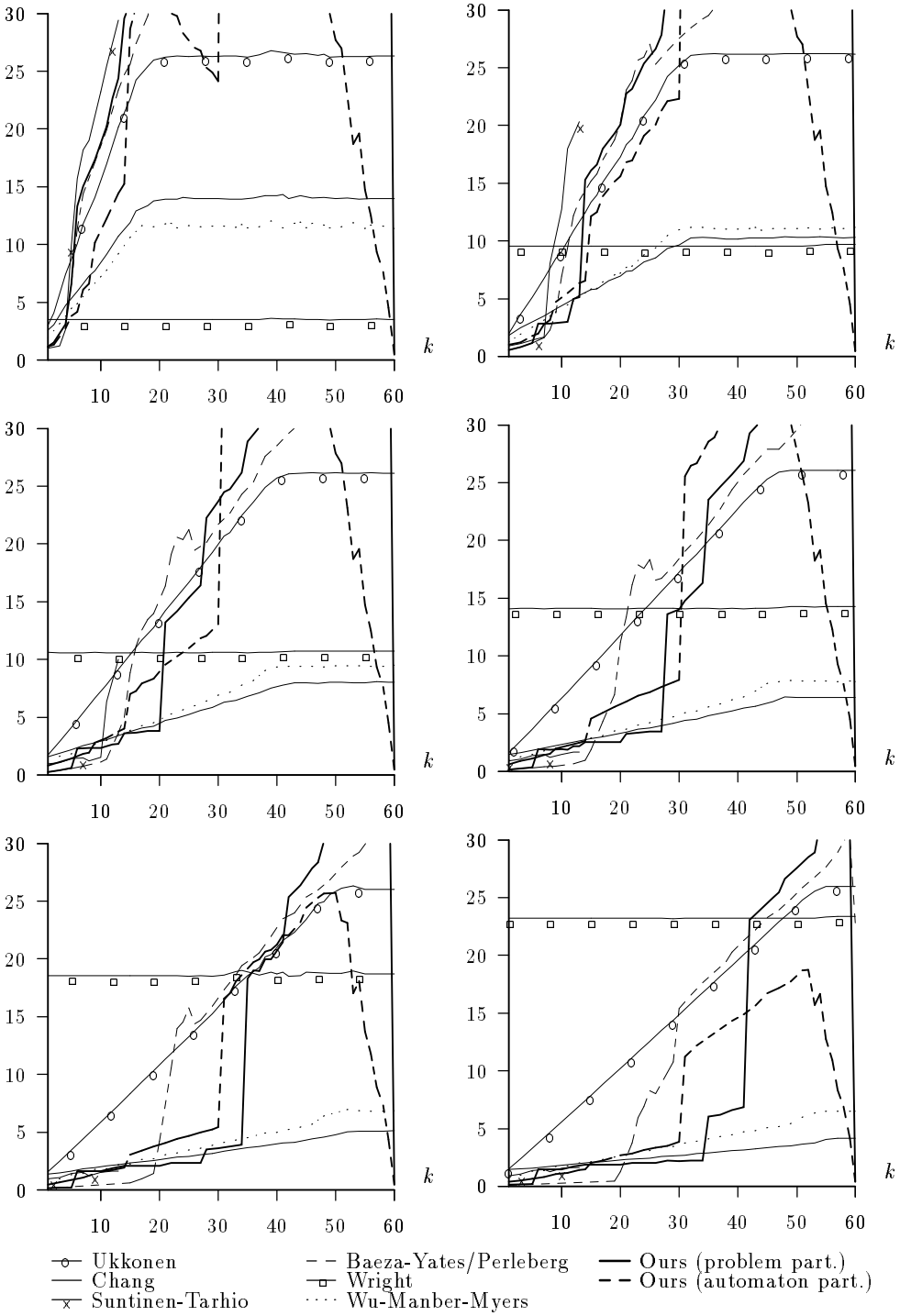

Fig. 7. Times in seconds for $m = 61$ and $k = 0..60$. From top to bottom and left to right, the plots are for $\sigma = 2$, 4, 8, 16, 32 and 64.

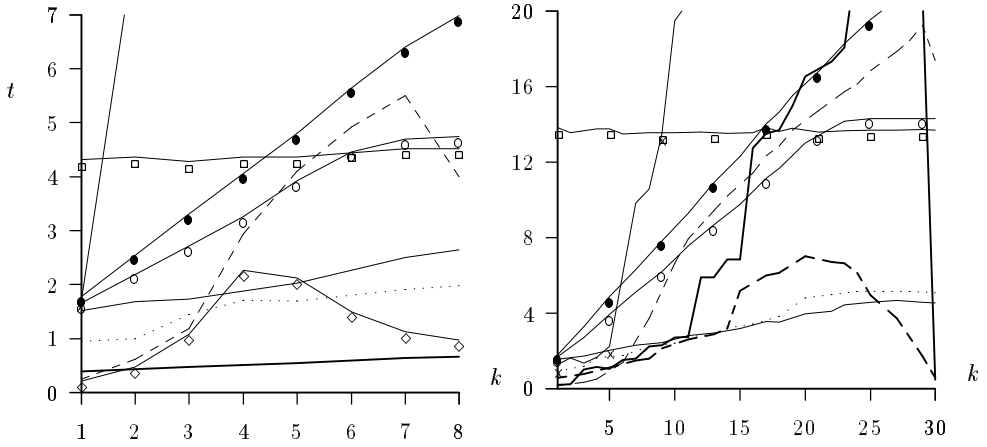

Fig. 8. Times in seconds for English text. The plots are for $m = 9$ and 31.

We know from [5] that Chang is $O(kn/\sqrt{\sigma})$, while from [20] we have that Wright is $O(mn\log(\sigma)/w)$. We use least squares to find the actual constants, in seconds for a 1 Mb text. Although the constants are for our machine, we use them only to compare among algorithms. We find very tight approximations, namely $2.0 + 0.26k/(1.04\sqrt{\sigma} - 0.99)$ for Chang, and $2m\lceil\log_2(\sigma)\rceil/w$ for Wright. We can use those numbers directly to select the best algorithm, or simplify the formula to obtain that we should use Chang up to

$$\alpha_{cw} \approx 7.7\frac{(\sqrt{\sigma}-1)\lceil\log_2\sigma\rceil}{w}$$

which depends only on the alphabet size and the size of the computer word.

– For $\alpha > \alpha_3$, our automaton partitioning is the best choice. The selection of α_3 proceeds by comparing the real costs drawn in the previous item with our real costs. The corresponding times for our different types of automata are: (a) $1.00 \times IJ$, (b) $0.83 \times IJ$, (c) $1.26 \times IJ$, vertical $1.04 \times I_v$. This practical prediction method succeeded in our experiments.

Figure 9 summarizes the results, showing in which case should each algorithm be applied, and the practical performance achieved.

5 Concluding Remarks

We studied the practical performance of a new algorithm for approximate string matching. We made a number of experiments comparing the fastest on-line algorithms that solve this problem, and found the situations in which each algorithm should be applied.

This provides not only an experimental analysis of the performance of our new algorithm against others, but also the fastest known heuristic for on-line

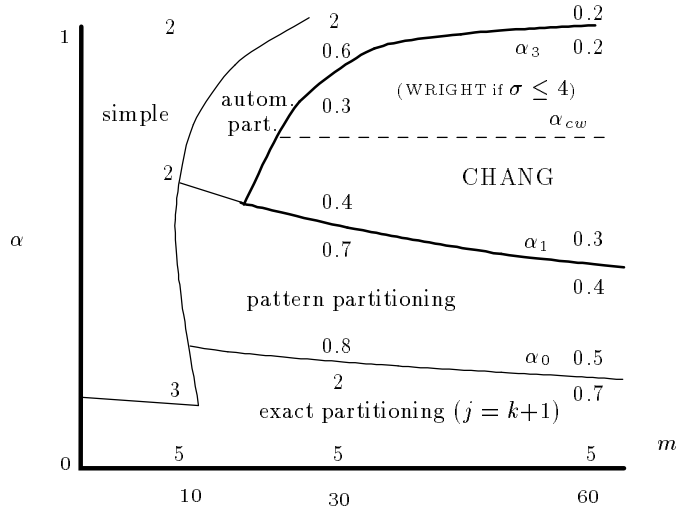

Fig. 9. The optimal heuristic. The numbers inside the plot are the search performance, in Mb per second, for our machine, with $\sigma = 32$. For this value, α_{cw} is outside the relevant area (i.e. between α_1 and α_3). Although we draw a dashed ficticious α_{cw} to show its shape, the numbers inside the area correspond to Chang.

approximate string matching, for m not extremely large (which is the case in text searching). This heuristic involves the use of our algorithms, as well as those of Wright [20] and Chang [5].

To complete this study, the case of very large m must be included. In that case, sublinear algorithms that are not practical for medium m become relevant (essentially, filtration algorithms [16, 17, 15, 14, 6, 7]).

Acknowledgments

We thank Gene Myers and Udi Manber for their helpful comments on a previous version of this work. We also thank all those 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 [22, 8, 18]), Gene Myers, Erkki Suntinen, and Tadao Takaoka.

References

1. 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.

2. R. Baeza-Yates. A unified view to pattern-matching problems. In *Proc. PANEL'96, Bogotá, Colombia*, 1996. To appear. `ftp://sunsite.dcc.uchile.cl/-pub/users/rbaeza/unified.ps.gz`.

3. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. of CPM'96*, Laguna Beach, California, June 1996. To appear. `ftp//-sunsite.dcc.uchile.cl/pub/users/gnavarro/cpm96.ps.gz`.

4. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, pages 185–192. Springer-Verlag, 1992. LNCS 644.

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

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

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

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

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

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

11. 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.

12. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

13. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.

14. E. Suntinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. of ESA'95*. Springer-Verlag, 1995. LNCS 979.

15. T. Takaoka. Approximate pattern matching with samples. In *Proc. of ISAAC'94*, pages 234–242. Springer-Verlag, 1994. LNCS 834.

16. 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.

17. E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.

18. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

19. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

20. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.

21. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.

22. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.

23. S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *J. of Algorithms*, 19:346–360, 1995.

24. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

This article was processed using the LaTeX macro package with CUP_CS class