Multiple Approximate String Matching *

Ricardo Baeza-Yates Gonzalo Navarro

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

Abstract. We present two new algorithms for on-line multiple approximate string matching. These are extensions of previous algorithms that search for a single pattern. The single-pattern version of the first one is based on the simulation with bits of a non-deterministic finite automaton built from the pattern and using the text as input. To search for multiple patterns, we superimpose their automata, using the result as a filter. The second algorithm partitions the pattern in sub-patterns that are searched with no errors, with a fast exact multipattern search algorithm. To handle multiple patterns, we search the sub-patterns of all of them together. The average running time achieved is in both cases $O(n)$ for moderate error level, pattern length and number of patterns. They adapt (with higher costs) to the other cases. However, the algorithms differ in speed and thresholds of usefulness. We analyze theoretically when each algorithm should be used, and show experimentally that they are faster than previous solutions in a wide range of cases.

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 and a pattern of length m (both sequences over an alphabet Σ of size σ), 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 use the term "error ratio" to refer to $\alpha = k/m$.

The solutions to this problem differ if the algorithm has to be on-line (i.e. 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 for a single pattern is $O(mn)$ running time [?].

In the last years several algorithms have improved the classical one. Some achieve $O(kn)$ cost by using the properties of the dynamic programming matrix [?, ?, ?, ?, ?]. Others filter the text to quickly eliminate uninteresting parts [?, ?, ?, ?, ?], some of them being sublinear on average for moderate α. Yet other approaches use bit-parallelism [?] to reduce the number of operations [?,

* This work has been supported in part by FONDECYT grants 1950622 and 1960881.

?, ?, ?]. In [?] the search is modeled with a non-deterministic finite automaton, whose execution is simulated in parallel on machine words of w bits, achieving $O(kmn/w)$ time. In [?], we simulate the same automaton in a different way, achieving $O(n)$ time for small patterns. This algorithm is shown to be the fastest in that case (see also [?]).

The problem of approximately searching a set of patterns (i.e. the occurrences of anyone of them) has been considered only recently. A trivial solution is to do r searches, where r is the number of patterns. As far as we know, the only previous works on this problem are [?] and [?]. The first approach uses hashing to search many patterns with *one* error, being efficient even for one thousand patterns. The second one filters the text by counting matching positions, keeping many counters in a single computer word and updating them in a single operation.

In this work, we present two new algorithms that are extensions of previous ones to the case of multiple search. In Sections 2 and 3 we explain and extend [?]. In Section 4 we do the same for [?]. In Section 5 and 6 we analyze our algorithms and compare them against [?] and [?].

Although [?] allows to search for many patterns, it is limited to only one error. We allow any number of errors, and improve [?] when the number of patterns is not very large (say, less than 60). We improve [?] except for intermediate error ratios. We also improve the trivial algorithm (i.e. one separate search per pattern) when the error ratio is moderate. The extension of [?] is the fastest for small error ratios, while that of [?] adapts better to more errors.

2 Bit-Parallelism by Diagonals

In this section we review the main points of the algorithm [?]. We refer the reader to the original article for more details.

Consider the NFA for searching `"this"` with at most $k = 2$ errors shown in Figure 1 (for now disregard the "$(+\ x)$"). 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). The loop at the initial state allows to consider any character as a potential starting point of a match. The automaton accepts a character (as the end of a match) whenever a rightmost state is active. Initially, the active states at row i ($i \in 0..k$) are those at the columns from 0 to i, to represent the deletion of the first i characters of the pattern, referred here as $pat[1..m]$.

Many algorithms for approximate string matching consist fundamentally in simulating this automaton by rows or columns. The dependencies introduced by the diagonal empty transitions prevent the parallel computation of the new

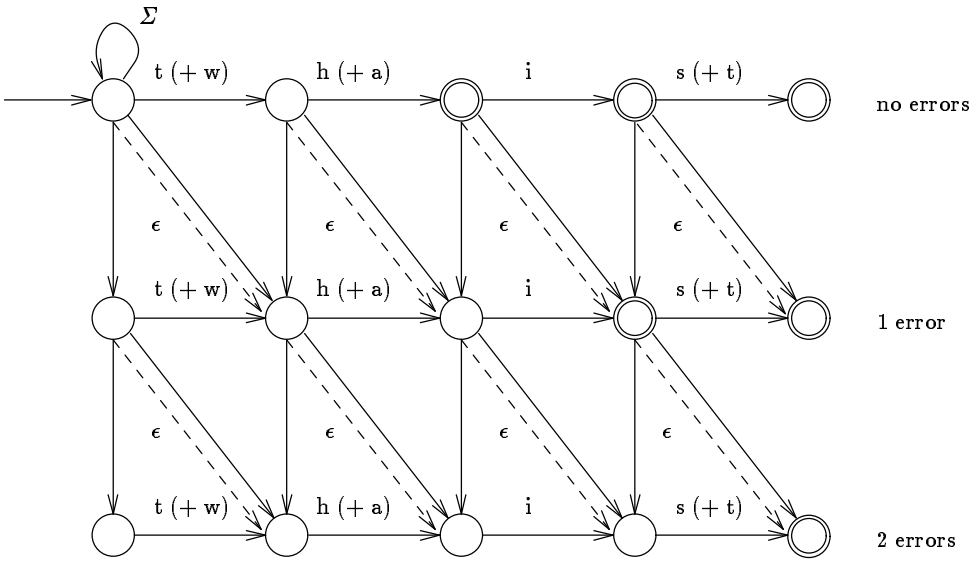

Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character.

values. In [?] we show that by simulating the automaton by diagonals, it is possible to compute all values in parallel.

Because of the empty transitions, once a state in a diagonal is active, all the subsequent states in that diagonal become active too, so we can define the minimum active row of each diagonal, D_i (diagonals are numbered by looking the column they start at). The new values for D_i $(i \in 1..m-k)$ after we read a new text character c can be computed by

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

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

We use bit-parallelism to represent the D_i's in unary. With some pattern preprocessing, the parallel update is $O(1)$ cost and very fast in practice. A central part of this preprocessing is the definition of an m bits long mask $t[c]$, representing match or mismatch against the pattern, for each character c. The resulting algorithm is linear whenever the representation fits in a computer word (i.e. $(m-k)(k+2) \leq w$, where w is the number of bits in a computer word).

Observe that the $t[]$ table mechanism allows more sophisticated searching: at each position of the pattern, we can allow not a single character, but a class of characters, at no additional search cost. It suffices to set $t[c]$ to "match" at position i for every $c \in pat[i]$. For example, we can search in case-insensitive by allowing each position to match the upper-case and lower-case versions of the letter. We use this property to allow multiple patterns.

3 Superimposed Automata

Suppose we have to search r patterns $P_1, ..., P_r$. We are interested in the occurrences of any one of them, with at most k errors. We can extend the previous bit-parallelism approach by building the automaton for each one, and then "superimpose" all the automata.

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

The superimposition is defined as follows: we build the $t[]$ table for each pattern, and then take the bitwise-*or* of all the tables. The resulting $t[]$ table matches at position i with the i-th 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, namely

$$\{P_1[1], ..., P_r[1]\}\ \{P_1[2], ..., P_r[2]\}\ ...\ \{P_1[m], ..., P_r[m]\}$$

for example, if the search is for `"this"` and `"wait"`, the string `"whit"` is accepted with *zero* errors. See Figure 1, this time paying attention to the two characters present at each horizontal transition.

For a moderate number of patterns, the filter is strict enough at the same cost of a single search. Each occurrence reported by the automaton has to be verified for all the involved patterns (we use [?] for this step).

If the number of patterns is so large that the filter does not work well, we partition the set of patterns into groups of r' patterns each, build the automaton of each group and perform $\lceil r/r' \rceil$ independent searches. The cost of this search is $O(r/r'\ n)$, where r' is small enough to make the number of verifications negligible. This r' always exists, since for $r' = 1$ we have a single pattern per automaton and no verification is needed.

If the length of the patterns does not allow to put their automata in single computer words (i.e. $(m-k)(k+2) > w$), we partition the problem. We adapt the two partitioning techniques defined in [?].

3.1 Pattern Partitioning

The following lemma proved in [?] suggests a way to partition a large pattern.

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

Thus, we can reduce the size of the problem if we divide the pattern in j parts, provided we search all the sub-patterns with $\lfloor k/j \rfloor$ errors. Each match of a sub-pattern must be verified to determine if it is in fact a complete match.

To perform the partition, we pick the smallest j such that the problem fits in a single computer word (i.e. $(\lceil m/j \rceil - \lfloor k/j \rfloor)(\lfloor k/j \rfloor + 2) \leq w$). We divide the pattern in j subpatterns as evenly as possible. The limit of this method is

reached for $j = k+1$, since in that case we search with zero errors. The resulting algorithm is qualitatively different and is described later.

Once we partition all the patterns, we are left with jr subpatterns to be searched with $\lfloor k/j \rfloor$ errors. We simply group them as if they were independent patterns to search with the general method. The only difference is that we have to verify the complete patterns when we find a possible occurrence. To avoid verifying all the patterns at each match, we try as much as possible to include subpatterns of the same patterns in the groups.

3.2 Automaton Partitioning

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

The idea is as follows: once the (large) automata have been superimposed, we partition the automaton into a matrix of subautomata, each one fitting in a computer word. Those subautomata behave differently than the simple one, since they must propagate bits to their neighbors.

Once the automaton is partitioned, we run it over the text updating its subautomata. Observe, however, that it is not necessary to update all the subautomata. In the same spirit as [**?**], we work only on "active" diagonals.

The technique of grouping in case of a very relaxed filter is used here too. We use the heuristic of sorting the patterns and packing neighbors in the same group, trying to have the same first characters.

4 Exact Partitioning Extended to Multiple Patterns

We first briefly review the algorithm [**?**] (studied more in detail in [**?**, **?**]). We refer the reader to the original articles for details.

A particular case of the lemma of Section 3.1 shows that if a pattern matches a text position with k errors, and we split the pattern in $k+1$ pieces, then at least one of the pieces must be present with no errors in each occurrence (this is a folklore property which has been used several times [**?**, **?**, **?**]). Searching with zero errors leads to a completely different technique.

Since there are efficient algorithms to search for a set of patterns exactly, we partition the pattern in $k+1$ pieces (of similar length), and apply a multipattern exact search for the pieces. Each occurrence of a piece is verified to check if it involves a complete match. If there are no too many verifications, this algorithm is extremely fast.

From the many algorithms for multipattern search, an extension of Boyer-Moore-Horspool-Sunday (BMHS) [**?**] gave the best results. We build a trie with the sub-patterns. At each text position we search the text that follows into the trie, until a leaf is found (match) or there is no path to follow (mismatch). The jump to the next text position is precomputed as the minimum of the jumps allowed in each sub-pattern by the BMHS algorithm.

Observe that we can easily add more patterns to this scheme. Suppose we have to search for r patterns $P_1, ..., P_r$. We cut each one into $k+1$ pieces and search in parallel for *all* the $r(k+1)$ pieces. When a piece is found in the text, we use a classical algorithm to verify its pattern in the candidate area (this time we normally know which pattern to verify). As in the previous case, this constitutes a good filter if the number of patterns and errors is not too high. Unlike the previous case, grouping is of no use here, since there are no more matches in the union of patterns than the sum of the individual matches.

5 Analysis

We are interested in the restrictions that α and r must satisfy for each mechanism to be efficient in filtering most of the unrelevant part of the text. We are also interested in the complexity of the algorithms, especially in when they are linear on average and when they are *useful* (i.e. better than r sequential searches).

5.1 Superimposed Automata

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

In [**?**] we prove that the probability of a given text position matching a random pattern with error ratio α is $O(a^m)$, where $a = 1/(\sigma^{1-\alpha}\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)})$. It is also proven that $a < 1$ whenever $\alpha < 1 - e/\sqrt{\sigma}$.

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

The algorithm is linear on average whenever the total cost of verifications is $O(1)$ per character. Since each verification costs $O(m^2)$ per pattern in the superimposed group, we pay $O(r'm^2)$ to verify the whole group. Thus, we want the probability of a verification to be $O(1/(r'm^2))$, which happens for $a < 1$.

To decide which is the optimal size of the groups, we state that the total cost of verifications must be at most one per character, since if it is larger we would prefer to make two separate searches with much less verifications. Hence, our r' satisfies $a^m = 1/(r'm^2)$, i.e.

$$r' = \left(\frac{\sigma^{m-k}\alpha^{2k}(1-\alpha)^{2(m-k)}}{m^2} \right)^{\frac{1}{m-k+1}}$$

which our algorithm uses to determine the correct grouping.

For m not too small, we can simplify and bound the above result to

$$r' = \sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2\left(1+O\left(\frac{\log m}{m}\right)\right) \geq \frac{\sigma(1-\alpha)^2}{e^2}\left(1+O\left(\frac{\log m}{m}\right)\right)$$

where the last step is valid because $1 \geq \alpha^{\frac{\alpha}{1-\alpha}} \geq e^{-1}$ for $0 \leq \alpha \leq 1$.

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

Observe that this means a linear algorithm for $r = O(\sigma)$ (taking the error ratio as a constant), and that for $\alpha > 1 - e/\sqrt{\sigma}$, the cost is $O(rn)$, not better than the trivial solution (i.e. $r' = 1$ and hence no superimposition occurs).

Pattern Partitioning We have now jr patterns to search with $\lfloor k/j \rfloor$ errors. The error level is the same for subproblems (recall that the subpatterns are of length m/j). Since we group as many subpatterns of the same pattern as possible, we have that if our sets have size r', they have sub-patterns of at most $1 + \lceil r'/j \rceil$ distinct patterns. Therefore, the cost of a verification is $O(m^2 r'/j)$ and our equation to define r' is $a^{m/j} = j/(r'm^2)$, i.e.

$$r' = \left((j/m^2)^j \sigma^{m-k}\alpha^{2k}(1-\alpha)^{2(m-k)}\right)^{\frac{1}{m-k+j}}$$

To approximate this value we expand j, whose formula is obtained in [**?**]. We rephrase it as $j = (m-k)d$, with $d = d(\alpha, w) = \left(1+\sqrt{1+\frac{w\,\alpha}{1-\alpha}}\right)/w = O(1/\sqrt{w})$. The order is valid because $1 - \alpha \geq e/\sqrt{\sigma}$ (past this point the verifications are too many even with no superimposition). We then have $r' = (d/m)^{\frac{d}{d+1}}\sigma^{\frac{1}{d+1}}\alpha^{\frac{2\alpha}{(1-\alpha)(d+1)}}(1-\alpha)^{\frac{d+2}{d+1}}$, and the total cost is $O(jr/r'\ n)$, i.e.

$$\frac{d^{\frac{1}{d+1}} m^{\frac{2d+1}{d+1}}\ rn}{\sigma^{\frac{1}{d+1}}\alpha^{\frac{2\alpha}{(1-\alpha)(d+1)}}(1-\alpha)^{\frac{1}{d+1}}} = \frac{dm\ rn}{\sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)}\ \left(1+O\left(\frac{1}{\sqrt{w}}\right)\right) = O\left(\frac{m}{\sigma\sqrt{w}}\ rn\right)$$

which is linear for $r = O(\sigma\sqrt{w}/m)$ (the order is valid for constant α), and is *useful* when the total cost is less than rn, i.e. $dm < \sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)$. This condition that can be pessimistically bounded by $\alpha \leq 1 - e^2 m/(\sigma\sqrt{w})$.

Automaton Partitioning The analysis for this case is similar to the simple one, except because each step of the large automaton takes time proportional to the total number of subautomata. In [**?**] we show that this number is $O(k(m-k)/w)$. Therefore, the cost formula is

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

which is linear in n for $r = O(\sigma w/m^2)$, and *useful* for $\alpha \leq \sigma w/((em)^2 + \sigma w)$.

5.2 Exact Partitioning

In [?] we analyze this algorithm as follows. Except for verifications, the search time is linear (e.g. by using an Aho-Corasick machine [?] although, as mentioned before, we use an algorithm which is faster in practice). Hence, we are interested in the cases where there are few verifications.

Since we cut the pattern in $k+1$ pieces, they are of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$. On average, half of the subpatterns have each length. The probability of each piece matching is $1/\sigma^{\lfloor m/(k+1) \rfloor}$ (the case $1/\sigma^{\lceil m/(k+1) \rceil}$ is exponentially smaller than this one, so we disregard it). Ignoring equal pieces, the probability of any piece matching is $(k+1)/(2\sigma^{\lfloor m/(k+1) \rfloor})$.

We can easily extend that analysis to the case of multiple search, since we have now $r(k+1)$ pieces of the same length. Hence, the probability of verifying is $r(k+1)/(2\sigma^{\lfloor \frac{m}{k+1} \rfloor})$. This must be $O(1/m^2)$ (i.e. the cost of one verification, since in this algorithm we know which pattern to verify) to ensure that verifications do not affect on average the linearity of the search. This happens pessimistically for $\alpha \leq 1/(3\log_\sigma m + \log_\sigma r)$ (although roundoffs must be observed in practice). The method is better than r sequential searches for $\alpha \leq 1/3\log_\sigma m$.

6 Experimental Results

We experimentally study our algorithms and compare them against previous work [2]. We tested with 1 Mb of random text ($\sigma = 30$) and lower-case English text. The patterns were randomly generated in the first case and randomly selected from the the same text (at the start of non-stopwords) in the second case. We use a Sun SparcStation 4 running Solaris 2.4, with 32 Mb of RAM, and $w = 32$. Each data point was obtained by averaging the Unix's user time over 10 trials.

We first show the degree of parallelism achieved by our different algorithms, in terms of the ratio between the parallel version and r applications of the same single-pattern algorithm. Figure 2 shows the behavior in terms of r, and Figure 3 in terms of k. We observe that for low error ratio the parallel versions take 0.3 to 0.6 of their sequential versions, and that exact partitioning works better for more patterns, while the others quickly stabilize. On the other hand, only automaton partitioning works well for a moderate number of errors, while the other two quickly become worse than their single-pattern counterparts. We tried other m and r values with very similar results (omitted here for lack of space).

We compare now our algorithms against Muth-Manber [?] and Navarro [?]. Figure 4 shows a comparison for $k = 1$ and varying r. Exact partitioning is the best for a moderate number of patterns (i.e. near 60 patterns, except for the bad case of short English patterns). For more patterns, Muth-Manber is better. Pattern and automaton partitioning are better than Muth-Manber for a small number of patterns (10-15) and better than Navarro on random text.

[2] We thank Robert Muth and Udi Manber for their implementation of [?].

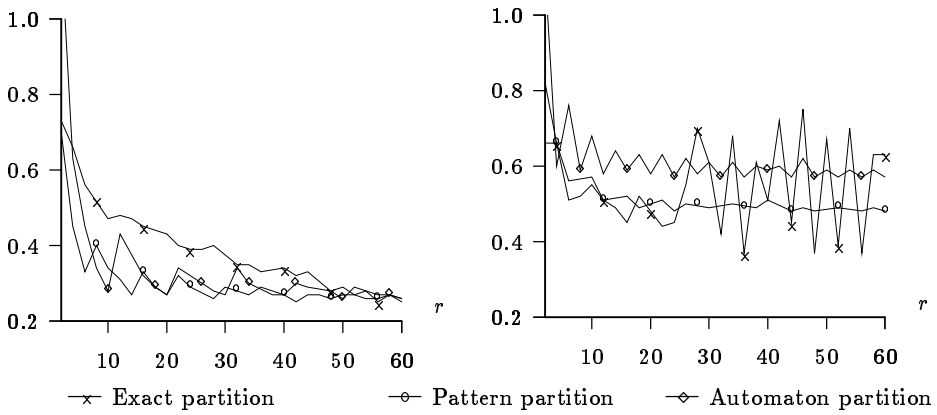

Fig. 2. Parallelism ratio for $m = 10$ and $k = 1$. The left plot is for random text ($\sigma = 30$), the right plot for English text.

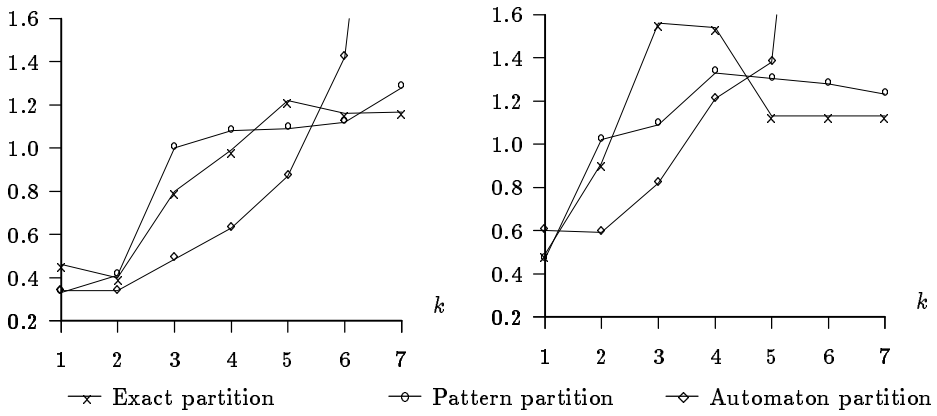

Fig. 3. Parallelism ratio for $m = 10$ and $r = 15$. The left plot is for random text ($\sigma = 30$), the right plot for English text.

Figure 5 shows a comparison (excluding Muth-Manber because $k > 1$) for fixed r and varying k. Exact partitioning is the fastest algorithm for low error ratios. For moderate error ratios the verifications make it useless and Navarro becomes the best algorithm. When the error ratio increases further, automaton partitioning becomes the best choice. The usefulness of this last algorithm ends for a large number of errors, when it becomes similar to r sequential searches.

7 Concluding Remarks

We are working on a number of heuristic optimizations to our algorithms, for instance

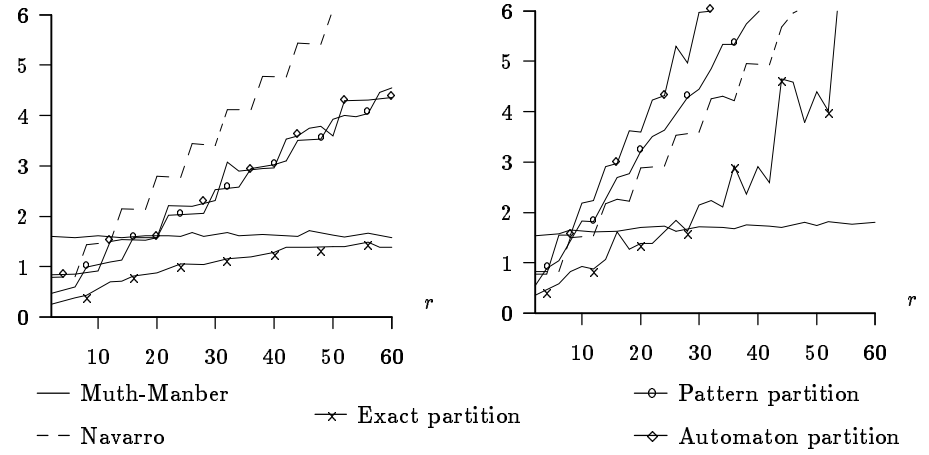

Fig. 4. Times in seconds for $m = 10$ and $k = 1$. The left plot is for random text ($\sigma = 30$), the right plot for English text.

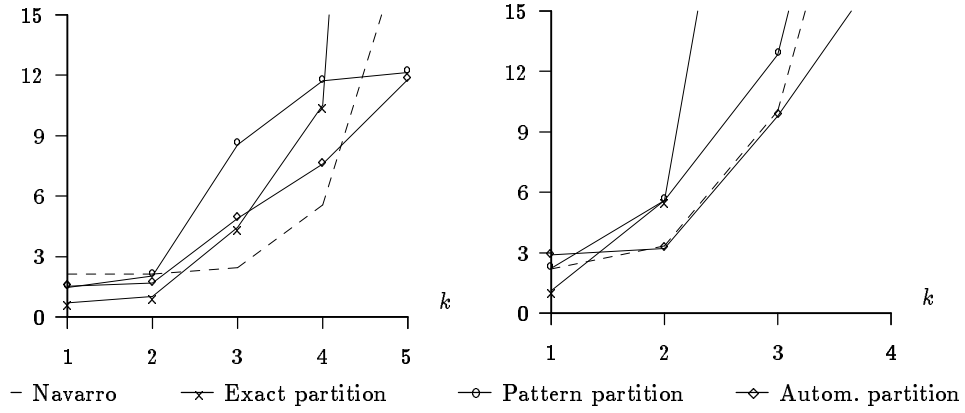

Fig. 5. Times in seconds for $m = 10$ and $r = 15$. The left plot is for random text ($\sigma = 30$), the right plot for English text.

- If the patterns have different length, we truncate them to the shortest one when superimposing automata. We can select cleverly the substrings to use, since having the same character at the same position in two patterns improves the filtering mechanism.
- We used simple heuristics to group subpatterns in superimposed automata. These can be improved to maximize common letters too.
- We are free to partition each pattern in $k+1$ pieces as we like in exact partitioning. This is used to minimize the expected number of verifications when the letters of the alphabet do not have the same probability of occurrence (e.g. in English text). We have an $O(m^3)$ dynamic programming algorithm to select the best partition. The same can be done in superimposed automata.

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