Improving an Algorithm for Approximate Pattern Matching *

Gonzalo Navarro Ricardo Baeza-Yates

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

Abstract

We study a recent algorithm for fast on-line approximate string matching. This is the problem of searching a pattern in a text allowing errors in the pattern or in the text. The algorithm is based on a very fast kernel which is able to search short patterns using a non-deterministic finite automaton, which is simulated using bit-parallelism. A number of techniques to extend this kernel for longer patterns are presented in that work. However, the techniques can be integrated in many ways and the optimal interplay among them is by no means obvious.

The solution to this problem starts at a very low level, by obtaining basic probabilistic information about the problem which was not previously known, and ends integrating analytical results with empirical data to obtain the optimal heuristic. The conclusions obtained via analysis are experimentally confirmed. We also improve many of the techniques and obtain a combined heuristic which is faster than the original work.

This work shows an excellent example of a complex and theoretical analysis of algorithms used for design and for practical algorithm engineering, instead of the common practice of first designing an algorithm and then analyzing it.

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, a (short) pattern of length m, and a maximal number of errors allowed k, find all segments (called "occurrences" or "matches") whose *edit distance* to the pattern is at most k. Text and pattern are sequences of characters from an alphabet Σ of size σ. We call $\alpha = k/m$ the *error ratio* or *error level.*

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 $0 < k < m$, i.e. $0 < \alpha < 1$.

The solutions to this problem differ if the algorithm is 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 [20, 21].

*This work has been supported in part by FONDECYT grant 1950622.

In the last years several algorithms have been presented that achieve $O(kn)$ comparisons in the worst-case [28, 11, 14, 15] or in the average case [29, 11, 8], by taking advantage of the properties of the dynamic programming matrix (e.g. values in neighbor cells differ at most in one). The best average complexity achieved under this approach is $O(kn/\sqrt{\sigma})$ [8].

Other approaches attempt to filter the text, reducing the area in which dynamic programming needs to be used [26, 30, 25, 24, 9, 10, 18, 7, 22]. The filtration is based on the fact that some portions of the pattern must appear with no errors even in an approximate occurrence. These algorithms achieve "sublinear" expected time in many cases for low error ratios (i.e. not all text characters are read, $O(kn \log_\sigma m/m)$ is a typical figure), but the filtration is not effective for larger ratios, and some algorithms are not practical if m is not very large.

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 [19, 13] 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 [1, 2, 33]. This technique *simulates* parallelism on a *sequential* processor using bit operations. This takes advantage of the fact that the processor operates in all the bits of the computer word in parallel. In a RAM machine of word length $w = \Omega(\log n)$ bits, this can reduce the number of real operations by a factor of $O(1/w)$. In [31] the cells of the dynamic programming matrix are packed in diagonals to achieve $O(mn \log(\sigma)/w)$ time complexity. In [34] a Four Russians approach is used to pack the matrix in machine words (they end up in fact with a DFA where they can trade the number of states for their internal complexity). In [33], a non-deterministic finite automaton (NFA) that recognizes the approximate occurrences of the pattern is used, which has only a few states and a regular structure. They achieve $O(kmn/w)$ time by parallelizing in bits the work of such automaton. A recent work in this trend is [17], which parallelizes the dynamic programming algorithm to obtain $O(mn/w)$ cost in the worst case and $O(kn/w)$ on average.

In [3, 5] we proposed a new algorithm based on the bit-parallel simulation of the same NFA of [33]. The simulation, however, is completely different and a core algorithm which is $O(n)$ time for small patterns (independently of k) was obtained. The algorithm is the fastest one in that case. A number of techniques to extend that algorithm for longer patterns were shown:

- *Automaton partitioning* simulates the NFA using many computer words, triyng not to work on inactive portions of the automaton.
- *Pattern partitioning* cuts the pattern in pieces and searches all them with less errors, building up the occurrences of the complete pattern from the matches of the pieces.
- *Superimposition* searches many pieces using a single automaton which serves as a filter for the multipattern search. Its matches have to be verified to check which piece (if any) actually matched.

However, those techniques can be combined in non-trivial ways and their optimal interaction is not obvious and was not obtained in [3, 5]. This optimal arrangement depends on the *parameters* of the search problem, the most important of which being

- The error level tolerated (α). In general, filtering techniques (such as pattern partitioning or superimposition) work well only for moderately low error levels. As we see in Section 3, the behavior of the problem changes drastically depending on the error level to tolerate.
- The pattern length (m). Our basic techniques work for short patterns and we need to extend them to work on longer ones.
- The alphabet size (σ). The larger the alphabet, the less probable is that two random strings match, which improves the efficiency of filtration algorithms.
- The length in bits of the computer word (w). We simulate automata using the bits of computer words, so the longer it is, the longer patterns can be accommodated.

In this paper we study in depth the techniques, finding out the range of parameters where each one can be applied and the optimal way to combine them. The analysis starts at a very low level, finding the probability of an approximate occurrence. Each technique is analytically and experimentally studied to determine its expected behavior, and then optimized. At the end, we combine optimally the techniques.

Throughout the work, experimental validation of the analytical results is provided and sometimes used as part of the heuristic results. As a separate contribution obtained in part thanks to the analysis, we improve many of the techniques themselves. Those improvements translate into better execution times in the combined algorithm which were not possible to obtain using the original techniques, even using them in the optimal way. Moreover, due to the new techniques, most previous analyses are significantly modified.

As a side effect, we show the interplay between algorithm analysis and design and the feedback between theory and practice. We highlight the use of analysis for design as well as experimental results for design.

The main improvements obtained over our previous work of [3, 5] are:

- We use our algorithms to verify potential matches instead of relying on classical dynamic programming. This improves the algorithms for high error levels when the patterns are not very long.
- We use a new technique to verify potential matches which is capable of early discarding uninteresting candidates. This makes all our algorithms more resistant to the error level, which translates into better execution times even for low error levels.
- We improve the search time when the pattern is long and the error level is not very high, by improving the register usage of the algorithm. This doubles the searching performance in some cases. More importantly, it allows to keep the same performance of the core algorithm in patterns up to 8 times longer.
- We improve search times for high error levels in up to 20% by using code that assumes a high error level. This is achieved by avoiding performing some expensive bookkeeping which pays off only for low error levels.

- We obtain a heuristic to automatically combine the techniques in an optimal way, based on analytical and empirical results.

A summary of the organization and results presented in the paper follows. In section 2 we explain the main features of the core algorithm presented in [3, 5]. In section 3 we study the main statistics of the problem that drive all the average-case analysis that follows. We consider specifically the probability of matching when errors are allowed and the portion of the NFA which is active on average. In section 4 we present the automaton partitioning technique, explain the general idea, optimize the way to partition the NFA and present new techniques to improve register usage. In section 5 we explain the pattern partitioning technique, optimize the partitioning scheme and present a new technique to integrate the matches of the pattern pieces. In section 6 we describe the technique of superimposing automata and show how to optimize the amount of superimposition. In section 7 we build the complete heuristic based on the previous results, finding the optimal way to combine the above techniques. In section 8 we compare experimentally the combined algorithm against the fastest algorithms we know. Finally, we present our conclusions and future work directions in section 9.

A compiled version of the complete algorithm is publicly available (see Section 7.2). All the experimental results of this paper were obtained on a Sun UltraSparc-1 of 167 MHz running Solaris 2.5.1, with 64 Mb of RAM. This is a 32-bit machine, i.e. $w = 32$. All the times are measured in seconds of user (CPU) time per megabyte of text. Except otherwise stated, our experiments have a standard deviation of 10%.

2 A Bit-Parallel Core Algorithm

In this section we review the main points of the algorithm [3, 5]. We refer the reader to the original articles for more details.

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 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.

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

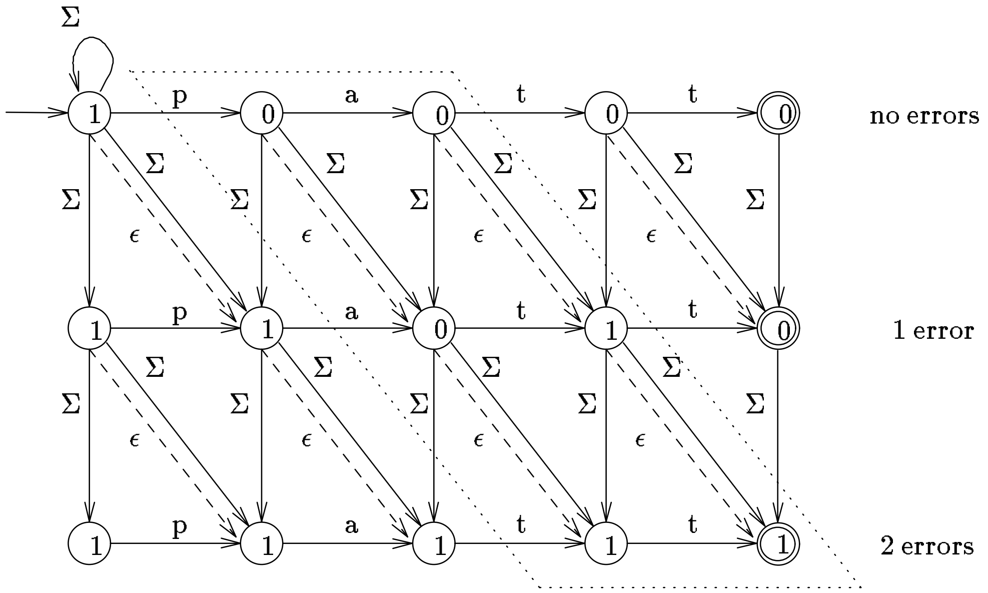


Figure 1: An example NFA for approximate string matching. After processing the text "xat", active states (those containing a "1") are $(2,3)$, $(2,4)$ and $(1,3)$, besides those always active of the lower-left triangle. We enclose in dotted the states actually represented by the algorithm.

Many algorithms for approximate string matching consist fundamentally in simulating this automaton by rows or columns. The dependencies introduced by the diagonal empty transitions prevented the bit-parallel computation of the new values. In [3, 5] we have shown that by simulating the automaton by diagonals (i.e. packing the bits of the diagonals in machine words), it is possible to compute all values in parallel (using bit-parallelism). Hence, when all the bits to represent fit in a computer word, the parallel update formula for each new text character read is $O(1)$ cost and very fast in practice.

For this simulation, it suffices to represent only the complete diagonals of the automaton (excluding the first one). The total number of bits needed to represent the automaton is $(m-k)(k+2)$. If we call w the number of bits in the computer word, the core algorithm is $O(n)$ time in the worst and average case whenever $(m-k)(k+2) \leq w$.

A central part of the algorithm is the definition of an m bits long mask $t[c]$, representing match or mismatch against the pattern for each character c. That is, if the pattern is $patt = p_1...p_m$ (with $p_i \in \Sigma$), then $t[c] = b_1...b_m$, where the bit b_i is set whenever $p_i \neq c$. This $t[\ ]$ table is similar to that used in the Shift-Or algorithm for exact string matching [2], and it allows more sophisticated searching: at each position of the pattern, we can allow not only a single character, but a class of characters, at no additional search cost. This is expressed as an *extended pattern*, which has a set of characters at each position, i.e. it belongs to $\mathcal{P}(\Sigma)^*$ instead of Σ^*. Those patterns are denoted as $patt = C_1...C_m$, where $C_i \subseteq \Sigma$. To search an extended pattern it suffices to set $t[c]$ to "match" at position i for every $c \in C_i$. 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 show later other applications of this ability for our purposes.

An separate speedup technique considers that any occurrence of the pattern must begin with one of its $k+1$ initial letters (otherwise we will spend more than k errors in inserting those letters). We can therefore traverse the text with a fast search for one of those characters, and start the automaton only when we find one. The filtering is resumed when the automaton runs out of active states. As shown in [5] this can double in practice the performance of the automaton for low error levels, although its performance (and even its convenience) depends on α, σ and m.

This technique can be used for other algorithms as well, as long as they are slower than the search for one character (which outrules most filtration algorithms), and they can easily be restarted and terminated.

3 The Statistics of the Problem

As we see later, a number of techniques to handle a long pattern rely on searching pieces of the pattern or even more complex constructions, and then performing a costly verification step each time a piece is found. Therefore, for an average case analysis and to compare different heuristics, it is essential to determine which is the probability of finding a pattern in a text position allowing errors. Another statistical information which is necessary for our average-case analysis is related to which portion of the NFA has active states, as our algorithms try to simulate only the active portion of the NFA.

In all the average-case analysis of this paper we assume that the patterns are not extended. An easy way to consider extended patterns is to replace σ by σ/s in all the formulas, where s is the size of the C_i sets corresponding to pattern positions. This is because the probability of crossing a horizontal edge of the automaton is not $1/\sigma$ anymore, but s/σ.

3.1 Probability of Matching

Given a pattern of length m which is searched in a text, both pattern and text being random sequences over an alphabet of size σ (the letters are selected with uniform probability), we want to find the probability $f(m,k)$ of a match with k errors or less at a given text position. Recall that we use $\alpha = k/m$.

As we show shortly, this probability grows very abruptly as a function of α, being exponentially decreasing with m for small α. The importance of being exponentially decreasing with m is that the cost to verify a text position is $O(m^2)$, and therefore if that event occurs with probability $O(\gamma^m)$ for some $\gamma < 1$ then the total cost of verifications is $O(m^2\gamma^m) = o(1)$, which makes the verification cost negligible. On the other hand, as soon as the cost ceases to be exponentially decreasing it begins to be at least $1/m$, which yields a total verification cost of $O(mn)$. This is the same cost of plain dynamic programming.

In [3, 5] it is shown that $f(m,k) \le \gamma^m$, where

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

and therefore $f(m,k)$ is exponentially decreasing with m whenever $\gamma < 1$, i.e.

$$\alpha \;<\; \alpha^* \;=\; 1 - \frac{e}{\sqrt{\sigma}} \tag{2}$$

On the other hand, the only optimistic bound we can prove is based on considering that only replacements are allowed (i.e. Hamming distance). In this case, given a pattern of length m, the number of strings that are at distance i from it are obtained by considering that we can freely determine the i places of mismatch, and at those places we can put any character except that of the pattern, i.e.

$$\binom{m}{i}(\sigma-1)^i \;=\; \binom{m}{i}\;\sigma^i\;(1+O(1/\sigma))$$

Although we should sum the above probabilities for i from zero to k, we use the largest $i = k$ as a (tight) lower bound. Hence, the probability of matching is obtained by dividing the above formula (with $i = k$) by σ^m (the total number of possible text windows of length m), to obtain

$$f(m,k) \geq \binom{m}{k}\frac{1}{\sigma^{m-k}} \;=\; \frac{m^m}{k^k(m-k)^{m-k}\sigma^{m-k}\sqrt{m}}\;\Theta(1) \;=\; \left(\frac{1}{\alpha^{\frac{\alpha}{1-\alpha}}(1-\alpha)\sigma}\right)^{(1-\alpha)m} \Theta(m^{-1/2})$$

(where we used Stirling's approximation to the factorial). Since $e^{-1} \leq \alpha^{\frac{\alpha}{1-\alpha}} \leq 1$, the above expression can be lower bounded by $f(m,k) \geq \delta^m\; m^{-1/2}$, where

$$\delta = \left(\frac{1}{(1-\alpha)\sigma}\right)^{1-\alpha}$$

Therefore an upper bound for the maximum allowed value for α is $\alpha \leq 1 - 1/\sigma$, since otherwise we can prove that $f(m,k)$ is *not* exponentially decreasing on m (i.e. it is $\Omega(m^{-1/2})$).

Hence, the limit $\alpha < 1 - e/\sqrt{\sigma}$ corresponds to the maximum error level up to where we can prove that the algorithms based on filtration can work well, and we can prove that they cannot work well for $\alpha > 1 - 1/\sigma$.

We now verify this analysis empirically. The experiment we performed 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, finding how many text positions matched for each k value. We 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 for $m = 300$. The curve $\alpha = 1 - 1/\sqrt{\sigma}$ is included to show its closeness to the experimental data. Least squares give the approximation $\alpha^* = 1 - 1.09/\sqrt{\sigma}$, with a relative error smaller than 1%.

Figure 3 validates other theoretical assumptions. On the left we show that the matching probability undergoes a sharp increase at α^*. On the right we show that this point is essentially independent on m. Notice, however, that our assumptions are a bit optimistic since for short patterns the matching probability is somewhat higher.

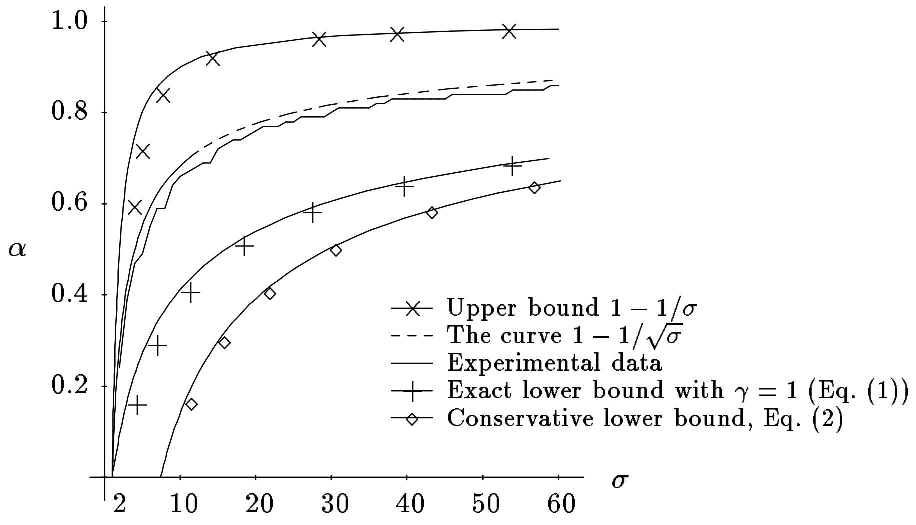


Figure 2: Theoretical and practical bounds for α, for $m = 300$.

3.2 Active Columns

A question related to the previous one is: which is the average number of automaton columns which have active states? That is, if we call c_r the smallest row with active states in column r of our NFA, which is the largest r satisfying $c_r \leq k$? Those columns satisfying $c_r \leq k$ are called *active*, and columns past the last active one need not be updated. Since our simulation avoids working on the inactive portions of the automaton, the question of the active columns is important for the average-case analysis of our algorithm (especially for partitioned automata).

Ours is not the first algorithm profiting from active columns. Ukkonen defined active columns in [29], and modified the dynamic programming algorithm so that it does not work past the last active column. The algorithm keeps track of the current last active column. At the end of each iteration this last column may increase in one (if a horizontal automaton arrow is crossed from the last active column to the next one), or may decrease in one or more (if the last active column runs out of active states, the next-to-last may be well before it). In this case the algorithm goes backward in the matrix looking at the new last active column.

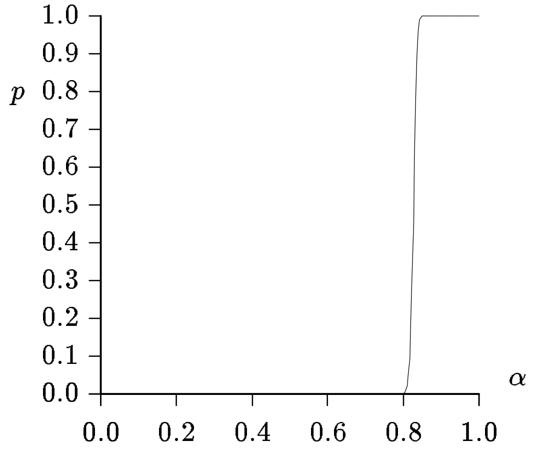
Ukkonen conjectured that the last active column was $O(k)$ on average and therefore his algorithm was $O(kn)$ on average. However, this was proved much later by Chang and Lampe [8]. We found in [3, 5] a tighter bound, namely

$$\frac{k}{1 - e/\sqrt{\sigma}} + O(1) \tag{3}$$

which is $O(k)$. The e of the formula has the same source as before and hence can be replaced by 1.09 in practice. By using least squares on experimental data we find that a very accurate formula is

$$0.9 \frac{k}{1 - 1.09/\sqrt{\sigma}} \tag{4}$$

with a relative error smaller than 3.5%.



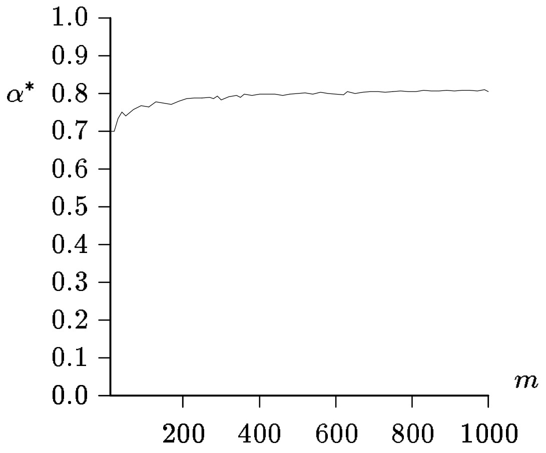
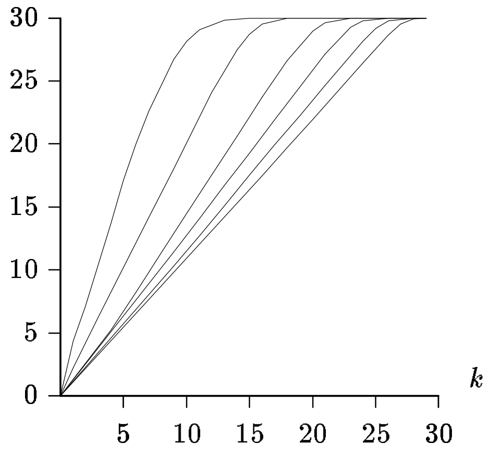


Figure 3: On the left, probability of an approximate match as a function of the error level ($m = 300$). On the right, maximum allowed error level as a function of the pattern length. Both cases correspond to random text with $\sigma = 32$.

Figure 4 (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. The results are the same for any k less than $m(1 - 1.09/\sqrt{\sigma})$.



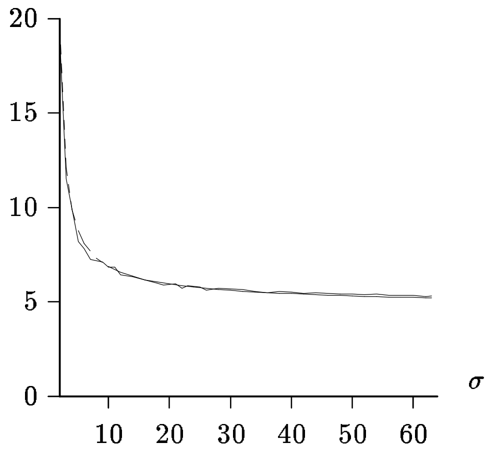


Figure 4: On the left, last active column for $\sigma = 2$, 4, 8, 16, 32 and 64 (curves read from left to right). On the right, last active column for $k = 5$, experimental (solid line) and theoretical (dashed line). We used $m = 30$.

4 Automaton Partitioning

This technique, presented in [3, 5], is the simplest way to extend the algorithm to handle longer patterns. We first present the general method and then optimize it.

4.1 General Method

If the automaton does not fit in a single word, we just partition it using a number of machine words for the simulation. Those subautomata behave differently than the simple one, since they must communicate their first and last diagonals with their lateral neighbors, as well as propagate active states down to the cells below (see Figure 5). We say that the automaton is partitioned in $I \times J$ "cells", arranged in I "d-rows" (set of rows packed in a cell) and J "d-columns" (set of automaton *diagonals* packed in a cell).

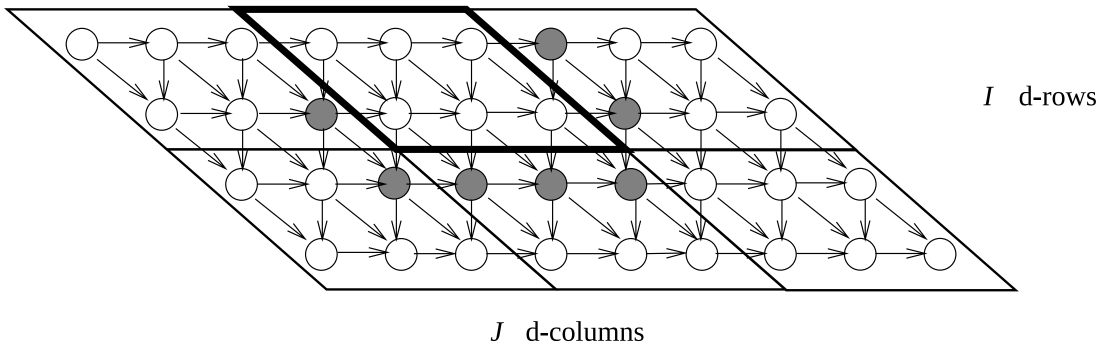


Figure 5: A 2×3 partitioned automaton where $\ell_c = 3$, $\ell_r = 2$, $I = 2$, $J = 3$. We selected a cell (bold edges) and shaded all the nodes of other cells affected by it. The bold-edged cell must communicate with those neighbors that own shaded nodes.

Let's suppose first 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. Note that we need that at least one automaton diagonal fits in a single machine word, i.e. $k + 2 \leq w$.

Suppose, on the other hand, that k is 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. These subautomata must propagate the ϵ-transitions down to all subsequent subautomata. 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. The correct choice is a matter of optimization.

If we divide the automaton in $I \times J$ 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 (see Section 3.2). The expense of working on less d-columns is having to keep account of the possible variation of the last active column for each text character.

4.2 Theoretical Analysis

Since automaton partitioning gives us some freedom to arrange the cells, we find out now the best arrangement.

In Section 3.2 we obtained the expected value for the last active column in the automaton (Eq. (3)). This measures active *columns* and we work on active *diagonals*. To obtain the last active diagonal we subtract k, to obtain that on average we work on $ke/(\sqrt{\sigma} - e)$ diagonals.

This is because the last active column depends on the error level k. Hence, at automaton row i (where only i errors are allowed) the last active column is $lcol(i) = i/(1 - e/\sqrt{\sigma})$. Hence, the last active column defines a diagonal line across the automaton whose slope is $1/(1 - e/\sqrt{\sigma})$. Figure 6 illustrates the situation. All the active states of the automaton are to the left of the dashed diagonal. The number of diagonals affected from the first one (thick line) to the dashed one is $k/(1 - e/\sqrt{\sigma}) - k$.

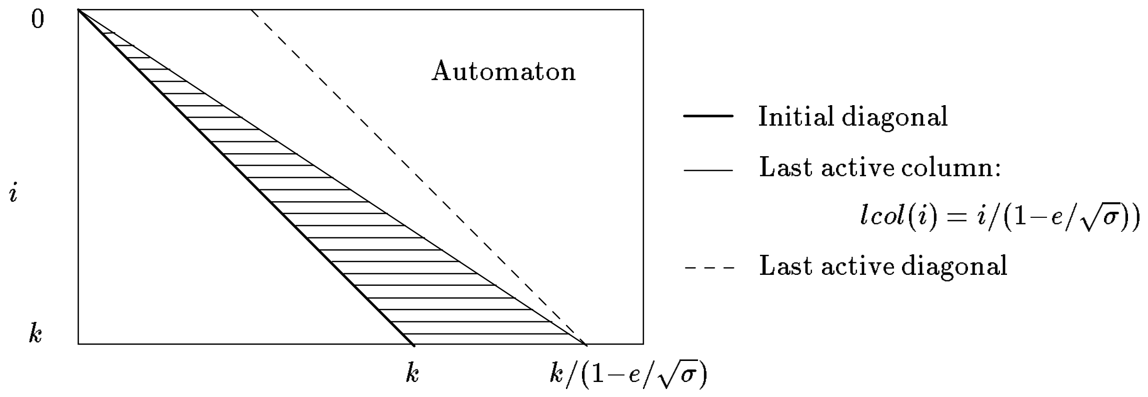


Figure 6: Converting active columns to active diagonals. The shaded area represents the active states of the automaton.

Since we pack $(m - k)/J$ diagonals in a single cell, we work on average on $ke/(\sqrt{\sigma} - e) \times J/(m - k)$ d-columns. Each d-column must work on all its I cells. On the other hand, there are only J d-columns. Hence our total complexity is

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

which shows that any choice for I and J is the same for a fixed IJ. Since $IJ \approx (m - k)(k + 2)/w$ (total number of bits to place divided by the size of the computer word), the final cost expression is independent (up to round-offs) of I and J:

$$\min\left(m - k \; , \; \frac{ke}{\sqrt{\sigma} - e}\right) \; \frac{k + 2}{w} \; n \qquad (5)$$

This formula has two parts. First, for $\alpha < 1 - e/\sqrt{\sigma}$, it is $O(k^2 n/(\sqrt{\sigma} w))$ time. Second, if the error ratio is high ($\alpha \geq 1 - e/\sqrt{\sigma}$), it is $O((m-k)kn/w)$. This last complexity is also the worst case of this algorithm. Recall that in practice the value e should be replaced by 1.09 and the average number of active columns is that of Eq. (4).

4.3 Practical Tuning

Since the gross analysis does not give us any clue about which is the optimal selection for I and J, we perform more detailed considerations.

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

$$(\ell_r + 1)\ell_c \leq w \;, \quad I = \left\lceil \frac{k+1}{\ell_r} \right\rceil \;, \quad J = \left\lceil \frac{m-k}{\ell_c} \right\rceil$$

Notice that in some configurations the cells are better occupied than in others, due to round-offs. That is, once we select ℓ_r and ℓ_c, the best possible packing leaves some bits unused, namely $w - (\ell_r + 1)\ell_c$.

Given the freedom that the above conditions give us, we compare now the alternatives we have, to find out the best one. One could, in fact, try every I and J and pick the configuration with less cells. Since we work proportionally to the number of cells, this seems to be a good criterion. Some configurations need more cells than others because, due to round-offs, they use less bits in each computer word (i.e. cell). In the worst possible configuration, $w/2 + 1$ bits can be used out of w, and in the best one all the w bits can be used. It is clearly not possible to use as few as $w/2$ bits or less, since in that case there is enough room to pack the bits of two cells in one, and the above equations would not hold. Hence, the best we can obtain by picking a different I and J is to reduce the number of cells by a factor of 2.

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 to 6 times faster than the general one), since some cases of communication with the neighbors are not present. Moreover, a more horizontal automaton makes the strategy of active columns work better.

This much faster update formula is more important than the possible 2-fold gains due to round-offs. Hence, we prefer to take minimal I, i.e.

$$I = \lceil (k+1)/(w-1) \rceil \;, \qquad \ell_r = \lceil (k+1)/I \rceil \;, \qquad \ell_c = \lfloor w/(\ell_r + 1) \rfloor \;, \qquad J = \lceil (m-k)/\ell_c \rceil$$

However, the three cases mentioned do not cover (d) a purely vertical partitioning, (i.e. $J = 1$), which is applicable whenever $2(m-k) \leq w$ and has also a simple update formula. The selection for vertical partitioning is $J_v = 1$, $\ell_{vc} = m - k$, $\ell_{vr} = \lfloor w/(m-k) \rfloor - 1$, $I_v = \lceil (k+1)/\ell_{vr} \rceil$. Figure 7 shows an experimental comparison between (c) and (d).

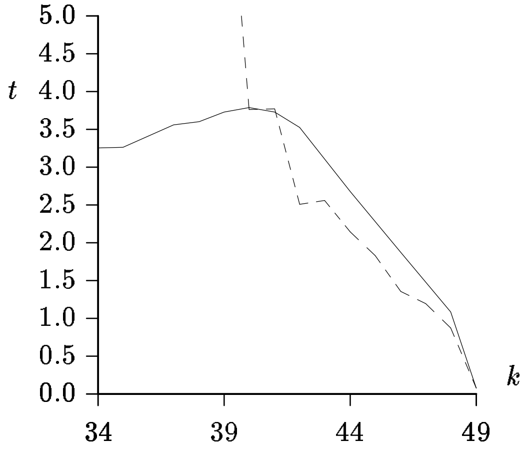


Figure 7: Time in seconds for vertical partitioning (dashed line) versus minimal rows partitioning (solid line). We use $m = 50$, $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

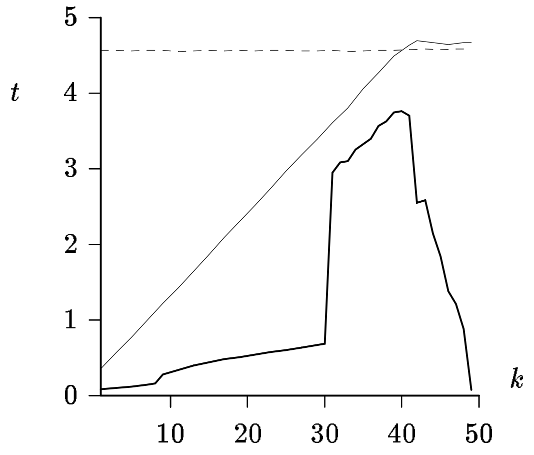
The mechanism we use to determine the optimal setup and predict its search cost integrates experimental and analytical results, as follows.

- We experimentally obtain the time that each type of automaton spends per text character (using least squares over real measures). We express those costs normalized so that the cost of the core algorithm is 1.00. These costs have two parts:
  - A base cost that does not depend on the number of cells: (a) 1.02, (b) 1.13, (c) 0.12, (d) 1.66.
  - A cost per processed cell of the automaton: (a) 1.25, (b) 0.83, (c) 2.27, (d) 1.36.
  - A cost spent in keeping account of which is the last active diagonal: (a) 0.68, (b) 0.20, (c) 1.66. Notice that although at a given text position this work can be proportional to the number of active columns, the amortized cost is $O(1)$ per text position. To see this, consider that at each text character we can at most increment in one the last active column, and therefore no more than n increments and n decrements are possible in a text of size n. Hence the correct choice is to consider this cost as independent on the number of cells of the automaton.
- We analytically determine using Eq. (4) the expected number of active d-columns.
- Using the above information, we determine whether it is convenient to keep track of the last active column or just modify all columns (normally the last option is better for high error ratios). We also determine which is the most promising partition.

Since this strategy is based on very well-behaved experimental data, it is not surprising that it predicts very well the cost of automaton partitioning and that it selected the best strategy in almost all cases we tried (in some cases it selected a strategy 5% slower than the optimal, but not more).

Finally, notice that the worst case complexity of $O(k(m-k)/w)$ per inspected character is worse than the $O(m)$ of dynamic programming when the pattern length gets large, i.e. $m > w/(\alpha(1-\alpha))$. This ensures that automaton partitioning is better for $m \leq 4w$, which is quite large. In fact, we should also account for the constants involved. The constant for partitioned automata is nearly twice as large as that of dynamic programming, which makes sure that this method is better for $m \leq 2w$. We use therefore a partitioned automaton instead of dynamic programming as our verification engine for potential matches in the sections that follow.

Figure 8 shows an experimental comparison between plain dynamic programming, the Ukkonen cutoff variant [29] and our partitioned automaton for large patterns. In the worst moment of the partitioned automaton, it is still faster than dynamic programming up to $m = 60$, which confirms our assumptions. The peaks in the right plot is not due to variance, but to integer round-offs which are inherent to our algorithm (this is explained more in detail in Section 8).



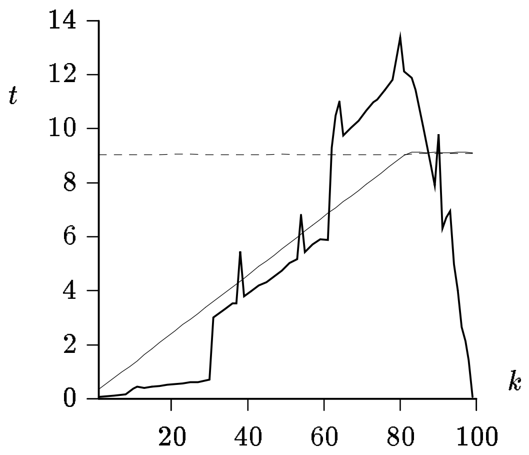


Figure 8: Time in seconds for partitioned automaton (thick line) versus dynamic programming (dashed line) and the Ukkonen's improvement (solid thin line). The left plot is for $m = 50$ and the right one for $m = 100$. We use $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

4.4 Improving Register Usage

We finish this section explaining an improvement in the engineering of the algorithms that leads to triplicating the performance in some cases. The improvement is based on better usage of the computer registers.

The main difference in the cost between the core algorithm and an horizontally partitioned automaton is that in the first case we can put in a register the machine word which simulates the automaton. This cannot be done in a partitioned automaton, since we use an array of words. The locality of accesses of those words is very low, i.e. if there are a active d-columns, we update for each text character all the words from the first one to the a-th. Hence, we cannot keep them in registers.

An exception to the above statement is the case $a = 1$. This represents having active only the first cell of the horizontal automaton. We can, therefore, put that cell in a register and traverse the text updating it, until the last diagonal inside the cell becomes active. At that point, it is possible that the second cell will be activated at the next character and we must resume the normal searching with the array of cells. We can return to the one-cell mode when the second cell becomes inactive again.

With this technique, the search cost for a pattern is equal to that of the core algorithm until the second automaton is activated, which in some cases is a rare event. In fact, we must adjust the above prediction formulas, so that the horizontal automata cost the same as the core algorithm (1.00), and we add the above computed cost only whenever their last diagonal is activated. The probability of this event is $f(\ell_c + k, k)$.

This technique elegantly generalizes a (non-elegant) truncation heuristic proposed in earlier work. It stated that, for instance, if we had $m = 12$, $k = 1$, better than partitioning the automaton in two we could just truncate the pattern in one letter, use the core algorithm and verify each occurrence. With the present technique we would automatically achieve this, since the last letter will be isolated in the second cell of the horizontal automaton.

Notice that this idea cannot be applied to the case $I > 1$, since in that case we have always more than one active cell. In order to use the technique also for this case, and in order to extend the idea to not only the first cell, we could develop specialized code for two cells, for three cells, and so on, but the effort involved and the complexity of the code are not worth it.

Figure 9 shows the improvements obtained over the old version. The better register usage is more noticeable for low error levels (horizontal partitioning). This version of our partitioned automaton is automatically determining whether to use the speedup technique of the end of Section 2 or not.

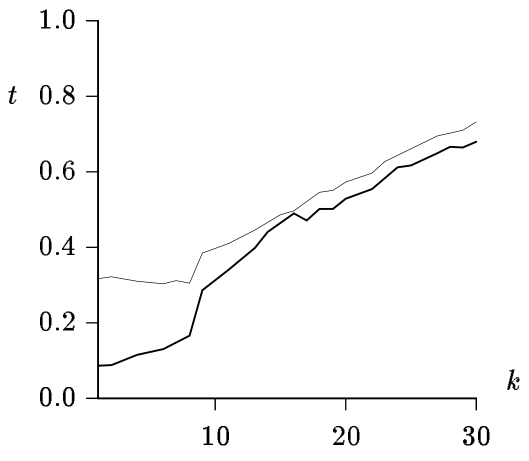


Figure 9: Time in seconds for partitioned automata before (thin line) and after (thick line) improving register usage. We use $m = 60$, $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

5 Pattern Partitioning

We present now a different technique to cope with long patterns. This technique was developed in [3, 5], and is improved here. We first explain the general method and then optimize it.

5.1 General Method

The following lemma, proved in [3, 5], suggests a way to partition a large problem into smaller ones.

Lemma: If $segm = Text[a..b]$ matches $patt$ with k errors, and $patt = P_1 ... P_j$ (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.

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. Suppose we find at text position i the end of a match for the subpattern ending at position s in the pattern. Then, the potential match must be searched in the area between positions $i - s + 1 - k$ and $i - s + 1 + m + k$ of the text, an $(m + 2k)$-wide area. This checking must be done with an algorithm resistant to high error levels, such as our partitioned automaton[1].

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

If we partition the pattern in j parts, we have to perform j searches. Moreover, those searches will together trigger more verifications as j grows (i.e. a piece split in two will trigger all the verifications triggered by the original piece plus spurious ones). This fact is reflected in the formula for the match probability of Section 3.1 (Eq. (1)), since the match probability is now $O(\gamma^{m/j})$, which may be much larger than $O(\gamma^m)$ even for a single piece. Therefore, we prefer to keep j small.

A first alternative is to make j just large enough for the subproblems to fit in a computer word, that is

$$j^* = \min \left\{ j \;/\; \left(\left\lceil \frac{m}{j} \right\rceil - \left\lfloor \frac{k}{j} \right\rfloor \right) \left(\left\lfloor \frac{k}{j} \right\rfloor + 2 \right) \leq w \;\wedge\; \left\lfloor \frac{m}{j} \right\rfloor > \left\lfloor \frac{k}{j} \right\rfloor \right\}$$

where the second guard 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 if $k < m$. Solving the above equation (disregarding roundoffs) we obtain

$$j^* = \frac{m - k + \sqrt{(m-k)^2 + wk(m-k)}}{w} \quad = \quad m\; d(w, \alpha)$$

[1]In fact, we used plain dynamic programming in previous work, but as shown in Section 4.4 the partitioned automaton is faster except for very long patterns. As we see shortly, however, we elaborate more on this verification technique.

where

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

As a function of α, $d(w,\alpha)$ is convex and is maximized for $\alpha = 1/2\ (1 - 1/(\sqrt{w}-1))$, where it takes the value $1/(2(\sqrt{w}-1))$. To give an idea of the reduction obtained, this maximum value is 0.11 for $w = 32$ and 0.07 for $w = 64$.

Excluding verifications, the search cost is $O(j^*n)$. For very low error ratios ($\alpha < 1/w$), $j^* = O(m/w)$ and the cost is $O(mn/w)$. For higher error ratios, $j^* = O(\sqrt{mk/w})$ and then the search cost is $O(\sqrt{mk/w}\ n)$. Both cases can be obviously bounded by $O(mn/\sqrt{w})$.

A second alternative is to use a smaller j (and therefore the automata still do not fit in a computer word) and combine this technique with automaton partitioning for the subpatterns. We consider this alternative next.

5.2 Optimal Selection for j

It is possible to use just automaton partitioning (Section 4) to solve a problem of any size. It is also possible to use just pattern partitioning, with j large enough for the pieces to be tractable with the kernel algorithm directly (i.e. $j = j^*$).

It is also possible to merge both techniques: partition the pattern into pieces. Those pieces may or may not be small enough to use the kernel algorithm directly. If they are not, search them using automaton partitioning. This has the previous techniques as particular cases.

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 (using Eq. (5))

$$\frac{\frac{ke/j}{\sqrt{\sigma}-e}\ (k/j+2)}{w}\ jn \quad = \quad \frac{ke(k/2+2)}{(\sqrt{\sigma}-e)w}\ n$$

which shows that the lowest cost is obtained with the largest j value, and therefore $j = j^*$ is the best choice.

However, this is just an asymptotic result. In practice the best option is more complicated due to simplifications in the analysis, constant factors, and integer roundoffs. For instance, a pattern with 4 pieces can be better searched with two horizontal automata of size $(I = 1, J = 2)$ than with four simple automata (especially given the improvements of Section 4.4). The cost of each automaton depends heavily on its detailed structure. Therefore, to determine the best option in practice we must check all the possible j values, from 1 to j^* and predict the cost of each strategy. This cost accounts for running j automata of the required type (which depends on j), as well as for the cost to verify the potential matches multiplied by their probability of occurrence (using Eq. (1)).

5.3 A Hierarchical Verification Algorithm

The original proposal for pattern partitioning (presented in [3, 5]) stopped working long before the limit $\alpha < 1 - 1.09/\sqrt{\sigma}$, as it can be seen in the original references and in Figure 11. This was because all the pattern was verified whenever any piece matched. Hence, the total cost for verifications for a *single* piece was $O(m^2 \gamma^{m/j^*})$. For that cost to be $O(1)$, we need $\gamma \leq 1/m^{2j^*/m}$, i.e.

$$\alpha \leq 1 - \frac{e}{\sqrt{\sigma}} m^{\frac{j*}{m-k}} = 1 - \frac{e}{\sqrt{\sigma}} m^{\frac{2d(w,\alpha)}{1-\alpha}}$$

which clearly decreases as m grows. Therefore, the original method degraded for longer patterns. This was caused mainly because a large pattern was verified although the probability to verify it increased with j^* (i.e. with m).

We propose now a different verification technique which does not degrade as the pattern gets longer. The idea is to try to quickly determine that the match of the small piece is not in fact part of a complete match. A technique similar to this hierarchical verification was mentioned in [16], in the context of indexed searching.

First assume that j is a power of 2. Then, we recursively split the pattern in two halves of size $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$ (halving also the number of errors, i.e. $\lfloor k/2 \rfloor$) until the pieces are small enough to be searched with the core algorithm (i.e. $(m-k)(k+2) \leq w$, where m and k are the parameters for the subpatterns). Those pieces (leaves of the tree) are searched in the text. Each time a leaf reports an occurrence, its parent node checks the area looking for its pattern (whose size is close to twice the size of the leaf pattern). Only if the parent node finds the longer pattern, it reports the occurrence to its parent, and so on. The occurrences reported by the root of the tree are the final answers.

This construction is correct because the partitioning lemma applies to each level of the tree, i.e. any occurrence reported by the root node *must* include an occurrence reported by one of the two halves, so we search both halves. The argument applies then recursively to each half.

Figure 10 illustrates this concept. If we search the pattern "aaabbbcccddd" with four errors in the text "xxxbbxxxxxxx", and split the pattern in four pieces, the piece "bbb" will be found in the text. In the original approach, we would verify the complete pattern in the text area, while with the new approach we verify only its parent "aaabbb" and immediately determine that there cannot be a complete match.

In the Appendix (Eq. (8)) we analyze this method and show that the total amount of verification work for each piece is $O((m/j)^2 \gamma^{m/j})$. This is much better than $O(m^2 \gamma^{m/j})$, and in particular it is $O(1)$ whenever $\gamma < 1$. Hence, with this verification method the acceptable error level does not degrade as the pattern grows.

If j is not a power of two we try to build the tree as well balanced as possible. This is because an unbalanced tree will force the verification of a long pattern because of the match of a short pattern (where the long pattern is more than twice as long as the short one). The same argument shows that it is not a good idea to use ternary or higher arity trees. Finally, we could increase j to have a perfect binary partition, but the shorter pieces trigger more verifications.

In order to handle partitions which are not a power of two, we need a stronger version of the

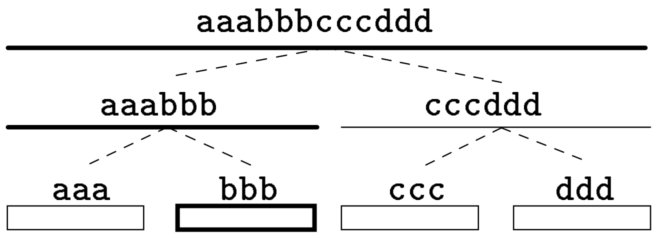


Figure 10: The hierarchical verification method. The boxes (leaves) are the elements which are really searched, and the root represents the whole pattern. At least one pattern at each level must match in any occurrence of the complete pattern. If the bold box is found, all the bold lines may be verified.

partitioning lemma of Section 5.1. For instance, if we determine $j = 5$, we have to partition the tree in, say, a left child with three pieces and a right child with two pieces. The standard partitioning lemma tells us that each subtree could search its pattern with $\lfloor k/2 \rfloor$ errors, but this will increase the verifications of the subtree with the shorter pattern. In fact, we can search the left subtree with $\lfloor 3k/5 \rfloor$ errors and the right one with $\lfloor 2k/5 \rfloor$ errors. Continuing with this policy we arrive to the leaves, which are searched with $\lfloor k/5 \rfloor$ errors each as expected. The stronger version of the Lemma follows

Stronger Lemma: If $segm = Text[a..b]$ matches $patt$ with k errors, and $patt = P_1...P_j$ (a concatenation of subpatterns), then $segm$ includes a segment that matches at least one of the P_i's, with $\lfloor a_i k/A \rfloor$ errors, where $A = \sum_{i=1}^{j} a_i$.

Proof: Otherwise, each P_i matches with at least $\lfloor a_i k/A \rfloor + 1 > a_i k/A$ errors. Summing up the errors of all the pieces we have more than $Ak/A = k$ errors and therefore a match is not possible.

Although when there are few matches (i.e. low error level) plain and hierarchical verification behave similarly, there is an important difference for medium error levels: hierarchical verification is more tolerant to errors. We illustrate this fact in Figure 11. As it can be seen, both methods eventually are overwhelmed by verifications *before* reaching the limit $\alpha = 1 - 1.09/\sqrt{\sigma}$. This is because, as j grows, the cost of verifications $O((m/j)^2 \gamma^{m/j})$ increases. In the case $\sigma = 32$, the theoretical limit is $\alpha^* = 0.83$ (i.e. $k = 50$), while the plain method ceases to be useful for $k = 35$ (i.e. $\alpha = 0.58$) and the hierarchical one works well up to $k = 42$ (i.e. $\alpha = 0.7$). For English text the limit is $\alpha^* = 0.69$, while the plain method works up to $k = 30$ ($\alpha = 0.50$) and the hierarchical one up to $k = 35$ ($\alpha = 0.58$).

It is also noticeable that hierarchical verification works a little harder in the verifications once they become significative (very high error levels). This is because the hierarchy of verifications makes it to check many times the same text area. On the other hand, we notice that the use of partitioned automata instead of dynamic programming for the verification of possible matches is especially advantageous in combination with our hierarchical verification, since in most cases we verify only a short pattern, where the automaton is much faster than dynamic programming.

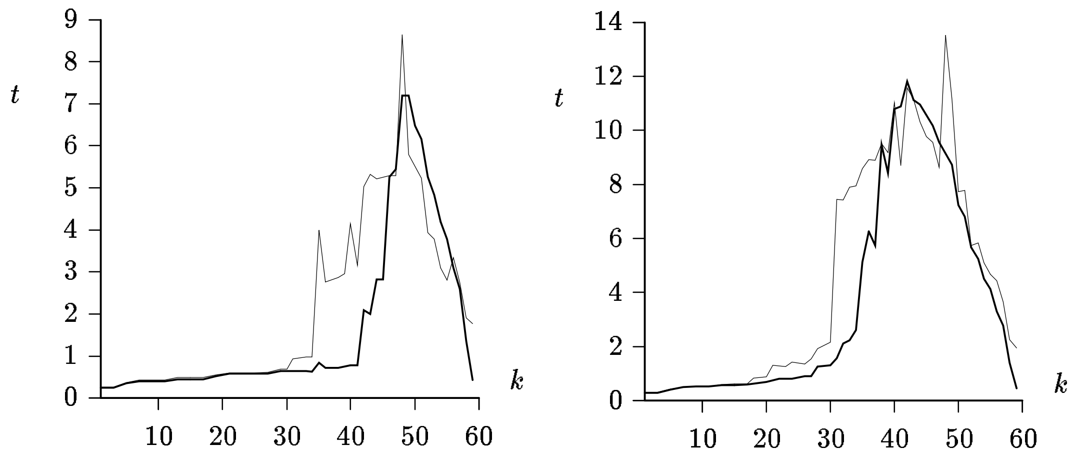


Figure 11: Time in seconds for pattern partitioning using plain (thin line) and hierarchical (thick line) verification. We use $m = 60$, $w = 32$, and $n = 1$ Mb. On the left, random text ($\sigma = 32$). On the right, English text.

6 Superimposed Automata

This technique was first presented in [4] for multipattern approximate search, and integrated into the single-pattern algorithm in [5]. We first explain it and then find the optimal form to use it.

6.1 General Method

When we use pattern partitioning, the search is divided into a number of subsearches for smaller patterns $P_1, ..., P_j$. The aim of this technique is to avoid searching each subpattern separately, by collapsing a number r of searches in a single one.

In pattern partitioning all the patterns have almost the same length. If they differ (at most in 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 (Section 2), and then take the bitwise-*or* of all the tables. The resulting $t[\ ]$ table matches in its position i with the i-th character of *any* of the patterns involved. 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, an extended pattern, see the end of Section 2), namely $C_1...C_m$, with $C_i = \{P_1[i], ..., P_r[i]\}$. For example, if the search is for `patt` and `wait`, the string `watt` is accepted with *zero* errors (see Figure 12). Each occurrence reported by the automaton has to be verified for all the patterns involved.

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

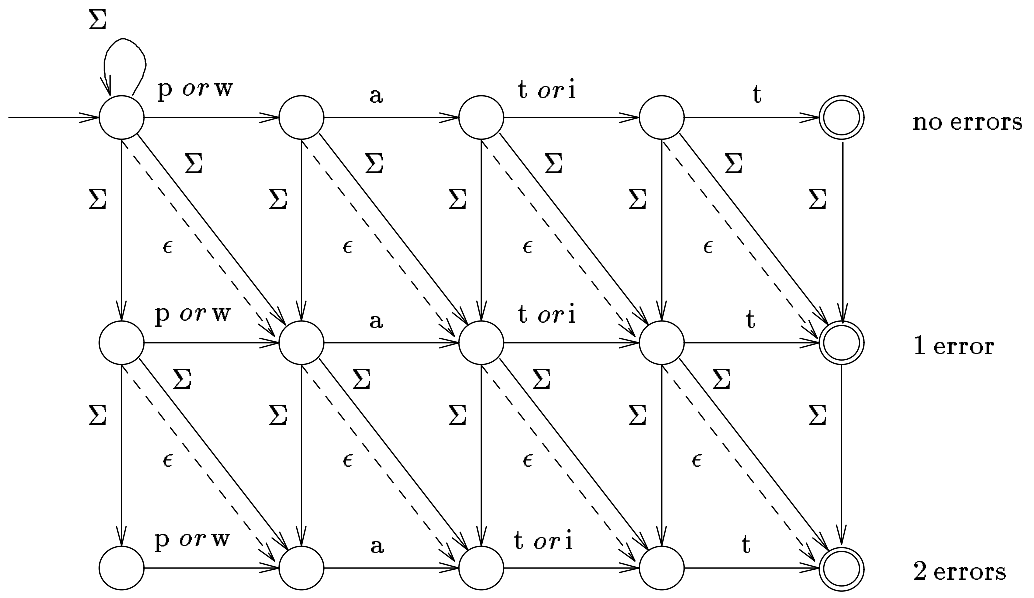


Figure 12: An NFA to filter the parallel search of `patt` and `wait`.

ones. This limits the amount of possible superimposition.

If we use pattern partitioning in j pieces and superimpose in groups of r pieces, we must perform $\lceil j/r \rceil$ superimposed searches. We keep the groups of almost the same size, namely $\lfloor j/\lceil j/r \rceil \rfloor$ and $\lceil j/\lceil j/r \rceil \rceil$.

We group subpatterns which are contiguous in the pattern. When an occurrence is reported we cannot know which of the superimposed subpatterns caused the match (since the mechanism does not allow to know), so we check whether the concatenation of the subpatterns appears in the area. From that point on, we use the normal hierarchical verification mechanism.

6.2 Optimizing the Amount of Superimposition

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, with the consequent increase in the cost of verifications. We analyze now how many patterns can we superimpose.

As shown in Section 3.1 (Eq. (1)), the probability of a given text position matching a random pattern is $O(\gamma^m)$, where γ depends on α and σ. This cost 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 random characters being equal). To extend this result, we notice that we have r characters on each edge now, so the above mentioned probability is $1-(1-1/\sigma)^r \approx r/\sigma$. The (pessimistic) approximation is tight for $r << \sigma$. We use the approximation because in practice r will be quite modest compared to σ.

Hence, the value of γ when superimposing r patterns (which we call γ' to keep unchanged the old

γ value) is

$$\gamma' \;=\; \left(\frac{r}{\sigma \alpha^{\frac{2\alpha}{1-\alpha}} (1-\alpha)^2}\right)^{1-\alpha} \;=\; r^{1-\alpha}\,\gamma \qquad (6)$$

and therefore the new limit for α is

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

or alternatively the limit for r (i.e. the maximum amount of superimposition r_{lim} that can be used given the error level) is

$$r_{lim} \;=\; \frac{\sigma\ (1-\alpha)^2}{e^2}$$

which for constant error level is $O(\sigma)$ independent on m. However, this is not the only restriction on r.

If we use pattern partitioning in j pieces and superimpose in groups of r pieces, we must perform j/r superimposed searches. In the last part of the Appendix (Eq. (9)) we show that the expected cost due to verifications is $O(\gamma'^{\ell} r^2 \ell^2)$ per search, where $\ell = m/j$. For this cost to be $O(1)$ we need a new (stricter) condition on r. This is obtained by expanding γ' using Eq. (6):

$$\left(\frac{re^2}{\sigma(1-\alpha)^2}\right)^{\ell(1-\alpha)} r^2\ell^2 = O(1)$$

which yields

$$r^* = \Theta\left(\frac{r_{lim}}{(r_{lim}\ell)^{\frac{2}{2+\ell(1-\alpha)}}}\right) \;=\; \Theta\left(r_{lim}^{\,1-\frac{2}{2+\ell(1-\alpha)}}\right)$$

which approaches r_{lim} for large ℓ (i.e. partitioning the pattern into less pieces). In the analysis that follows we make the simplifying assumption $r^* = r_{lim}$[2].

Notice that superimposition may give more arguments to partition a pattern in $j < j^*$ pieces. On the other hand, thanks to the new verification mechanism of Section 5.3 we can superimpose more patterns than in the original work, which translates into better performance everywhere, not only when the error level is becoming high.

Considering the above limit, the total search cost becomes $1/r^* = O(1/(\sigma\ (1-\alpha)^2))$ times that of pattern partitioning. For instance, if we partition in j^* pieces (so that they can be searched with the core algorithm), the search cost becomes

$$O\left(\frac{m\ d(w,\alpha)}{\sigma(1-\alpha)^2}\ n\right)$$

which for $\alpha \le 1/w$ is $O(mn/(w\sigma))$, and for higher error level becomes $O(\sqrt{mk/(w\sigma)}\ n)$ (this is because $1-\alpha$ is lower bounded by $e/\sqrt{\sigma}$). Again, a general bound is $O(mn/\sqrt{w\sigma})$.

[2] A recent work on multipattern approximate searching shows that by applying the idea of hierarchical verification to the number r of patterns we achieve in fact $r^* = r_{lim}$, since the cost to verify r superimposed patterns does not depend on r anymore [6].

A natural question is for which error level can we superimpose all the j^* patterns to perform just one search, i.e. when $r^* = j^*$ holds. That is

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

whose approximate solution is

$$\alpha \;\; < \;\; \alpha_1 \;\; = \;\; 1 - \frac{e^2 m}{\sigma\sqrt{w}} \tag{7}$$

where as always we must replace e by 1.09 in practice. As we see in the experiments, this bound is pessimistic because of the roundoff factors which affect j^* for medium-size patterns.

Notice that superimposition stops working when $r^* = 1$, i.e. when $\alpha = 1 - e/\sqrt{\sigma}$. This is the same point when pattern partitioning stops working. We show in Figure 13 the effect of superimposition on the performance of the algorithm and its tolerance to the error level. As we see in Section 8, we achieve almost constant search time until the error level becomes medium. This is because we automatically superimpose as much as possible given the error level.

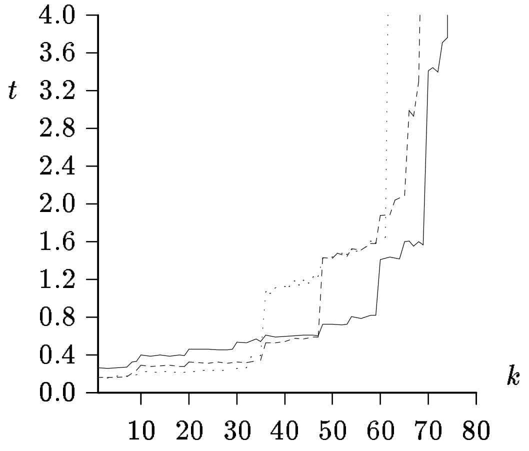


Figure 13: Times in seconds for superimposed automata. Superimposition is forced to $r = 2$ (solid line), 4 (dashed line) and 6 (dotted line). The larger r, the faster the algorithm but it stops working for lower error levels. We use $m = 100$, $w = 32$, and $n = 1$ Mb and random text and patterns with $\sigma = 32$.

6.3 Optimal Grouping and Aligning

Two final aspects allow further optimization. A first one is that it is possible to try to form the groups so that the patterns in each group are similar (e.g. they are at small edit distance among them, or they share letters at the same position). This would decrease the probability of finding spurious matches in the text. A possible disadvantage of this heuristic is that since the subpatterns are not contiguous we cannot simply verify whether their concatenation appears, but we have to check if any of the corresponding leaves of the tree appears. The probability that the concatenation appears is much lower.

A second one is that, since we may have to prune the longer subpatterns of each group, we can determine whether to eliminate the first or the last character (the patterns differ at most in one), using the same idea of trying to make the patterns as similar as possible.

None of these heuristics have been tested yet.

7 Combining All the Techniques

At this point, a number of techniques have been described, analyzed and optimized. They can be used in many combinations for a single problem. A large pattern can be split in one or more subpatterns (the case of "one" meaning no splitting at all). Those subpatterns can be small enough to be searched with the kernel algorithm or they can be still large and need to be searched with a partitioned automaton. Moreover, we can group those automata (simple or partitioned) to speed up the search by using superimposition.

The analysis helped us to find more efficient verification techniques and to determine the cases where each technique can be used. However, a number of questions still arise. Which is the correct choice to split the pattern versus the size of the pieces? Is it better to have less pieces or smaller pieces? How does the superimposition affect this picture? Is it better to have more small pieces and superimpose more pieces per group or is it better to have larger pieces and smaller groups?

We study the optimal combination in this section. We begin showing the result of a theoretical analysis and then explain the heuristic we use.

7.1 A Theoretical Approach

The analysis recommends using the maximal possible superimposition, $r = r^*$, to reduce the number of searches. As proved in Section 5.2, it also recommends to use the maximal $j = j^*$. This gives the following combined (simplified) average complexity for our algorithm, illustrated in Figure 14:

- If the problem fits in a machine word (i.e. $(m-k)(k+2) \leq w$), the core algorithm is used at $O(n)$ average and worst-case search cost.
- If the error level is so low that we can cut the pattern in j^* pieces and superimpose all them (i.e. $\alpha < \alpha_1$, Eq. (7)) then superimposed automata gives $O(n)$ average search cost.
- If the error level is not so low but it is not too high (i.e. $\alpha < \alpha^*$, Eq. (2)), then use pattern partitioning in j^* parts, to obtain $O(\sqrt{mk/(w\sigma)}\ n)$ average search cost.
- If the error level is too high (i.e. $\alpha > \alpha^*$) we must use automaton partitioning at $O(k(m-k)n/w)$ average and worst-case search cost.

On the other hand, the worst-case search cost is $O(k(m-k)/w\ n)$ in all cases. This is the same worst-case cost of the search using the automaton. This is because we use such an automaton to verify the matches, and we never verify a text position twice with the same automaton. We keep the state of the search and its last text position visited to avoid backtracking in the text due to overlapping verification requirements. This argument is valid even with hierarchical verification.

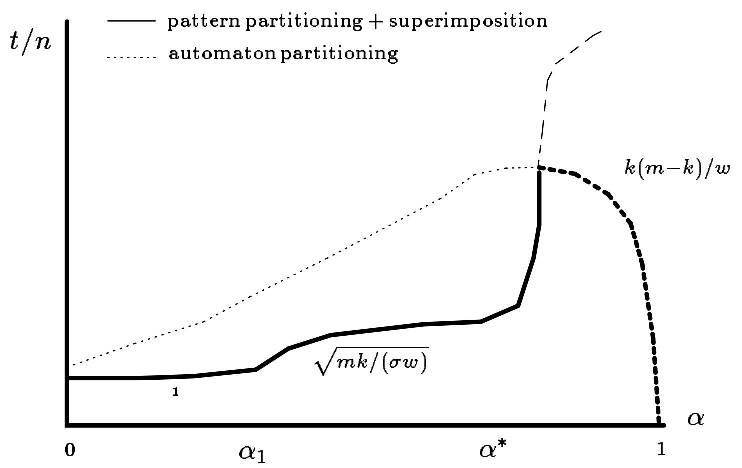


Figure 14: The simplified complexity of our algorithm.

7.2 A Practical Heuristic and a Searching Software

Clearly the theoretical analysis alone is insufficient at this point. The results are asymptotic and do not account for many details which are important in practice, such as roundoffs and constant factors.

The real costs are so complex that the best way to find the optimal combination relies on trying all the possible values of j, from 1 to j^* and for r, from 1 to j. For each possible value of r and j, we compute the cost of performing the $\lceil j/r \rceil$ searches with simple or partitioned automata as explained in Section 4.3. We also account for the probability of matching the (possibly superimposed) automata in the text, which is obtained from Eq. (6), as well as the cost of such verification. This is an inextricable mix of theoretical and empirical results. This prediction algorithm costs $O(k^2)$, which is quite modest. Its outcome is not only the recommended combination of techniques to use, but also the expected cost of the search.

This heuristic algorithm has been implemented as a software system, which is publicly available from `http://www.dcc.uchile.cl/~gnavarro/pubcode`. This software uses the techniques in an optimal way, but it also allows to force the use of any combination for test purposes. It also allows to force or avoid using the twist mentioned at the end of Section 2 (to compare against other algorithms that could also use it). It reports the combination of parameters used, the time spent in the search and the number of matches found. It can optionally print those matches. Currently the software needs to be provided with the value of σ (more exactly, with the inverse of the probability that two random letters match, which is σ on random text and close to 13 in lowercase English text). We plan in the future a self-adjusting feature that makes it able not only of determine the type of text it is in, but also to change the strategy if the selected combination proves bad.

8 Experimental Comparison

In this section we experimentally compare our combined heuristic against the fastest previous algorithms we are aware of. Since we compare only the fastest algorithms, we leave aside [21, 28, 11, 14, 25, 31, 33, 29, 24], which are not competitive in the range of parameters we study here. Our algorithm is shown using and not using speedup explained at the end of Section 2, since it could be applied to many other algorithms as well (but generally not to filtration algorithms).

We tested random patterns against 10 Mb of random text on a Sun UltraSparc-1 of 167 MHz running Solaris 2.5.1, with 64 Mb of RAM. This is a 32-bit machine, i.e. $w = 32$. We use $\sigma = 32$. We also tested lower-case 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. To check the influence of the fact that we selected the patterns from the same text (and therefore there are always matches) we also tested the effect of selecting the patterns from another text, but there were no noticeable differences.

Each data point was obtained by averaging the Unix's user time over 20 trials. We present all the times in seconds per megabyte of text. The standard deviation of the results is below 5%. We measure preprocessing and searching time together, since preprocessing time is totally negligible. The slowest preprocessing time was 2 milliseconds, which is less than 1% of the fastest searching time on 10 Mb.

The algorithms included in this comparison are (in alphabetical code order)

Agrep [32] is a widely distributed exact and approximate search software oriented to natural language text. It is limited (although not intrinsically) to $m \leq 32$ and $k \leq 8$.

BM is a filter based on applying a Boyer-Moore-type machinery [27]. The code is from the authors.

BPM (bit-parallel matrix) is a recent work [17] based on the bit-parallel simulation of the dynamic programming matrix. The code is from the author and has different versions for one and for multiple machine words.

Count is a counting filter proposed in [12], which slides a window over the text counting the number of letters in the text window that are present in the pattern. When the number is high enough, the area is verified. We use our own variant, implemented in [18] (window of fixed size).

CP is the column partitioning algorithm (kn.clp) of [8], which computes only the places where the value of the dynamic programming matrix does not change along each column. The code is from the authors.

DFA converts the NFA into a deterministic automaton which is computed in lazy form. The algorithm is proposed in [13] and studied more in detail in [19], whose implementation we use.

EP (exact partitioning) is the filtering algorithm proposed in [33] which splits the pattern in $k+1$ pieces and searches them using a Boyer-Moore multipattern algorithm, as suggested in [7]. The code is ours and uses an extension of the Sunday [23] algorithm.

Four-Russians applies a Four Russians technique to pack many automaton transitions in computer words. The code is from the authors [34], and is used with $r = 5$ as suggested in their paper (r is related with the size of the Four Russians tables).

NFA - NFA/NS is our combined heuristic, with and without the speed-up technique.

Figure 15 shows the results for random text with $\sigma = 32$. As it can be seen, our algorithm is more efficient than any other when the problem fits in a single word ($m = 9$), except for low error level, where EP is unbeaten. For longer patterns, our algorithm is the fastest ones up to shortly after $\alpha = 1/2$. Again, EP is the exception, since it is faster up to $\alpha = 1/3$ approximately. For $\alpha > 1/2$, BPM is the fastest one, except when the pattern is longer than w letters and the error level is high. In this final case, Four Russians is the winner.

Figure 16 shows the results for English text. The results are similar but the allowed error ratios are reduced: our algorithm is the fastest up to $\alpha = 1/3$ approximately, except for EP which is faster for $\alpha \leq 1/5$. Agrep is also very efficient for low error levels, quite close to EP. The strange behavior for Agrep occurs because as soon as it finds a match in the line it reports the line and abandons the search of that line, hence being faster for very high error ratios.

Finally, Figure 17 shows the results for long patterns and fixed error level. The results show that for long patterns our algorithm and BPM are the fastest if the error level is not too high. For low error levels the algorithm EP is better, but it degrades as m grows.

The reader may be curious about the strange behavior of some of the curves in our algorithms. Those are not caused by statistical deviations in the tests but are due to integer round-offs, which are intrinsic to our algorithms. 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. The use of the smart heuristic eliminates most of those peaks, but some remain.

We end by noticing that as a general rule, the use of the speedup technique described at the end of Section 2 seems to pay off up to the same point where the EP algorithm stops working. We analyzed in [5] this point, obtaining that it is close to $1/(3 \log_\sigma m)$. We use this value to determine whether or not to use the technique.

9 Conclusions and Future Work

We presented a case study of analysis of algorithms applied to the improvement and tuning of a practical algorithm. The algorithm consists of a fast kernel to search a pattern in a text allowing errors which works only for short patterns, and a number of techniques to extend it to longer patterns. The techniques can be applied in combination and their optimal interplay was not trivial to deduce.

We used the analysis and the empirical data to obtain the optimal combination and showed experimentally the correctness of the conclusions. This work shows an excellent example of a complex and theoretical analysis of algorithms applied to practical algorithm engineering.

We obtained also a significant improvement in the error level which our algorithms tolerate (thanks to a new scheme to verify potential matches) and in the practical performance of many algorithms (thanks to better usage of the machine registers). The new performance figures could not have been obtained by just tuning the original algorithms. The combined heuristic was converted into a software which is publicly available.

A number of options have resisted the analysis and are open to experimentation of different heuristics. These are related to the optimal grouping and alignment of patterns for superimposed automata and are subject of future work. We also plan to develop a self-adjusting heuristic able to dynamically determine the best combination to use, so that it is able to modify its choice in the middle of the search if the selected combination is not working well.

We are also working on applying the hierarchical verification technique to filtration algorithms in order to improve their tolerance to errors.

Acknowledgments

We thank all the people who sent us working versions of their algorithms, what made the tests a lot easier and, in some sense, more fair: William Chang, Alden Wright, Gene Myers, Erkki Sutinen, Tadao Takaoka and Jorma Tarhio. We also thank the anonymous referees, which made a careful work and gave us a number of suggestions that improved this paper.

References

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

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

[3] 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, 1996.

[4] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. of WADS'97*, LNCS 1272, pages 174–184, 1997.

[5] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 1998. To appear.

[6] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. Technical Report TR/DCC-98-10, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/multi.ps.gz`.

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

[8] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, 1992.

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

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

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

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

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

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

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

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

[17] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, number 1448 in LNCS, pages 1–13, 1998.

[18] G. Navarro. Multiple approximate string matching by counting. In *Proc. of Fourth South American Workshop on String Processing (WSP'97)*, pages 125–139. Carleton University Press, 1997.

[19] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. of Fourth South American Workshop on String Processing (WSP'97)*, pages 112–124. Carleton University Press, 1997.

[20] 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.

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

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

[23] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.

[24] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In P. Spirakis, editor, *Proc. ESA'95*, LNCS 979. Springer-Verlag, 1995.

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

[26] 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, 1990.

[27] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM J. on Computing*, 22(2):243–260, 1993.

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

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

[33] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.

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

Appendix: Analysis of the Verification Algorithm

We analyze the average amount of verification work to perform per character for each subpattern. We assume that $j = 2^r$ for integer r, and that $m = 2^r \ell$. Hence, we search j subpatterns of length ℓ. Recall that since we search patterns of length m/j with k/j errors, the error level is almost the same for any subproblem. Hence, the probability for any subpattern of length m' to match in the text is $O(\gamma^{m'})$ (where $\gamma < 1$, Eq. (1)). To verify the occurrence of a pattern of length $\ell = m/j$, an area of length $\ell' = m/j + 2k/j = O(\ell)$ must be checked. The cost of that verification is $O(\ell^2)$.

Consider a given subpattern of length ℓ. It matches a given text position with probability $O(\gamma^\ell)$. This match causes its parent to perform a verification in an area of length $2\ell'$ (since the parent is of length 2ℓ). With some probability the parent pattern is not found and the verification ends here. Otherwise the parent is found and we must proceed upward in the tree. The probability of having to continue the verification is

$$P(\text{parent node / child node}) \;=\; \frac{P(\text{parent} \;\wedge\; \text{child})}{P(\text{child})} \;\leq\; \frac{P(\text{parent})}{P(\text{child})} \;=\; \frac{\gamma^{2\ell}}{\gamma^{\ell}} \;=\; \gamma^{\ell}$$

and therefore with probability γ^{ℓ} we pay the next verification which spans an area of length $4\ell'$, and so on. Notice that the next verification will find the longer pattern with probability $\gamma^{2\ell}$.

This process continues until we either find the complete pattern or we fail to find a subpattern. The total amount of work to perform is

$$\gamma^{\ell}\;(\;(2\ell)^2\;+\;\gamma^{\ell}\;(\;(4\ell)^2\;+\;\gamma^{2\ell}\;(\;(8\ell)^2\;+\ldots)))\;\;=\;\;\gamma^{\ell}\;(2\ell)^2\;+\;\gamma^{2\ell}\;(4\ell)^2\;+\;\gamma^{4\ell}\;(8\ell)^2\;+\ldots$$

which formally is

$$\sum_{i=1}^{r} \gamma^{\ell 2^{i-1}}\;(2^i\ell)^2 \;\;=\;\; 4\ell^2 \sum_{i=0}^{r-1} 4^i \left(\gamma^{\ell}\right)^{2^i}$$

The summation can be bounded with an integral to find that it is between $C(\ell) - C(m)$ and $\gamma^{\ell} + C(\ell) - C(m/2)$, where $C(x) = \gamma^x(x\ln(1/\gamma) - 1)/(\ell^2 \ln^2(1/\gamma)\ln 2)$. Therefore, the summation is $\Theta(\gamma^{\ell})$ and the total verification cost is

$$O(\ell^2\;\gamma^{\ell}) \tag{8}$$

We now generalize the above analysis by assuming that the verification tree is pruned when the patterns are of length ℓr, and the subtrees are searched using superimposed automata, which match with probability γ'^{ℓ} (for $\gamma' < 1$, Eq. (6)).

Once a leaf is found, we must verify an area of length ℓr to determine whether their concatenation appears. That concatenation is found with probability $O(\gamma^{\ell r})$. When it is found, its parent is verified (an area of length $2\ell r$), which continues the verification with probability $\gamma^{2\ell r}$. Hence, the verification cost per piece is

$$\gamma'^{\ell}\;4(\ell r)^2\;(1 + \gamma^{\ell r} + C(\ell r) - C(m/2)) \;\;=\;\; O(\gamma'^{\ell}\ell^2 r^2(1+\gamma^{\ell r})) \;\;=\;\; O(\gamma'^{\ell}\ell^2 r^2) \tag{9}$$

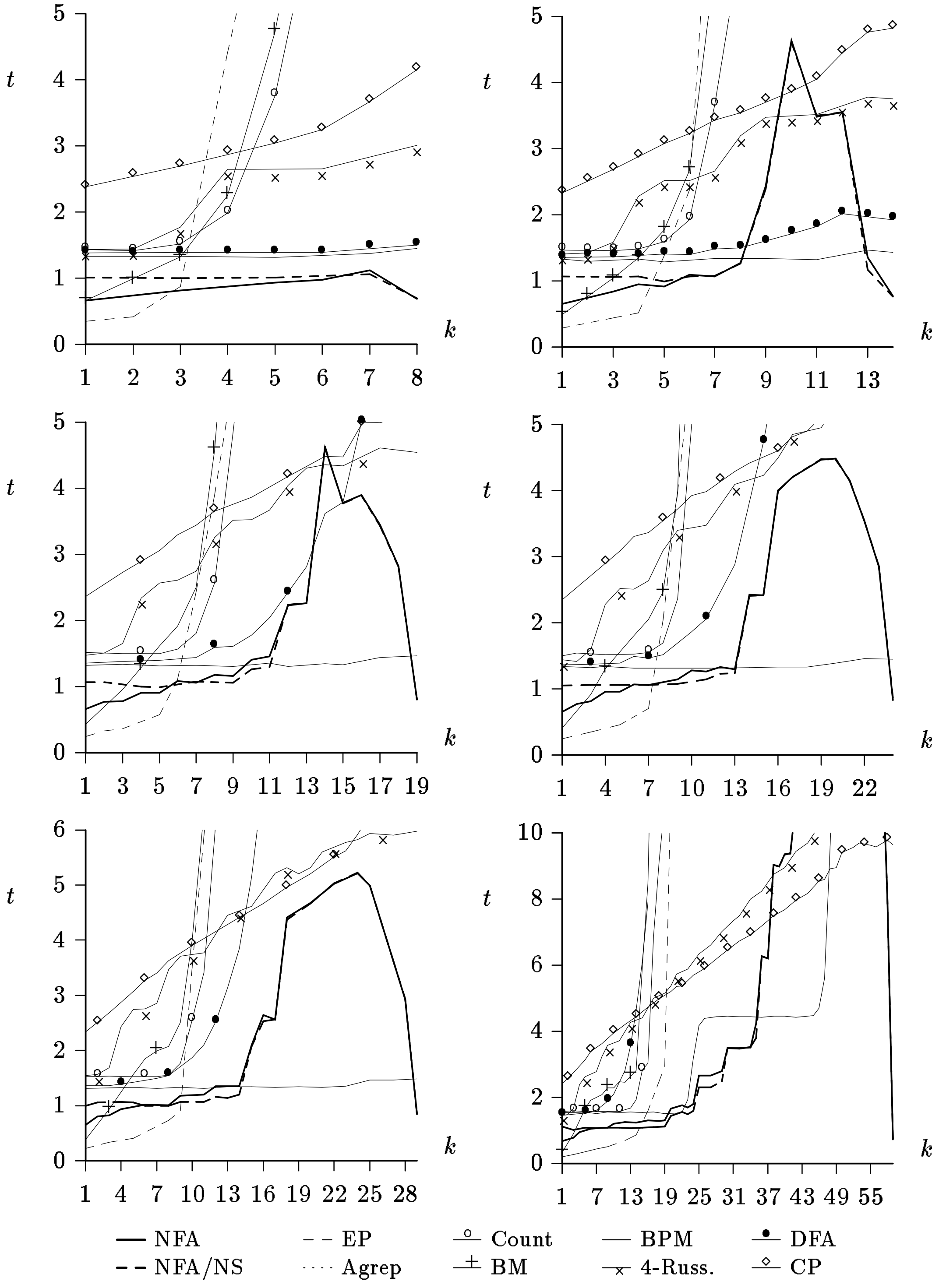


Figure 15: Experimental results for random text ($\sigma = 32$). From top to bottom and left to right, m =9, 15, 20, 25, 30 and 60.

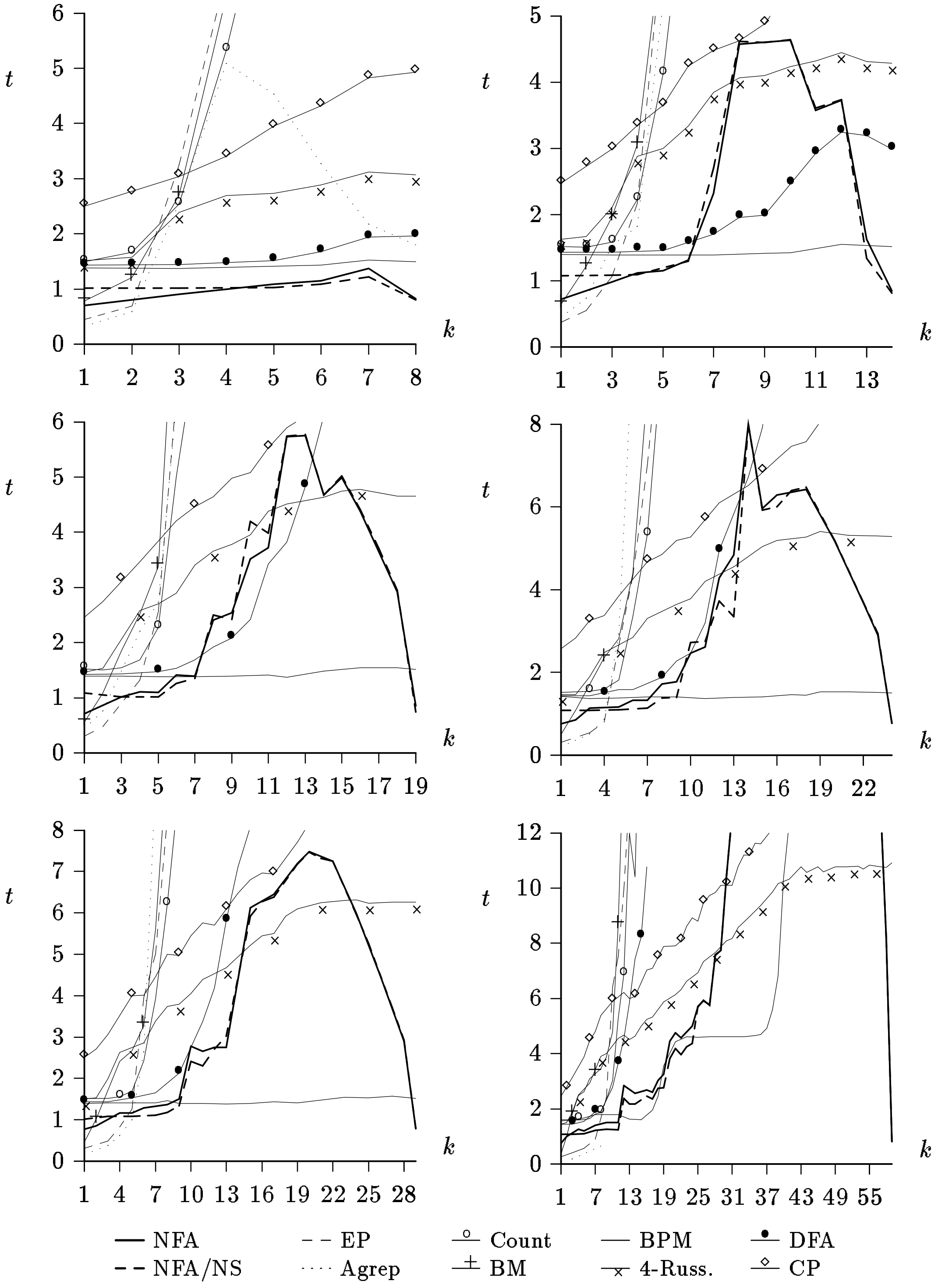


Figure 16: Experimental results for English text. From top to bottom and left to right, m =9, 15, 20, 25, 30 and 60.

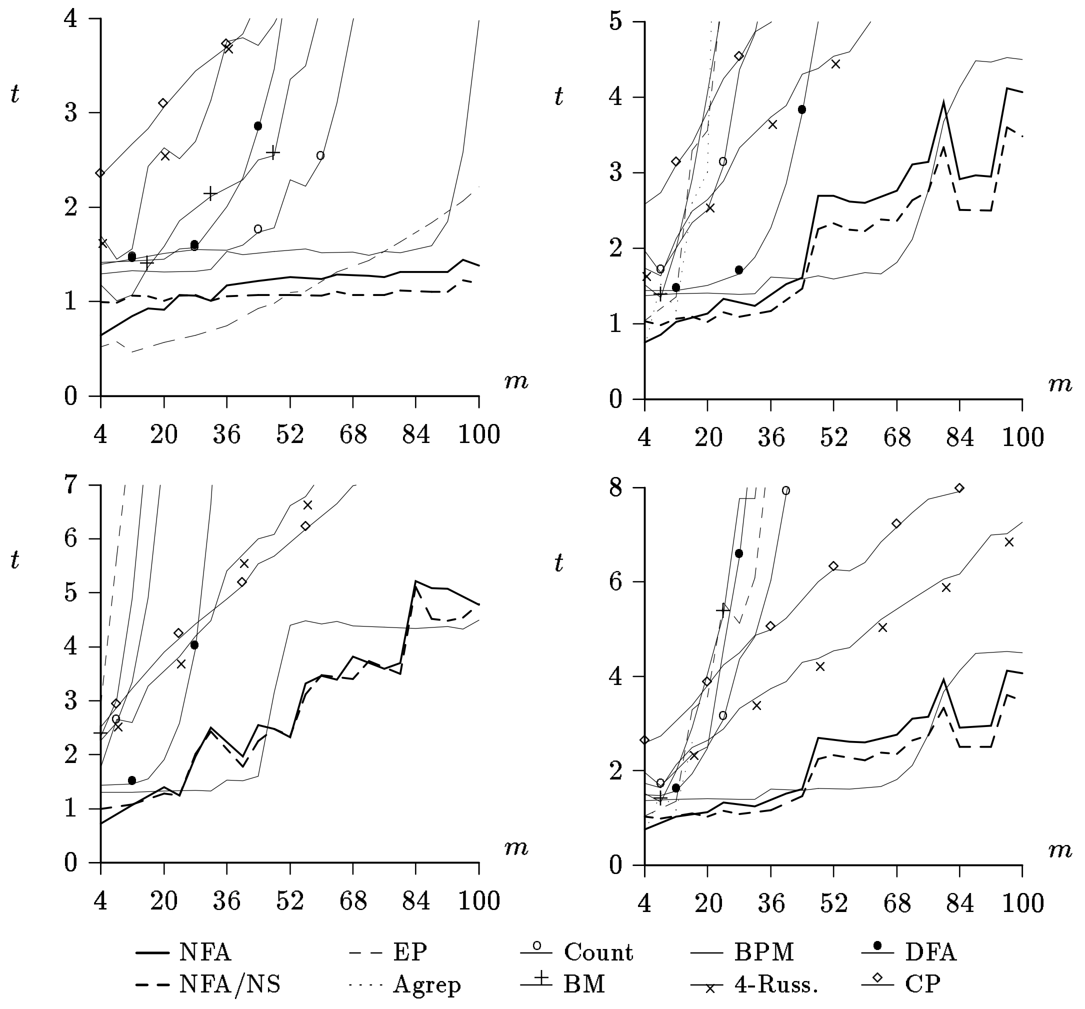


Figure 17: Experiments for long patterns. On the left, random text ($\sigma = 32$), on the right, English text. The plots on the top are for $\alpha = 0.25$ and those on the bottom are for $\alpha = 0.5$.