# Improved Single and Multiple Approximate String Matching

Kimmo Fredriksson[1] * and Gonzalo Navarro[2] **

[1] Department of Computer Science, University of Joensuu
kfredrik@cs.joensuu.fi
[2] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

**Abstract.** We present a new algorithm for multiple approximate string matching. It is based on reading backwards enough $\ell$-grams from text windows so as to prove that no occurrence can contain the part of the window read, and then shifting the window. Three variants of the algorithm are presented, which give different tradeoffs between how much they work in the window and how much they shift it. We show analytically that two of our algorithms are optimal on average. Compared to the first average-optimal multipattern approximate string matching algorithm [Fredriksson and Navarro, CPM 2003], the new algorithms are much faster and are optimal up to difference ratios of $1/2$, contrary to the maximum of $1/3$ that could be reached in previous work. This is also a contribution to the area of single-pattern approximate string matching, as the only average-optimal algorithm [Chang and Marr, CPM 1994] also reached a difference ratio of $1/3$. We show experimentally that our algorithms are very competitive, displacing the long-standing best algorithms for this problem. On real life texts, our algorithms are especially interesting for computational biology applications.

## 1 Introduction and Related Work

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 $T_{1...n}$, a pattern $P_{1...m}$, and a maximal number of differences permitted, $k$, we want to find all the text positions where the pattern matches the text up to $k$ differences. The differences can be substituting, deleting or inserting a character. We call $\alpha = k/m$ the *difference ratio*, and $\sigma$ the size of the alphabet $\Sigma$. For the average case analyses it is customary to assume a random text over a uniformly distributed alphabet.

A natural extension to the basic problem consists of *multipattern searching*, that is, searching for $r$ patterns $P^1 \dots P^r$ simultaneously in order to report all their occurrences with at most $k$ differences. This has also several applications such as virus and intrusion detection, spelling, speech recognition, optical

* Supported by the Academy of Finland, grant 202281.
** Partially supported by Fondecyt grant 1-020831.

character recognition, handwriting recognition, text retrieval under synonym or thesaurus expansion, computational biology, multidimensional approximate matching, batch processing of single-pattern approximate searching, etc. Moreover, some single-pattern approximate search algorithms resort to multipattern searching of pattern pieces. Depending on the application, $r$ may vary from a few to thousands of patterns. The naive approach is to perform $r$ separate searches, so the goal is to do better.

The single-pattern problem has received a lot of attention since the sixties [10]. For low difference ratios (the most interesting case) the so-called *filtration algorithms* are the most efficient ones. These algorithms discard most of the text by checking for a necessary condition, and use another algorithm to verify the text areas that cannot be discarded. For filtration algorithms, the two important parameters are the filtration speed and the maximum difference ratio $\alpha$ up to which they work.

In 1994, Chang & Marr [2] showed that the average complexity of the problem is $O((k + \log_\sigma m)n/m)$, and gave the first (filtration) algorithm that achieved that average-optimal cost for $\alpha < 1/3 - O(1/\sqrt{\sigma})$.

The multipattern problem has received much less attention, not because of lack of interest but because of its difficulty. There exist algorithms that search permitting only $k = 1$ difference [8], that handle too few patterns [1], and that handle only low difference ratios [1]. No effective algorithm exists to search for many patterns with intermediate difference ratio. Moreover, as the number of patterns grows, the difference ratios that can be handled get reduced, as the most effective algorithm [1] works for $\alpha < 1/\log_\sigma(rm)$.

Very recently [4], we have proposed the first average-optimal multiple approximate string matching algorithm, with average complexity $O((k+\log_\sigma(rm))n/m)$. It consists of a direct extension of Chang & Marr algorithm to multiple patterns. The algorithm performs well in practice. However, just as Chang & Marr on single patterns, it can only cope with difference ratios $\alpha < 1/3$, and even lower in practice due to memory limitations. Both algorithms are based on reading enough $\ell$-grams (substrings of length $\ell$) from text windows so as to prove that no occurrence can contain the part of the window read, and then shifting the window.

In this paper we present a new algorithm in this trend. Compared to our previous algorithm [4], the present one is algorithmically more novel, as it is not a direct extension of an existing single pattern algorithm but a distinct approach. We use a sliding window rather than fixed windows. This permits us having a window length of $m - k$ instead of $(m - k)/2$, which is the crux of our ability to handle higher difference ratios. In order to maximize shifts on a sliding window, we have to read it backwards, which was not an issue in [2, 4]. In this sense, our algorithm inherits from ABNDM algorithm [11]. We present also two variants of our algorithm, which trade amount of work done inside a window by amount of window shifting obtained.

The analysis of the new algorithms is more difficult than previous ones [2, 4]. We manage to prove that two of our new algorithms are also average-optimal.

However, they are superior to previous ones because they are optimal for difference ratios up to $\alpha < 1/2$, while previous work permitted up to $\alpha < 1/3$. Hence, the new algorithms not only improve our previous algorithm but also, in the area of single pattern matching, are better than what was achieved by any previous average-optimal algorithm.

In practice, our new algorithms are much faster than our previous algorithm [4], and they are also faster than all other multipattern search algorithms for a wide range of parameters that include many cases of interest in computational biology (DNA and protein searching). Moreover, our new algorithms are also relevant in the much more competitive area of single pattern matching algorithms, where they turn out to be the fastest for low difference ratios.

## 2 Our Algorithm

### 2.1 Basic Version

Given $r$ search patterns $P^1 \dots P^r$, the preprocessing fixes value $\ell$ (to be discussed later) and builds a table $D : \Sigma^\ell \to \mathbb{N}$ telling, for each possible $\ell$-gram, the minimum number of differences necessary to match the $\ell$-gram *anywhere* inside *any* of the patterns.

The scanning phase proceeds by sliding a window of length $m-k$ over the text. The invariant is that any occurrence starting before the window has already been reported. For each window position $i+1 \dots i+m-k$, we read successive $\ell$-grams backwards: $S^1 = T_{i+m-k-\ell+1 \dots i+m-k}$, $S^2 = T_{i+m-k-2\ell+1 \dots i+m-k-\ell}$, $\dots$, $S^t = T_{i+m-k-t\ell+1 \dots i+m-k-(t-1)\ell}$, and so on. Any occurrence starting at the beginning of the window must fully contain those $\ell$-grams. We accumulate the $D$ values for the successive $\ell$-grams read, $M_u = \sum_{1 \le t \le u} D[S^t]$. If, at some point, the sum $M_u$ exceeds $k$, then it is not possible to have an occurrence containing the sequence of $\ell$-grams read $S^u \dots S^1$, as merely matching those $\ell$-grams inside *any* pattern in *any* order needs more than $k$ differences.

Therefore, if at some point we obtain $M_u > k$, then we can safely shift the window to start at position $i+m-k-u\ell+2$, which is the first position not containing the $\ell$-grams read.

On the other hand, it might be that we read all the $U = \lfloor (m-k)/\ell \rfloor$ $\ell$-grams fully contained in the window without surpassing threshold $k$. In this case we must check the text area of the window with a non-filtration algorithm, as it might contain an occurrence. We scan the text area $T_{i+1 \dots i+m+k}$ for each of the $r$ patterns, so as to cover any occurrence starting at the beginning of the window and report any match found. Then, we shift the window by one position and resume the scanning.

This scheme may report the same ending position of occurrence several times, if there are several starting positions for it. A way to avoid this is to remember the last position scanned when verifying the text, so as to prevent retraversing the same text areas but just restarting from the point we left the last verification. We use Myers' algorithm [9] for the verification of single patterns, which makes

the cost $O(m^2/w)$ per pattern, $w$ being the number of bits in the computer word. Otherwise, the standard $O(m^2)$ dynamic programming could be used [12].

Several optimizations are proposed over our previous algorithm [4]: optimal choice of $\ell$-grams, hierarchical verification, reduction of preprocessing costs, and bit-parallel packing of counters. All these can be equally applied to our new algorithm, but we have omitted this discussion for lack of space. In the experiments, however, we have applied all these improvements except the optimal choice of $\ell$-grams, that did not work well in practice. The preprocessing cost with hierarchical verification is $O(r\sigma^\ell m/w)$.

### 2.2 A Stricter Matching Condition

In the basic algorithm we permit that the $\ell$-grams match anywhere inside the patterns. This has the disadvantage of being excessively permissive. However, it has an advantage: When the $\ell$-grams read accumulate more than $k$ differences, we know that no pattern occurrence can contain them *in any position*, and hence can shift the window next to the first position of the leftmost $\ell$-gram read. We show now how the matching condition can be made stricter without losing this property.

First consider $S^1$. In order to shift the window by $m-k-\ell+1$, we must ensure that $S^1$ cannot be contained in any pattern occurrence, so the condition $D[S^1] > k$ is appropriate. If we consider $S^2 : S^1$, to shift the window by $m-k-2\ell+1$, we must ensure that $S^2 : S^1$ cannot be contained inside any pattern occurrence. The basic algorithm uses the sufficient condition $D[S^1] + D[S^2] > k$.

However, a stricter condition can be enforced. In an approximate occurrence of $S^2 : S^1$ inside the pattern, where pattern and window are aligned at their initial positions, $S^2$ cannot be closer than $\ell$ positions from the end of the pattern. Therefore, for $S^2$ we precompute a table $D_2$, which considers its best match in the area $P_{1\ldots m-\ell}$ rather than $P_{1\ldots m}$. In general, $S^t$ is input to a table $D_t$, which is preprocessed so as to contain the number of differences of the best occurrence of its argument inside $P_{1\ldots m-(t-1)\ell}$, for any of the patterns $P$. Hence, we will shift the window to position $i+m-k-u\ell+2$ as soon as we find the first (that is, smallest) $u$ such that $M_u = \sum_{t=1}^{u} D_t[S^t] > k$.

The number of tables is $U = \lfloor (m-k)/\ell \rfloor$ and the length of the area for $D_U$ is at most $2\ell - 1$. Fig. 1(a) illustrates.

Since $D_t[S] \geq D[S]$ for any $t$ and $S$, the smallest $u$ that permits shifting the window is never smaller than for the basic method. This means that, compared to the basic method, this variant never examines more $\ell$-grams, verifies more windows, nor shifts less. So this variant can never work more than the basic algorithm, and usually works less. In practice, however, it has to be shown whether the added complexity, preprocessing cost and memory usage of having several $D$ tables instead of just one, pays off. We consider this issue in Section 4. The preprocessing cost is increased to $O(r(\sigma^\ell(m/w+m)+U)) = O(r\sigma^\ell m)$.

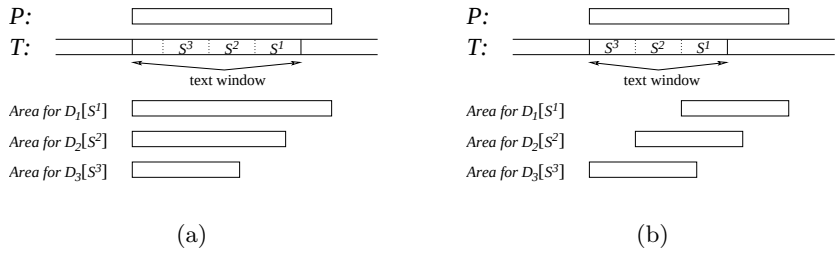


**Fig. 1.** Pattern $P$ aligned over a text window. (a): The text window and $\ell$-grams for the algorithms of Secs. 2.1 and 2.2. The window is of length $m-k$. The areas correspond to the Sec. 2.2. All the areas for $S^t$ for the basic algorithm are the same as for $D_1$. (b): The areas for the algorithm of Sec. 2.3. The areas overlap by $\ell+k-1$ characters. The window length is $U\ell$.

### 2.3 Shifting Sooner

We now aim at abandoning the window as soon as possible. Still the more powerful variant developed above is too permissive in this sense. Actually, the $\ell$-grams should match the pattern more or less at the same position they have in the window.

This fact has been previously used in algorithm LAQ [13]. The idea is that we are interested in occurrences of $P$ that start in the range $i-\ell+1 \ldots i+1$. Those that start before have already been reported and those that start after will be dealt with by the next windows. If the current window contains an approximate occurrence of $P$ beginning in that range, then the $\ell$-gram at window position $(t-1)\ell+1 \ldots t\ell$ can only match inside $P_{(t-1)\ell+1 \ldots t\ell+k}$.

We have $U = \lfloor (m-k-\ell+1)/\ell \rfloor$ tables $D_t$, one per $\ell$-gram position in the window. Table $D_{U-t+1}$ gives distances to match the $t$-th window $\ell$-gram, so it gives minimum distance in the area $P_{(t-1)\ell+1 \ldots t\ell+k}$ instead of in the whole $P$, see Fig. 1(b).

At a given window, we compute $M_u = \sum_{0 \le t < u} D_t[S^t]$ until we get $M_u > k$ and then shift the window. It is clear that $D_t$ is computed over a narrower area that before, and therefore we detect sooner that the window can be shifted. The window is shifted sooner, working less per window.

The problem this time is that it is not immediate how much can we shift the window. The information we have is only enough to establish that we can shift by $\ell$. Hence, although we shift the window sooner, we shift less. The price for shifting sooner has been too high.

A way to obtain better shifting performance resorts to bit-parallelism. Values $D_t[S]$ are in the range $0 \ldots \ell$. Let us define $l$, as the number bits necessary to store one value from each table $D_t$. If our computer word contains at least $Ul$ bits, then the following scheme can be applied.

Let us define table $D[S] = D_1[S] : D_2[S] : \ldots : D_U[S]$, where the $U$ values have been concatenated, giving $l$ bits to each, so as to form a larger number ($D_1$ is in the area of the most significant bits of $D$). Assume now that we accumulate

$M_u = \sum_{t=1}^{u}(D[S^t] << (t-1)l)$, where "$<< (t-1)l$" shifts all the bits to the left by $(t-1)l$ positions, enters zero bits from the right, and discards the bits that fall to the left. The leftmost field of $M_u$ will hold the value $\sum_{t=1}^{u} D_t[S^t]$, that is, precisely the value that tells us that we can shift the window when it exceeds $k$. Similar idea was briefly proposed in [13], but their (single pattern) proposal was based on direct extension of Chang & Marr algorithm [2].

In general, the $s$-th field of $M_u$, counting from the left, contains the value $\sum_{t=s}^{u} D_t[S^{t-s+1}]$. In order to shift by $\ell$ after having read $u$ $\ell$-grams, we need to ensure that the window $S^U \ldots S^1$ contains more than $k$ differences. A sufficient condition is the familiar $\sum_{t=1}^{u} D_t[S^t] > k$, that is, when the leftmost field exceeds $k$. In order to shift by $2\ell$, we need also to ensure that the window $S^{U-1} \ldots S^1 S^0$ contains more than $k$ differences. Here $S^0$ is the next $\ell$-gram we have not yet examined. A lower bound to the number of differences in that window is $\sum_{t=2}^{u} D_t[S^{t-1}] > k$, where the summation is precisely the content of the second field of $M_u$, counting from the left. In general, we can shift by $s\ell$ whenever all the $s$ leftmost fields of $M_u$ exceed $k$.

Let us consider the value for $l$. The values in $D_t$ are in the range $0 \ldots \ell$. In our $M_u$ counters we are only interested in the range $0 \ldots k+1$, as knowing that the value is larger than $k+1$ does not give us any further advantage. We will manage to keep our counters in the range $0 \ldots k+1$, so after computing $M_{u+1} = M_u + (D[S^{u+1}] << ul)$, the counters may have reached $k+1+\ell$. So we define $l = 1 + \lceil \log_2(k+\ell+2) \rceil$. We actually store value $2^{l-1} - (k+1) + c$, where $c$ is the value of the counter. Therefore, the counters activate their highest bits whenever they exceed $k$, and they still can reach $k+\ell+1$ without overflowing. After the sum occurs, we have to reset them to $2^{l-1}$ if they are not smaller than $2^{l-1}$. A simple way to do this is $X \leftarrow M_u \;\&\; (10^{l-1})^U$, $M_u \leftarrow M_u \;\&\; \sim (X - (X >> (l-1)))$. The "&" is the bitwise "and", and $(10^{l-1})^U$ is a constant with bits sets at the highest counter positions, so $X$ has bits set at the highest bits of the exceeded counters. Later, $X - (X >> (l-1))$ has all (but the highest) bits set in the fields of the exceeded counters. The "$\sim$" operation reverses the bits, so at the end we clean the trailing bits of the counters that exceed $2^{l-1}$, leaving them exactly at $2^{l-1}$.

The same mask $X$ serves for the purpose of determining the shift. We want to shift by $s\ell$ whenever the $s$ leftmost highest counter bits in $X$ are set. Let us rather consider $Y = X \hat{} (10^{l-1})^U$, where "$\hat{}$" is the bitwise "xor" operation, so we now we are interested in how many leftmost highest counter bits in $Y$ are *not* set. This means that we want to know which is the leftmost set bit in $Y$. If this corresponds to the $s$-th counter, then we can shift by $(s-1)\ell$. But the leftmost bit set in $Y$ is at position $y = \lfloor \log_2 Y \rfloor$, so we shift by $(U-1-y/l)\ell$. The logarithm can be computed fast by casting the number to float (which is fast in modern computers) and then extracting the exponent from the standardized real number representation.

If there are less than $Ul$ bits in the computer word, there are several alternatives: space the $\ell$-grams in the window by more than $\ell$, prune the patterns to reduce $m$, or resort to using more computer words for the counters. In our

implementation we have used a combination of the first and last alternatives: for long patterns we use simulated 64 bit words (in our 32 bit machine), directly supported by the compiler. If this is not enough, we use less counters, i.e. use spacing $h$, where $h > \ell$, for reading the $\ell$-grams. This requires that the areas are $\ell + h - 1 + k$ characters now, instead of $2\ell - 1 + k$, and there are only $\lfloor (m - k - h + 1)/h \rfloor$ tables $D_t$. On the other hand, this makes the shifts to be multiples of $h$.

The same idea can be applied for multiple patterns as well, by storing the minimum distance over all the patterns in $D_t$. The preprocessing cost is this time $O(r(\sigma^\ell(m/w + m))) = O(r\sigma^\ell m)$, computing the minimum distances bit-parallelly.

This variant is able of shifting sooner than previous ones. In particular, it never works more than the others in a window. However, even with the bit-parallel improvement, it can shift less. The reason is that it may shift "too soon", when it has not yet gathered enough information to make a longer shift. For example, consider the basic algorithm with $D[S^1] = 1$ and $D[S^2] = k$. It will examine $S^1$ and $S^2$ and then will shift by $m - k - 2\ell + 1$. If the current variant finds $D_1[S^1] = k + 1$ and $D_t[S^t] = k$ for $t \geq 2$, it will shift right after reading $S^1$, but will shift only by $\ell$.

This phenomenon is well known in exact string matching. For example, the non-optimal Horspool algorithm [5] shifts as soon as the window suffix mismatches the pattern suffix, while the optimal BDM [3] shifts when the window suffix does not match *anywhere* inside the pattern. Hence BDM works more inside the window, but its shifts are longer and at the end it has better average complexity than Horspool algorithm.

## 3 Analysis

We analyze our basic algorithm and prove its average-optimality. It will follow that the variant of Section 2.2, being never worse than it, is also optimal.

We analyze an algorithm that is necessarily worse than our basic algorithm for every possible text window, but simpler to analyze. In every text window, the simplified algorithm *always* reads $1 + \lfloor k/(c\ell) \rfloor$ consecutive $\ell$-grams, for some constant $0 < c < 1$ that will be considered shortly. After having read them, it checks whether *any* of the $\ell$-grams produces less than $c\ell$ differences in the $D$ table. If there is at least one such $\ell$-gram, the window is verified and shifted by 1. Otherwise, we have at least $1 + \lfloor k/(c\ell) \rfloor > k/(c\ell)$ $\ell$-grams with at least $c\ell$ differences each, so the sum of the differences exceeds $k$ and we can shift the window to one position past the last character read.

Note that, for this algorithm to work, we need to read $1 + \lfloor k/(c\ell) \rfloor$ $\ell$-grams from a text window. This is at most $\ell + k/c$ characters, a number that must not exceed the window length. This means that $c$ must observe the limit $\ell + k/c \leq m - k$, that is, $c \geq k/(m - k - \ell)$.

It should be clear that the real algorithm can never read more $\ell$-grams from any window than the simplified algorithm, can never verify a window that the

simplified algorithm does not verify, and can never shift a window by less positions than the simplified algorithm. Hence an average-case analysis of this simplified algorithm is a pessimistic average-case analysis of the real algorithm. We later show that this pessimistic analysis is tight.

Let us divide the windows we consider in the text into *good* and *bad* windows. A window is good if it does not trigger verifications, otherwise it is bad. We will consider separately the amount of work done over either type of window.

In good windows we read at most $\ell + k/c$ characters. After this, the window is shifted by at least $m-k-(\ell+k/c)+1$ characters. Therefore, it is not possible to work over more than $\lfloor n/(m-k-(\ell+k/c)+1)\rfloor$ good windows. Multiplying the maximum number of good windows we can process by the amount of work done inside a good window, we get an upper bound for the total work over good windows:

$$\frac{\ell+k/c}{m-k-(\ell+k/c)+1}\ n \quad = \quad O\left(\frac{\ell+k}{m}\ n\right), \tag{1}$$

where we have assumed $k + k/c < x(m-\ell)$ for some constant $0 < x < 1$, that is, $c > k/(x(m-\ell)-k)$. This is slightly stricter than our previous condition on $c$.

Let us now focus on bad windows. Each bad window requires $O(rm^2)$ verification work, using plain dynamic programming over each pattern. We need to show that bad windows are unlikely enough. We start by restating two useful lemmas proved in [2], rewritten in a way more convenient for us.

**Lemma 1** [2] The probability that two random $\ell$-grams have a common subsequence of length $(1-c)\ell$ is at most $a\sigma^{-d\ell}/\ell$, for constants $a = (1+o(1))/(2\pi c(1-c))$ and $d = 1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)$. The probability decreases exponentially for $d > 0$, which surely holds if $c < 1 - e/\sqrt{\sigma}$.

**Lemma 2** [2] If $S$ is an $\ell$-gram that matches inside a given string $P$ (larger than $\ell$) with less than $c\ell$ differences, then $S$ has a common subsequence of length $\ell - c\ell$ with some $\ell$-gram of $P$.

Given Lemmas 1 and 2, the probability that a given $\ell$-gram matches with less than $c\ell$ differences inside some $P^i$ is at most that of having a common subsequence of length $\ell - c\ell$ with some $\ell$-gram of some $P^i$. The probability of this is at most $mra\sigma^{-d\ell}/\ell$. Consequently, the probability that any of the considered $\ell$-grams in the current window matches is at most $(1 + k/(c\ell))mra\sigma^{-d\ell}/\ell$.

Hence, with probability $(1 + k/(c\ell))mra\sigma^{-d\ell}/\ell$ the window is bad and costs us $O(m^2r)$. Being pessimistic, we can assume that *all* the $n - (m-k) + 1$ text windows have their chance to trigger verifications (in fact only some text windows are given such a chance as we traverse the text). Therefore, the average total work on bad windows is upper bounded by

$$(1 + k/(c\ell))mra\sigma^{-d\ell}/\ell\ O(rm^2)\ n \quad = \quad O(r^2m^3n(\ell + k/c)a\sigma^{-d\ell}/\ell^2). \tag{2}$$

As we see later, the complexity of good windows is optimal provided $\ell = O(\log_\sigma(rm))$. To obtain overall optimality it is sufficient that the complexity of

bad windows does not exceed that of good windows. Relating Eqs. (1) and (2) we obtain the following condition on $\ell$:

$$\ell \geq \frac{4\log_\sigma m + 2\log_\sigma r + \log_\sigma a - 2\log_\sigma \ell}{d} = \frac{4\log_\sigma m + 2\log_\sigma r - O(\log\log(mr))}{d},$$

and therefore a sufficient condition on $\ell$ that retains the optimality of good windows is

$$\ell = \frac{4\log_\sigma m + 2\log_\sigma r}{1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)}. \tag{3}$$

It is time to define the value for constant $c$. We are free to choose any constant $k/(x(m-\ell)-k) < c < 1 - e/\sqrt{\sigma}$, for any $0 < x < 1$. Since this implies $k/(m-k) = \alpha/(1-\alpha) < 1 - e/\sqrt{\sigma}$, the method can only work for $\alpha < (1 - e/\sqrt{\sigma})/(2 - e/\sqrt{\sigma}) = 1/2 - O(1/\sqrt{\sigma})$. On the other hand, for any $\alpha$ below that limit we can find a suitable constant $x$ such that, asymptotically on $m$, there is space for constant $c$ between $k/(x(m-\ell)-k)$ and $1 - e/\sqrt{\sigma}$. For this to be true we need that $r = O(\sigma^{o(m)})$ so that $\ell = o(m)$. (For example, $r$ polynomial in $m$ meets the requirement.)

If we let $c$ approach $1 - e/\sqrt{\sigma}$, the value of $\ell$ goes to infinity. If we let $c$ approach $k/(m-\ell-k)$, then $\ell$ gets as small as possible but our search cost becomes $O(n)$. Any fixed constant $c$ will let us use the method up to some $\alpha < c/(c+1)$, for example $c = 3/4$ works well for $\alpha < 3/7$. Having properly chosen $c$ and $\ell$, our algorithm is on average

$$O\left(\frac{n(k + \log_\sigma(rm))}{m}\right) \tag{4}$$

character inspections. We remark that this is true as long as $\alpha < 1/2 - O(1/\sqrt{\sigma})$, as otherwise the whole algorithm reduces to dynamic programming. (Let us remind that $c$ is just a tool for a pessimistic analysis, not a value to be tuned in the real algorithm.) Our preprocessing cost is $O(mr\sigma^\ell)$, thanks to the smarter preprocessing of [4]. Given the value of $\ell$, this is $O(m^5 r^3 \sigma^{O(1)})$. The space with plain verification is $\sigma^\ell = m^4 r^2 \sigma^{O(1)}$ integers.

It has been shown that $O(n(k+\log_\sigma(rm))/m)$ is optimal [4]. Moreover, used on a single pattern we obtain the same optimal complexity of Chang & Marr [2], but our filter works up to $\alpha < 1/2 - O(1/\sqrt{\sigma})$. The filter of Chang & Marr works only up to $\alpha < 1/3 - O(1/\sqrt{\sigma})$. Hence we have improved the only average-optimal simple approximate search algorithm with respect to its area of applicability.

### 3.1 Worst-case Complexity

In the worst case the shift is always 1, and we work $O(m)$ for each each of the $O(n)$ windows, and verify each window for each pattern. The verifications are done incrementally, by saving the state of dynamic programming algorithm, to avoid scanning the same text characters more than once. Hence in the worst case our algorithm takes $O(nm + nr\lceil m/w \rceil)$ time. The worst case verification cost can be improved to $O(rnk)$ by replacing the dynamic programming algorithm with a

$O(nk)$ worst case algorithm [7]. The backward scanning filter can be improved to $O(n)$ worst case by combining it with the $O(n)$ time forward scanning filter [4]. The idea is that we stop the backward scanning if $(m-k)/2$ characters have been examined in some window, stopping at character $p$, and then switch to forward scanning and scan the next $m-k$ characters from $p$, and then switch again to backward scanning. This guarantees that each text character is examined at most twice. Variations of this trick has been used before, see e.g. [3].

## 4 Experimental Results

We have implemented the algorithms in C, compiled using `icc 7.1` with full optimizations. The experiments were run in a 2GHz Pentium 4, with 512MB RAM, running Linux 2.4.18. The computer word length is $w = 32$ bits. We measured user times, averaged over five runs.

We ran experiments for alphabet sizes $\sigma = 4$ (DNA) and $\sigma = 20$ (proteins). The test data were randomly generated 64MB files. The randomly generated patterns were 64 characters long. We show also experiments using real texts. These are: the E.coli DNA sequence from Canterbury Corpus[3], and real protein data from TIGR Database (TDB)[4]. In this case the patterns were randomly extracted from the texts. In order to better compare with the experiments with random data, we replicated the texts up to 64MB.

For lack of space we omit the experiments on preprocessing times. The most important result is that the maximum values we can use in practice are $\ell \leq 8$ for DNA, and $\ell \leq 3$ for proteins, otherwise the preprocessing time exceeds by far any reasonable search time. The search times that follow were measured for these maximum $\ell$ values unless otherwise stated, and we show the total times, preprocessing included.

Observe that the $\ell$ values required by our analysis (Section 3) in order to have optimal complexity are, depending on $r$, $\ell = 12 \ldots 20$ for DNA, and $\ell = 6 \ldots 10$ for proteins. These are well above the values we can handle in practice. The result is that, although our algorithms are very fast as expected, they can cope with difference ratios much smaller than those predicted by the analysis. This is the crux of the difference between our theoretical and practical results.

We compare our algorithm against previous work, both for searching single and multiple patterns. Following [10] we have included only the relevant algorithms in the comparison. These are:

**Ours:** Our basic algorithm (Sec. 2.1).
**Ours, strict:** Our algorithm, stricter matching (Sec. 2.2).
**Ours, strictest:** Our algorithm, strictest matching (Sec. 2.3).
**CM:** Our previous algorithm [4] based on [2].
**LT:** Our previous linear time filter [4], a variant of CM that cannot skip characters but works up to $\alpha = 1/2$.

[3] `http://corpus.canterbury.ac.nz/descriptions/`
[4] `http://www.tigr.org/tdb`

**EXP:** Partitioning into exact search [1], an algorithm for single and multiple approximate pattern matching, implemented by its authors.

**MM:** Muth & Manber algorithm [8], the first multipattern approximate search algorithm we know of, able of searching only with $k = 1$ differences and until now unbeatable in its niche, when more than 50–100 patterns are searched for. The implementation is also from its authors.

**ABNDM:** Approximate BNDM algorithm [11, 6], a single pattern approximate search algorithm extending classical BDM. The implementation is by its authors. We used the version of [11] which gave better results in our architecture, although it is theoretically worse than [6].

**BPM:** Bit-parallel Myers [9], currently the best non-filtering algorithm for single patterns, using the implementation of its author. We do not expect this algorithm to be competitive against filtering approaches, but it should give a useful control value.

For our algorithms we have used hierarchical verification, and for the algorithms of Secs. 2.1 and 2.2 we also applied bit-parallel counters. Both optimizations were applied for our previous algorithms CM and LT too.

We have modified ABNDM and BPM to use the superimposition technique [1] to handle multiple patterns, combined with hierarchical verification. The superimposition is useful only if $r/\sigma$ is reasonably small. In particular, it does not work well on DNA, and in that case we simply run the algorithms $r$ times. For proteins we superimposed a maximum of 16 patterns at a time. Note that the optimal group size depends on $k$, decreasing as $k$ increases. However, we used the above group sizes for all the cases.

Since MM is limited to $k = 1$, we compare this case separately in Table 1. As it can be seen, our algorithm is better by far, 4–40 times faster depending on the alphabet size.

| | $r=1$ | $r=16$ | $r=64$ | $r=256$ | $r=1$ | $r=16$ | $r=64$ | $r=256$ |
|---|---|---|---|---|---|---|---|---|
| Alg. | DNA (random) | | | | DNA (Ecoli) | | | |
| MM | 1.30 | 3.97 | 12.86 | 42.52 | 1.27 | 3.85 | 13.96 | 45.29 |
| Ours | 0.08 | 0.12 | 0.21 | 0.54 | 0.10 | 0.15 | 0.26 | 0.91 |
| Alg. | proteins (random) | | | | proteins (real) | | | |
| MM | 1.17 | 1.19 | 1.26 | 2.33 | 1.15 | 1.17 | 1.23 | 2.21 |
| Ours | 0.08 | 0.11 | 0.18 | 0.59 | 0.08 | 0.13 | 0.22 | 0.75 |

**Table 1.** Comparison against Muth & Manber, for $k = 1$.

Figures 2, and 3 show results for the case $r = 1$ (single patterns), as well as larger $r$ values. Our algorithm is the fastest in the majority of cases. EXP beats our algorithm for large $k$, and this happens sooner for larger $r$.

Our main algorithm is in most cases the clear winner, as expected from the analysis. The filtering capability of Chang & Marr extension (CM) collapses

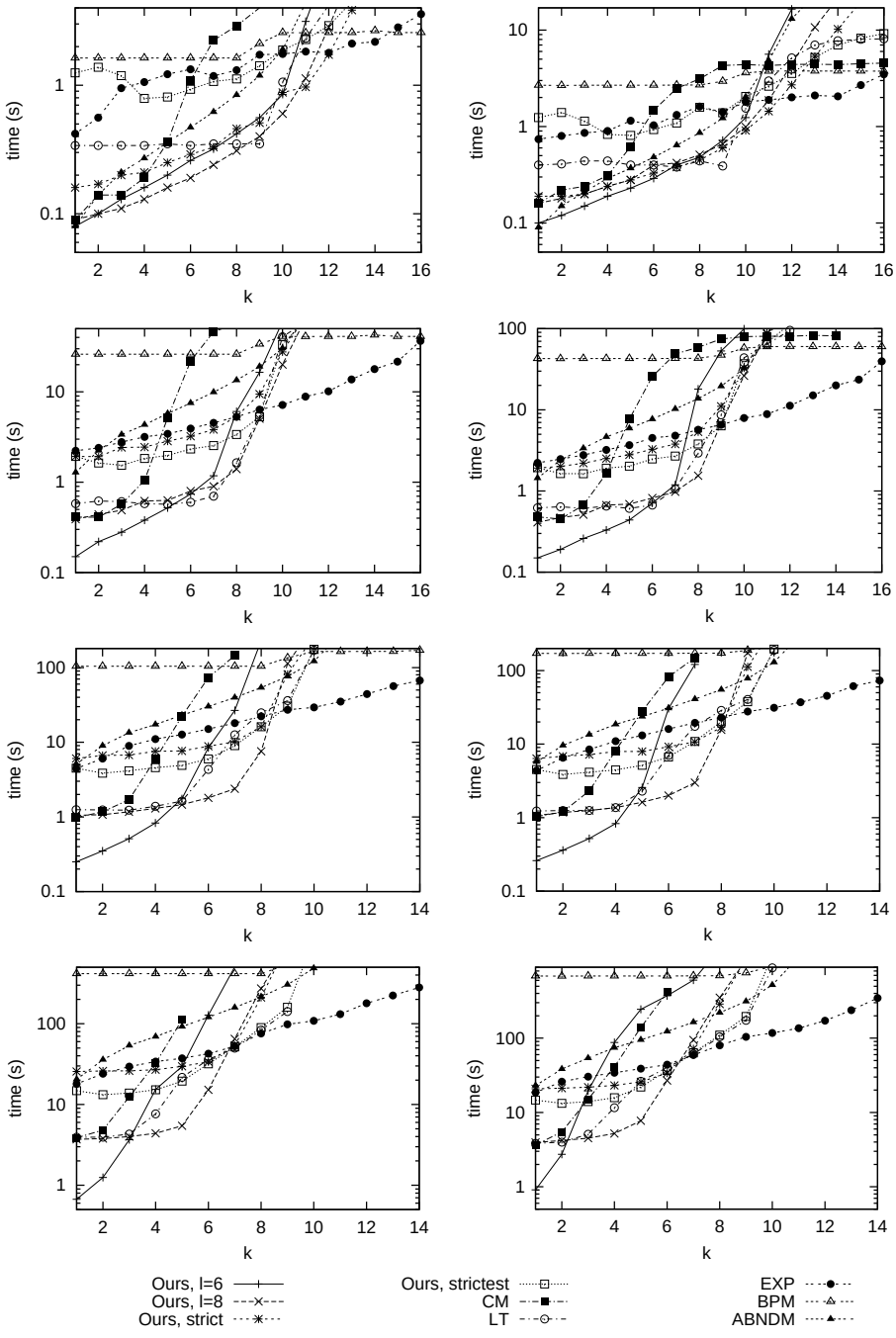


**Fig. 2.** Comparing different algorithms in DNA, considering both preprocessing and searching time. On the left, random data; on the right E.coli. From top to bottom: $r = 1, 16, 64, 256$. $\ell$ looks as "l". Note the logarithmic scale.

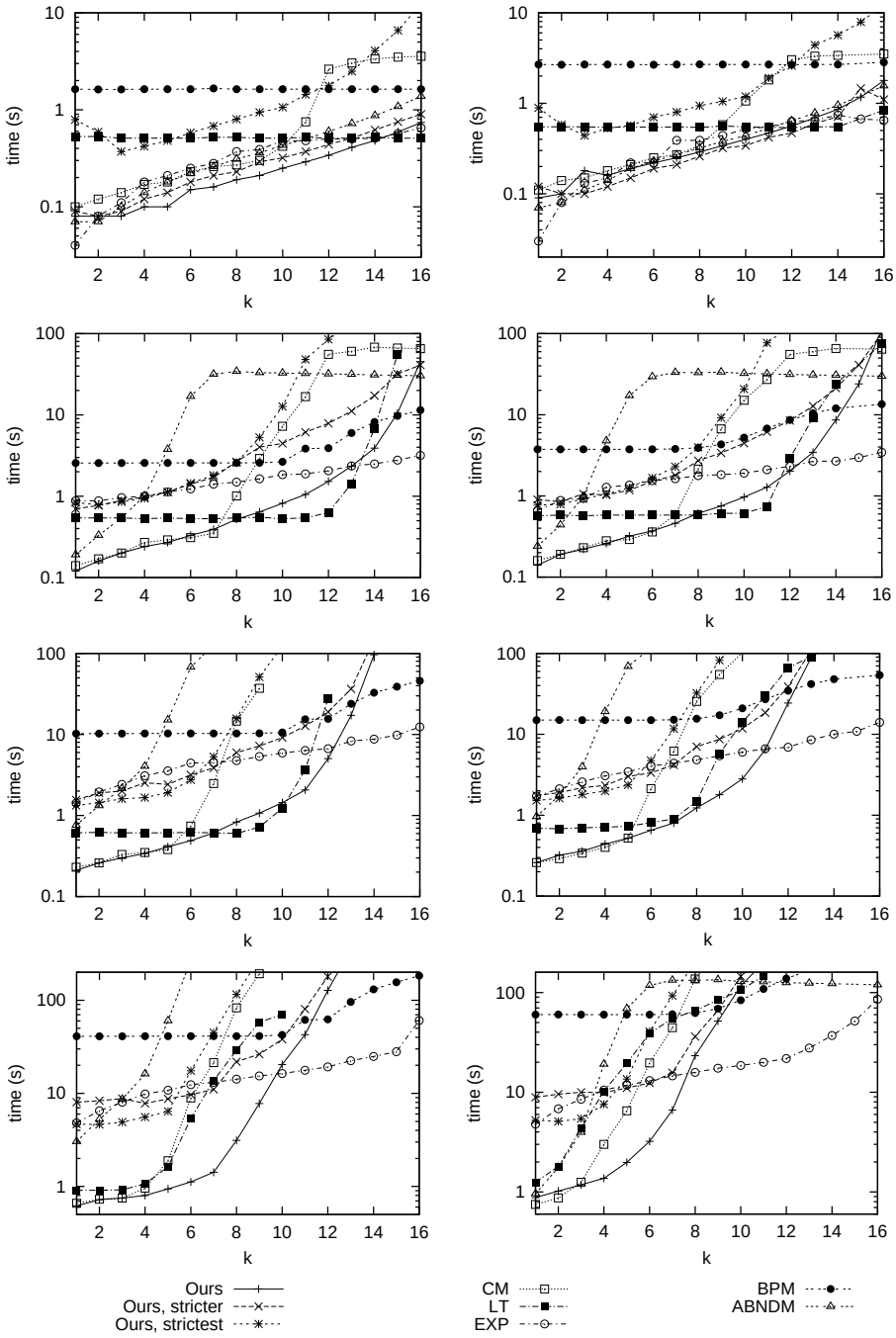


**Fig. 3.** Comparing different algorithms in proteins, considering both preprocessing and searching time. On the left, random data; on the right real proteins. From top to bottom: $r = 1, 16, 64, 256$. We used $\ell = 3$ for our algorithm. Note the logarithmic scale.

already with quite small difference ratios, due to the use of small text blocks, and this collapse is usually very sharp. Our new algorithms use larger search windows, so they trigger less verifications and permits larger difference rations. In addition, even with small difference ratios they skip more characters, and hence it is faster for all values of $k/m$. Our algorithm is usually faster than the linear time filter (LT), because the latter cannot skip text characters. This fact stays true even for large difference ratios.

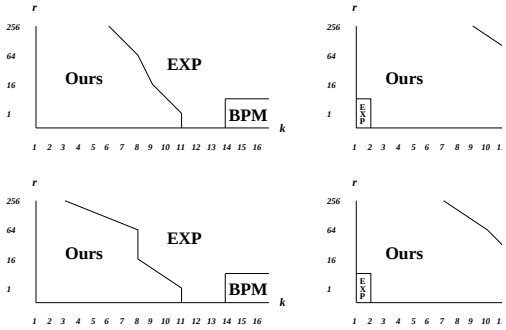


**Fig. 4.** Areas where each algorithm performs best. From left to right, DNA ($m = 64$), and proteins ($m = 64$). Top row is for random data, bottom row for real data.

Figure 4 shows the areas where each algorithm is best. Our new algorithm becomes the fastest choice for low $k$, although, the more patterns are sought, the smaller $k$ value is dominated by our algorithm. We have displaced the previously fastest algorithm for this case [1] to the area of intermediate difference ratios. Finally, we note that, when applied to just one pattern, our algorithm becomes indeed the fastest for low difference ratios.

Our basic algorithm beats the extensions proposed in Secs. 2.2 and 2.3, with the exception of single pattern real protein, where the algorithm of Sec. 2.2 is better. The success of our basic algorithm is due to lower preprocessing cost and the fact that the $D$ tables better fit into cache, which is important factor in modern computers. Note that this is true only if we use the same parameter $\ell$ for both algorithms. If we are short of memory we can use the variant of Sec. 2.2 with smaller $\ell$, and beat the basic algorithm. We have verified this, but omit the detailed experiments for lack of space.

Note also that our algorithm would be favored on even longer texts, as its preprocessing depends only on $rm$, and hence its relative cost decreases when $n$ increases.

## 5 Conclusions

We have presented a new multipattern approximate string matching algorithm. Our algorithm is optimal on average for low and intermediate difference ra-

tios (up to 1/2), improving our last-year result [4], which was the first average-optimal multipattern approximate string matching and worked only for $\alpha < 1/3$. Even for $\alpha < 1/3$, our algorithm is in practice much faster. We have also presented two variants, the first variant being also optimal. This algorithm inspects less text characters than the basic version, but becomes interesting only when we have strict limit for the maximum amount of memory we can use for the $D$ tables.

We have experimentally shown that our algorithms perform well in handling from one to a very large number of patterns, and low difference ratios. Our algorithms become indeed the best alternatives in practice for these cases. In particular, this includes being the fastest single-pattern search algorithms for low difference ratios, which is a highly competitive area. On real life texts, we show that our algorithms are especially interesting for computational biology applications (DNA and proteins).

## References


1. R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. *Random Structures and Algorithms (RSA)*, 20:23–49, 2002.
2. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 259–273, 1994.
3. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.
4. K. Fredriksson and G. Navarro. Average-optimal multiple approximate string matching. In *Proc. 14th Combinatorial Pattern Matching (CPM 2003)*, LNCS 2676, pages 109–128, 2003.
5. R. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.
6. H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 203–224, 2002.
7. G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.
8. R. Muth and U. Manber. Approximate multiple string search. In *Proc. 7th Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 75–86, 1996.
9. E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
10. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
11. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.
12. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.
13. E. Sutinen and J. Tarhio. Filtration with $q$-samples in approximate string matching. In *Proc. 7th Combinatorial Pattern Matching*, LNCS 1075, pages 50–63, 1996.