Average-Optimal Single and Multiple Approximate String Matching

KIMMO FREDRIKSSON
University of Joensuu
and
GONZALO NAVARRO
University of Chile

We present a new algorithm for multiple approximate string matching. It is based on reading backwards enough ℓ-grams from text windows so as to prove that no occurrence can contain the part of the window read, and then shifting the window.

We show analytically that our algorithm is optimal on average. Hence our first contribution is to fill an important gap in the area, since no average-optimal algorithm existed for multiple approximate string matching.

We consider several variants and practical improvements to our algorithm, and show experimentally that they are resistant to the number of patterns and the fastest for low difference ratios, displacing the long-standing best algorithms. Hence our second contribution is to give a practical algorithm for this problem, by far better than any existing alternative in many cases of interest. On real life texts, our algorithm is especially interesting for computational biology applications.

In particular, we show that our algorithm can be successfully used to search for one pattern, where many more competing algorithms exist. Our algorithm is also average-optimal in this case, being the second after that of Chang and Marr. However, our algorithm permits higher difference ratios than Chang and Marr, and this is our third contribution.

In practice, our algorithm is competitive in this scenario too, being the fastest for low difference ratios and moderate alphabet sizes. This is our fourth contribution, which also answers affirmatively the question of whether a practical average-optimal approximate string matching algorithm existed.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching, Computations on discrete structures*; H.3.3 [**Information storage and retrieval**]: Information Search and Retrieval—*Search process*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Algorithms, approximate string matching, multiple string matching, optimality, biological sequences

Supported by the Academy of Finland, grant 202281 (first author), partially supported by Fondecyt grant 1-020831 (second author).

Authors' address: Kimmo Fredriksson, Department of Computer Science, P.O. Box 111, FI-80101 Joensuu, Finland. Email: `kfredrik@cs.joensuu.fi`. Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. Email: `gnavarro@dcc.uchile.cl`.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Given a text $T_{1...n}$, 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 σ the size of the alphabet Σ.

A natural extension to the basic problem consists of *multipattern searching*, that is, searching for r patterns $P^1 \ldots P^r$ simultaneously in order to report all their occurrences with at most k differences. This has also several applications such as virus and intrusion detection [Kumar and Spafford 1994], spelling [Kukich 1992], speech recognition [Dixon and Martin 1979], optical character recognition [Elliman and Lancaster 1990], handwriting recognition [Lopresti and Tomkins 1994], text retrieval under synonym or thesaurus expansion [Baeza-Yates and Ribeiro-Neto 1999], computational biology [Sankoff and Kruskal 1983; Waterman 1995], multidimensional approximate matching [Baeza-Yates and Navarro 2000], batch processing of single-pattern approximate searching, etc. Moreover, some single-pattern approximate search algorithms resort to multipattern searching of pattern pieces [Navarro and Baeza-Yates 2001]. 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.

In this paper we use the terms "approximate matching" and "approximate searching" indistinctly, meaning the search for approximate occurrences of short strings (the patterns) inside a longer string (the text). This is one particular case of the wider concept of *sequence alignment*, pervasive in computational biology. Actually, the area of sequence alignment is divided into global alignment (where two strings are wholly compared), semi-global alignment (where one string is compared against any substring of the other, our focus in this paper), and local alignment (where all substrings of both strings are compared). *Multiple alignment* refers to comparing a set of strings so as to find common parts to all of them. Note that this is quite different from multiple string matching, where many (pattern) strings are compared against one (the text).

For average case analyses of text search algorithms it is customary to assume random text and patterns drawn over a uniformly distributed alphabet. Average-case complexity thus refers to averaging the cost of an algorithm over all possible texts of length n and patterns of length m, giving equal weight to each combination. The average complexity of a problem refers to the best possible average case of an algorithm solving the problem, and an average-optimal algorithm is one whose average-case complexity matches the average complexity of the problem.

The single-pattern problem has received a lot of attention since the sixties [Navarro 2001]. After the first dynamic-programming-based $O(mn)$ time solution to the problem [Sellers 1980], many faster techniques have been proposed, both for the worst and the average case. 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 algo-

rithms, the two important parameters are the filtration speed and the maximum difference ratio α up to which they work. Note that, in any case, one needs a good non-filtration algorithm to verify the text that the filter cannot discard.

The best non-filtration algorithms are in practice by Baeza-Yates & Navarro [Baeza-Yates and Navarro 1999; Navarro and Baeza-Yates 2001], and by Myers [Myers 1999], with average complexities of $O(kmn/w)$ and $O(kn/w)$, respectively, where w is the length in bits of the computer word. In 1994, Chang & Marr [Chang and Marr 1994] 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 $\alpha < 1/3 - O(1/\sqrt{\sigma})$. In practice, the fastest algorithms, based on filtration, have a complexity of $O(k \log_\sigma(m)n/m)$ and work for $\alpha < 1/\log_\sigma m$ [Navarro and Baeza-Yates 1999]. Hence the quest for an average-optimal algorithm whose optimality shows up in practice has been advocated [Navarro and Raffinot 2002].

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 [Muth and Manber 1996], that handle too few patterns [Baeza-Yates and Navarro 2002], that handle only low difference ratios [Baeza-Yates and Navarro 2002], and that handle only very short patterns [Hyyrö et al. 2004]. 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 [Baeza-Yates and Navarro 2002] works for $\alpha < 1/\log_\sigma(rm)$. No optimal algorithms have been devised, and the existing algorithms are not that fast. Hence multiple approximate string matching is a rather undeveloped area.

In this paper we present several contributions to the approximate search problem, for a single pattern and for multiple patterns:

—We present a new algorithm for multiple approximate string matching. It is based on sliding a window over the text. At each window, we read backwards successive ℓ-grams until we can prove that no pattern occurrence can contain those ℓ-grams read. This is done by a preprocessing that counts the minimum number of differences needed to match each ℓ-gram inside any pattern. Then the window is displaced far enough to avoid containing the scanned ℓ-grams.

—We show analytically that the average complexity of our algorithm is $O((k + \log_\sigma(rm))n/m)$ for $\alpha < 1/2 - O(1/\sqrt{\sigma})$. We show that this is average-optimal, which makes our algorithm the first average-optimal multiple approximate string matching algorithm. If applied to a single pattern, the algorithm is also optimal, like that of Chang & Marr. However, our algorithm works for higher difference ratios, since Chang & Marr works only for $\alpha < 1/3 - O(1/\sqrt{\sigma})$.

—We consider several practical improvements to our algorithm, such as optimal choice of the window ℓ-grams, reduction of the cost to check windows that cannot be discarded, reduction of preprocessing costs for many patterns, and so on. As a result, our algorithm turns out to be very competitive in practice, apart from theoretically optimal.

—We perform exhaustive experiments to evaluate the different aspects of our algorithm and to compare it against others. We show that our algorithm is resistant

to the number of patterns and that it works well for low difference ratios. In these cases, our algorithm displaces the long-standing best algorithms [Muth and Manber 1996; Baeza-Yates and Navarro 2002] (since 1996) in most cases, largely improving their search times. Our algorithm is competitive even to search for one single pattern, where many more competing alternatives exist. In this case our algorithm becomes the fastest choice for low difference ratios and not very large alphabets. On real life texts, we show that our algorithm is especially interesting for computational biology applications (searching DNA and proteins).

—As a side contribution, we explore several variants of our algorithm, which turn out to be more or less direct extensions to multipattern searching of Chang & Marr [Chang and Marr 1994], LET by Chang & Lawler [Chang and Lawler 1994], LAQ by Sutinen & Tarhio [Sutinen and Tarhio 1996]. We also consider combining the basic idea of LAQ with our main algorithm. In most cases our new algorithm is superior, although these variants are of interest in some particular situations.

Hence we have positively settled the question of the existence of a practical average-optimal approximate string matching algorithm [Navarro and Raffinot 2002]. It is interesting that our algorithm is theoretically optimal for low and intermediate difference ratios, but good in practice only for low difference ratios. The reason has to do with the space requirement of our algorithm, that is polynomial but high, and that forces us in practice to use suboptimal parameter settings.

Preliminary partial versions of this paper appeared in [Fredriksson and Navarro 2003; 2004].

2. RELATED WORK

In this section we cover the existing algorithms for multiple approximate string matching, as well as the techniques for simple approximate string matching that are relevant to our work. We explain them up to the level necessary to understand their relevance for us.

2.1 Multiple Approximate String Matching

The naive approach to multipattern approximate searching is to perform r separate searches, one per pattern. If we use the optimal single-pattern algorithm [Chang and Marr 1994], the average search time becomes $O((k+\log_\sigma m)rn/m)$ for the naive approach. On the other hand, if we use the classical $O(mn)$ algorithm [Sellers 1980] the time is $O(rmn)$.

Few algorithms exist for multipattern approximate searching under the k differences model. The first one was presented by Muth and Manber [Muth and Manber 1996]. It permits searching with $k = 1$ differences only, but it is rather tolerant to the number of patterns r, which can reach the hundreds without affecting much the cost of the search. They show that, if an approximate occurrence of P in T ends at position j and we take $P' = T_{j-m+1...j}$, then there are character positions s and s' such that P and P' become equal if we remove their s-th and s'-th character, respectively. Therefore they take all the m choices of removing a single character of every pattern P^i and store the rm strings in a hash table. Later, they consider every text window of the form $T_{j-m+1...j}$, and for each of the m choices of removing a single window character, they search for the resulting substring in

the hash table. If the substring is found in the table, they check the appropriate pattern in the text window. The preprocessing time is $O(rm)$ and the average search time is $O(mn(1 + rm^2/M))$, where M is the size of the hash table. This adds up $O(rm + nm(1 + rm^2/M))$, which is $O(m(r + n))$ if $M = \Omega(m^2r)$. The time is basically independent of r if n is large enough. However, in order to permit a general number k of differences, they should consider every way of removing k characters from the windows, resulting in an $O(m^k(r + n))$ time algorithm, which is clearly impractical for all but very small k values.

Baeza-Yates and Navarro [Baeza-Yates and Navarro 2002] have presented several algorithms for this problem. One of them, *partitioning into exact search*, uses the fact that, if P is cut into $k+1$ pieces, then at least one of the pieces appears inside every occurrence with no differences. Hence the algorithm splits every pattern into $k + 1$ pieces and searches for the $r(k + 1)$ pieces with an exact multipattern search algorithm. The preprocessing takes $O(rm)$ time. If they used an optimal multipattern exact search algorithm like MultiBDM [Crochemore and Rytter 1994], the search time would have been $O(k \log_\sigma(rm)n/m)$ on average. For practical reasons they used another algorithm, more suitable to searching for short pieces (of length $\lfloor m/(k + 1)\rfloor$), albeit with worse theoretical complexity. This technique can be applied for $\alpha < 1/\log_\sigma(rm)$, a limit that gets stricter as m or r increase.

They also presented other algorithms that, although can handle higher difference ratios, are linear on r, which means that they give a speedup only up to a constant number c of patterns and then just divide the search into r/c groups that are searched for separately. *Superimposition* uses a standard search technique on a set of "superimposed" patterns, which means that the i-th character of the superimposition matches the i-th character of any of the superimposed patterns. Implemented over a newer bit-parallel algorithm [Myers 1999], superimposition would yield average time $O(rn/(\sigma(1-\alpha)^2))$ for $\alpha < 1-e\sqrt{r/\sigma}$ on patterns shorter than the number of bits in the computer word, w (typically $w = 32$ or 64). Different techniques are used to cope with longer patterns, but the times are worse. *Counting* extends a single-pattern algorithm that slides a window of length m over the text checking in linear time whether it shares at least $m - k$ characters with the pattern (regardless of the order). The multipattern version keeps several counters in a single computer word, achieving an average search time of $O(rn \log(m)/w)$ for $\alpha < e^{-m/\sigma}$.

Hyyrö et al. [Hyyrö et al. 2004] have recently presented another bit-parallel technique to simultaneously search for a few short patterns. When $m \le w$, they need $O(\lceil r/\lfloor w/m\rfloor\rceil n)$ worst-case time instead of $O(rn)$. On average this improves to $O(\lceil r/\lfloor w/\max(k, \log m)\rfloor\rceil n)$.

2.2 Dynamic Programming and Myers' Bit-parallel Algorithm

The classical algorithm to find the approximate occurrences of pattern P inside text T [Sellers 1980] is to compute a matrix $C_{i,j}$ for $0 \le i \le m$ and $0 \le j \le n$, using dynamic programming as follows:

$$C_{i,0} = i\,, \quad C_{0,j} = 0$$
$$C_{i+1,j+1} = \text{if } P_{i+1} = T_{j+1} \text{ then } C_{i,j} \text{ else } 1 + \min(C_{i,j}, C_{i,j+1}, C_{i+1,j})$$

which can be computed, for example, column-wise. We need only the previous column in order to compute the current one. Every time the current column value $C_{m,j} \leq k$ we report text position j as the endpoint of an occurrence. The algorithm needs $O(mn)$ time and $O(m)$ space.

The same matrix can be used to compute the edit distance (number of differences necessary to convert one string into the other) just by changing $C_{0,j} = j$ in the initial conditions. The distance is then $C_{m,n}$.

One of the most successful non-filtration approximate string matching algorithms consists of a bit-parallel version of the dynamic programming matrix [Myers 1999]. On a computer word of w bits, the algorithm obtains $O(mn/w)$ search time. The central idea is to represent the current column $C_{*,j}$ using two streams of m bits, one indicating positive and the other negative differences of the form $C_{i,j} - C_{i-1,j}$. Since consecutive cells in C differ at most by ± 1, this representation is sufficient. Myers manages to update those vectors in constant time per column.

2.3 The Algorithm of Tarhio and Ukkonen

Tarhio and Ukkonen [Tarhio and Ukkonen 1993; Jokinen et al. 1996] presented the first filtration algorithm (1991), using a Boyer-Moore-like technique to filter the text. The idea is to align the pattern with a text window and scan the text backwards. The scanning ends where more than k "bad" text characters are found. A "bad" character is one that not only does not match the pattern position it is aligned with, but it also does not match any pattern character at a distance of k characters or less. More formally, assume that the window starts at text position $j+1$, and therefore T_{j+i} is aligned with P_i. Then T_{j+i} is bad when $Bad(i, T_{j+i})$, where $Bad(i, c)$ has been precomputed as $c \notin \{P_{i-k}, P_{i-k+1}, ..., P_i, ...P_{i+k}\}$.

The idea of the bad characters is that we know for sure that we have to pay a difference to match them, that is, they will not match as a byproduct of inserting or deleting other characters. When more than k characters that are differences for sure are found, the current text window can be abandoned and shifted forward. If, on the other hand, the beginning of the window is reached, the area $T_{j+1-k..j+m}$ must be checked with a classical algorithm.

To know how much can we shift the window, the authors show that there is no point in shifting P to a new position j' where none of the $k+1$ text characters that are at the end of the current window $(T_{j+m-k}, ...T_{j+m})$ match the corresponding character of P, that is, where $T_{j+m-s} \neq P_{m-s-(j'-j)}$. If those differences are fixed with substitutions we make $k+1$ differences, and if they can be fixed with less than $k+1$ operations, then it is because we aligned some of the involved pattern and text characters using insertions and deletions. In this case, we would have obtained the same effect aligning the matching characters from start.

So for each pattern position $i \in \{m-k..m\}$ and each text character a that could be aligned to position i (that is, for all $a \in \Sigma$) the shift to align a in the pattern is precomputed, that is, $Shift(i, a) = \min_{s>0}\{P_{i-s} = a\}$ (or m if no such s exists). Later, the shift for the window is computed as $\min_{i \in m-k..m} Shift(i, T_{j+i})$. This last minimum is computed together with the backward window traversal.

The (simplified) analysis [Tarhio and Ukkonen 1993; Navarro 2001] shows that the search time is $O(k^2 n/\sigma)$, for $\alpha < e^{-(2k+1)/\sigma}$. The analysis is valid for $m \gg \sigma > k$. The algorithm is competitive in practice for low difference ratios. Interestingly, the

version $k = 0$ corresponds exactly to Horspool algorithm [Horspool 1980]. Like Horspool, it does not take proper advantage of very long patterns.

2.4 The Algorithm of Chang and Marr

Chang and Marr [Chang and Marr 1994] show that no approximate search algorithm for a single pattern can be faster than $O((k + \log_\sigma m)n/m)$ on the average. This is not hard to prove, and we give more details in Section 4.

In the same paper [Chang and Marr 1994], Chang and Marr presented an algorithm achieving that optimal average time complexity. In the preprocessing phase they build a table D as follows. They choose a number ℓ in the range $1 \leq \ell \leq \lceil (m-k)/2 \rceil$, whose exact value we will consider shortly. For every string S of length ℓ (ℓ-gram), they search for S in P and store in $D[S]$ the smallest number of differences needed to match S inside P (this is a number between 0 and ℓ). Hence D requires space for σ^ℓ entries and is computed in $O(\sigma^\ell \ell m)$ time. A numerical representation of Σ^ℓ permits constant time access to D.

The text scanning phase consists of logically dividing the text into blocks of length $b = \lceil (m-k)/2 \rceil$, which ensures that any approximate occurrence of P (which must be of length at least $m-k$) contains at least one whole block. Each block $T_{ib+1...ib+b}$ is processed as follows. They take the first ℓ-gram of the block, $S^1 = T_{ib+1...ib+\ell}$, and obtain $D[S^1]$. Then they take the next ℓ-gram, $S^2 = T_{ib+\ell+1...ib+2\ell}$, and obtain $D[S^2]$, and so on. If, before reaching the end of the block, they have obtained $\sum_{1 \leq t \leq u} D[S^t] > k$, then they can safely skip the block because no occurrence of P can contain the block, as merely matching those t ℓ-grams anywhere inside P requires more than k differences. If, on the other hand, they reach the end of the block without surpassing k total differences, the block must be checked. In order to check for $T_{ib+1...ib+b}$ they run the classical dynamic programming algorithm over $T_{ib+b+1-m-k...ib+m+k}$.

In order to keep the space requirement polynomial in m, it is required that $\ell = O(\log_\sigma m)$. On the other hand, in order to achieve the claimed complexity, it is necessary that $\ell \geq x \log_\sigma m$ for some constant $x > 3$, so the space is $O(m^x)$. The optimal complexity holds as long as $\alpha < 1/3 - O(1/\sqrt{\sigma})$.

A somewhat related algorithm is LET, by Chang & Lawler [Chang and Lawler 1994]. In this case the window is not fixed but slides over the text and they keep count of the minimum number of differences necessary to match the current window. To slide the window they include text at its right and discard from its left. LET, however, does not permit approximate but just exact matching of the window substrings inside P, and therefore its tolerance to differences is lower.

2.5 The Approximate BNDM Algorithm

Navarro and Raffinot [Navarro and Raffinot 2000] presented a novel approach based on suffix automata. A suffix automaton built over a string recognizes all the suffixes of the string. In [Navarro and Raffinot 2000], they adapted an exact string matching algorithm, BDM, to allow differences.

The idea of the original BDM algorithm is as follows [Crochemore and Rytter 1994]. The deterministic suffix automaton of the *reverse* pattern is built, so it recognizes the reverse prefixes of the pattern. Then the pattern is aligned with a text window, and the window is scanned backwards with the automaton (this is

why the pattern is reversed). The automaton is active as long as what it has read is a substring of the pattern. Each time the automaton reaches a final state, it has seen a pattern prefix, so we remember the last time it happened. If the automaton arrives with active states to the beginning of the window then the pattern has been found, otherwise what is there is not a substring of the pattern and hence the pattern cannot be in the window. In any case the last window position that matched a pattern prefix gives the next initial window position.

The algorithm BNDM [Navarro and Raffinot 2000] is a bit-parallel implementation of the nondeterministic suffix automaton, which is much faster in practice and allows searching for extended patterns. This nondeterministic automaton is then modified to match any suffix of the pattern with up to k differences, and then essentially the same BDM algorithm is applied.

The window will be abandoned when no pattern substring matches with k differences what was read. The window is shifted to the next pattern prefix found with k differences. The matches must start exactly at the initial window position. The window length is $m-k$, not m, to ensure that if there is an occurrence starting at the window position then a substring of the pattern occurs in any suffix of the window (so we do not abandon the window before reaching the occurrence). Reaching the beginning of the window does not guarantee a match, however, so we have to check the area by computing edit distance from the beginning of the window (at most $m+k$ text characters).

In [Navarro 2001] it is shown that the algorithm inspects on average $O(n(k+\log_\sigma(m)/m))$ characters, and the filter works well for $\alpha < 1/2 - O(1/\sqrt{\sigma})$. This looks optimal and works for higher difference ratios than Chang & Marr. However, characters cannot be inspected in constant time. In the original algorithm [Navarro and Raffinot 2000], the nondeterministic automaton is simulated in $O(km/w)$ operations per character. Later [Hyyrö and Navarro 2002] this was improved to $O(m/w)$ by adapting Myers' bit-parallel simulation. However, optimality is attained only for $m = O(w)$.

In practice, the result is competitive for low difference ratios and for pattern lengths close to w.

3. OUR ALGORITHM

We explain now the basic idea of our algorithm, and later consider different practical improvements.

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

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 ℓ-grams backwards, that is, $S^1 = T_{i+m-k-\ell+1 \dots i+m-k}$, $S^2 = T_{i+m-k-2\ell+1 \dots i+m-k-\ell}$, $\dots$, $S^t = T_{i+m-k-t\ell+1 \dots i+m-k-(t-1)\ell}$, and so on. Any occurrence starting at the beginning of the window must fully contain those ℓ-grams. We accumulate the D values for the successive ℓ-grams read, $M_u = \sum_{1 \le t \le u} D[S^t]$. If, at some point, the

$P^1 =$ aacaccgaaa $P^2 =$ gaacgaacac $P^3 =$ ttcggcccgg

	aa	ac	ag	at	ca	cc	cg	ct	ga	gc	gg	gt	ta	tc	tg	tt
D_{P^1}	0	0	1	1	0	0	0	1	0	1	1	1	1	1	1	2
D_{P^2}	0	0	1	1	0	1	0	1	0	1	1	1	1	1	1	2
D_{P^3}	2	1	1	1	1	0	0	1	1	0	0	1	1	0	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
D	0	0	1	1	0	0	0	1	0	0	0	1	1	0	1	0

Fig. 1. An example of the table D, computed for the set of three DNA patterns and for $\ell = 2$. For example, the 2-gram ga occurs in P^1 and P^2 with 0 errors and in P^3 with 1 error. The table D is obtained by taking the minimum of the table entries for each of the patterns.

Current window position is $T_{25..33}$:

i	24	25	26	27	28	29	30	31	32	33	34	35	36	37	
T ...	c	c	t	a	g	g	*t*	*a*	*a*	*t*	t	t	a	c	...

$$M = D[T_{32...33}] + D[T_{30..31}] = D[\texttt{at}] + D[\texttt{ta}] = 2 > k$$
$\Rightarrow$ The window can be shifted past $T[30]$:

i	30	31	32	33	34	35	36	37	38	39	40	41	42	43	
T ...	t	a	*a*	*t*	*t*	*t*	*a*	c	*a*	*a*	t	t	a	g	...

$$M = D[T_{38...39}] + D[T_{36..37}] + D[T_{34..35}] + D[T_{32..33}] = D[\texttt{aa}] + D[\texttt{ac}] + D[\texttt{tt}] + D[\texttt{at}] = 1 \le k$$
$\Rightarrow$ Must verify the area $T_{31..41}$. The next window position is $T_{32..40}$.

Fig. 2. An example of **CanShift**(24, D) using the D table of Figure 1. The parameters are $m = 10$, $k = 1$, and $\ell = 2$.

sum M_u exceeds k, it is not possible to have an occurrence containing the sequence of ℓ-grams read $S^u \dots S^1$, as merely matching those ℓ-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 ℓ-grams read.

On the other hand, it might be that we read all the ℓ-grams fully contained in the window and do not surpass 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 $T_{i+1...i+m+k}$ for each of the r patterns, so as to cover any possible 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. Figure 2 illustrates.

The above 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, c, so as to prevent retraversing the same text areas but just restarting from the point we left the last verification.

We use Myers' algorithm [Myers 1999] for the verification of single patterns, which makes the cost $O(m^2/w)$ per pattern, being w the number of bits in the computer word.

Figure 3 gives the code in its simplest form. We present next several improvements over the basic idea and change accordingly some modules of this code. **Search** is the main module, using variables i and c denoting the position preceding the first character of the window and the first position not yet verified, respectively. The verification algorithm must be reinitialized when verification must restart ahead of the last verified position, otherwise the last verification is resumed from position c. **CanShift**(i, D) considers window $i + 1 \ldots i + m - k$ and scans ℓ-grams backwards from it until determining that it can be shifted or not. It returns a position pos such that the new window can start at position $pos + 1$. If $pos = i$ it means that the window must be verified. **Preprocess** computes ℓ and D. The computation of ℓ takes the largest value such that it will not exceed the window length, the D table will fit in memory, and the preprocessing cost will not exceed the average search cost. Finally, **MinDist**(S, P) computes the minimum edit distance of S inside P, using the formula of Section 2.2 to find the "pattern" S inside the "text" P. In this case $0 \le i \le \ell$ and $0 \le j \le m$, and it turns out to be more convenient for presenting our later developments to compute the matrix row-wise rather than column-wise.

3.1 Optimal Choice of ℓ-grams

The basic algorithm uses the last consecutive ℓ-grams of the block in order to find more than k differences. This is simple, but not necessarily the best choice. Note that any set of non-overlapping ℓ-grams found inside the window whose total number of differences inside P exceeds k permits us discarding the window. Hence the question of using the best possible set is raised.

The optimization problem is as follows. Given the text window $T_{i+1 \ldots i+m-k}$ we have $m - k - \ell + 1$ possible ℓ-grams, namely $T_{i+1 \ldots i+\ell}$, $T_{i+2 \ldots i+\ell+1}$, $\ldots$, $T_{i+m-k-\ell+1 \ldots i+m-k}$. From this set we want a subset of non-overlapping ℓ-grams $S^1 \ldots S^u$ such that $\sum_{1 \le t \le u} D[S^t] > k$. Moreover, we want to process the set right to left and detect a good enough subset as soon as possible.

This is solved by redefining M_u as the maximum sum that can be obtained using disjoint ℓ-grams that start at positions $i + u \ldots i + m - k$. Initially we start with $M_u = 0$ for $m - k - \ell + 2 < u \le m - k + 1$. Then we traverse the block computing, for decreasing u values,

$$M_u \quad \leftarrow \quad \max(D[T_{i+u \ldots i+u+\ell-1}] + M_{u+\ell}\ ,\ M_{u+1}), \tag{1}$$

where the first term accounts for the fact that we choose to use the ℓ-gram that starts at u and add to it the best solution to the right that does not overlap this ℓ-gram; and the second term accounts for the fact that we do not use the ℓ-gram that starts at u.

We compute M_u for decreasing u until either (i) $M_u > k$, in which case we shift the window, or (ii) $u = 0$, in which case we have to verify the window. Figure 4

```
Search (T_{1...n}, P^1_{1...m} ... P^r_{1...m}, k)
 1.   Preprocess ( )
 2.   i ← 0, c ← 0
 3.   While i ≤ n − (m − k) Do
 4.       pos ← CanShift (i, D)
 5.       If pos = i Then
 6.           If i + 1 > c Then
 7.               Initialize verification algorithm
 8.               c ← i + 1
 9.           Run verification in text area T_{c...i+m+k}
10.           c ← i + m + k + 1
11.           pos ← pos + 1
12.       i ← pos
```

```
CanShift (i, D)
 1.   M ← 0
 2.   p ← m − k
 3.   While p ≥ ℓ Do
 4.       p ← p − ℓ
 5.       M ← M + D[T_{i+p+1...i+p+ℓ}]
 6.       If M > k Then Return i + p + 1
 7.   Return i
```

```
Preprocess ( )
 1.   ℓ ← according to Eq. (4)
 2.   For S ∈ Σ^ℓ Do
 3.       D[S] ← ℓ
 4.       For i ∈ 1 ... r Do
 5.           D[S] ← min(D[S], MinDist(S, P^i))
```

```
MinDist (S_{1...ℓ}, P_{1...m})
 1.   For j ∈ 0 ... m Do C_j ← 0
 2.   For i ∈ 1 ... ℓ Do
 3.       Cold ← C_0, C_0 ← i
 4.       For j ∈ 1 ... m Do
 5.           If S_i = P_j Then Cnew ← Cold
 6.           Else Cnew ← 1 + min(Cold, C_j, C_{j−1})
 7.           Cold ← C_j, C_j ← Cnew
 8.   Return min_{0≤j≤m} C_j
```

Fig. 3. Simple description of the algorithm. The main variables are global in the rest of the paper, to simplify the presentation.

gives the code.

Note that the cost of choosing the best set of ℓ-grams is that, if we abandon the block after considering position x, then we work $O(x/\ell)$ with the simple method and $O(x)$ with the current one. (This assumes we can read an ℓ-gram in constant time.) However, x itself may be smaller with the optimization method.

CanShift $(i,\ D)$
1. **For** $u \in m-k-\ell+2 \ldots m-k+1$ **Do** $M_u \leftarrow 0$
2. $u \leftarrow m-k-\ell+1$
3. **While** $u \geq 0$ **Do**
4. $\quad M_u \leftarrow \max(D[T_{i+u \ldots i+u+\ell-1}] + M_{u+\ell}\ ,\ M_{u+1})$
5. $\quad$ **If** $M > k$ **Then Return** $i+u+1$
6. $\quad u \leftarrow u-1$
7. **Return** i

Fig. 4. Optimization technique to choose the set of overlapping ℓ-grams that maximize the sum of differences.

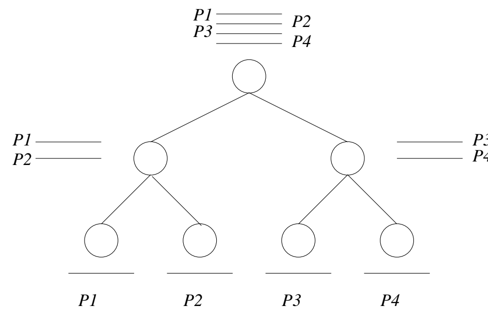

Fig. 5. Pattern hierarchy for 4 patterns.

3.2 Hierarchical Verification

On the windows that have to be verified, we could simply run the verification for every pattern, one by one. A more sophisticated choice is *hierarchical verification* (already presented in previous work [Baeza-Yates and Navarro 2002]). We form a tree whose nodes have the form $[i, j]$ and represent the group of patterns $P^i \ldots P^j$. The root is $[1, r]$. The leaves have the form $[i, i]$. Every internal node $[i, j]$ has two children $[i, \lfloor (i+j)/2 \rfloor]$ and $[\lfloor (i+j)/2 \rfloor + 1, j]$.

The hierarchy is used as follows. For every internal node $[i, j]$ we have a table D computed using the minimum distances between each ℓ-gram and patterns $P^i \ldots P^j$. This is done by computing first the leaves (that is, each pattern separately) and then computing every cell of D in the internal node as the minimum over the corresponding cell in its two children. In order to scan the text, we use the D table of the root node, which corresponds to the full set of patterns. Every time a window has to be verified with respect to a node in the hierarchy (at first, the root node), we rescan the window considering the two children of the current node. It is possible that the window can be discarded for both children, for one, or for none. We recursively repeat the process for every child that does not permit discarding the window, see Figure 5. If we process a leaf node and still have to verify the window, then we run the verification algorithm for the corresponding single pattern.

The idea of using the hierarchy instead of plainly checking the r patterns one by one is that it is possible that the grouping of the patterns matches a block, but

```
PreprocessD (i)
 1.     For S ∈ Σ^ℓ Do
 2.         D_{i,i}[S] ← MinDist(S, P^i)

HierarchyPreprocess (i, j)
 1.     If i = j Then PreprocessD(i)
 2.     Else
 3.         p ← ⌊(i + j)/2⌋
 4.         HierarchyPreprocess (i, p, S)
 5.         HierarchyPreprocess (p + 1, j, S)
 6.         For S ∈ Σ^ℓ Do
 7.             D_{i,j}[S] ← min(D_{i,p}[S], D_{p+1,j}[S])

Preprocess ( )
 1.     ℓ ← according to Eq. (6)
 2.     HierarchyPreprocess(1, r)
```

Fig. 6. Preprocessing to build the hierarchy. It produces global tables $D_{i,j}$ to be used by **HierarchyVerify**. Table D is $D_{1,r}$.

that none of its halves match. In this case we save verification time. On the other hand, the plain technique needs $O(\sigma^\ell)$ space, while hierarchical verification needs much more, $O(r\sigma^\ell)$. This means that, given an amount of main memory, we can use a larger ℓ with the plain technique, which may result in less verifications.

Another possible drawback of hierarchical verification is that, in the worst case, it will work $O(rm)$ to reach the leaves and still will pay $O(rm^2/w)$ for all the verifications, just like the plain algorithm. However, if this becomes an issue, this means that the whole scheme will work poorly because α is too high.

Figures 6 and 7 give code for hierarchical preprocessing and verification. Since now verification proceeds pattern-wise rather than as a block for all the patterns, we need to record in c_i the last text position verified for P^i. **HierarchyVerify** is in charge of doing all necessary verifications and finally shift the window. It first tries to discard the window for all the patterns in one shot. If not possible, it divides the set of patterns into two and yields the smallest shift among the two subsets. When faced against a single pattern that cannot shift the window, it verifies the window for that pattern.

Note that verification would benefit if the patterns we group together are as similar as possible, in terms of numbers of differences. The more similar the patterns are, the larger are the average ℓ-gram distances. A simple clustering method is to group consecutive patterns after sorting them as follows: We start with any pattern, follow with the pattern that minimizes the edit distance to the previous pattern, and so on. This simple clustering technique requires $O(r^2m^2)$ time.

3.3 Reducing Preprocessing Time

Either if we use plain or hierarchical verification, preprocessing time is an issue. We have to search every pattern for every ℓ-gram, resulting in $O(r\ell m\sigma^\ell)$ preprocessing time. In the case of hierarchical verification we pay an additional $O(r\sigma^\ell)$ time to

```
HierarchyVerify (i, j, pos)
 1.   npos ← CanShift(pos, D_{i,j})
 2.   If pos ≠ npos Then Return npos
 3.   If i ≠ j Then
 4.       p ← ⌊(i + j)/2⌋
 5.       Return min(HierarchyVerify(i, p, pos), HierarchyVerify(p + 1, j, pos))
 6.   If pos + 1 > c_i Then
 7.       Initialize verification algorithm of P^i
 8.       c_i ← pos + 1
 9.   Run verification for P^i in text area T_{c_i...pos+m+k}
10.   c_i ← pos + m + k + 1
11.   Return npos + 1
```

```
Search (T_{1...n}, P^1_{1...m} ... P^r_{1...m}, k)
 1.   Preprocess ( )
 2.   For j ∈ 1 ... r Do c_j ← 0
 3.   i ← 0
 4.   While i ≤ n − (m − k) Do
 5.       i ← HierarchyVerify(1, r, i)
```

Fig. 7. The search algorithm using hierarchical verification.

create the D tables of the internal nodes, but this is negligible compared to the cost to process the individual patterns.

We present now a method to reduce the preprocessing time to $O(rm\sigma^\ell)$, which has been used before in the context of indexed approximate string matching [Navarro et al. 2000]. Instead of running the ℓ-grams one by one over a pattern P, we form a trie data structure of all the ℓ-grams. For every trie node whose path from the root spells out the string S, we compute the last row of the C matrix corresponding to searching for S inside P. For this sake we use the previous matrix row, which was computed for the parent node. Hence, if we traverse the trie using a classical depth first search recursion and compute a new matrix row at each invocation, then the execution stack contains the matrix computed up to now, so we use the row computed at the invoking process to compute the row of the invoked process. Since we work $O(m)$ at every trie node and there are $O(\sigma^\ell)$ nodes, the overall process takes $O(m\sigma^\ell)$ time. It needs just space for the stack, $O(m\ell)$. By repeating this over each pattern we obtain $O(rm\sigma^\ell)$ time.

Note finally that the trie of ℓ-grams does not need to be explicitly built, as we know that we have every possible ℓ-gram and hence can use an implicit method to traverse all them without actually storing them. Only the minima over the final rows are stored into the corresponding D entries. Figure 8 shows the code.

Actually, we use Myers' algorithm [Myers 1999] rather than dynamic programming to compute the matrix rows, which makes the preprocessing time $O(rm\sigma^\ell/w)$. For this sake we need to modify Myers' algorithm so that it takes the ℓ-gram as the text and P^i as the pattern. This means that the matrix is transposed, so the current "column" starts with zeros and at the i-th step its first cell has the value i. The necessary modifications are simple and are described, for example, in [Hyyrö and Navarro 2002].

```
RecPreprocessD (P, S, Cold, D)
1.   If |S| = ℓ Then D[S] ← min_{0≤j≤m} Cold_j
2.   Else
3.       For s ∈ Σ Do
4.           Cnew_0 ← |S|
5.           For j ∈ 1...m Do
6.               If s = P_j Then Cnew_j ← Cold_{j-1}
7.               Else Cnew_j ← 1 + min(Cold_{j-1}, Cold_j, Cnew_{j-1})
8.           RecProcessD (P, Ss, Cnew, D)
```

```
PreprocessD (i)
1.   For j ∈ 0...m Do C_j ← 0
2.   RecPreprocessD (P^i, ε, C, D_{i,i})
```

Fig. 8. Faster preprocessing for a single table. ε denotes the empty string.

The only complication is how to obtain the value $\min_{0 \leq j \leq m} C_{\ell,j}$ from Myers' compressed representation of C as a bit vector of increments and decrements. A solution is to use bit magic, so as to store preprocessed answers that give the total increment and minimum value for every bit mask of a given length. Since C is represented using two bit vectors of m bits (one for increments and the other for decrements), we need $O(2^{2x})$ space in order to process the bit vector in $O(m/x)$ time. A reasonable choice not affecting the time complexity is $x = w/4$ for 32-bit machines or $x = w/8$ for 64-bit machines (for a table of 2^{16} entries). This was done, for example, in [Fredriksson 2003].

3.4 Packing Counters

Our final optimization resorts to bit-parallelism, that is, to storing several values inside the same computer word (this has been also used, for example, in the counting algorithm [Baeza-Yates and Navarro 2002]). For this sake we will denote the bitwise *and* operation as "&", the *or* as "|", and the bit complementation as "∼". Shifting i positions to the left (right) is represented as "$<< i$" ("$>> i$"), where the bits that fall are discarded and the new bits that enter are zero. We can also perform arithmetic operations over the computer words. We use exponentiation to denote bit repetition, such as $0^3 1 = 0001$, and write the most significant bit at the leftmost position.

In our process of adding up differences, we start with zero differences and grow at most up to $k + \ell$ differences before abandoning the window. This means that it suffices to use $B = \lceil \log_2(k + \ell + 1) \rceil$ bits to store a counter. Instead of taking minima over several patterns, we could separately store their counters in a single computer word C of w bits ($w = 32$ or 64 in current architectures). This means that we could store $A = \lfloor w/B \rfloor = O(w/\log k)$ counters in a single machine word C.

Consequently, we should keep several difference counts in the same machine word of a D cell. We can still add up our counter and the corresponding D cell and all the counters will be added simultaneously, so the cost is exactly the same as for one single counter or pattern.

Every text window must be traversed until *all* the counters exceed k, so we need a mechanism to check for this condition over all the counters in a single operation. A solution is to initialize the counters not at zero but at $2^{B-1}-k-1$, which ensures that the highest bit in each counter will be activated as soon as the counter reaches the value $k+1$. However, this means that the values stored inside the counters may now reach $2^{B-1}+\ell-1$. This will not cause overflow as long as $2^{B-1}+\ell-1 < 2^B$, that is, $2\ell \leq 2^B$. So in fact B should be chosen such that $2^B > \max(k+\ell, 2\ell-1)$. Moreover, we have to ensure that $2^{B-1}-k-1 \geq 0$ to properly initialize the counters. Overall, this means $B = \lceil \log_2 \max(k+\ell+1, 2\ell, 2k+1) \rceil$.

With this arrangement, in order to check whether all the counters have exceeded k, we simply check whether all the highest bits of all the counters are set. This is achieved using the bitwise *and* operation: Let $H = (10^{B-1})^A$ be the bit mask where all the highest bits of the counters are set. Then, all the counters have exceeded k if and only if $H \ \& \ C = H$. In this case we can abandon the window.

Note that it is still possible that our counters overflow, because we can have that some of them have exceeded $k+\ell$ while others have not. We avoid using more bits for the counters and at the same time ensure that, once a counter has its highest bit set, it will stay with this bit set. Before adding $C \leftarrow C + D[S]$, we remove all the highest bits from C, that is, we assign $O \leftarrow H \ \& \ C$, and replace the simple sum by the assignment $C \leftarrow ((C \ \& \sim H) + D[S]) \mid O$. Since we have selected B such that $\ell \leq 2^{B-1}$, adding $D[S]$ to a counter with its highest bit clear cannot cause an overflow. Note also that highest bits that are already set are always preserved.

This technique permits us searching for $A = \lfloor w/B \rfloor$ patterns at the same time. If we have more patterns we resort to grouping. In a plain verification scenario, we can group r/A patterns in a single counter and search for the A groups simultaneously, with the advantage of having to verify only r/A patterns instead of all the r patterns whenever a window requires verification. In a hierarchical verification scenario, the result is that our hierarchy tree has arity A instead of two, and has no root. That is, there are A tree roots that are searched for together, and each root packs r/A patterns. If one such node has to be verified, then we consider its A children nodes (that pack r/A^2 patterns each), all together, and so on. This reduces not only verification costs but also the preprocessing space, since we need less tables.

We have also to consider how this is combined with the optimization algorithm of Section 3.1, since the best choice to maximize one counter may not be the best choice to maximize another. The solution is to pack also the different values of M_u in a single computer word. The operation of Eq. (1) can be perfectly done in parallel for several counters, as long as we replace the sum by the above technique to avoid overflows. The only obstacle is the maximum, but this has already been solved [Paul and Simon 1980].

If we have to compute $\max(X, Y)$, where X and Y contain several counters properly aligned, in order to obtain the counter-wise maxima, we need an extra highest bit per counter, which is always zero. Say that counters have now $B+1$ bits, counting this new highest bit. We precompute the bit mask $J = (10^B)^A$ (where now $A = \lfloor w/(B+1) \rfloor$) and perform the operation $F \leftarrow ((X \mid J) - Y) \ \& \ J$. The result is that, in F, each highest bit is set if and only if the counter of X is larger than that of Y. We now compute $F \leftarrow F - (F >> B)$, so that the counters

CanShift $(i,\ D)$
1. $B \leftarrow \lceil \log_2 \max(k+\ell+1, 2\ell, 2k+1) \rceil$
2. $A \leftarrow \lfloor w/(B+1) \rfloor$
3. $H \leftarrow (010^{B-1})^A$
4. $J \leftarrow (10^B)^A$
5. **For** $u \in m-k-\ell+2 \ldots m-k+1$ **Do**
6. $\quad M_u \leftarrow (2^{B-1}-k-1) \times (0^B1)^A$
7. $u \leftarrow m-k-\ell+1$
8. **While** $u \geq 0$ **Do**
9. $\quad X \leftarrow M_{u+\ell}$
10. $\quad O \leftarrow X \ \&\ H$
11. $\quad X \leftarrow ((X \ \&\ \sim H) + D[T_{i+u \ldots i+u+\ell-1}]) \mid O$
12. $\quad Y \leftarrow M_{u+1}$
13. $\quad F \leftarrow ((X \mid J) - Y) \ \&\ J$
14. $\quad F \leftarrow F - (F >> B)$
15. $\quad M_u \leftarrow (X \ \&\ F) \mid (Y \ \&\ \sim F)$
16. $\quad$ **If** $H \ \&\ M_u = H$ **Then Return** $i+u+1$
17. $\quad u \leftarrow u-1$
18. **Return** i

Fig. 9. The bit-parallel version of **CanShift**. It requires that D is preprocessed by packing the values of A different patterns in the same way. Lines 1–6 can in fact be done once at preprocessing time.

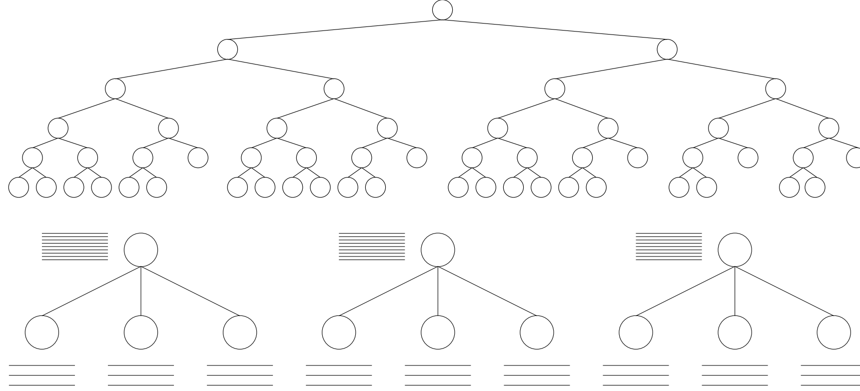

Fig. 10. On top, the basic pattern hierarchy for 27 patterns. On the bottom, pattern hierarchy with bit-parallel counters (27 patterns).

where X is larger than Y have all their bits set in F, and the others have all the bits in zero. Finally, we choose the maxima as $\max(X,Y) \leftarrow (X \ \&\ F) \mid (Y \ \&\ \sim F)$.

Figure 9 shows the bit-parallel version of the counter accumulation, and Figure 10 shows an example of pattern hierarchy.

4. ANALYSIS

We start by analyzing our basic algorithm and proving its average-optimality. In particular, our basic analysis does not make any use of bit-parallelism, so as to be comparable with classical developments. Later we consider the impact of the

diverse practical improvements proposed on the average performance. This section can be safely skipped by readers interested only in the practical aspects of the algorithm.

4.1 Basic Algorithm

We analyze an algorithm that is necessarily worse than ours 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 ℓ-grams, for some constant $0 < c < 1$ that will be considered shortly. After having read them, it checks whether *any* of the ℓ-grams produces less than $c\ell$ differences in the D table. If there is at least one such ℓ-gram, the window is verified and shifted by 1. Otherwise, we have at least $1 + \lfloor k/(c\ell) \rfloor > k/(c\ell)$ ℓ-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 $1 + \lfloor k/(c\ell) \rfloor$ ℓ-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 ℓ-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:

$$\frac{\ell + k/c}{m - k - (\ell + k/c) + 1}\, n \quad = \quad O\left(\frac{\ell + k}{m}\, n\right), \tag{2}$$

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 [Chang and Marr 1994], rewritten in a way more convenient for us.

Lemma 1 [Chang and Marr 1994] The probability that two random ℓ-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 [Chang and Marr 1994] If S is an ℓ-gram that matches inside a given

string P (larger than ℓ) with less than $c\ell$ differences, then S has a common subsequence of length $\ell - c\ell$ with some ℓ-gram of P.

Given Lemmas 1 and 2, the probability that a given ℓ-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 ℓ-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 ℓ-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

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

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. (2) and (3) we obtain the following condition on ℓ:

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

and therefore a sufficient condition on ℓ that retains the optimality of good windows is

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

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 α 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 ℓ goes to infinity. If we let c approach $k/(m-\ell-k)$, then ℓ 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 ℓ, our algorithm is on average

$$O\left(\frac{n(k + \log_\sigma(rm))}{m}\right) \qquad (5)$$

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.)

Recall that our preprocessing cost is $O(mr\sigma^\ell)$, thanks to the smarter preprocessing of Section 3.3. Given the value of ℓ, this is $O(m^5r^3\sigma^{O(1)})$. The space with

plain verification is $\sigma^\ell = m^4 r^2 \sigma^{O(1)}$ integers. We must consider the impact of preprocessing time and space in the overall time.

The preprocessing cost must not exceed the average search time of the algorithm. This means that it must hold $mr\sigma^\ell = O((k+\ell)n/m)$, that is $\ell \le \log_\sigma(kn/(m^2 r))$, or $r = O(n^{1/3}/m^2)$. Also, it is necessary that $\ell \le m-k$, which means $r = O(\sigma^{(m-k)/2}/m^2)$. This is looser than our previous limit $r = O(\sigma^{o(m)})$. Finally, we must have enough memory $\mathcal{M}$ to hold σ^ℓ entries, so $\ell \le \log_\sigma \mathcal{M}$. Therefore, the claimed complexity is obtained provided the above limits on r and $\mathcal{M}$ hold.

To summarize, we have shown that we are able to work, on average, $O(n(k+\log_\sigma(rm))/m)$ time whenever $\alpha < (1-e/\sqrt{\sigma})/(2-e/\sqrt{\sigma}) = 1/2 - O(1/\sqrt{\sigma})$, $r = O(\min(n^{1/3}/m^2, \sigma^{o(m)}))$ and we have $\mathcal{M} = O(m^4 r^2 \sigma^{O(1)})$ memory available.

It has been shown that, for a single pattern, $O(n(k+\log_\sigma m)/m)$ is optimal [Chang and Marr 1994]. This comes from adding up two facts. The first is that it is necessary to inspect at least $k+1$ characters in order to skip a given text window of length m, so we need at least $\Omega(kn/m)$ character inspections. The second is that the $\Omega(n \log_\sigma(m)/m)$ lower bound of Yao [Yao 1979] for exact string matching applies to approximate searching too, as exact searching is included in the approximate search problem (that is, we have to report the exact occurrences of P as well). When searching for r patterns, this second lower bound becomes $\Omega(n \log_\sigma(rm)/m)$ [Navarro and Fredriksson 2004]. Hence our algorithm is average-optimal.

Moreover, used on a single pattern we obtain the same optimal complexity of Chang & Marr [Chang and Marr 1994], 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 obtained not only the first average-optimal multipattern approximate search algorithm, but also improved the only average-optimal simple approximate search algorithm with respect to its area of applicability.

Note that, except that on the difference ratio, all the conditions of applicability can be weakened by reducing r. Therefore, if some of those conditions is not met, we can find the maximum $r' < r$ such that they hold, divide our pattern set into $\lceil r/r' \rceil$ groups of at most r' patterns each, and search for each set separately. The resulting algorithm is not optimal anymore, but we can still apply the technique.

In particular, if the problem is that our main memory available is $\mathcal{M} < \sigma^\ell$, we can use some space tradeoff techniques. A first one is that we can use the maximum value $\ell' = \log_\sigma \mathcal{M}$ that can be handled with our available memory. The r value that makes the search optimal with that ℓ' value is $r' = \mathcal{M}^{d/2}/m^2$. Hence, if we search for r/r' separate groups of r' patterns each, we get a total search time of

$$O\left(\frac{rm^2}{\mathcal{M}^{d/2}} \, \frac{k+\log_\sigma \mathcal{M}}{m} \, n\right) \quad = \quad O\left(\frac{rm(k+\log_\sigma \mathcal{M})}{\mathcal{M}^{d/2}} \, n\right).$$

Another simple technique is to map the alphabet onto a smaller alphabet set, so that several characters are merged into one. This is particularly interesting when the alphabet is not uniform, so we can pack several characters of low probability into one. (It should be clear that the optimum is to make the merged probabilities as uniform as possible, as this maximizes average distances.) The result effectively reduces σ, so that σ^ℓ can now fit in our memory. However, to retain optimality, ℓ should get larger. It turns out that, in order to fit the available memory, we must

merge the alphabet onto $\sigma' = (2c \ln c + 2(1-c)\ln(1-c))/(\ln(m^4 r^2)/\ln \mathcal{M} - (1-c))$ different characters, for a total complexity of

$$O\left(\left(k + \frac{\log(rm)}{\log \mathcal{M}}\right)\frac{n}{m}\right).$$

Comparing, we find out that alphabet mapping is analytically more promising than grouping, under reasonable assumptions ($\mathcal{M}^{\ln \mathcal{M}} > rm$). Later, we test in practice how these techniques perform.

4.2 Improvements

We analyze now the diverse practical optimizations proposed over our basic scheme, except for that of Section 3.3, that is already included in the basic analysis.

The optimal choice of ℓ-grams (Section 3.1), as explained, results in equal or better performance for every possible text window. Hence the average complexity cannot change because it is already optimal.

On the other hand, if we do not use the optimization, we can manage to read whole ℓ-grams in single computer instructions, for an average number of $O((1 + k/\log_\sigma(rm))n/m)$ instructions executed. This breaks the lower bound simply because the lower bound counts number of characters read and we are counting computer instructions. An ℓ-gram must fit in a computer word of length $\log_2 \mathcal{M}$ because $\ell \log_2 \sigma \le \log_2 \mathcal{M}$, otherwise $\mathcal{M}$ is not enough to hold the σ^ℓ entries of D.

The fact that we use Myers' algorithm [Myers 1999] instead of dynamic programming reduces the $O(m^2)$ costs in the verification and preprocessing to $O(m^2/w)$. This does not change the complexities but it permits reducing ℓ a bit in practice.

Let us now analyze the effect of hierarchical verification (Section 3.2). This time we start with r patterns, and if the block requires verification, we run two new scans for $r/2$ patterns, and continue the process until a single pattern asks for verification. Only then we perform the dynamic programming verification. Let $p = (1 + k/(c\ell))ma\sigma^{-d\ell}/\ell$. Then the probability of verifying the root node is pr. For a non-root node, the probability that it requires verification given that the parent requires verification is $Pr(child/parent) = Pr(child \wedge parent)/P(parent) = Pr(child)/Pr(parent) = p(r/2)/(pr) = 1/2$, since if the child requires verification then the parent requires verification. Then the number of times we scan the whole block is on average

$$pr(2 + 2(1/2(2 + 2(1/2 \ldots \quad = \quad 2pr \log_2 r.$$

Hence the total character inspections for the scans that require verifications is $O(pmr \log r)$. Finally, each individual pattern is verified provided an ℓ-gram of the text block matches inside it. This accounts for $O(prm^2)$ verification cost. Hence the overall cost of bad windows under hierarchical verification is

$$O\left(\frac{(1 + k/(c\ell))am^2 r(m + \log r)}{\ell}\sigma^{-d\ell}\ n\right),$$

which is clearly better than the cost with plain verification. The condition on ℓ to obtain the same search time of Eq. (5) is now

$$\ell \ \ge \ \frac{\log_\sigma(m^3 r(m + \log_2 r))}{d} \ = \ \frac{3\log_\sigma m + \log_\sigma r + \log_\sigma(m + \log_2 r)}{1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)}, \qquad (6)$$

which is smaller and hence requires less preprocessing effort. This time the preprocessing cost is $O(m^4r^2(m + \log r)\sigma^{O(1)}/w)$, lower than with plain verification (regardless of the "$/w$"). The space requirement of hierarchical verification, however, is $2r\sigma^\ell = 2m^3r^2(m + \log_2 r)\sigma^{O(1)}$, larger than with plain verification. However, we notice that it is only slightly larger, $1 + \log(r)/m$ times rather than r times larger as initially supposed. The reason is that ℓ itself is smaller thanks to hierarchical verification. Therefore, it seems clear that hierarchical verification brings only benefits.

Finally, let us consider the use of bit-parallel counters (Section 3.4). This time the arity of the tree is $A = \lfloor w/(1 + \lceil \log_2(k + 1)\rceil)\rfloor$ and it has no root. We have r/A tables in the leaves of the hierarchical tree. The total space requirement is less than $r/(A - 1)$ tables. The verification effort is now $O(pmr \log_A r)$ for scanning and re-scanning, and $O(prm^2)$ for dynamic programming. This puts a less stringent condition on ℓ:

$$\ell \;\geq\; \frac{\log_\sigma(m^3 r(m + \log_A r))}{d} \;=\; \frac{3\log_\sigma m + \log_\sigma r + \log_\sigma(m + \log_A r)}{1 - c + 2c\log_\sigma c + 2(1 - c)\log_\sigma(1 - c)},$$

and reduces the preprocessing effort to $O(m^4r^2(m + \log_A r)\sigma^{O(1)}/w)$. The space requirement is $\lceil r/(A - 1)\rceil\sigma^\ell = m^3r^2(m + \log_A r)\sigma^{O(1)}/(A - 1)$. With plain verification the space requirement is still smaller, but the difference is this time even less significant.

Actually, using bit-parallel counters the probability of having a bad window is reduced, because the sum of distances must exceed k inside some of the A groups. However, the difference goes unnoticed under our simplified analysis (where a window is bad if some considered ℓ-gram matches with few enough differences inside any pattern).

5. SOME VARIANTS OF OUR ALGORITHM

We briefly present in this section several variants of our algorithm. They are not analytically superior to our basic algorithm, but in some cases they turn out to be interesting alternatives in practice.

5.1 Direct Extension of Chang & Marr

The algorithm of Chang & Marr [Chang and Marr 1994] (Section 2.4) can be directly extended to handle multiple patterns. We perform exactly the same preprocessing of our algorithm in order to build the D table and then the scanning phase is the same as in Section 2.4: If we surpass k differences inside a block we are sure that none of the patterns match, since there are t ℓ-grams inside the block that need overall more than k differences in order to be found inside any pattern. Otherwise, we check the patterns one by one over the block. All the improvements we have proposed in Section 3 for our algorithm can be applied to this version too.

Figure 11 gives the code, including hierarchical verification. **HierarchyVerify** is the same of Figure 7 except that the c_j markers are not used. Rather, for block $bi + 1 \ldots bi + b$ we check the area $T_{ib+b+1-m-k \ldots ib+m+k}$ as explained in Section 2.4. The other difference is inside **Preprocess**, where the computation of ℓ must be done according to the following analysis.

Search $(T_{1\dots n},\ P^1_{1\dots m}\dots P^r_{1\dots m},\ k)$
1. **Preprocess** ( )
2. $b \leftarrow \lceil (m-k)/2 \rceil$
3. **For** $i \in 0 \dots \lfloor n/b \rfloor - 1$ **Do**
4. **HierarchyVerify** $(1, r, b \cdot i)$

Fig. 11. Direct extension of Chang & Marr algorithm.

The analysis of this algorithm is a bit simpler than that of our central algorithm, because there are no sliding but just fixed windows. Overall, there are $\lfloor 2n/(m-k) \rfloor = O(n/m)$ windows to consider. Hence we can apply the same simplification of Section 4 and obtain a very similar result about the probability of a window being verified.

There are two main differences, however. The first is that now there are $O(n/m)$ candidates to bad windows, while in the original analysis there are potentially $O(n)$ bad windows. Hence the overall complexity in this case is

$$O\left(\frac{n}{m}\left((1+k/(c\ell))m^3r^2a\sigma^{-d\ell}/\ell + \ell + \frac{k}{c}\right)\right),$$

where the first summand corresponds to the average plain verification cost and the others to the average (fixed in our simplified model) block scanning cost. This slightly reduces the minimum value ℓ must have in order to achieve optimality to

$$\ell \;=\; \frac{3\log_\sigma m + 2\log_\sigma r}{1 - c + 2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)},$$

so it is possible that this algorithm performs better than our central algorithm when the latter is unable of using the right ℓ because of memory limitations.

The second difference is that windows are of length $(m-k)/2$ instead of $m-k$, and therefore $\ell + k/c$ must not reach $(m-k)/2$. This reduces the difference ratio up to which the method is applicable to $\alpha < 1/3 - O(1/\sqrt{\sigma})$, so we expect our central algorithm to work well for α values where this method does not work anymore.

5.2 A Linear Time Algorithm

When the difference ratio becomes relatively high, it is possible that our central algorithm retraverses many times the same text in the scanning phase, apart from verifying a significant part of the text. A way to alleviate the first problem is to slide the window over the text ℓ-gram by ℓ-gram, updating the cumulative sum of D values over all the ℓ-grams of the window.

That is, the window contains t ℓ-grams, where $t = \lfloor (m-k+1)/\ell \rfloor - 1$. If we only consider text windows for the form $T_{i\ell+1\dots i\ell+t\ell}$, then we are sure that every occurrence contains a complete window. Then, if the ℓ-grams inside the window add up more than k differences, we can move to the next window. Otherwise, before moving we must verify the area $T_{i\ell+t\ell-(m+k)\dots i\ell+(m+k)}$

Since consecutive windows overlap with each other by $t-1$ ℓ-grams, we are able to update our difference accumulation from one text window to the next in constant time. This is rather easy, although it does not permit anymore the use of the optimization of Section 3.1.

Search $(T_{1\ldots n},\ P^1_{1\ldots m}\ldots P^r_{1\ldots m},\ k)$
1. **Preprocess** ( )
2. $t \leftarrow \lfloor (m-k+1)/\ell \rfloor - 1$
3. $M \leftarrow 0$
4. **For** $i \in 0\ldots t-2$ **Do** $M \leftarrow M + D[T_{i\ell+1\ldots i\ell+\ell}]$
5. **For** $i \in t-1\ldots \lfloor n/\ell \rfloor - 1$ **Do**
6. $\quad M \leftarrow M + D[T_{i\ell+1\ldots i\ell+\ell}]$
7. $\quad$ **If** $M \le k$ **Then** Verify $T_{(i-t)\ell-2k+2\ldots(i-t+1)\ell+m+k}$
8. $\quad M \leftarrow M - D[T_{(i-t+1)\ell+1\ldots(i-t+1)\ell+\ell}]$

Fig. 12. Our linear time algorithm. The verification area has been optimized a bit.

The resulting algorithm takes $O(n)$ time. Its difference ratios of applicability are the same of our central algorithm, $\alpha < 1/2 - O(1/\sqrt{\sigma})$, since this basically depends on the window length. Therefore, analytically the algorithm does not bring any benefit. However, we expect it to be of some interest in practice for high difference ratios.

Figure 12 shows this algorithm. The verification can also be done hierarchically. The result reminds algorithm LET of Chang & Lawler [Chang and Lawler 1994]. However, the latter used exact matching of varying-length window substrings, rather than approximate matching of length-ℓ window substrings. As all the filters based on exact matching, the difference ratio of applicability of LET is just $\alpha < 1/\log_\sigma m + O(1)$. On the other hand, if $\ell = 1$, then our linear time filter becomes an elaborate implementation of the counting filter [Grossi and Luccio 1989; Baeza-Yates and Navarro 2002].

5.3 Enforcing Order Among ℓ-grams

Note that our algorithms do not induce order on the ℓ-grams. They can appear in any order, as long as their total distance to the patterns is at most k. The filtering efficiency can still be improved by requiring that the ℓ-grams from the pattern must appear in approximately same order in the text. This approach was used in [Sutinen and Tarhio 1996], in particular in algorithm LAQ.

The idea is that we are interested in occurrences of P that fully contain the current window, so an ℓ-gram at window position $u\ell+1\ldots u\ell+\ell$ can only match inside $P_{u\ell-\ell+1\ldots u\ell+\ell+k-1} = P_{(u-1)\ell+1\ldots(u+1)\ell+k-1}$, since larger displacements mean that the occurrence is also contained in an adjacent pattern area. Hence, the D table is processed in a slightly more complex way. We have t tables D, one per ℓ-gram position in the window. Table D_u gives distances to match an ℓ-gram at position $u\ell$ in the window, so it gives minimum distance in the area $P_{(u-1)\ell+1\ldots(u+1)\ell+k-1}$ instead of in the whole P.

At a given window $i\ell+1\ldots i\ell+t\ell$ we compute $M_i = \sum_{0\le u<t} D_u[T_{i\ell+u\ell+1\ldots i\ell+u\ell+\ell}]$. If M exceeds k we do not need to verify the window. In order to shift the window, however, the current value of M is of no use, which defeats the whole purpose of the linear filter. However, bit-parallelism is useful here. We can store t consecutive M values in a single computer word, which we call again M. There is a single table D where $D[S]$ contains the concatenation of the counters $D_t[S]$, $D_{t-1}[S]$, …, $D_1[S]$. When we read the i-th text ℓ-gram, S, we update $M = (M << v) + D[S]$, being

```
Preprocess ( )
 1.    ℓ ← according to Eq. (4)
 2.    t ← ⌊(m − k + 1)/ℓ⌋ − 1
 3.    v ← ⌈log_2(tℓ + 1)⌉
 4.    For S ∈ Σ^ℓ Do
 5.        D[S] ← 0
 6.        For u ∈ 1 . . . t Do
 7.            min ← ℓ
 8.            For i ∈ 1 . . . r Do
 9.                min ← min(min, MinDist(S, P^i_{(u−1)ℓ+1...(u+2)ℓ+k−1}))
 10.           D[S] ← D[S] | (min << (u − 1)v)

Search (T_{1...n}, P^1_{1...m} . . . P^r_{1...m}, k)
 1.    Preprocess ( )
 2.    M ← 0
 3.    For i ∈ 0 . . . t − 2 Do M ← M + D[T_{iℓ+1...iℓ+ℓ}]
 4.    For i ∈ 0 . . . ⌊n/ℓ⌋ − 1 Do
 5.        M ← (M << v) + D[T_{iℓ+1...iℓ+ℓ}]
 6.        If (M >> (t − 1)v) ≤ k Then Verify T_{(i−t)ℓ−2k+2...(i−t+1)ℓ+m+k}
```

Fig. 13. Extension of LAQ algorithm to multiple patterns. We assume that the computer word is exactly of length tv for simplicity, otherwise all the values must be properly aligned to the left.

v the width in bits of the counters. As we do this over t consecutive ℓ-grams, the t-th counter in M contains M_{i-t+1} properly computed.

The same idea can be applied for multiple patterns as well, by storing the minimum distance over all the patterns in D_u. Figure 13 gives the pseudocode. Hierarchical verification is of course possible here as well, although for simplicity we have written down the plain version of **Preprocess**.

5.4 Ordered ℓ-grams for Backward Matching

In the basic algorithm we permit that the ℓ-grams match anywhere inside the patterns. This has the disadvantage of being excessively permissive. However, it has an advantage: When the ℓ-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 ℓ-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 ℓ 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 $P_{1...m-\ell}$ rather than $P_{1...m}$. 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

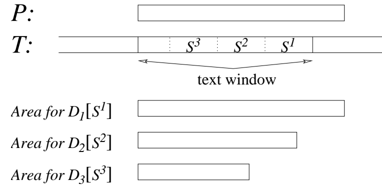

Fig. 14. Pattern P aligned over a text window. The text window and ℓ-grams correspond to the basic algorithm (Section 3) and for the algorithm of Section 5.4. The window is of length $m-k$. The areas correspond to the Section 5.4. All the areas for S^t for the basic algorithm are the same as for D_1.

CanShift $(i,\ D)$
1. $M \leftarrow 0$
2. $p \leftarrow m-k$
3. $t \leftarrow 1$
4. **While** $p \geq \ell$ **Do**
5. $\quad p \leftarrow p-\ell$
6. $\quad M \leftarrow M + D_t[T_{i+p+1\ldots i+p+\ell}]$
7. $\quad$ **If** $M > k$ **Then Return** $i+p+1$
8. $\quad t \leftarrow t+1$
9. **Return** i

Fig. 15. The algorithm for stricter matching condition.

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. 14 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 ℓ-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. The preprocessing cost is increased to $O(r(\sigma^\ell(m/w+m)+U)) = O(r\sigma^\ell m)$. Fig. 15 gives the code.

5.5 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 ℓ-grams should match the pattern more or less at the same position they have in the window. We therefore combine the ideas of Sections 5.3 and 5.4.

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 ℓ-gram

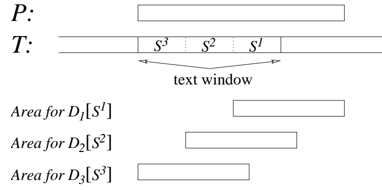

Fig. 16. Pattern P aligned over a text window. The areas for the algorithm of Section 5.5. The areas overlap by $\ell + k - 1$ characters. The window length is $U\ell$.

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 ℓ-gram position in the window. Table D_{U-t+1} gives distances to match the t-th window ℓ-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. 16.

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 ℓ. 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 $M_u = \sum_{t=1}^{u} (D[S^t] << (t-1)l)$. 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 [Sutinen and Tarhio 1996], but their (single pattern) proposal was based on direct extension of Chang & Marr algorithm [Chang and Marr 1994].

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 ℓ after having read u ℓ-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 2ℓ, 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 ℓ-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

---

CanShift $(i,\ D)$

1. $l \leftarrow \lceil \log_2 \max(k+\ell+1, 2\ell, 2k+1) \rceil$
2. $U = \lfloor (m-k-\ell+1)/\ell \rfloor$
3. $H \leftarrow (10^{l-1})^U$
4. $O \leftarrow 10^{Ul-1}$
5. $C \leftarrow (2^{l-1}-k-1) \times (0^{l-1}1)^U$
6. $p \leftarrow U\ell$
7. **While** $p > 0$ **Do**
8. $\quad p \leftarrow p - \ell$
9. $\quad C \leftarrow ((C \ \&\ \sim H) + D[T_{i+p+1\ldots i+p+\ell}])\ |\ (C \ \&\ H)$
10. $\quad$ **If** $C \ \&\ O = O$ **Then Return** $i + (U - 1 - \lfloor \log_2((C \ \&\ H)\hat{}H) \rfloor / l)\ell$
11. **Return** i

Fig. 17. **CanShift** with fast shifting. It requires that D is preprocessed as in LAQ. Lines 1–5 can in be done once at preprocessing time.

the value is larger than $k + 1$ does not give us any further advantage. We can therefore apply the same bounding technique that was used in Section 3.4, so that $l = O(\log_2(k + \ell))$ bits is sufficient.

The problem with this method is how to compute the shift. Recall from Section 3.4 that the bit mask C contains our counters, and the bit mask H has all the highest bits of the counters set, and zeros elsewhere. We want to shift by $s\ell$ whenever the s leftmost highest counter bits in C are set. Let us rather consider $Y = (C \ \&\ H)\hat{}H$, where "^" is the bitwise "xor" operation, so 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 ℓ-grams in the window by more than ℓ, 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 ℓ-grams. This requires that the areas are $\ell + h - 1 + k$ characters long, 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 ℓ.

This phenomenon is well known in exact string matching. For example, the non-optimal Horspool algorithm [Horspool 1980] shifts as soon as the window suffix mismatches the pattern suffix, while the optimal BDM [Crochemore et al. 1994] 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.

6. 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. This was sufficient as the variance was very low.

We ran experiments for alphabet sizes $\sigma=4$ (DNA), $\sigma=20$ (proteins) and $\sigma=96$ (ASCII text). The test data were randomly generated 64MB files. By default, the randomly generated patterns were 64 characters long for DNA and proteins, and 16 characters long for ASCII. However, we also show some experiments with varying pattern lengths.

We show later some experiments using real texts. These are: the E.coli DNA sequence (4,638,690 characters) from Canterbury Corpus[1], real protein data (5,050,292 characters) from TIGR Database (TDB)[2], and the Bible (4047392 characters), from Canterbury Corpus. 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. We have experimentally verified that there is no statistical difference between using real-life texts of 64MB versus replicating a 5MB one.

All the files were represented in plain ASCII, hence we read ℓ-grams at $O(\ell)$ cost. Note that if the texts were stored used only 2 (DNA) and 5 bits (protein) per character, that would have allowed $O(1)$ time access to the ℓ-grams. See [Fredriksson and Navarro 2003] for experiments with this "compressed" searching.

6.1 Preprocessing and Optimality

Table I gives the preprocessing times for various alphabets, number of patterns and ℓ values. We have considered hierarchical verification because it gave consistently better results, so the preprocessing timings include all the hierarchy construction. In particular, they correspond to a binary hierarchy. Using the bit-parallel counters technique produces a shorter hierarchy but it requires slightly more time. The space usage for the D tables is at most $2r\sigma^\ell$ bytes for the binary hierarchy, and slightly less with bit-parallel counters. For example, this means 128MB of memory for DNA with 256 patterns and $\ell=9$. The LAQ algorithm needs much more preprocessing effort, although our implementation was highly optimized in this case (we

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

Table I. Preprocessing times in seconds for $r \in \{1, 16, 64, 256\}$ and for various ℓ-gram lengths. The pattern lengths are $m = 64$ for DNA and proteins, and $m = 16$ for ASCII. Left: binary hierarchy; right: binary hierarchy with LAQ for $k = 0$.

DNA	1	16	64	256	w/ LAQ	1	16	64	256
5	0.00	0.00	0.02	0.06	5	0.00	0.02	0.08	0.33
6	0.00	0.01	0.06	0.23	6	0.01	0.06	0.25	1.02
7	0.01	0.06	0.23	0.91	7	0.02	0.24	0.91	3.72
8	0.01	0.23	0.93	3.80	8	0.05	0.28	3.64	14.53
9	0.07	0.98	3.96	15.94	9	0.20	3.47	14.33	

proteins	1	16	64	256	w/ LAQ	1	16	64	256
2	0.00	0.00	0.01	0.03	2	0.00	0.02	0.06	0.24
3	0.02	0.02	0.10	0.39	3	0.02	0.32	1.31	5.35
4	0.03	1.50	6.11	24.17	4	0.56	9.58		

ASCII	1	16	64	256	w/ LAQ	1	16	64	256
2	0.00	0.03	0.10	0.41	2	0.04	0.65	2.57	10.47
3	0.23	4.76	19.31	77.44	3	18.30			

preprocess each pattern block separately). The values shown are for $k = 0$, which gives the maximum number of blocks, and hence the slowest preprocessing times.

As it can be seen, the maximum values in practice are $\ell \leq 8$ for DNA, $\ell \leq 3$ for proteins, and $\ell \leq 2$ for ASCII. The search times that follow were measured for these maximum values unless otherwise stated. For $r = 256$ patterns, the associated memory requirements using those maximum ℓ values are 32MB, 4MB and 4.5MB, respectively. For Figures 18, 19, and 20 we show only the search times (as we compare our own algorithms), while for the other figures we show the total times (searching and all preprocessing).

Observe that the ℓ values required by our analysis (Section 4) in order to have optimal complexity are, depending on $r \in 1 \ldots 256$, $\ell = 12 \ldots 20$ for DNA, $\ell = 6 \ldots 10$ for proteins, and $\ell = 3 \ldots 5$ for ASCII. In the case of the simpler algorithm of Section 5.1 the values are slightly lower: $\ell = 9 \ldots 17$ for DNA, $\ell = 4 \ldots 8$ for proteins, and $\ell = 2 \ldots 5$ for ASCII. For example, for $r = 256$, the space required for those ℓ values is 8TB (terabytes) for DNA, 12TB for proteins, and 4TB for ASCII.

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.

6.2 Comparing Variants of Our Algorithm

In this section we compare several combinations over variants of our algorithm. They are identified in the plots as follows:

`-Sb`*:* The main algorithm of Section 3. "Sb" stands for "sublinear, reading window backwards".

`-Sf`*:* The sublinear time filter that extends Chang & Marr, Section 5.1. The "f" stands for forward ℓ-gram scanning inside windows.

No label:. The linear time filter of Section 5.2.

L:. The extension of LAQ algorithm, Section 5.3.

Furthermore, we add some modifiers to the above labels to denote different optimizations. If the label is suffixed by -O, then the optimization method of Section 3.1 is applied as well. Suffix -L denotes the stricter matching condition of Section 5.4, and the additional suffix -F denotes the method of Section 5.5 to obtain the shift faster. If the label is prefixed by -B, then also bit-parallel counters (Section 3.4) were used. Note that several possible variations are omitted here (e.g. -Sb -L -O), and several variations are not possible in general (e.g. -B -Sb -L -F). As mentioned, all the algorithms use hierarchical verification.

Figures 18, 19, and 20 show search times for the DNA, proteins, and ASCII alphabets.

As it can be seen, our main algorithm is in most cases the clear winner, as expected from the analysis. The filtering capability of Chang & Marr extension collapses already with quite small difference ratios, due to the use of small text blocks, and this collapse is usually very sharp. Our main algorithm uses larger search windows, so it triggers less verifications and permits larger difference rations. In addition, even with small difference ratios it skips more characters, and hence it is faster for all values of k/m. The main algorithm is usually faster than the linear time filter, because the latter cannot skip text characters. This fact stays true even for large difference ratios.

For LAQ we need bit-parallelism and soon we need more than 32 bits to hold the state of the search. For DNA and proteins we resort to simulated 64 bit counters and this slows down the algorithm. For ASCII we use the native 32-bit counters. These choices are sufficient for our parameters (m, k). If we did not have enough bits, we could use larger and fewer blocks, and correspondingly use sampling rates greater than ℓ (so there is some space between consecutive text ℓ-grams). This would make the filtering capability of the algorithm poorer, but note that on the other hand this could be used to speed up the search if k/m were small enough. We used the full number of blocks in each case, hence the choice of simulated 64 bits for long patterns. The algorithm is comparable with the linear time filter for low k/m (if we can do with 32 bits), but for large k/m it triggers much less verifications, and becomes comparable with our main algorithm, especially with large r and ASCII alphabet. However, LAQ needs much more preprocessing effort (not included in the plots, but see Table I), and hence always loses. Using smaller ℓ values for LAQ did not help.

Optimal choice of ℓ-grams helps sometimes, but is usually slower as it is more complicated. It usually helps significantly on the Chang & Marr variant, to delay the collapse of the algorithm. In a few cases it also helps our main algorithm significantly. The bit-parallel counters, on the other hand, speed up the search for large rm, or in other words, they allow larger r (using smaller ℓ). The use of bit-parallel counters never damages the performance. The combination of optimization with bit-parallel counters is not as attractive as without the counters. It is more complicated to implement with the bit-parallel counters, but in some cases it can push the limit for k/m a bit further.

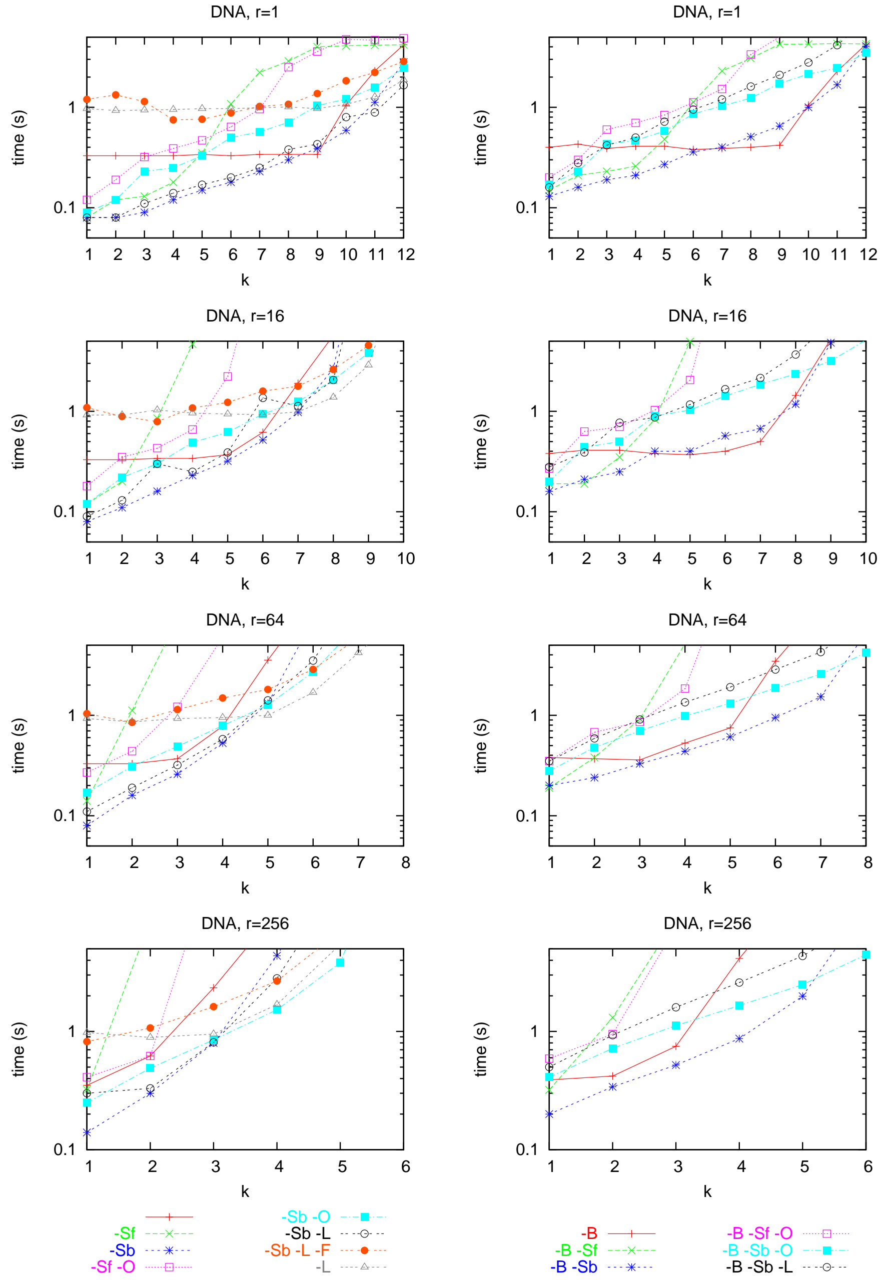

Fig. 18. Search times in seconds. Parameters are $\sigma = 4$, $m = 64$, and $\ell = 8$. The figures show, from top to bottom, $r = 1$, $r = 16$, $r = 64$, and $r = 256$. The right plots use bit-parallel counters. The x-axis is k, and y-axis time (logarithmic scale).

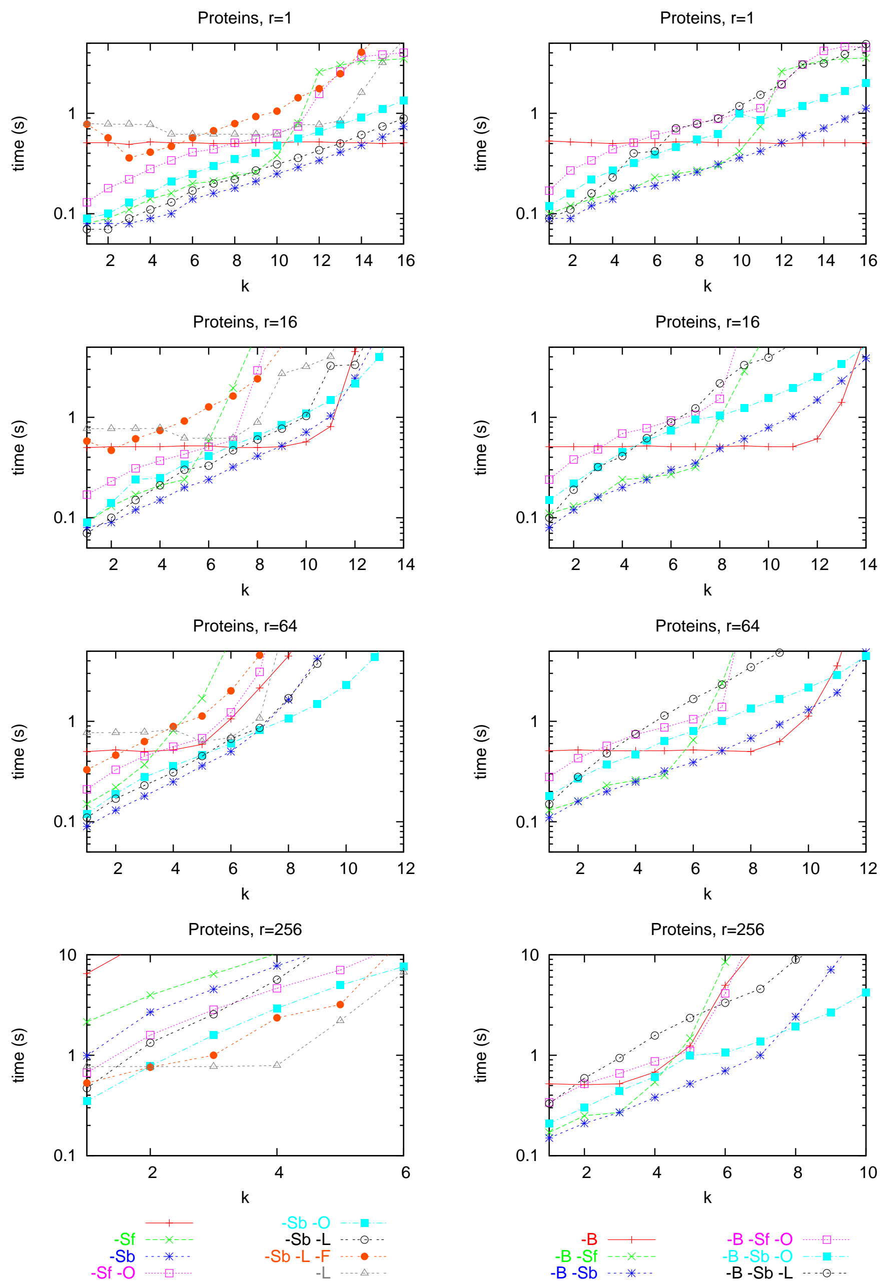

Fig. 19. Search times in seconds. Parameters are $\sigma = 20$, $m = 64$, and $\ell = 3$. The figures show, from top to bottom, $r = 1$, $r = 16$, $r = 64$, and $r = 256$. The right plots use bit-parallel counters. The x-axis is k, and y-axis time (logarithmic scale).

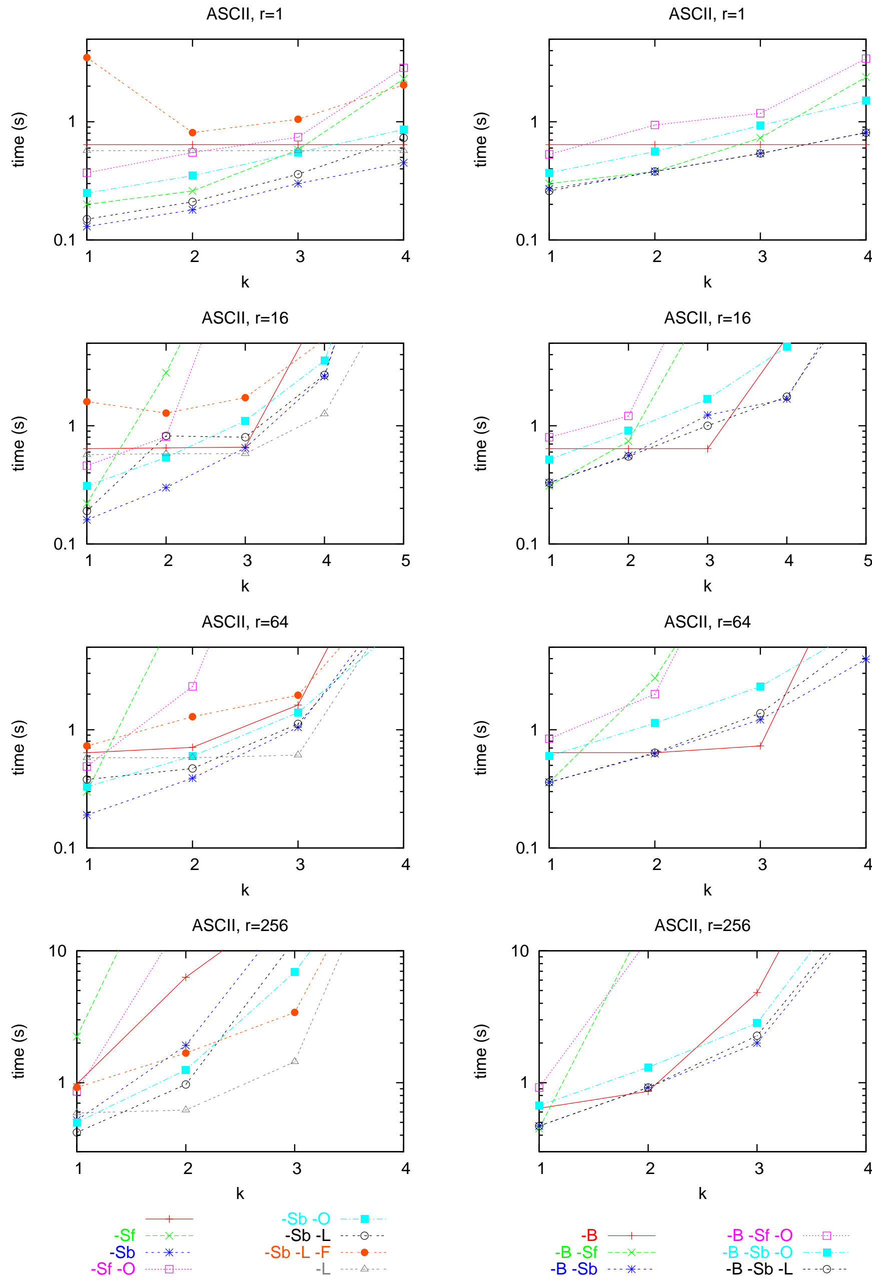

Fig. 20. Search times in seconds. Parameters are $\sigma = 96$, $m = 16$, and $\ell = 2$. The figures show, from top to bottom, $r = 1$, $r = 16$, $r = 64$, and $r = 256$. The right plots use bit-parallel counters. The x-axis is k, and y-axis time (logarithmic scale).

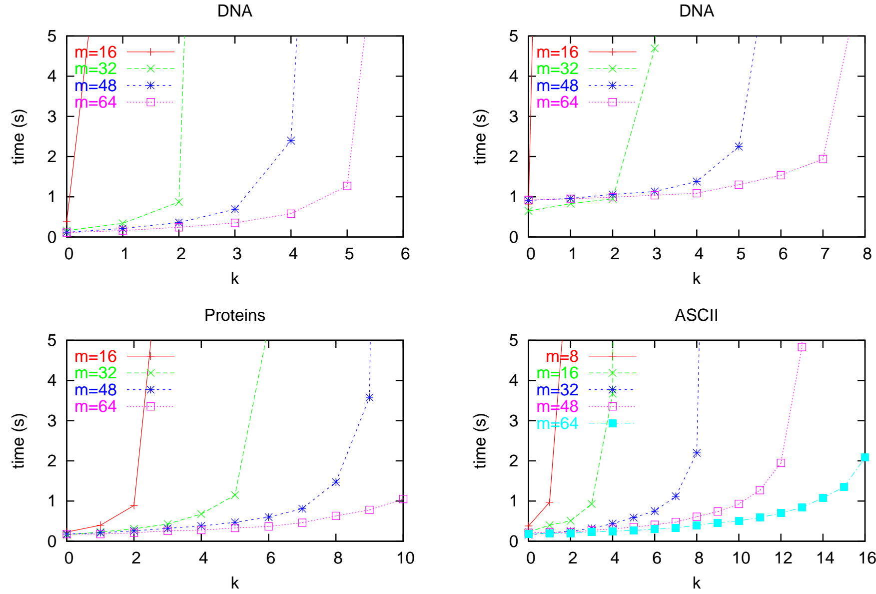

Fig. 21. Total times (preprocesing and searching) in seconds for different m and $r = 64$. Top row, left: DNA and $\ell = 6$, right: DNA and $\ell = 8$. Bottom row, left: proteins and $\ell = 3$, right: ASCII and $\ell = 2$.

Our basic algorithm beats the extensions proposed in Secs. 5.4 and 5.5, with the exception of single pattern real protein, where the algorithm of Section 5.4 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 ℓ for both algorithms. If we are short of memory we can use the variant of Section 5.4 with smaller ℓ, and beat the basic algorithm. We have verified this, but omit the detailed experiments.

From now on we will focus on the most successful algorithm, `-B -Sb`, that is, our main algorithm with bit-parallel counters and without optimization of ℓ-grams.

6.3 Tuning Our Algorithm

Figure 21 shows running times for different pattern lengths. Note that we can slightly increase the difference ratio for larger m if ℓ is "too" large. For example, $\ell = 6$ is faster than $\ell = 8$ for DNA if $m = 16$ and $k = 1$, because the total length of the ℓ-grams that fit into the text window is larger for $\ell = 6$. The corresponding search times (without preprocessing) are 12.27 for $\ell = 6$ and 45.75 seconds for $\ell = 8$. This shows that the choice of ℓ must be done with some care.

If k/m and/or rm becomes too large, we are not able to use large enough ℓ to keep the filtering efficient. Using bit-parallel counters helps when r becomes large, as it effectively searches for smaller pattern subsets in parallel. This helps only to some extent because the number of bits in computer word is limited (typically to 32 or 64). One possibility would be to simulate longer words using several native

words. However, the total time is likely to grow proportionally to the number of machine words that simulate a long virtual computer word. Following this line, we could explicitly divide the pattern set into smaller subsets, and search for each subset separately. This was considered at the end of Section 4.1. The optimal complexity will be lost, but this is already inevitable if the maximum ℓ value we can afford is fixed by our available main memory. Given that the ℓ value we can use is smaller than the optimal, searching in separate groups can reduce the total time.

Figure 22 shows experiments for explicitly dividing the pattern set into several subsets. This method clearly extends the practical value of the algorithms, allowing both larger k/m and r. It can also be seen that, the larger k, the less grouping is advisable. This coincides with the analysis, which recommends subgroup size $r' = \sigma^{d\ell'/2}/m^2$, being ℓ' the value we can afford.

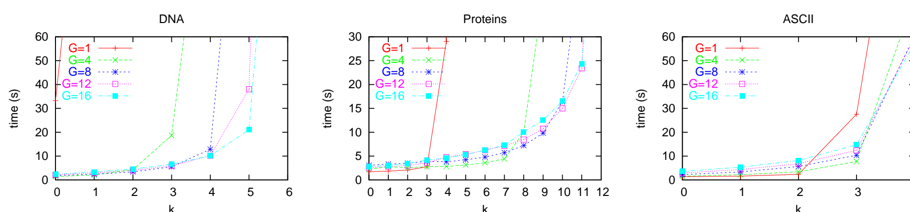

Fig. 22. Pattern grouping for algorithm `-B -Sb`. The parameters are $r = 1024$; $\ell = 6$ for DNA, $\ell = 3$ for proteins, and $\ell = 2$ for ASCII. The curves show total time (searching and preprocessing). From left to right, DNA, proteins, and ASCII. $G = x$ means that we used G groups of $1024/G$ patterns each.

As explained in Section 5, clustering the patterns to form groups may benefit the overall performance. We used real texts E.coli and the Bible for this experiment. Figure 23 gives the results. It shows that clustering helps, but seems not to be worth it. We repeated the experiment by generating pattern sets where the distance between subsequent patterns is only 2. The results show that clustering can be useful in some (extreme) cases. Note that we used only $\ell = 4$ for DNA here, to emphasize the effect. Using larger ℓ gives more or less the same timings for all the cases.

Alphabet mapping can be used to speed up the algorithm for skewed character distributions, as explained at the end of Section 4.1. We experimented with the Bible using different combinations of ℓ and number of character groups, denoted by $M = x$ in Figure 24, meaning that we map the characters into x groups, each having approximately the same total probability of occurrence. The optimal mapping depends on r and k, but the method clearly improves the results.

6.4 Comparing against Other Algorithms

We compare our (generally) best algorithm, `-B -Sb`, against previous work, both for searching single and multiple patterns. Following [Navarro 2001] we have included only the relevant algorithms in the comparison. These are:

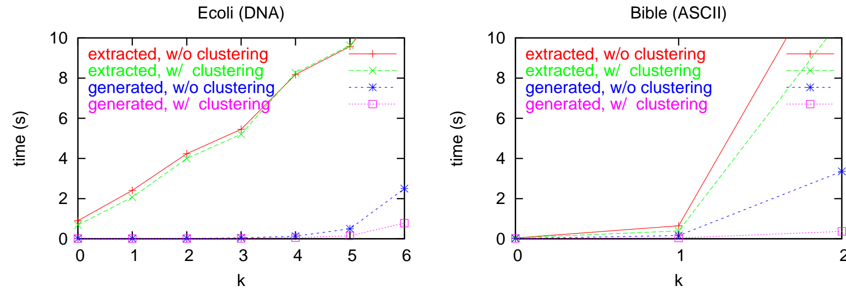

Fig. 23. The effect of clustering the patterns prior to searching (with algorithm `-B -Sb`). Left: Ecoli (DNA), $m = 64$, $r = 64$, $\ell = 4$. Right: Bible (ASCII), $m = 12$, $r = 245$, $\ell = 2$.

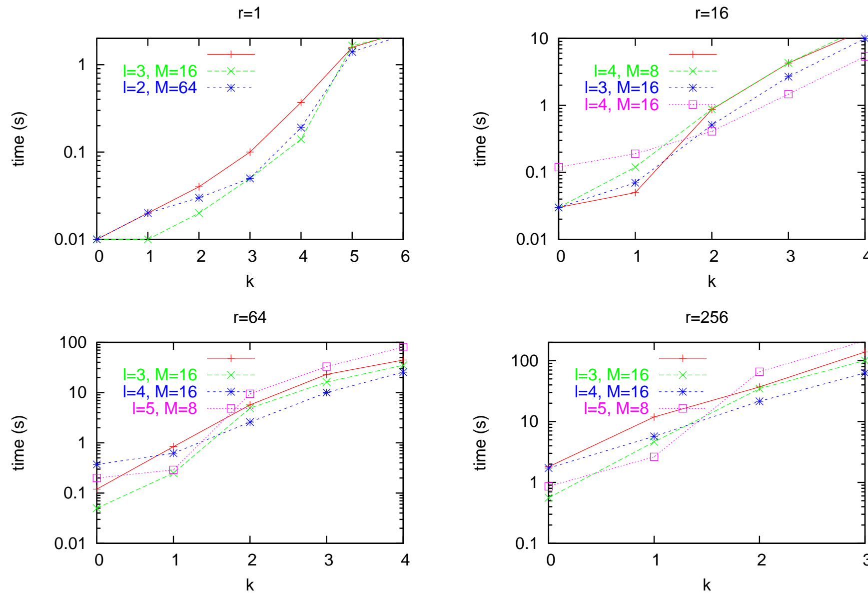

Fig. 24. Total times in seconds using alphabet mapping for the Bible, for different ℓ ("l" in the plots) and number of character groups M. The figures are from left to right, top to bottom, for $r = 1$, 16, 64, 256, and $m = 16$. The curve without label is for the standard search, without mapping. Note the logarithmic scale.

Ours:. Our `-B -Sb` algorithm, without pattern grouping, clustering, or alphabet mapping, except otherwise stated.

EXP:. Partitioning into exact search [Baeza-Yates and Navarro 2002], an algorithm for single and multiple approximate pattern matching, implemented by its authors. The algorithm is optimized with a bit-parallel verification machine and hierarchical verification, so it can be fairly compared with our `-B -Sb` version.

MM:. Muth & Manber algorithm [Muth and Manber 1996], 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

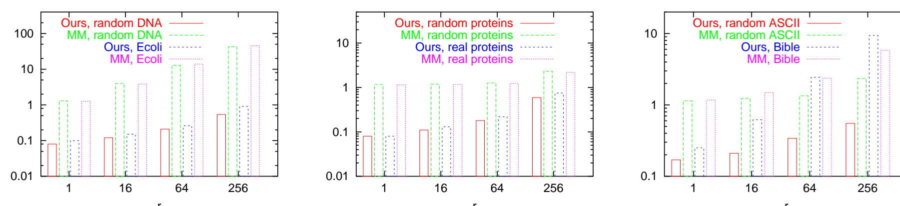

Fig. 25. Comparison against Muth & Manber, for $k = 1$. Note the logarithmic scale.

are searched for. The implementation is also from its authors.

TU:. Tarhio & Ukkonen algorithm [Tarhio and Ukkonen 1993], a Boyer-Moore type single pattern approximate search algorithm, reported in [Navarro 2001] as relevant in some special cases. Implemented by its authors.

ABNDM:. Approximate BNDM algorithm [Navarro and Raffinot 2000; Hyyrö and Navarro 2002], a single pattern approximate search algorithm extending classical BDM. The implementation is by its authors. We used the version of [Navarro and Raffinot 2000] which gave better results in our architecture, although it is theoretically worse than [Hyyrö and Navarro 2002]. Verification is also done using bit-parallelism, so the comparison against our algorithm is fair.

BPM:. Bit-parallel Myers [Myers 1999], 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.

We have modified ABNDM and BPM to use the superimposition technique [Baeza-Yates and Navarro 2002] to handle multiple patterns, combined with hierarchical verification. The superimposition is useful only if r/σ 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 and the Bible we superimposed a maximum of 16 patterns at a time, and for random ASCII, 64 patterns at a time. Note that the optimal group size depends on k, decreasing as k increases. However, we used the above group sizes, which is optimized for small k values, for all the cases.

Since MM is limited to $k = 1$, we compare this case separately in Figure 25. As it can be seen, our algorithm is better by far, 4–40 times faster depending on the alphabet size, except for real world ASCII for large r, where MM is about twice as fast for large r.

Figures 26, 27, and 28 show results for the other algorithms, for the case $r = 1$ (single patterns), as well as larger r values. As it can be seen, our algorithm is the fastest in the majority of cases.

EXP beats our algorithm for large k, and this happens sooner for larger r. On random ASCII our algorithm is the best only for small k and large r. Real world ASCII is hard for our algorithm, where EXP is in most cases the best choice. For ASCII we are not able to use large enough ℓ-grams. Using alphabet mapping and pattern grouping on real world ASCII allows the use of larger ℓ, but at the same time the mapping itself weakens the filtering sensitivity. The result is faster, but beats EXP only for $k = 1$.

In general, from the plots it seems that our algorithms are better on random than on real data. For ASCII this is partly due to the skewed character distribution, but the high preprocessing cost also affects.

Figure 29 shows the areas where each algorithm is best. As it can be seen, our new algorithm becomes the fastest choice for low k, and is useful to search for 1 to 1,000 patterns (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 [Baeza-Yates and Navarro 2002] 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. Note also that our algorithm would be favored on even longer texts, as its preprocessing depends only on rm.

7. CONCLUSIONS

Multiple approximate string matching is an important problem that arises in several applications, and for which the current state of the art is in a very primitive stage. Nontrivial solutions exist only for the case of very low difference ratios or very few patterns.

We have presented a new algorithm to improve this situation. Our algorithm is optimal on average for low and intermediate difference ratios (up to 1/2), filling an important gap in multipattern approximate string matching, where very few algorithms existed and no average-optimal algorithm was known.

We presented several variants, improvements, and tuning techniques over our algorithm. We have shown that, in practice, our algorithm performs well in handling from one to a very large number of patterns, and low difference ratios. Our algorithm becomes indeed the best alternative in practice for these cases. In particular, this includes being the fastest single-pattern search algorithm for low difference ratios, which is a highly competitive area. On real life texts, we show that our algorithm is especially interesting for computational biology applications (DNA and proteins).

The reason for the mismatch between theory and practice is that the space requirement of the algorithm, although polynomial in the size of the pattern set, is too large in practice and does not let us use the optimal ℓ-gram length. Space usage (and preprocessing time) is always one of the main concerns of multipattern search algorithms, and our algorithms are not an exception. We have studied several choices to alleviate this situation, which permit using our algorithm reasonably well in those extreme cases too.

Our algorithms can be extended in a number of ways. For example, it is very simple to adapt them to handle classes of characters in the patterns. This means that each pattern position can match a subset of the alphabet rather than just a single character. The only change needed by the algorithm is to set up correctly the minimum distances between ℓ-grams and patterns, by using a bit-parallel algorithm (that handles well classes of characters) both in the preprocessing and the verification. In particular, we have implemented case insensitive search for natural language text and IUB codes for DNA (standard degeneracy symbols, "wild cards").

Likewise, it is possible to extend our algorithms to handle edit distances with general real-valued weights, essentially without changes. It is enough to precompute

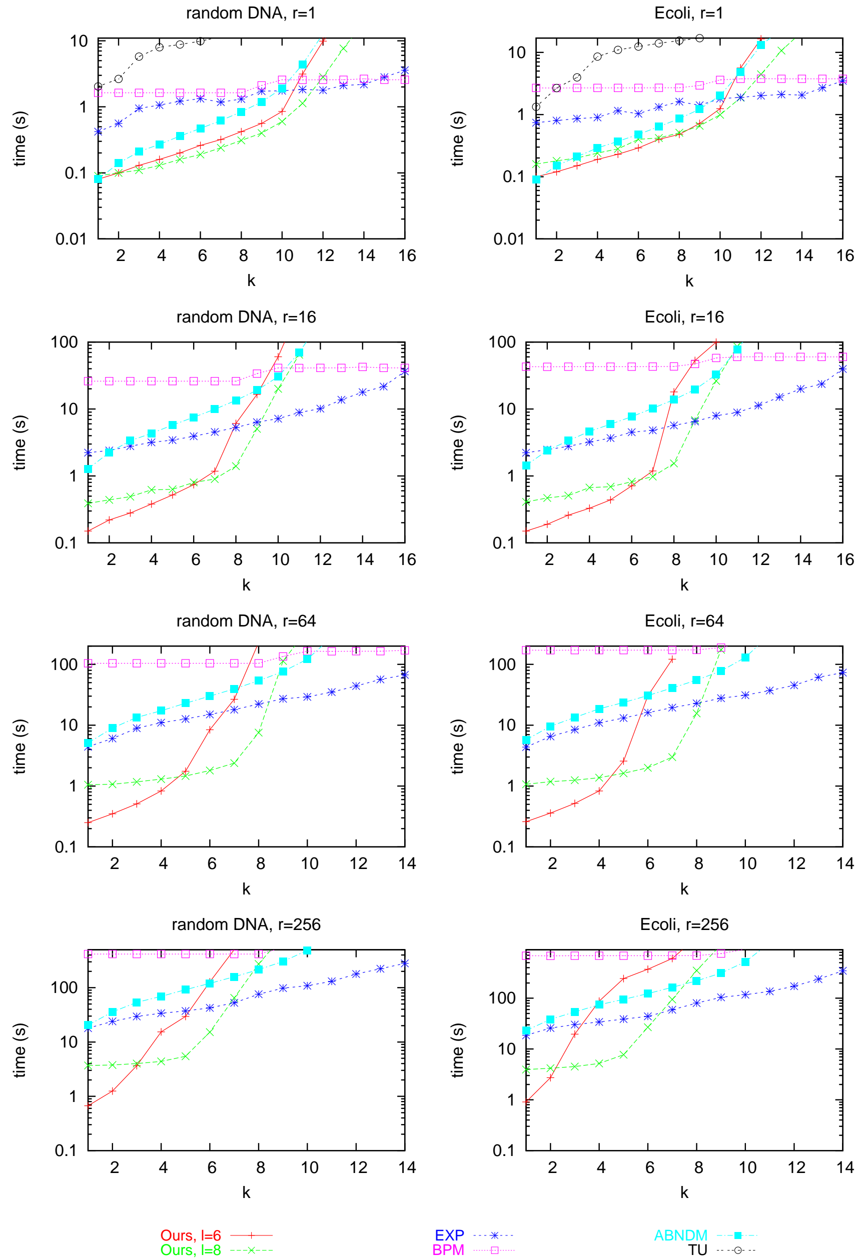

Fig. 26. Comparing different algorithms in DNA, considering both preprocessing and searching time. On the left, random data; on the right E.coli. From top to bottom: $r = 1, 16, 64, 256$. ℓ looks as "l". Note the logarithmic scale.

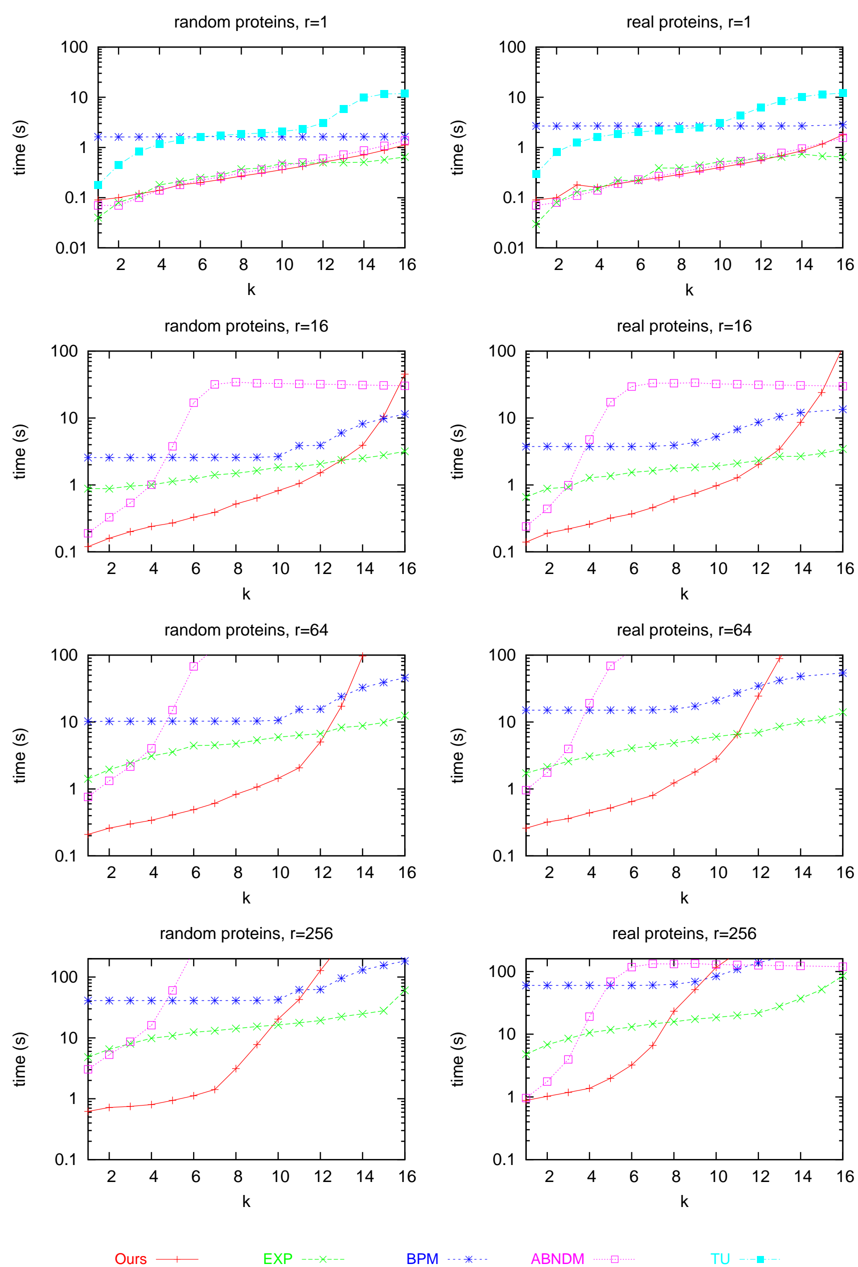

Fig. 27. Comparing different algorithms in proteins, considering both preprocessing and searching time. On the left, random data; on the right real proteins. From top to bottom: $r = 1, 16, 64, 256$. We used $\ell = 3$ for our algorithm. Note the logarithmic scale.

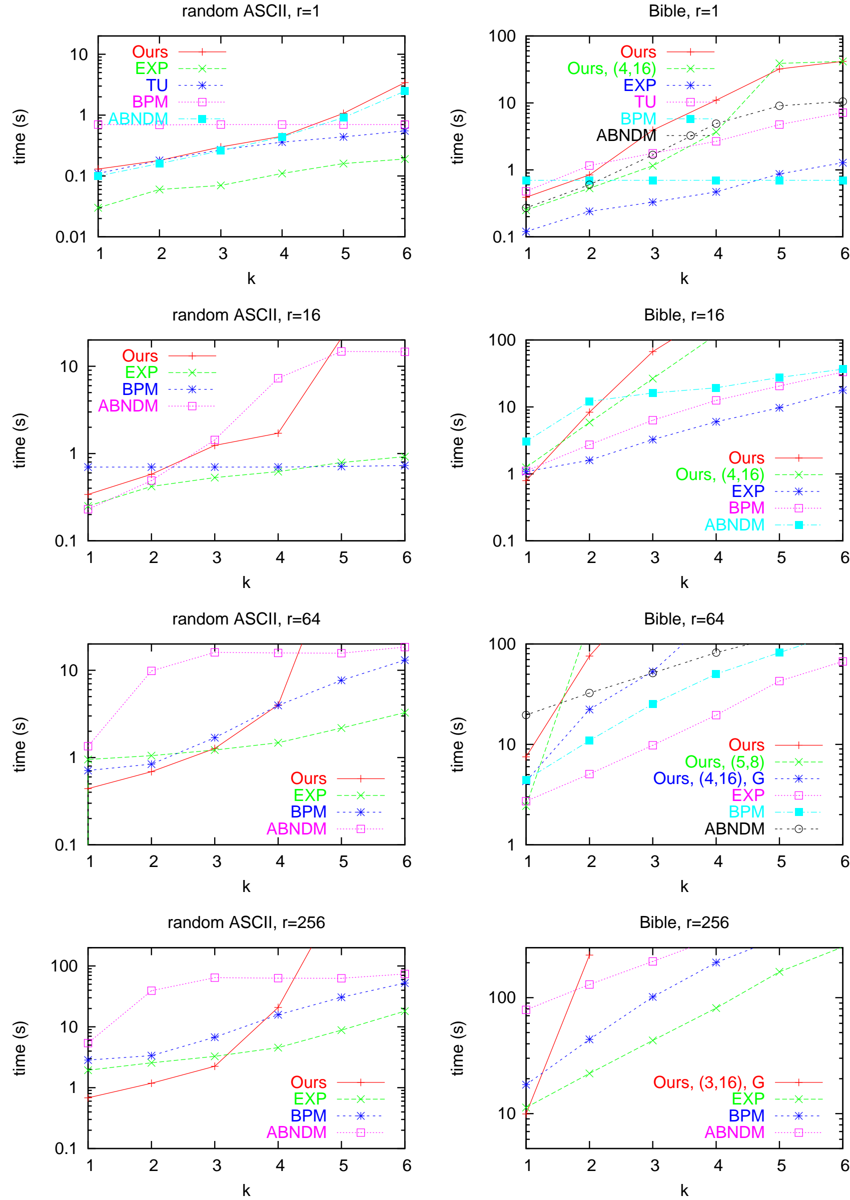

Fig. 28. Comparing different algorithms in ASCII, considering both preprocessing and searching time. On the left, random data; on the right, the Bible. From top to bottom: $r = 1, 16, 64, 256$. Ours (x, y) means alphabet mapping with $\ell = x$, and the number of character groups $= y$. G means grouping with 8 groups. All others are for the standard algorithm with $\ell = 2$. Note the logarithmic scale.

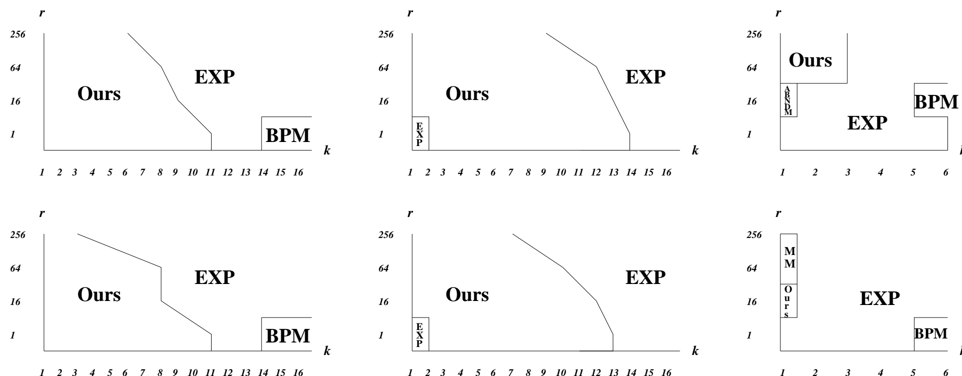

Fig. 29. Areas where each algorithm performs best. From left to right, DNA ($m = 64$), proteins ($m = 64$), and ASCII ($m = 16$). Top row is for random data, and bottom row for real world data.

the minimum weight necessary to match each ℓ-gram inside any pattern, and all the arguments remain the same. The only delicate point is to determine the minimum length of an occurrence of any pattern, which is not $m - k$ anymore. We should consider eliminating the characters of P, from cheaper to more expensive, until surpassing threshold k, and take the minimum length over all the patterns. In particular, we have implemented Hamming distance, a simplification where only replacements are permitted, at cost 1. This permits using windows of length m, as well as a bit smaller ℓ in practice (because the average ℓ-gram distances are a bit larger).

We believe that still several ideas can be pursued to reduce preprocessing time and memory usage.

One idea is lazy evaluation of the table cells. Instead of fully computing the D tables of size σ^ℓ for each pattern, we compute the cells only for the text ℓ-grams as they appear. If a given table cell is not yet computed, we compute it on the fly for all the patterns. This gives a preprocessing cost that is $O(rm\sigma^\ell(1 - e^{-n/\sigma^\ell}))$ on the average (using Myers' algorithm for the ℓ-grams inside the patterns, as $\lceil \ell/w \rceil = 1$). This, however, is advantageous only for very long ℓ-grams, namely $\ell + \Theta(\log\log\ell) > \log_\sigma n$.

Another possibility is to compute D only for those ℓ-grams that appear in a pattern with at most ℓ' differences, and assume that all the others appear with $\ell' + 1$ differences. This reduces the effectiveness at search time but, by storing the relevant ℓ-grams in a hash table, requires $O(rm(\sigma\ell)^{\ell'})$ space and preprocessing time (either for plain or hierarchical verification), since the number of strings at distance ℓ' to an ℓ-gram is $O((\sigma\ell)^{\ell'})$ [Ukkonen 1985]. With respect to plain verification, the space is reduced for $\ell' < (\ell - \log_\sigma(rm))/(1 + \log_\sigma \ell)$, and with respect to hierarchical verification, for $\ell' < (\ell - \log_\sigma m)/(1 + \log_\sigma \ell)$. These values seem reasonable.

REFERENCES

Baeza-Yates, R. and Navarro, G. 2000. New models and algorithms for multidimensional approximate pattern matching. *Journal of Discrete Algorithms (JDA) 1,* 1, 21–49. Special issue on Matching Patterns.

Baeza-Yates, R. and Navarro, G. 2002. New and faster filters for multiple approximate string matching. *Random Structures and Algorithms (RSA) 20*, 23–49.

Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

Baeza-Yates, R. A. and Navarro, G. 1999. Faster approximate string matching. *Algorithmica 23,* 2, 127–158.

Chang, W. and Lawler, E. 1994. Sublinear approximate string matching and biological applications. *Algorithmica 12,* 4/5, 327–344.

Chang, W. and Marr, T. 1994. Approximate string matching and local similarity. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*. LNCS 807. 259–273.

Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W. 1994. Speeding up two string matching algorithms. *Algorithmica 12,* 4/5, 247–267.

Crochemore, M. and Rytter, W. 1994. *Text Algorithms*. Oxford University Press.

Dixon, R. and Martin, T., Eds. 1979. *Automatic speech and speaker recognition*. IEEE Press.

Elliman, D. and Lancaster, I. 1990. A review of segmentation and contextual analysis techniques for text recognition. *Pattern Recognition 23,* 3/4, 337–346.

Fredriksson, K. 2003. Row-wise tiling for the Myers' bit-parallel approximate string matching algorithm. In *Proc. 10th Symposium on String Processing and Information Retrieval (SPIRE'03)*. LNCS 2857. 66–79.

Fredriksson, K. and Navarro, G. 2003. Average-optimal multiple approximate string matching. In *Proc. 14th Combinatorial Pattern Matching (CPM'03)*. LNCS 2676. 109–128.

Fredriksson, K. and Navarro, G. 2004. Improved single and multiple approximate string matching. In *Proc. 15th Combinatorial Pattern Matching (CPM'04)*. LNCS 3109. 457–471.

Grossi, R. and Luccio, F. 1989. Simple and efficient string matching with k mismatches. *Information Processing Letters 33,* 3, 113–120.

Horspool, R. 1980. Practical fast searching in strings. *Software Practice and Experience 10*, 501–506.

Hyyrö, H., Fredriksson, K., and Navarro, G. 2004. Increased bit-parallelism for approximate string matching. In *Proc. 3rd Workshop on Efficient and Experimental Algorithms (WEA'04)*. LNCS 3059. 285–298.

Hyyrö, H. and Navarro, G. 2002. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'02)*. LNCS 2373. 203–224. Extended version to appear in *Algorithmica*.

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

Kukich, K. 1992. Techniques for automatically correcting words in text. *ACM Computing Surveys 24,* 4, 377–439.

Kumar, S. and Spafford, E. 1994. A pattern-matching model for intrusion detection. In *Proc. National Computer Security Conference*. 11–21.

Lopresti, D. and Tomkins, A. 1994. On the searchability of electronic ink. In *Proc. 4th International Workshop on Frontiers in Handwriting Recognition*. 156–165.

Muth, R. and Manber, U. 1996. Approximate multiple string search. In *Proc. 7th Combinatorial Pattern Matching (CPM'96)*. LNCS 1075. 75–86.

Myers, E. W. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM 46,* 3, 395–415.

Navarro, G. 2001. A guided tour to approximate string matching. *ACM Computing Surveys 33,* 1, 31–88.

Navarro, G. and Baeza-Yates, R. 1999. Very fast and simple approximate string matching. *Information Processing Letters 72*, 65–70.

Navarro, G. and Baeza-Yates, R. 2001. Improving an algorithm for approximate pattern matching. *Algorithmica 30,* 4, 473–502.

Navarro, G. and Fredriksson, K. 2004. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science 321,* 2–3, 283–290.

Navarro, G. and Raffinot, M. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA) 5,* 4.

Navarro, G. and Raffinot, M. 2002. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press.

Navarro, G., Sutinen, E., Tanninen, J., and Tarhio, J. 2000. Indexing text with approximate q-grams. In *Proc. 11th Combinatorial Pattern Matching (CPM'00).* LNCS 1848. 350–363.

Paul, W. and Simon, J. 1980. Decision trees and random access machines. In *Proc. International Symposium on Logic and Algorithmic.* Zurich, 331–340.

Sankoff, D. and Kruskal, J., Eds. 1983. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley.

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

Sutinen, E. and Tarhio, J. 1996. Filtration with q-samples in approximate string matching. In *Proc. 7th Combinatorial Pattern Matching.* LNCS 1075. 50–63.

Tarhio, J. and Ukkonen, E. 1993. Approximate Boyer-Moore string matching. *SIAM Journal of Computing 22,* 2, 243–260.

Ukkonen, E. 1985. Finding approximate patterns in strings. *Journal of Algorithms 6*, 132–137.

Waterman, M. 1995. *Introduction to Computational Biology.* Chapman and Hall.

Yao, A. C. 1979. The complexity of pattern matching for a random string. *SIAM Journal of Computing 8,* 3, 368–387.