

Increased Bit-Parallelism for Approximate and Multiple String Matching

HEIKKI HYYRÖ

PRESTO, Japan Science and Technology Agency

University of Tampere

and

KIMMO FREDRIKSSON

University of Joensuu

and

GONZALO NAVARRO

University of Chile

Supported in part by the Academy of Finland, and by the Tampere Graduate School in Information Science and Engineering (first author). Supported by the Academy of Finland, grant 202281 (second author). Supported by Millenium Nucleus Center for Web Research, grant P04-067-F, Mideplan, Chile (third author).

A conference version of this paper appeared at *Proc. 3rd Workshop on Efficient and Experimental Algorithms (WEA '04)*, pages 285–298, 2004. LNCS 3059.

Authors' address: Heikki Hyyrö, Department of Computer Sciences, Kanslerinrinne 1, FIN-33014 University of Tampere, Finland. Email: Heikki.Hyyro@cs.uta.fi. 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

Bit-parallelism permits executing several operations simultaneously over a set of bits or numbers stored in a single computer word. This technique permits searching for the approximate occurrences of a pattern of length m in a text of length n in time $O(\lceil m/w \rceil n)$, where w is the number of bits in the computer word. Although this is asymptotically the optimal bit-parallel speedup over the basic $O(mn)$ time algorithm, it wastes bit-parallelism's power in the common case where m is much smaller than w , since $w - m$ bits in the computer words get unused.

In this paper we explore different ways to increase the bit-parallelism when the search pattern is short. First, we show how multiple patterns can be packed into a single computer word so as to search for all them simultaneously. Instead of spending $O(rn)$ time to search for r patterns of length $m \leq w/2$, we need $O(\lceil rm/w \rceil n)$ time. Second, we show how the mechanism permits boosting the search for a single pattern of length $m \leq w/2$, which can be searched for in $O(\lceil n/\lfloor w/m \rfloor \rceil)$ bit-parallel steps instead of $O(n)$. Third, we show how to extend these algorithms so that the time bounds essentially depend on k instead of m , where k is the maximum number of differences permitted. Finally, we show how the ideas can be applied to other problems such as multiple exact string matching and one-against-all computation of edit distance and longest common subsequences.

Our experimental results show that the new algorithms work well in practice, obtaining significant speedups over the best existing alternatives especially on short patterns and moderate number of differences allowed. This work fills an important gap in the field, where little work has focused on very short patterns.

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, Performance

Additional Key Words and Phrases: Approximate string matching, multiple string matching, bit-parallelism

1. INTRODUCTION

Approximate string matching is an old problem, with applications for example in spelling correction, bioinformatics and signal processing [Navarro 2001]. It refers in general to searching for substrings of a text that are within a predefined edit distance threshold from a given pattern. Let $T = T_{1\dots n}$ be a text of length n and $P = P_{1\dots m}$ a pattern of length m . Here $A_{a\dots b}$ denotes the substring of A that begins at its a th character and ends at its b th character, for $a \leq b$. Let $ed(A, B)$ denote the edit distance between the strings A and B , and k be the maximum allowed distance. Then the task of approximate string matching is to find all text indices j for which $ed(P, T_{h\dots j}) \leq k$ for some $h \leq j$.

The most common form of edit distance is Levenshtein distance [Levenshtein 1966]. It is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to make A and B equal. In this paper $ed(A, B)$ will denote Levenshtein distance. Another distance of interest is the indel distance, denoted by $id(A, B)$, where only character insertions and deletions are permitted. String B is a subsequence of string A if and only if we can transform A into B by removing zero or more characters from A . The indel distance is the dual of the

length of a longest common subsequence between two strings, which will be denoted $llcs(A, B)$ in this paper.

We also use w to denote the computer word size in bits, σ to denote the size of the text and pattern alphabet Σ , and $|A|$ to denote the length of the string A .

Bit-parallelism is the technique of packing several values in a single computer word and updating them all in a single operation. This technique has yielded the fastest approximate string matching algorithms if we exclude filtration algorithms (which need anyway to be coupled with a non-filtration one). In particular, the $O(\lceil m/w \rceil kn)$ algorithm of Wu and Manber [Wu and Manber 1992], the $O(\lceil km/w \rceil n)$ algorithm of Baeza-Yates and Navarro [Baeza-Yates and Navarro 1999], and the $O(\lceil m/w \rceil n)$ algorithm of Myers [Myers 1999] dominate for almost every value of m , k and σ .

In complexity terms, Myers' algorithm is superior to the others. In practice, however, Wu & Manber's algorithm is faster for $k = 1$ and Baeza-Yates and Navarro's is faster when $(k + 2)(m - k) \leq w$ or k/m is low. The reason is that, despite that Myers' algorithm packs better the state of the search (needing to update less computer words), it needs slightly more operations than its competitors. Except when m and k are small, the need to update less computer words makes Myers' algorithm faster than the others. However, when m is much smaller than w , Myers' advantage disappears because all the three algorithms need to update just one (or very few) computer words. In this case, Myers' representation wastes many bits of the computer word and is unable to take advantage of its more compact representation.

The case where m is much smaller than w is very common in several applications. Typically w is 32 or 64 bits in a modern computer, and for example the Pentium 4 processor allows one to use even words of 128 bits. Myers' representation uses m bits out of those w . In spelling, for example, it is usual to search for words whose average length is 6. In computational biology one can search for short DNA or amino acid sequences, of length as small as 4. Measuring edit distance or longest common subsequence against several short sequences is another common task. In signal processing applications one can search for streams composed of a few audio, MIDI, or video samples.

In this paper we concentrate on reducing the number of wasted bits in Myers' algorithm, so as to take advantage of its better packing of the search state even when $m \leq w$. This has been attempted previously [Fredriksson 2003], where $O(m \lceil n/w \rceil)$ time was obtained by processing the distance matrix by chunks of w columns at a time, each chunk in a row-wise manner. Our technique is different, as we focus on representing several patterns in a single computer word.

The contributions of this paper follow:

- We show how to search for several patterns simultaneously by packing them all in the same computer word. As a result, we can search for r patterns of length $m \leq w/2$ in $O(\lceil rm/w \rceil n + occ)$ rather than $O(rn)$ time, where $occ \leq rn$ is the total number of occurrences of all the patterns. Our experiments show that our algorithm is faster than all the others in many cases of interest, especially on short patterns.
- We show how the above idea can be used to boost the search for a single pattern,

so as to perform $O(\lceil n/\lfloor w/m \rfloor \rceil)$ instead of $O(n)$ bit-parallel steps, when $m \leq w/2$. Although we still perform $O(n)$ accesses to the text, those are much cheaper than the bit-parallel steps. Our experiments show that our algorithm is faster than all the previous ones in most interesting cases and by far, especially on short patterns.

- We show how to handle longer patterns, by considering only a sufficiently long prefix thereof. The result is an algorithm whose average search complexity is $O(\lceil r \max(k, \log m) / w \rceil n)$. Our experiments show that this technique extends the superiority of our multipattern search algorithm to the case of smaller k values, which are interesting for many applications.
- We show how the ideas developed for approximate searching can be extended to multiple exact string matching. Our algorithm can search for r patterns of any length in average time $O(\lceil r \log_\sigma w / w \rceil n)$. The same idea can be used to boost other bit-parallel algorithms such as MultiBDM [Navarro and Raffinot 2000; 2002]. Our experimental results show that the result is competitive against previous work, being the best by far when searching for a moderate number of short patterns.
- We show how the ideas for approximate searching can be adapted to compute edit distance or longest common subsequence for several short patterns against one. Our experimental results show that the idea is practical and permits noticeably boosting of the search time.

2. DYNAMIC PROGRAMMING

In the following ϵ denotes the empty string. To compute Levenshtein distance $ed(A, B)$, the dynamic programming algorithm fills an $(|A| + 1) \times (|B| + 1)$ table D , in which each cell $D[i, j]$ will eventually hold the value $ed(A_{1..i}, B_{1..j})$. Initially the trivially known *boundary values* $D[i, 0] = ed(A_{1..i}, \epsilon) = i$ and $D[0, j] = ed(\epsilon, B_{1..j}) = j$ are filled. Then the cells $D[i, j]$ are computed for $i = 1 \dots |A|$ and $j = 1 \dots |B|$ until the desired solution $D[|A|, |B|] = ed(A_{1..|A|}, B_{1..|B|}) = ed(A, B)$ is known. When the values $D[i - 1, j - 1]$, $D[i, j - 1]$ and $D[i - 1, j]$ are known, the value $D[i, j]$ can be computed by using the following well-known recurrence.

$$D[i, 0] = i, \quad D[0, j] = j.$$

$$D[i, j] = \begin{cases} D[i - 1, j - 1], & \text{if } A_i = B_j. \\ 1 + \min(D[i - 1, j - 1], D[i - 1, j], D[i, j - 1]), & \text{otherwise.} \end{cases}$$

This distance computation algorithm is easily modified to find approximate occurrences of A somewhere inside B [Sellers 1980]. This is done simply by changing the boundary condition $D[0, j] = j$ into $D[0, j] = 0$. In this case $D[i, j] = \min(ed(A_{1..i}, B_{h..j}), h \leq j)$, which corresponds to the earlier definition of approximate string matching if we replace A with P and B with T .

The values of D are usually computed by filling it in a column-wise manner for increasing j . This corresponds to scanning the string B (or the text T) one character at a time from left to right. At each character the corresponding column is completely filled in order of increasing i . This order makes it possible to save space by storing only one column at a time, since then the values in column j depend only on already computed values in it or values in column $j - 1$.

Some properties of matrix D are relevant to our paper [Ukkonen 1985b]:

- The diagonal property: $D[i, j] - D[i - 1, j - 1] = 0$ or 1 .
- The adjacency property: $D[i, j] - D[i, j - 1] = -1, 0$, or 1 , and $D[i, j] - D[i - 1, j] = -1, 0$, or 1 .

3. MYERS' BIT-PARALLEL ALGORITHM

In what follows we will use the following notation in describing bit-operations: '&' denotes bitwise "and", '|' denotes bitwise "or", '^' denotes bitwise "xor", '~' denotes bit complementation, and '<<' and '>>' denote shifting the bit vector left and right, respectively, using zero filling in both directions. The i th bit of the bit vector V is referred to as $V[i]$ and bit positions are assumed to grow from right to left. In addition we use superscripts to denote repetition. As an example let $V = 1011010$ be a bit vector. Then $V[1] = V[3] = V[6] = 0$, $V[2] = V[4] = V[5] = V[7] = 1$, and we could also write $V = 101^2010$ or $V = 101(10)^2$.

We describe here a version of the algorithm [Hyyrö 2001; Navarro and Raffinot 2002] that is slightly simpler than the original by Myers [Myers 1999]. The algorithm is based on representing the dynamic programming table D with vertical, horizontal and diagonal differences and precomputing the matching positions of the pattern into an array of size σ . This is done by using the following length- m bit vectors:

- . Vertical positive delta: $VP[i] = 1$ at text position j iff $D[i, j] - D[i - 1, j] = 1$.
- . Vertical negative delta: $VN[i] = 1$ at text position j iff $D[i, j] - D[i - 1, j] = -1$.
- . Horizontal positive delta: $HP[i] = 1$ at text position j iff $D[i, j] - D[i, j - 1] = 1$.
- . Horizontal negative delta: $HN[i] = 1$ at text position j iff $D[i, j] - D[i, j - 1] = -1$.
- . Diagonal zero delta: $DO[i] = 1$ at text position j iff $D[i, j] = D[i - 1, j - 1]$.
- . Pattern match vector PM_λ for each $\lambda \in \Sigma$: $PM_\lambda[i] = 1$ iff $P_i = \lambda$.

Fig. 1 shows a small example of a dynamic programming matrix and its representation with these bit vectors.

D	b	e	a	r	d	VP	b	e	a	r	d	VN	b	e	a	r	d		
0	0	0	0	0	0														
b	1	0	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0
a	2	1	1	1	2	2	1	1	0	0	1	1	0	0	0	0	0	0	0
n	3	2	2	2	2	3	1	1	1	1	0	1	0	0	0	0	0	0	0
d	4	3	3	3	3	2	1	1	1	1	1	0	0	0	0	0	0	0	1

HP	b	e	a	r	d	HN	b	e	a	r	d	DO	b	e	a	r	d	
b		0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
a		0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0
n		0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0
d		0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1

Fig. 1. The dynamic programming matrix to search "beard" for "band", and its representation with bit vectors. Each matrix column is represented by the corresponding bit vector columns.

Initially $VP = 1^m$ and $VN = 0^m$ to enforce the boundary condition $D[i, 0] = i$. At text position j the algorithm first computes vector $D0$ by using the old values VP and VN and the pattern match vector PM_{T_j} . Then the new HP and HN are computed by using $D0$ and the old VP and VN . Finally, vectors VP and VN are updated by using the new $D0$, HN and HP . Fig. 2 shows the complete formula for updating the vectors, and Fig. 3 shows the preprocessing of table PM and the higher-level search scheme. We refer the reader to [Hyyrö 2001; Myers 1999] for a more detailed explanation of the formula in Fig. 2.

Step(j)

1. $D0 \leftarrow (((PM_{T_j} \& VP) + VP) \wedge VP) \mid PM_{T_j} \mid VN$
2. $HP \leftarrow VN \mid \sim(D0 \mid VP)$
3. $HN \leftarrow VP \& D0$
4. $VP \leftarrow (HN \ll 1) \mid \sim(D0 \mid (HP \ll 1))$
5. $VN \leftarrow (HP \ll 1) \& D0$

Fig. 2. Updating the delta vectors at column j .

ComputePM(P)

1. **For** $\lambda \in \Sigma$ **Do** $PM_\lambda \leftarrow 0^m$
2. **For** $i \in 1 \dots m$ **Do** $PM_{P_i} \leftarrow PM_{P_i} \mid 0^{m-i}10^{i-1}$

Search(P, T, k)

1. **ComputePM(P)**
2. $VN \leftarrow 0^m$
3. $VP \leftarrow 1^m$
4. $currDist \leftarrow m$
5. **For** $j \in 1 \dots n$ **Do**
6. **Step(j)**
7. **If** $HP \& 10^{m-1} = 10^{m-1}$ **Then**
8. $currDist \leftarrow currDist + 1$
9. **Else If** $HN \& 10^{m-1} = 10^{m-1}$ **Then**
10. $currDist \leftarrow currDist - 1$
11. **If** $currDist \leq k$ **Then**
12. Report occurrence at j

Fig. 3. Preprocessing the PM -table and conducting the search.

The algorithm in Fig. 3 computes the value $D[m, j]$ explicitly in the $currDist$ variable by using the horizontal delta vectors (the initial value of $currDist$ is $D[m, 0] = m$). A pattern occurrence with at most k errors is found at text position j whenever $D[m, j] \leq k$.

We point out that the boundary condition $D[0, j] = 0$ is enforced on lines 4 and 5 in Fig. 2. After the horizontal delta vectors HP and HN are shifted left, their first bits correspond to the difference $D[0, j] - D[0, j - 1]$. This is the only phase

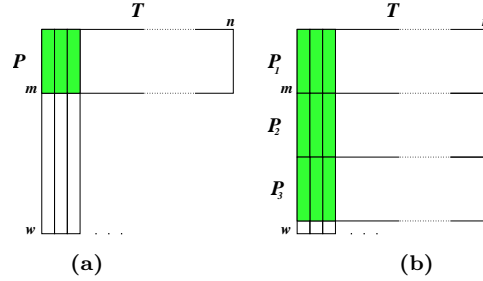


Fig. 4. For short patterns ($m < w$) Myers' algorithm (a) wastes $w - m$ bits. Our proposal (b) packs several patterns into the same computer word.

in the algorithm where the values from row 0 are relevant. And as we assume zero filling, the left shifts correctly set $HP[1] = HN[1] = 0$ to encode the difference $D[0, j] - D[0, j - 1] = 0$.

The running time of the algorithm is $O(n)$ when $m \leq w$, as there are only a constant number of operations per text character. The general running time is $O(\lceil m/w \rceil n)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit vectors of length w .

4. SEARCHING FOR SEVERAL PATTERNS SIMULTANEOUSLY

We show how Myers' algorithm can be used to search for r patterns of length m simultaneously, all with the same error threshold k . For simplicity we will assume $rm \leq w$; otherwise the search patterns must be split into groups of at most $\lfloor w/m \rfloor$ patterns each, and each group searched for separately. Our search time will be $O(\lceil r/\lfloor w/m \rfloor \rceil n + occ)$, as opposed to the $O(rn)$ time that would be achieved by searching for each pattern separately. Here $occ \leq rn$ stands for the total number of occurrences of all the patterns. When $w/m \geq 2$ (our case of interest), our complexity can be written as $O(\lceil rm/w \rceil n + occ)$.

Consider the situation where $w/m \geq 2$ and Myers' algorithm is used. Fig. 4a shows how the algorithm fails to take full advantage of bit-parallelism in that situation as at least one half of the bits in the bit vectors is not used. Fig. 4b depicts our proposal: encode several patterns into the bit vectors and search for them in parallel. There are several obstacles in implementing this simple idea correctly, which will be discussed next.

4.1 Updating the Delta Vectors

A natural starting point is the problem of encoding and updating several patterns in the delta vectors. Let us denote a parallel version of a delta vector with the superscript p . We encode the patterns consecutively into the vectors without leaving any space between them. For example $D\theta^p[i]$ corresponds to the bit $D\theta[(i-1) \bmod m] + 1$ in the $D\theta$ -vector of the $\lceil i/m \rceil$ th pattern. The pattern match vectors PM are computed in normal fashion for the concatenation of the patterns. This correctly aligns the patterns with their positions in the bit vectors.

When the parallel vectors are updated, we need to ensure that the values for different patterns do not interfere with each other and that the boundary values

$D[0, j] = 0$ are used correctly. From the update formula in Fig. 2 it is obvious that only the addition (“+”) on line 1 and the left shifts on lines 4 and 5 can cause incorrect interference.

The addition operation may be handled by temporarily setting off the bits in VP^p that correspond to the last characters of the patterns. When this is done before the addition, there cannot be an incorrect overflow, and the algorithm also stays correct in every other aspect: The value $VP^p[i]$ can affect only the values $D^p[i + h]$ for some $h > 0$. It turns out that a similar modification works also with the left shifts. If the bits that correspond to the last characters of the patterns are temporarily set off in HP^p and HN^p then, after shifting left, the positions in HP^p and HN^p that correspond to the first characters of the patterns will correctly have a zero bit. The first pattern gets the zero bits from zero filling of the shift. Therefore, this second modification both removes possible interference and enforces the boundary condition $D[0, j] - D[0, j - 1] = 0$.

Both modifications are implemented by *and*ing the corresponding vectors with the bit mask $ZM = (01^{m-1})^r$. Fig. 5 gives the code for a step.

MStep(j)

1. $XP \leftarrow VP \ \& \ ZM$
2. $D0 \leftarrow (((PM_{T_j} \ \& \ XP) + XP) \wedge XP) \mid PM_{T_j} \mid VN$
3. $HP \leftarrow VN \mid \sim(D0 \mid VP)$
4. $HN \leftarrow VP \ \& \ D0$
5. $XP \leftarrow (HP \ \& \ ZM) \ll 1$
6. $XN \leftarrow (HN \ \& \ ZM) \ll 1$
7. $VP \leftarrow (XN \mid \sim(D0 \mid XP))$
8. $VN \leftarrow XP \ \& \ D0$

Fig. 5. Updating the delta vectors at column j , when searching for multiple patterns.

4.2 Keeping the Scores

A second problem is computing the value $D[m, j]$ explicitly for each of the r patterns. We handle this by using bit-parallel counters in a somewhat similar fashion to [Hyyrö and Navarro 2002; Hyyrö and Navarro 2005]. Let MC be a length- w bit-parallel counter vector. We set up into MC an m -bit counter for each pattern. Let $MC(i)$ be the value of the i th counter. The counters are aligned with the patterns so that $MC(1)$ occupies the first m bits, $MC(2)$ the next m bits, and so on. We will represent value zero in each counter as $b = 2^{m-1} + k$, and the value $MC(i)$ will be translated to actually mean $b - MC(i)$. This gives each counter $MC(i)$ the following properties: (1) $b < 2^m$. (2) $b - m \geq 0$. (3) The m th bit of $MC(i)$ is set iff $b - MC(i) \leq k$. (4) In terms of updating the translated value of $MC(i)$, the roles of adding and subtracting from it are reversed.

The significance of properties (1) and (2) is that they ensure that the values of the counters will not overflow outside their regions. Their correctness depends on the assumption $k < m$. This is not a true restriction as it excludes only the case of trivial matching ($k = m$).

We use a length- w bit-mask $EM = (10^{m-1})^r$ to update MC . The bits set in $HP^p \& EM$ and $HN^p \& EM$ correspond to the last bits of the counters that need to be incremented and decremented, respectively. Thus, remembering to reverse addition and subtraction, MC may be updated by setting

$$MC \leftarrow MC + ((HN^p \& EM) \gg (m-1)) - ((HP^p \& EM) \gg (m-1)).$$

Property (3) means that the last bit of $MC(i)$ signals whether the i th pattern matches at the current position. Hence, whenever $MC \& EM \neq 0^{rm}$ we have an occurrence of some of the patterns in T . At this point we can examine the bit positions of EM one by one to determine which patterns have matched and report their occurrences. This, however, adds $O(r \min(n, occ))$ time in the worst case to report the occ occurrences of all the patterns. We show next how to reduce this to $O(occ)$.

Fig. 6 gives the code to search for the patterns $P^1 \dots P^r$.

```

MComputePM( $P^1 \dots P^r$ )
1.   For  $\lambda \in \Sigma$  Do  $PM_\lambda \leftarrow 0^{mr}$ 
2.   For  $s \in 1 \dots r$  Do
3.     For  $i \in 1 \dots m$  Do  $PM_{P_i^s} \leftarrow PM_{P_i^s} \mid 0^{m(r-s+1)-i} 10^{m(s-1)+i-1}$ 

```

```

MSearch( $P^1 \dots P^r, T, k$ )
1.   MComputePM( $P^1 \dots P^r$ )
2.    $ZM \leftarrow (01^{m-1})^r$ 
3.    $EM \leftarrow (10^{m-1})^r$ 
4.    $VN \leftarrow 0^{mr}$ 
5.    $VP \leftarrow 1^{mr}$ 
6.    $MC \leftarrow (2^{m-1} + k) \times (0^{m-1}1)^r$ 
7.   For  $j \in 1 \dots n$  Do
8.     MStep( $j$ )
9.      $MC \leftarrow MC + ((HN \& EM) \gg (m-1)) - ((HP \& EM) \gg (m-1))$ 
10.    If  $MC \& EM \neq 0^{rm}$  Then MReport( $j, MC \& EM$ )

```

Fig. 6. Preprocessing the PM -table and conducting the search for multiple patterns.

4.3 Reporting the Occurrences

Let us assume that we want to identify which bits in mask $OM = MC \& EM$ are set, in time proportional to the number of bits set. If we achieve this, the total time to report all the occ occurrences of all the patterns will be $O(occ)$. One choice is to precompute a table F that, for any value of OM , gives the position of the first bit set in OM . That is, if $F[OM] = s$, then we report an occurrence of the (s/m) th pattern at the current text position j , clear the s th bit in OM by doing $OM \leftarrow OM \& \sim (1 \ll (s-1))$, and repeat until OM becomes zero.

The only problem of this approach is that table F has 2^{rm} entries, which is too much. Fortunately, we can compute the s values efficiently without resorting to look-up tables. The key observation is that the position of the highest bit set in

OM is effectively the function $\lfloor \log_2(OM) \rfloor + 1$ (we number the bits from 1 to w), that is, it holds that

$$2^{\lfloor \log_2(x) \rfloor} \leq x < 2^{\lfloor \log_2(x) \rfloor + 1}, \text{ or which is the same,} \\ 1 \ll \lfloor \log_2(x) \rfloor \leq x < 1 \ll (\lfloor \log_2(x) \rfloor + 1).$$

The function $\lfloor \log_2(x) \rfloor$ for an integer x can be computed in $O(1)$ time in modern computer architectures by converting x into a floating point number and extracting the exponent, which requires only two additions and a shift. This assumes that the floating point number is represented in a certain way, in particular that the radix is 2, and that the number is normalized. The “industry standard” IEEE floating point representation meets these requirements. For the details and other solutions for the integer logarithm of base 2, refer for example to [Warren Jr 2003]. ISO C99 standard conforming C compilers also provide a function to extract the exponent directly, and many CPUs even have a dedicated machine instruction for $\lfloor \log_2(x) \rfloor$ function. Fig. 7 gives the code.

```

MReport( $j, OM$ )
1.   While  $OM \neq 0^{r^m}$  Do
2.      $s \leftarrow \lfloor \log_2(OM) \rfloor$ 
3.     Report occurrence of  $P^{(s+1)/m}$  at text position  $j$ 
4.      $OM \leftarrow OM \& \sim (1 \ll s)$ 

```

Fig. 7. Reporting occurrences at current text position.

For architectures where $\lfloor \log_2(x) \rfloor$ is hard to compute, we can still manage to obtain $O(\min(n, occ) \log r)$ time as follows. To detect the bits set in OM , we check its two halves. If some half is zero, we can finish there. Otherwise, we recursively check its two halves. We continue the process until we have isolated each individual bit set in OM . In the worst case, each such bit has cost us $O(\log r)$ halving steps.

4.4 Handling Different Lengths and Thresholds

For simplicity we have assumed that all the patterns are of the same length and are all searched with the same k . The method, however, can be adapted with little problems to different m and k for each pattern.

If the lengths are $m_1 \dots m_r$ and the thresholds are $k_1 \dots k_r$, we have to *and* the vertical and horizontal vectors with $ZM = 01^{m_r-1} 01^{m_{r-1}-1} \dots 01^{m_1-1}$, and this fixes the problem of updating the delta vectors. With respect to the counters, the i th counter must be represented as $b_i - MC(i)$, where $b_i = 2^{m_i-1} + k_i$.

One delicacy is the update of MC , since the formula we gave to align all the HP^p bits at the beginning of the counters involved “ $\gg (m-1)$ ”, and this works only when all the patterns are of the same length. If they are not, we could align the counters so that they start at the end of their areas, hence removing the need for the shift at all. To avoid overflows, we should sort the patterns in increasing length order prior to packing them in the computer word. The price is that we will need m_r extra bits at the end of the bit mask to hold the largest counter. An alternative

solution would be to handle the last counter separately. This would avoid the shifts, and effectively adds only a constant number of operations per text character.

Finally, reporting the occurrences works just as before, except that the pattern number we report is no longer $(s + 1)/m$ (Fig. 7). The correct pattern number can be computed efficiently for example using a look-up table indexed with s . The size of the table is only $O(w)$, as $s \leq w - 1$.

5. BOOSTING THE SEARCH FOR ONE PATTERN

Up to now we have shown how to take advantage of wasted bits by searching for several patterns simultaneously. Yet, if we only want to search for a single pattern, we still waste the bits. In this section we show how the technique developed for multiple patterns can be adapted to boost the search for a single pattern.

The main idea is to search for multiple copies of the same pattern P and parallelize the access to the text. Say that $r = \lfloor w/m \rfloor$. Then we search for r copies of P using a single computer word, with the same technique developed for multiple patterns. Each such copy will be used to search a *different* text segment. We divide the text T into r equal-sized subtexts $T = T^1 T^2 \dots T^r$. Text T^s , of length $n' = \lceil n/r \rceil$, will be searched for the s th copy of P , and therefore all the occurrences of P in T will be found.

Our search will perform $\lceil n/r \rceil$ steps, where step j will access r text characters $T_j, T_{j+n'}, T_{j+2n'}, \dots, T_{j+(r-1)n'}$. With those r characters $c_1 \dots c_r$ we should build the corresponding PM mask to execute a single step. This is easily done by using

$$PM \leftarrow PM_{c_1} | (PM_{c_2} \ll m) | (PM_{c_3} \ll 2m) | \dots | (PM_{c_r} \ll (r-1)m)$$

We must exercise some care at the boundaries between consecutive text segments. On the one hand, processing of text segment T^s ($1 \leq s < r$) should continue up to $m + k - 1$ characters in T^{s+1} in order to provide the adequate context for the possible occurrences in the beginning of T^{s+1} . On the other hand, the processing of T^{s+1} must avoid reporting occurrences at the first $m + k - 1$ positions to avoid reporting them twice. Finally, occurrences may be reported out of order if printed immediately, so it is necessary to store them in r buffer arrays in order to report them ordered at the end.

Thus the algorithm requires $\lceil n/r \rceil = O(\lceil n/\lfloor w/m \rfloor \rceil)$ bit-parallel steps, for $m \leq w/2$. In addition we perform n text accesses to compute PM . Albeit the formal complexity is still $O(n)$, the idea is interesting in practice because the bit-parallel steps are much more expensive than the text accesses.

6. LONG PATTERNS AND K -DIFFERENCES PROBLEM

We have shown how to utilize the bits in computer words economically, but our methods assume $m \leq w/2$. We now sketch a method that can handle longer patterns, and can pack more patterns in the same computer word. The basic assumption here is that we are only interested in pattern occurrences that have at most k differences. This is the situation that is most interesting in practice, and usually we can assume that k is much smaller than m . Our goal is to obtain similar time bounds as above, yet replacing m with k in the complexities. The difference will be that these become average case complexities now.

The method is similar to our basic algorithms, but now we use an adaptation of Ukkonen’s well-known “cut-off” algorithm [Ukkonen 1985a]. That algorithm fills the table D in column-wise order, and computes the values $D[i, j]$ in column j for only $i \leq \ell_j$, where

$$\ell_j = 1 + \max\{i \mid D[i, j - 1] \leq k\}.$$

The cut-off heuristic is based on the fact that the search result does not depend on cells whose value is larger than k . And from the definition of ℓ_j it holds that $D[i, j] > k$ for $i > \ell_j$.

After evaluating the current column of the matrix up to row ℓ_j , the value ℓ_{j+1} is computed, and the algorithm continues with the next column $j + 1$. The evaluation of ℓ_j takes $O(1)$ amortized time, and its expected value $L(k)$ is $O(k)$. Hence the whole algorithm takes only $O(nk)$ time.

Myers adapted his $O(n\lceil m/w \rceil)$ algorithm to use the cut-off heuristic as well. In principle the idea is very simple; since on average the search ends at row $L(k)$, it is enough to use only $L(k)$ bits of the computer word on average (actually he used $w\lceil L(k)/w \rceil$ bits), and only in some text positions (for example when the pattern matches) one has to use more bits. Only two modifications to the basic method are needed. We must be able to decide which is the last active row in order to compute the number of bits required for each text position, and we must be able to handle the communication between the boundaries of the consecutive computer words. Both problems are easy to solve, for details refer to [Myers 1999]. With these modifications Myers was able to obtain his $O(n\lceil L(k)/w \rceil)$ average time algorithm.

We can do exactly the same here. We use only $\beta = \max\{L(k), \lceil \log(m+k) \rceil + 1\}$ bits for each pattern and pack them into the same computer word just like in our basic method. We need $L(k)$ bits as $L(k)$ is the row number where the search is expected to end, and at least $\lceil \log(m+k) \rceil + 1$ bits to avoid overflowing the counters. Therefore we are going to search for $\lfloor w/\beta \rfloor$ patterns in parallel.

If for some text positions β bits are not enough, we use as many computer words as needed, each having β bits allocated for each pattern. Therefore, the β -bit blocks in the first computer word correspond to the first β characters of the corresponding patterns, and the β -bit blocks in the second word correspond to the next β characters of the patterns, and so on. In total we need $\lceil m/\beta \rceil$ computer words, but on average use only one for each text position.

The counters for each pattern have only β bits now, which means that the maximum pattern length is limited to $2^{\beta-1} - k$. The previous counters limited the pattern length to $2^{m-1} - k$, but at the same time assumed that the pattern length was $\leq w/2$. Using the cut-off method, we have less bits for the counters, but in effect we can use longer patterns, the upper bound being $m = 2^{w/2-1} - k$.

The tools we have developed for the basic method can be applied to modify Myers’ cut-off algorithm to search for $\lfloor w/\beta \rfloor$ patterns simultaneously. The only additional modification we need is that we must add a new computer word whenever the value of *any* of the pattern counters becomes less or equal to k , and this is trivial to detect with our counters model. On the other hand, this modification means that $L(k)$ must grow as the function of r . It has been shown in [Navarro 2001] that $L(k) = k/(1 - e/\sqrt{\sigma}) + O(1)$ for $r = 1$. For reasonably small r this bound should not be affected much, as the probability of a match is exponentially decreasing for

$m > L(k)$.

The result is that we can search for r patterns with at most k differences in $O(\lceil r\beta/w \rceil n)$ expected time for $\beta \leq w/2$. Finally, it is possible to apply the same scheme for single pattern search as well, resulting in $O(\lceil n/\lfloor w/\beta \rfloor \rceil)$ expected time. The method is useful even for short patterns (where we could apply our basic method also), because we can use tighter packing when $\beta < m$.

7. APPLICATIONS TO OTHER PROBLEMS

The ideas developed in previous sections for approximate string matching can be easily extended to other related problems. To illustrate this fact, we consider in this section three examples: multiple exact string matching, one-against-all edit distance computation, and one-against-all longest common subsequence computation.

7.1 Multiple Exact String Matching

A simple bit-parallel algorithm for exact searching of a single pattern is Shift-And [Wu and Manber 1992; Baeza-Yates and Gonnet 1992]. This algorithm is $O(\lceil m/w \rceil n)$ worst-case time and it works as follows. Consider the table PM of Section 3. The state of the search is maintained in a bit mask D of m bits. The invariant is that, after reading T_j , $D[i] = 1$ iff $P_{1\dots i} = T_{j-i+1\dots j}$. Thus, every text position where $D[m] = 1$ is the endpoint of an exact occurrence of P in T . To maintain the invariant, D is initialized at $D \leftarrow 0^m$, and then character T_j is processed as follows

$$D \leftarrow ((D \ll 1) \mid 0^{m-1}1) \& PM_{T_j},$$

and we report a match whenever $D \& 10^{m-1} \neq 0^m$.

It was already noted in [Wu and Manber 1992] that several patterns whose overall length does not exceed w could be searched for simultaneously using a slight extension of the above scheme: It is a matter of concatenating $P^1 \dots P^r$ and building the PM table for the concatenation, just as **MComputePM** in Fig. 6. It does not even matter that the shift moves the last bit representing P^s onto the first bit representing P^{s+1} because this first bit should anyway be set before *and*ing with PM . Let m_i be the length of P^i . Then we precompute masks $SM \leftarrow 0^{m_r-1}1 \dots 0^{m_1-1}1$ and $EM \leftarrow 10^{m_r-1} \dots 10^{m_1-1}$. The update formula is

$$D \leftarrow ((D \ll 1) \mid SM) \& PM_{T_j},$$

and we report an occurrence of some pattern whenever $D \& EM \neq 0^m$.

In the spirit of Section 6, we propose a variant that can handle longer patterns. The idea is that it is sufficient to consider a short prefix of each pattern, provided the prefix is long enough so that the probability of matching the text is low enough.

Assuming a uniform distribution, the probability of matching a pattern prefix of length β is $1/\sigma^\beta$. In this case we can pack up to $r' = \lfloor w/\beta \rfloor$ patterns in a single search. Every time the search finds any of the prefixes, the techniques of Section 4.3 permit us pointing out each of the matching prefixes in constant time. In turn, each matching prefix is checked for a complete occurrence in constant time as on average $\sigma/(\sigma - 1)$ probes are necessary to find the first differing character. Thus, processing each text character needs $O(1+r'/\sigma^\beta)$ average time. It is sufficient that $\beta = \lceil \log_\sigma r' \rceil$ to ensure that the average time per character is $O(1)$ and thus

the whole scanning is $O(n)$. Therefore, the number of patterns we can pack in a single search must satisfy $r' \leq \lfloor w / \lceil \log_\sigma r' \rceil \rfloor$. The solution has the form $r' = w \log(\sigma) / (\log(w) - \log \log(\sigma)) (1 + o(1))$.

In order to search for r patterns, we split them into groups of size r' and perform $\lceil r/r' \rceil$ searches, for a total average cost of $O(\lceil r/r' \rceil n) = O(\lceil r \log_\sigma w/w \rceil n)$. This is in practice as good as $O(rn/w)$ multiplied by a small constant. Note that the length of the patterns is actually irrelevant for the solution.

The idea of choosing a short prefix from each pattern could be applied to other bit-parallel approaches such as MultiBNDM [Navarro and Raffinot 2000; 2002]. In that case, using prefixes of length β , the search for each group takes average time $O(n \log_\sigma(r'\beta)/\beta)$. Interestingly, essentially the same condition on β as for our algorithm holds, and the final search complexity for r patterns turns out to be the same $O(rn \log_\sigma(w)/w)$. We compare both alternatives experimentally in Section 8.

Finally, just as in Section 5, we can adapt the idea to search for a single pattern by replicating it r' times and search r' text chunks simultaneously, for $O(n/r') = O(n \log_\sigma w/w)$ bit-parallel operations in addition to the n accesses to the text. This time, however, the Shift-And operations are so simple that the reduction in number of bit-parallel operations is not that significant compared to the accesses to text characters.

7.2 One-Against-All Edit Distance

Several biological applications require comparing whole sequences rather than matching one sequence inside another. That is, they require measuring edit distance rather than performing approximate string matching. Moreover, in many cases it is required to compare a sequence against many other short sequences. We show how our technique can be adapted to carry out several of those comparisons simultaneously.

Myers' bit-parallel algorithm is very simply adapted to compute edit distance [Hyvrö and Navarro 2002; Hyvrö and Navarro 2005]. The fact that $D[0, j] = j$ is translated into $HP[0] = 1$ instead of $HP[0] = 0$. Actually, $HP[0]$ is not represented in bit vector HP , but the fact that $HP[0] = 0$ is taken into account in **Step** (Fig. 2) whenever we let $HP \ll 1$ receive a zero fill from the right. Thus, $HP[0] = 1$ is enforced by replacing $(HP \ll 1)$ by $((HP \ll 1) | 0^{m-1}1)$ in lines 4 and 5 of Fig. 2. No other change is necessary to Myers' algorithm.

Let us assume that we want to compute the edit distance of a string B against several short strings $A^1 \dots A^r$, so that the length of A^i is m_i and $m_1 + \dots + m_r \leq w$. Then we can pack all the strings A^i into a single computer word just as we did to search for $P^1 \dots P^r$. The only difference to Section 4.1 is that now we must set rather than clear the bits in HP^p that will align to the first bits of the patterns. That is, we define $ZM = 01^{m_r-1} \dots 01^{m_1-1}$ in same fashion as in Fig. 5, as well as $SM = 0^{m_r-1}1 \dots 0^{m_1-1}1$. Then, the code of **MStep** in Fig. 5 has to be modified only in line 5, where XP is computed as $XP \leftarrow (HP \ll 1) | SM$.

This is the only change necessary to compute all r distances $ed(A^i, B)$ in $O(n)$ time, where n is the length of B , if $m_1 + \dots + m_r \leq w$. In general, r distances to strings A^i of length m can be computed in $O(\lceil rm/w \rceil n)$. After processing all the characters of B , the scores MC described in Section 4.2 contain the information on all the distances.

Note that $ed(A^i, B) \leq \max(n, m_i)$, and thus the i th score in MC will require in principle $\max(\lceil \log n \rceil, \lceil \log m_i \rceil)$ bits, yet we have allocated only m_i bits to it. This creates a potential problem if $m_i < \lceil \log n \rceil$. But we also note that $|n - m_i| \leq ed(A^i, B)$ and that $\max(n, m_i) - |n - m_i| = \min(n, m_i) \leq m_i$. This means that if we subtract $|n - m_i|$ from the i th score $MC(i)$ after all characters of B have been processed, the resulting value will fall within the range $0 \dots m_i$ and m_i bits will suffice. Hence the possible overflow within the counters in MC can be corrected by subtracting $|n - m_i|$ from each $MC(i)$ at the end. After doing this, each true value $ed(A^i, B)$ may be recovered as $ed(A^i, B) = MC(i) + |n - m_i|$.

In a variant of this subproblem, that of computing *thresholded* edit distances, we have a distance threshold k in similar fashion to approximate string matching. The task is to report the actual value $ed(A^i, B)$ only if $ed(A^i, B) \leq k$, and otherwise it is enough to declare that $ed(A^i, B) > k$. In this case we could replace Myers' algorithm by a modification of a technique [Hyyrö 2003] that represents only $O(k)$ cells per column within a central diagonal band of the dynamic programming matrix, resulting in the complexity $O(\lceil rk/w \rceil n)$.

7.3 One-Against-All Longest Common Subsequence

Another classic similarity measure between strings A and B is $llcs(A, B)$, the length of their longest common subsequence. The measure $llcs(A, B)$ and the indel edit distance $id(A, B)$ are related by the equation $2 \times llcs(A, B) = m + n - id(A, B)$, where $m = |A|$ and $n = |B|$. There exist several bit-parallel algorithms [Allison and Dix 1986; Crochemore et al. 2001; Hyyrö 2004] that compute $llcs(A, B)$ in $O(\lceil m/w \rceil n)$ time and work in similar fashion as Myers' algorithm. These algorithms can also be modified to encode several patterns into a single bit vector. In the following we concentrate on the variant of Hyyrö [Hyyrö 2004], which is a more efficient version of the algorithm of Crochemore et al. [Crochemore et al. 2001].

Let L be a $(m + 1) \times (n + 1)$ dynamic programming matrix that corresponds to computing $llcs(A, B)$. The equality $L[i, j] = llcs(A_{1..i}, B_{1..j})$ holds after L has been filled using the following well-known recurrence.

$$L[i, 0] = 0, \quad L[0, j] = 0.$$

$$L[i, j] = \begin{cases} L[i - 1, j - 1] + 1, & \text{if } A_i = B_j. \\ \max(L[i - 1, j], L[i, j - 1]), & \text{otherwise.} \end{cases}$$

Adjacent cell values in L fulfill the condition¹ $L[i, j] - L[i - 1, j] = 0$ or 1 . This means that a single length- m vertical delta vector is enough to encode the values of a column of L . Hyyrö's algorithm does this with the following length- m complemented vertical delta vector V' : $V'[i] = 1$ at position j of B iff $L[i, j] - L[i - 1, j] = 0$.

The algorithm uses the same pattern match vectors PM_λ as Myers' algorithm. It computes the vertical delta vector V' for $j \in 1 \dots n$, after which the value $llcs(A, B)$ can be computed as $llcs(A, B) = \sum_{i=1}^m (1 - V'[i])$ (ie. the number of zero bits in V'). Fig. 8 shows the complete algorithm.

Since the algorithm uses the same PM_λ vectors as Myers' algorithm, we can directly reuse the procedure `MComputePM` (Fig. 6) to encode r patterns A^i, \dots, A^r of length m into a single length- rm bit vector. Given a string B of length n , the

¹Also the symmetric condition $L[i, j] - L[i, j - 1] = 0$ or 1 holds, but it is not needed here.

```

LLCS( $A, B$ )
1.   ComputePM( $A$ )
2.    $V' \leftarrow 1^m$ 
3.   For  $j \in 1 \dots n$  Do
4.      $U \leftarrow V' \& PM_{B_j}$ 
5.      $V' \leftarrow (V' + U) | (V' - U)$ 
6.    $llcs(A, B)$  is equal to the number of zero bits in  $V'$ 

```

Fig. 8. Hyyrö's algorithm for computing $llcs(A, B)$.

goal is to compute the r $llcs(A^i, B)$ values in $O(\lceil rm/w \rceil n)$ time. The case of non-uniform lengths $m_1 \dots m_r$ can be handled as in Section 4.4.

We will again use masking with $ZM = (01^{m-1})^r$ to prevent the bit regions of different patterns from interfering with each other (see Section 4.1). Consider the lines 4-5 in Fig. 8. We first note that subtracting the vector $U = V' \& PM_{B_j}$ from V' does not create any carry effects. So the only possible source of interference between different bit regions is the addition $V' + U$, and this can be fixed by changing the addition into the form $(V' \& ZM) + (U \& ZM)$. To confirm that this modification does not affect the correct behaviour of the algorithm, we note the following: If $V'[m] = 0$ before the addition, then also $U[m] = 0$ and the modification has no effect. If $V'[m] = 1$ and $U[m] = 1$ before the addition, then the first m bits of the result are the same: the modification just removes the $(m + 1)$ th carry bit. Finally, if $V'[m] = 1$ and $U[m] = 0$ before the addition, then the m th bit of the result of the addition is not important: the result is anyway *ored* with $(V' - U)$, which has its m th bit set in this case.

After all characters of B have been processed, each value $llcs(A^i, B)$ can be read from V' by counting the number of 0 bits in the length- m_i bit region of the string A^i . Fig. 9 shows the modified algorithm for computing multiple $llcs(A^i, B)$ values in parallel.

```

MLLCS( $A^1 \dots A^r, B$ )
1.   MComputePM( $A^1 \dots A^r$ )
2.    $V' \leftarrow 1^{rm}$ 
3.    $ZM \leftarrow (01^{m-1})^r$ 
4.   For  $j \in 1 \dots n$  Do
5.      $U \leftarrow V' \& PM_{B_j}$ 
6.      $V' \leftarrow ((V' \& ZM) + (U \& ZM)) | (V' - U)$ 
7.    $llcs(A^i, B)$  is equal to the number of zero bits in the bit region  $V'[m(i-1) + 1 \dots mi]$ 

```

Fig. 9. Our algorithm for computing multiple $llcs(A^i, B)$ values in parallel.

8. EXPERIMENTAL RESULTS

We have conducted experiments in order to evaluate the performance of our algorithms. The results are presented and discussed in the following subsections. The

experiments were run on an AMD Athlon64 3200+ with 1.5 GB RAM and running a 64-bit Linux operating system. The word size was $w = 64$. All included algorithms were implemented in C and compiled with GCC 3.3.1 using optimization.

8.1 Approximate String Matching: A Single Pattern

In the first experiment we tested our single-pattern approximate string matching techniques from Sections 5 and 6 against most relevant existing alternatives.

The texts we used were composed of DNA from baker's yeast and natural language English text from the TREC collection. Each text was copied repeatedly to be 40 million characters long. We tested with pattern lengths $m \in \{8, 16, 32, 64\}$, and the patterns were selected randomly from the texts. For each combination (m, k) , we measured the average time over searching for 100 patterns. The set of patterns was the same for each algorithm. We omitted using the values $k \in \{0, m - 1, m\}$ because the first reduces to exact string matching, the second to finding occurrences of any character of the pattern, and the last to declaring matches at all text positions.

The algorithms included in the experiments were those known to be the most promising from previous experiments [Navarro 2001; Navarro and Raffinot 2002]:

BPR. Non-deterministic finite state automaton bit-parallelized by rows [Wu and Manber 1992]. The complexity is $O(\lceil m/w \rceil kn)$. We used our implementation with hand optimized special code for each different tested k value.

BPD. Non-deterministic finite state automaton bit-parallelized by diagonals [Baeza-Yates and Navarro 1999]. The complexity is $O(\lceil km/w \rceil n)$. Implemented by its original authors.

BPM. Myers' original algorithm [Myers 1999], whose complexity is $O(\lceil m/w \rceil n)$. We used our implementation, which was roughly 20 % faster than the original code of Myers on the test computer.

BPP. A combined heuristic [Navarro and Baeza-Yates 2001] using pattern partitioning, superimposition and hierarchical verification. Built upon *BPD* and implemented by its original authors.

PEX. Partitioning the pattern into $k+1$ pieces and using hierarchical verification with *BPD* in verifying full matches [Navarro and Baeza-Yates 1999], implemented by its original authors. The average case complexity is $O(nk \log_{\sigma}(m)/m)$.

OPT. Average-optimal filtering method [Fredriksson and Navarro 2004], using hierarchical verification with *BPM* in verifying full matches. Implemented by its original authors. Its average case complexity is $O(n(k + \log_{\sigma} m)/m)$. We found that the best options for this test were to use ordered ℓ -grams and backward matching. For DNA we used ℓ -grams in the range 6...8, and for English texts in the range 3...4. For the latter case we used also alphabet mapping, mapping the original alphabet to 16 character groups, each having (approximately) the same probability of appearing in the text.

PAR. Our basic parallelized single-pattern search algorithm (Section 5), which is applicable when $m \leq w/2 = 32$.

PAR-CO. Our parallelized single-pattern search algorithm (Section 5) using cut-off (Section 6). For each case we report the best time over all choices of $\beta < m$.

We used a minimum value $\beta = 8$, so *PAR-CO* was used when $m > 8$.

The results are shown in Fig. 10. Our algorithms perform very well especially with short patterns. On English text, the cases where either *PAR* or *PAR-CO* is the fastest are $m = 8$ for all k , $m = 16$ for all k , $m = 32$ for all $k > 1$, and $m = 64$ for $k \in \{7 \dots 17\}$. On DNA, *PAR* or *PAR-CO* was the best in the cases $m = 8$ for all k , $m = 16$ for all k , $m = 32$ for $k > 2$, and $m = 64$ for $k \in \{8, 9\}$. Note that we could interpret our *PAR/PAR-CO* techniques to subsume the basic Myers' algorithm as the case $r = 1/\beta = m$. The result would be that *PAR* or *PAR-CO* would be the fastest also in the cases $m = 64$ and $k \in \{21 \dots 62\}$ on English text, and $m = 64$ and $k \in \{10 \dots 61\}$ on DNA.

The comparison between *PAR* and *PAR-CO* is as follows. On English text, *PAR-CO* is faster than *PAR* when $m = 16$ and $k \leq 3$, and when $m = 32$ and $k \leq 8$. On DNA, *PAR-CO* is faster than *PAR* when $m = 32$ and $k \leq 5$.

Overall, the result is that our new single-pattern algorithm is clearly superior to any other in the vast majority of cases. A possible drawback of *PAR-CO* is that we need to know the best β to obtain the times shown. Fig. 12 shows, however, that this optimum value is a rather predictable linear function of k , at least for moderate k values (the most interesting ones). Values of β to the optimum ones do not yield significantly worse performance, as long as one is conservative. That is, while using a value slightly smaller than the optimum may quickly degrade the performance, using slightly larger values does not have an important impact in performance.

Finally, one may wonder how our algorithm behaves when m is not a power of 2. The times essentially depend on the numbers packed in the computer word, $\lfloor w/m \rfloor$, so the lengths we have chosen correspond to some relevant transition points in the search times. Note, on the other hand, that if we use the cutoff technique the search times do not depend on m .

8.2 Approximate String Matching: Several Patterns

In the second experiment we tested our techniques for conducting approximate string matching for several patterns (Sections 4 and 6). The texts and patterns were exactly the same as in the experiments of Section 8.1. We measured the average time per pattern when searching for a set of 100 patterns of equal length.

The algorithms included in these experiments were the most promising ones according to the most recent work [Fredriksson and Navarro 2004]:

Single. Handling each pattern separately with a single pattern approximate string matching algorithm. For each case, we took the best time over all single pattern algorithms in the experiments of Section 8.1.

MM. A hashing-based algorithm handling only the case $k = 1$ but known to be resistant to high number of patterns [Muth and Manber 1996].

MOPT. Average-optimal filtering method to search for several patterns [Fredriksson and Navarro 2004], with average complexity $O(n(k + \log_{\sigma}(rm))/m)$. Implemented by its original authors. The optimum choices for this test were to use ordered ℓ -grams and backward matching with bit-parallel counters. For DNA we used ℓ -grams in the range 6...8, and for English texts in the range 3...4. For the latter case we used also alphabet mapping, mapping the original alphabet to 16

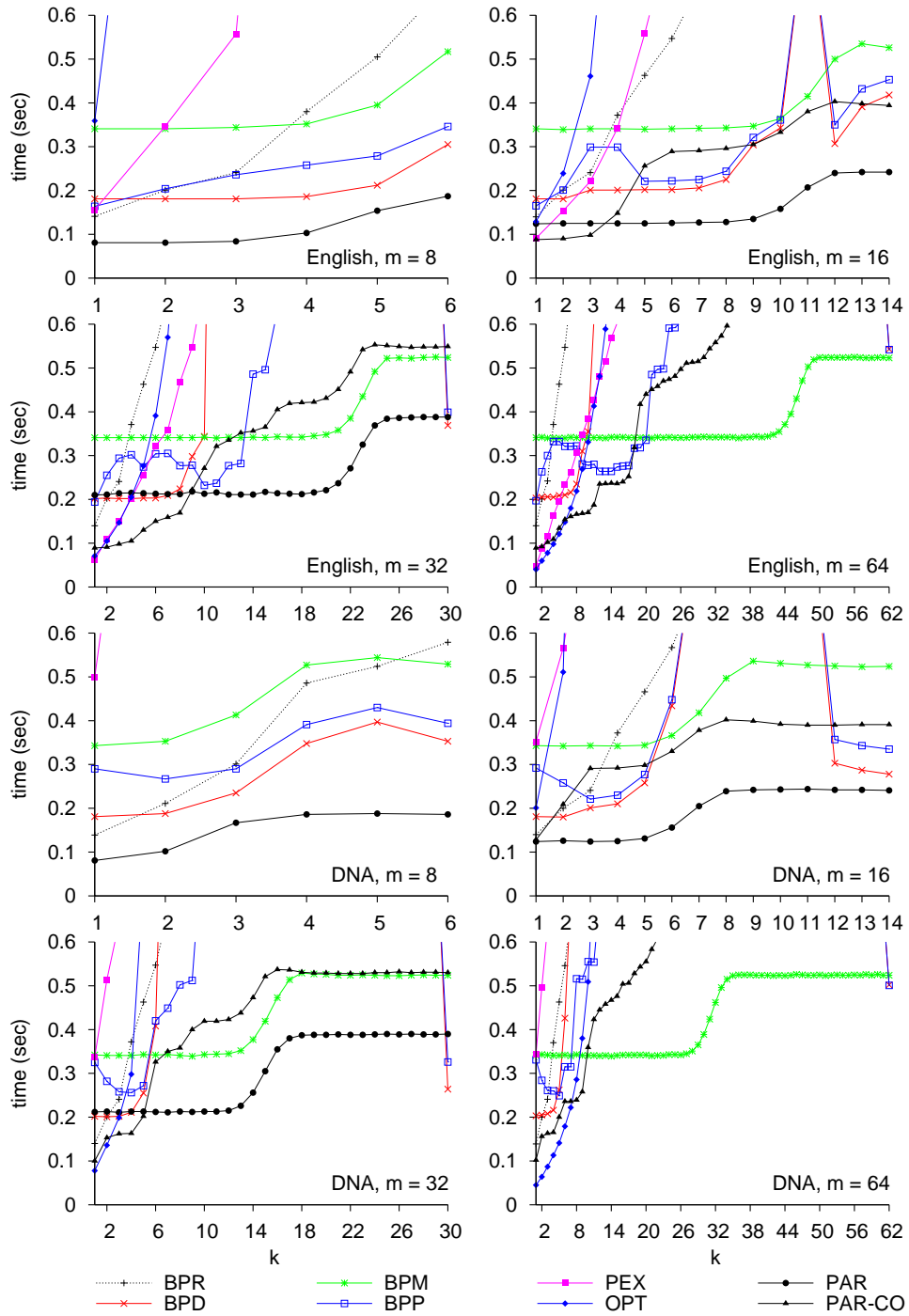


Fig. 10. Approximate string matching: a single pattern. The plots show the average time over searching for 100 patterns.

character groups, each having (approximately) the same probability of appearing in the text. Hierarchical verification was used in both cases.

MPAR. Our basic parallelized algorithm to search for several patterns (Section 4), which is applicable when $m \leq w/2 = 32$.

MPAR-CO. Our basic parallelized algorithm to search for several patterns (Section 4) using cut-off (Section 6). For each case we report the best time over all choices of $\beta < m$. We used a minimum value $\beta = 8$, so *PAR-CO* was used when $m > 8$.

The results are shown in Fig. 11. Our algorithms perform again very well especially with short patterns. On English text, the cases where either *MPAR* or *MPAR-CO* is the fastest are $m = 8$ for all $k > 1$, $m = 16$ for $k > 2$, $m = 32$ for $k > 4$, and $m = 64$ for $k \in \{10 \dots 27, 48 \dots 62\}$. On DNA, *MPAR* or *MPAR-CO* was the best in the cases $m = 8$ for all k , $m = 16$ for $k > 1$, $m = 32$ for $k > 4$, and $m = 64$ for $k \in \{9 \dots 17, 33 \dots 62\}$.

The comparison between *MPAR* and *MPAR-CO* is as follows. On English text, *MPAR-CO* is faster than *MPAR* when $m = 16$ and $k \leq 4$, and when $m = 32$ and $k \leq 11$. On DNA, *MPAR-CO* is faster than *MPAR* when $m = 16$ and $k = 1$, and when $m = 32$ and $k \leq 5$.

Fig. 12 also shows what the best choices of β were for *MPAR-CO*.

8.3 Multiple Exact String Matching

In the third experiment we tested our technique for multiple exact string matching (Section 7.1). The texts and patterns were again as in the experiments of Section 8.1. For each pattern length we tested set sizes $r = \{x \mid \lfloor w/x \rfloor \neq \lfloor w/(x+1) \rfloor\} = \{2, \dots, 10, 12, 16, 21, 32\}$, and the overall number of patterns processed in each case was the nearest integer to 100 that is divisible by r .

We have considered both multipattern algorithms and repeated applications of single-pattern algorithms. The algorithms included in these experiments (chosen as the most promising in previous experiments [Navarro and Raffinot 2002]) follow. All the implementations are by the authors of [Navarro and Raffinot 2002] except of course for our new algorithms, which were implemented by ourselves.

SO. Shift-Or single pattern exact string matching algorithm [Baeza-Yates and Gonnet 1992], which is $O(\lceil m/w \rceil n)$ worst-case and $O(n)$ average-case time.

BNDM. The bit-parallel version [Navarro and Raffinot 2000; 2002] of the classical average-optimal BDM single pattern exact string matching algorithm. Its average-case complexity is $O(n \log_\sigma(rm)/w)$.

AC. Aho-Corasick multiple exact string matching algorithm [Aho and Corasick 1975], which has $O(n)$ worst-case search time.

BMH. Set-Horspool, a multipattern version of the Boyer-Moore-Horspool exact string matching algorithm [Horspool 1980; Navarro and Raffinot 2002].

MBOM. Multipattern version of the Backward Oracle Matching algorithm [Al-lauzen and Raffinot 1999; Navarro and Raffinot 2002], a simplification of Multiple BDM.

MBNDM. A bit-parallel version [Navarro and Raffinot 2000; 2002] of Multiple BDM.

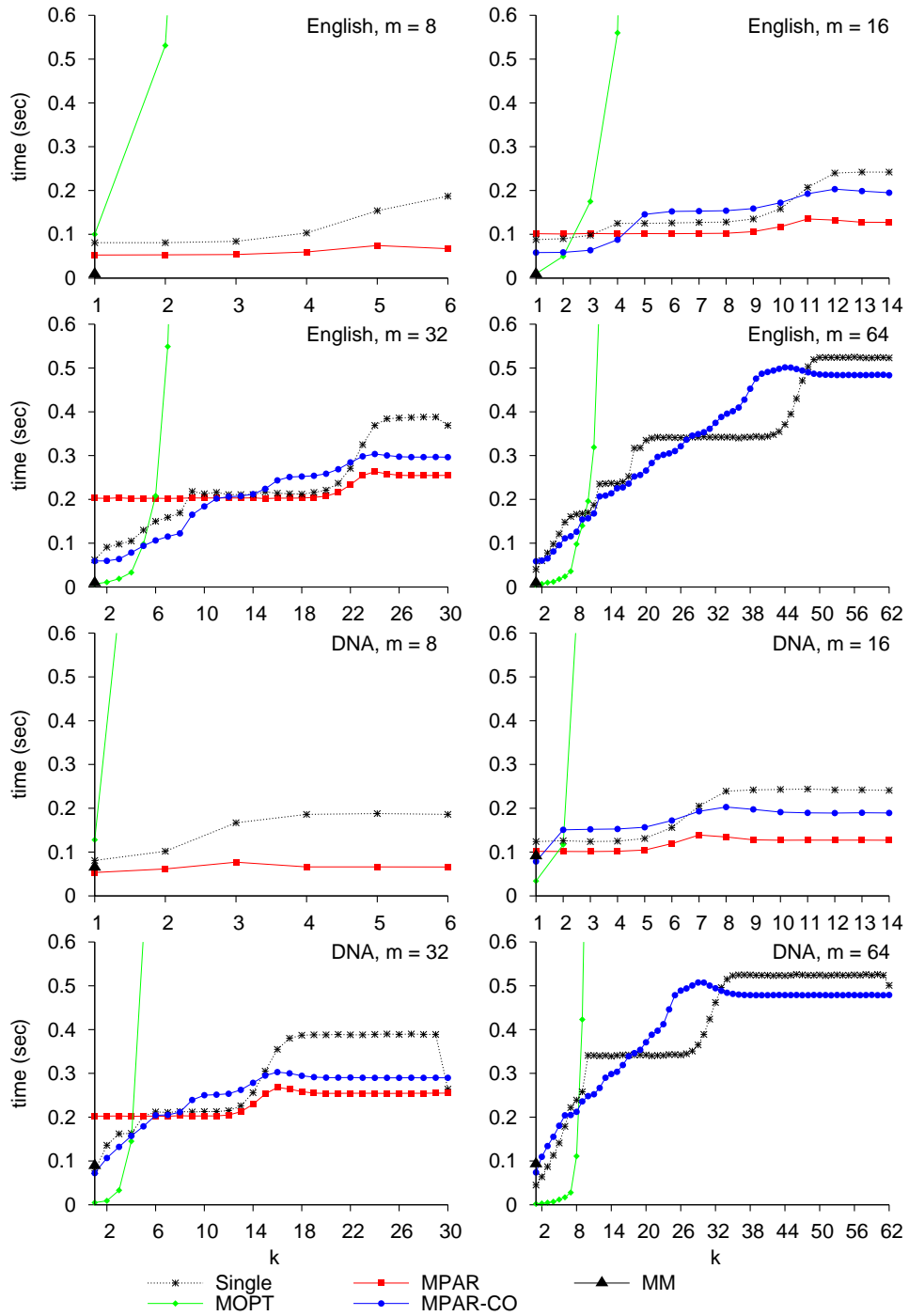


Fig. 11. Approximate string matching: several patterns. The plots show the average time per pattern when searching for a set of 100 patterns.

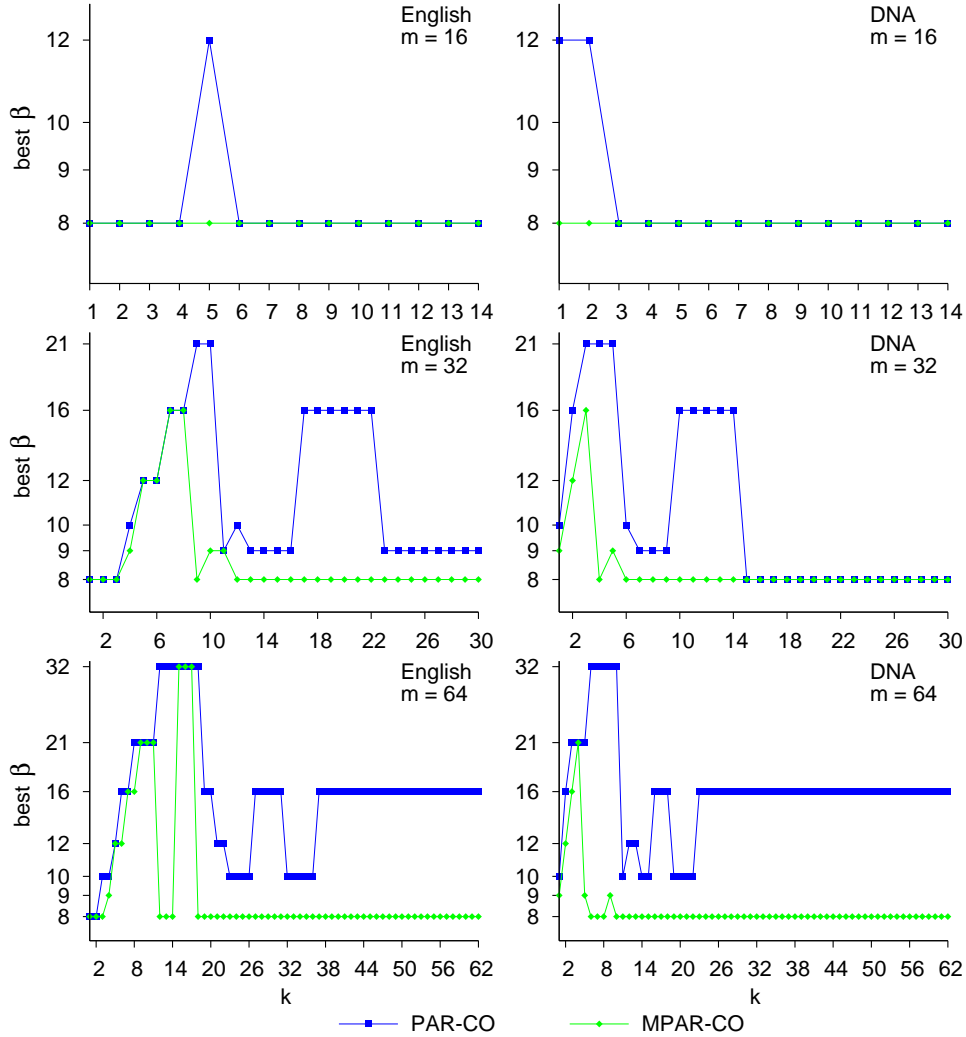


Fig. 12. The plots show the best choice of β , the number of bits allocated for each pattern, when using the cut-off technique (Section 6) in approximate string matching.

PMBNDM. Our modification of MultiBNDM to allocate β bits per pattern (Section 7.1).

MSA. Shift-Or modified to search for several patterns simultaneously by allocating β bits per pattern (Section 7.1).

Fig. 13 shows the results when *PMBNDM* and *MSA* allocate always $\beta = \lfloor w/r \rfloor$ bits per pattern. This shows how the efficiency depends on β . Fig. 14 shows the results when *PMBNDM* and *MSA* use optimal choices for β . For example if $r = 32$, it may be faster to conduct four runs of 8 patterns with $\beta = \lfloor w/8 \rfloor = 8$, than to do a single run of 32 patterns with $\beta = \lfloor w/32 \rfloor = 2$. On English text, our methods are not beaten in the cases $m = 8$ and $r \in \{9, 10, 12, 16, 21, 32\}$, and

$m = 16$ and $r \in \{5, \dots, 10, 12, 16\}$. On DNA, our methods win in the cases $m = 8$ and $r \in \{2, \dots, 10, 12, 16, 21\}$, $m = 16$ and $r \in \{5, \dots, 10, 12, 16, 21\}$, and $m = 32$ and $r \in \{6, \dots, 10\}$.

The comparison between *PMBNDM* and *MSA* is as follows. On English text, *PMBNDM* is faster than *MSA* when $m = 8$ and $r \leq 6$, and when $m \in \{16, 32, 64\}$ and $r \leq 7$. On DNA, *PMBNDM* is faster than *MSA* when $m \in \{16, 32, 64\}$ and $r \leq 4$.

Overall, our methods display considerable improvements over previous algorithms especially on a moderate number of short patterns.

8.4 One-Against-All String Comparison

In the fourth, and last, experiment we evaluated our techniques for computing several edit distances simultaneously (Sections 7.2 and 7.3). This was done in the form of all-against-all string comparison: given a set of x patterns $P^1 \dots P^x$, the task was to compute the set of distances $\{ed(P^i, P^j) \mid 1 \leq i < j \leq x\}$, or in similar fashion, the set of similarity values $\{llcs(P^i, P^j) \mid 1 \leq i < j \leq x\}$. The tested pattern lengths were $m \in \{10, 12, 16, 21, 32\}$, and for each length we used a pattern set of size $x = 6000$. Each case involved computing roughly 18 million distance/similarity values, and we measured the overall time. In this test the patterns were randomly generated with alphabet size $\sigma = 100$ (this choice is arbitrary and does not really affect processing times). For each pattern length m , we tested handling $r = 1 \dots \lfloor w/m \rfloor$ computations in parallel. In this test we evaluated the effect of the number of different computations that are done simultaneously.

Fig. 15 shows the results. It can be seen that doing several computations in parallel pays off. The gain from increasing the number of simultaneous computations is clearly noticeable for $r = 2 \dots 4$. Using $r > 4$ does not seem to offer noticeable gain over using $r = 4$.

9. CONCLUSIONS

Bit-parallel algorithms are currently the fastest approximate string matching algorithms for many relevant applications. In particular, the algorithm of Myers [Myers 1999] dominates the field when the pattern is long enough, thanks to its better packing of the search state in the bits of the computer word. In this paper we showed how this algorithm can be modified to take advantage of the wasted bits when the pattern is short. We have shown two ways to do this. The first one permits searching for several patterns simultaneously. The second one boosts the search for a single pattern by processing several text positions simultaneously. We have also shown how the same ideas can be used for multiple exact string matching and for one-against-all distance computation of different sorts.

We have shown, both analytically and experimentally, that our algorithms are significantly faster than all the other bit-parallel algorithms when the pattern is short or if the error threshold is moderate with respect to the alphabet size. This fills an important gap in the field, as there has been no previous work focusing on very short patterns, which is the case in many relevant applications.

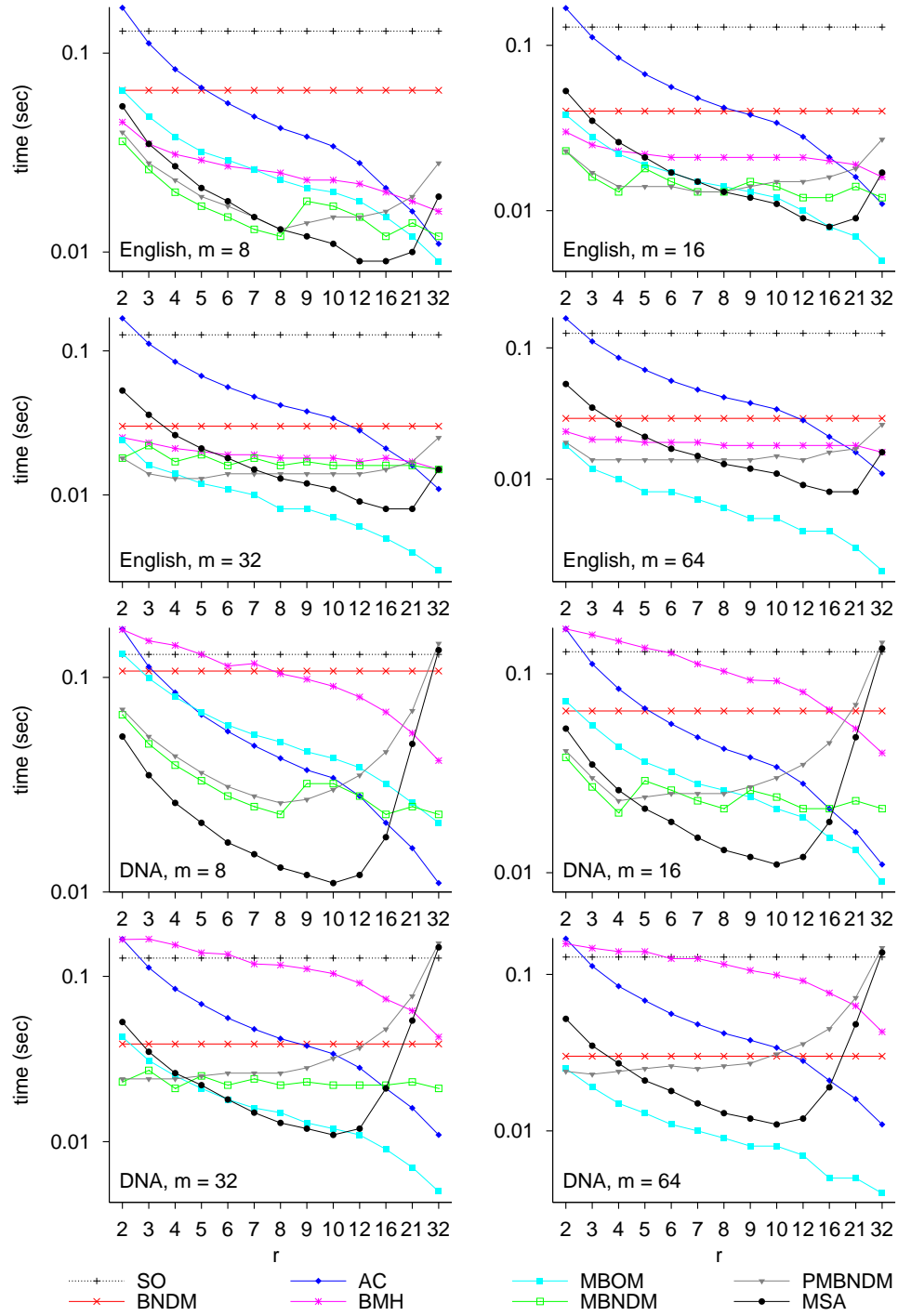


Fig. 13. Multiple exact string matching. The plots show the average time per pattern when searching for roughly 100 patterns in groups of r patterns.
 ACM Journal Name, Vol. V, No. N, Month 20YY.

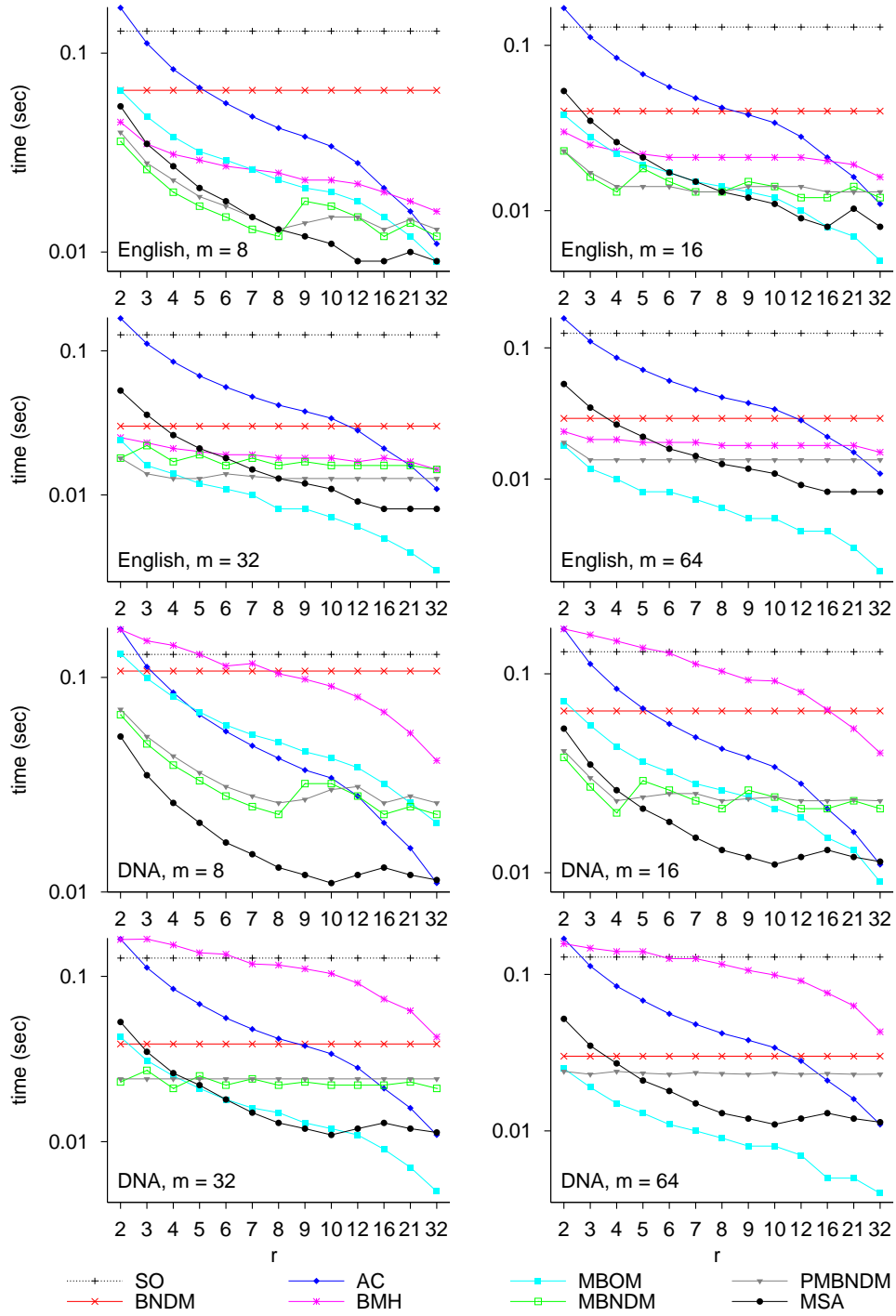


Fig. 14. Multiple exact string matching. The plots are otherwise as in Fig. 13, but now PMBNDM and MSA divide each group of r patterns into possibly smaller subgroups in an optimal manner.

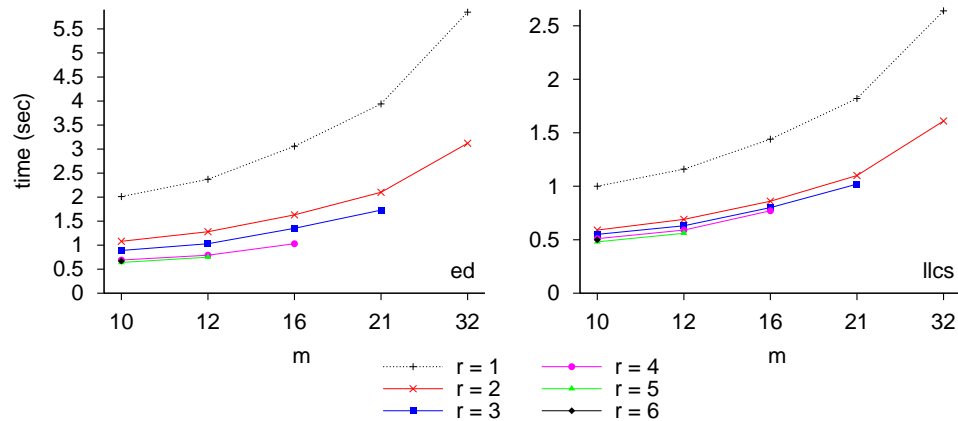


Fig. 15. All-against-all string comparison. The plots show the overall time for a set of 6000 patterns. The left side is for computing edit distance and the right side for computing the length of a longest common subsequence.

REFERENCES

- AHO, A. AND CORASICK, M. 1975. Efficient string matching: an aid to bibliographic search. *Communications of the ACM* 18, 6, 333–340.
- ALLAUZEN, C. AND RAFFINOT, M. 1999. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée.
- ALLISON, L. AND DIX, T. L. 1986. A bit-string longest common subsequence algorithm. *Information Processing Letters* 23, 305–310.
- BAEZA-YATES, R. AND GONNET, G. 1992. A new approach to text searching. *Communications of the ACM* 35, 10, 74–82.
- BAEZA-YATES, R. AND NAVARRO, G. 1999. Faster approximate string matching. *Algorithmica* 23, 2, 127–158.
- CROCHEMORE, M., ILIOPOULOS, C. S., PINZON, Y. J., AND REID, J. F. 2001. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters* 80, 279–285.
- FREDRIKSSON, K. 2003. Row-wise tiling for the Myers’ bit-parallel dynamic programming algorithm. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE’03)*. LNCS 2857. Springer, Berlin, Germany, 66–79.
- FREDRIKSSON, K. AND NAVARRO, G. 2004. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics (JEA)* 9, 1.4.
- HORSPOOL, R. N. 1980. Practical fast searching in strings. *Software Practice and Experience* 10, 6, 501–506.
- HYRÖ, H. 2001. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Tech. Rep. A-2001-10, Department of Computer and Information Sciences, University of Tampere, Tampere, Finland.
- HYRÖ, H. 2003. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing* 10, 1–11.
- HYRÖ, H. 2004. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA’04)*. 16–27.
- HYRÖ, H. AND NAVARRO, G. 2002. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM’02)*. LNCS 2373. Springer, Berlin, Germany, 203–224.
- HYRÖ, H. AND NAVARRO, G. 2005. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica* 41, 3, 203–231.
- ACM Journal Name, Vol. V, No. N, Month 20YY.

- LEVENSHEIN, V. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 8, 707–710. Original in Russian in *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.
- MUTH, R. AND MANBER, U. 1996. Approximate multiple string search. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*. LNCS 1075. Springer-Verlag, 75–86.
- MYERS, G. 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 string matching. *Algorithmica* 30, 4, 473–502.
- 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, Cambridge, UK. ISBN 0-521-81307-7.
- SELLERS, P. 1980. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms* 1, 359–373.
- UKKONEN, E. 1985a. Algorithms for approximate string matching. *Information and Control* 64, 1–3, 100–118.
- UKKONEN, E. 1985b. Finding approximate patterns in strings. *Journal of Algorithms* 6, 132–137.
- WARREN JR, H. S. 2003. *Hacker's Delight*. Addison-Wesley, Boston, MA. ISBN 0-201-91465-4.
- WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Communications of the ACM* 35, 10, 83–91.