Increased Bit-Parallelism for Approximate String Matching

Heikki Hyyrö[1,2] [⋆], Kimmo Fredriksson[3] [⋆⋆], and Gonzalo Navarro[4] [⋆⋆⋆]

[1] PRESTO, Japan Science and Technology Agency
[2] Department of Computer Sciences, University of Tampere, Finland
`Heikki.Hyyro@cs.uta.fi`
[3] Department of Computer Science, University of Joensuu, Finland
`Kimmo.Fredriksson@cs.joensuu.fi`
[4] Department of Computer Science, University of Chile
`gnavarro@dcc.uchile.cl`

Abstract. 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 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 in a single computer word so as to search for multiple patterns simultaneously. Instead of paying $O(rn)$ time to search for r patterns of length $m < w$, we obtain $O(\lceil r/\lfloor w/m \rfloor \rceil n)$ time. Second, we show how the mechanism permits boosting the search for a single pattern of length $m < w$, which can be searched for in time $O(n/\lfloor w/m \rfloor)$ instead of $O(n)$. Finally, 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.

Our experimental results show that that the algorithms work well in practice, and are the fastest alternatives for a wide range of search parameters.

1 Introduction

Approximate string matching is an old problem, with applications for example in spelling correction, bioinformatics and signal processing [7]. It refers in general to searching for substrings of a text that are within a predefined edit distance

[⋆] Supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering.

[⋆⋆] Supported by the Academy of Finland.

[⋆⋆⋆] Supported in part by Fondecyt grant 1-020831.

threshold from a given pattern. Let $T = T_{1...n}$ be a text of length n and $P = P_{1...m}$ a pattern of length m. Here $A_{a...b}$ denotes the substring of A that begins at its ath character and ends at its bth 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 $ed(P, T_{h...j}) \leq k$ for some $h \leq j$.

The most common form of edit distance is Levenshtein distance [5]. 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. We also use w to denote the computer word size in bits, σ to denote the size of the 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 [13], the $O(\lceil km/w \rceil n)$ algorithm of Baeza-Yates and Navarro [1], and the $O(\lceil m/w \rceil n)$ algorithm of Myers [6] 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 in a modern computer, and for example the Pentium 4 processor allows one to use even words of size 128. 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. In signal processing applications one can search for sequences 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 [2], where $O(m\lceil n/w \rceil)$ time was obtained. Our technique is different. We first show how to search for several patterns simultaneously by packing them all in the same computer word. We can search for r patterns of length $m \leq w$ in $O(\lceil r/\lfloor m/w \rfloor \rceil n + occ)$ rather than $O(rn)$ time, where $occ \leq rn$ is the total number of occurrences of all the patterns. We then show how this idea can be pushed further to boost the search for a single pattern, so as to obtain $O(n/\lfloor w/m \rfloor)$ time instead of $O(n)$ for $m \leq w$.

Our experimental results show that the presented schemes work well in practice.

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 $i = 1 \ldots |A|$ and $j = 1 \ldots |B|$ 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 [9]. 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 [11]:

-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 ith 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 [3, 8] that is slightly simpler than the original by Myers [6]. 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 if and only if $D[i,j] - D[i-1,j] = 1$.
-Vertical negative delta: $VN[i] = 1$ at text position j if and only if $D[i,j] - D[i-1,j] = -1$.
-Horizontal positive delta: $HP[i] = 1$ at text position j if and only if $D[i,j] - D[i,j-1] = 1$.
-Horizontal negative delta: $HN[i] = 1$ at text position j if and only if $D[i,j] - D[i,j-1] = -1$.
-Diagonal zero delta: $D0[i] = 1$ at text position j if and only if $D[i,j] = D[i-1,j-1]$.
-Pattern match vector PM_λ for each $\lambda \in \Sigma$: $PM_\lambda[i] = 1$ if and only if $P_i = \lambda$.

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. 1 shows the complete formula for updating the vectors, and Fig. 2 shows the preprocessing of table PM and the higher-level search scheme. We refer the reader to [3, 6] for a more detailed explanation of the formula in Fig. 1.

```
Step($j$)
 1.   $D0 \leftarrow (((PM_{T_j}\ \&\ VP) + VP)\ \hat{}\ VP)\ |\ PM_{T_j}\ |\ VN$
 2.   $HP \leftarrow VN\ |\ \sim(D0\ |\ VP)$
 3.   $HN \leftarrow VP\ \&\ D0$
 4.   $VP \leftarrow (HN << 1)\ |\ \sim(D0\ |\ (HP << 1))$
 5.   $VN \leftarrow (HP << 1)\ \&\ D0$
```

Fig. 1. Updating the delta vectors at column j.

The algorithm in Fig. 2 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. 1. After the horizontal delta vectors HP and HN are shifted

```
ComputePM($P$)
1.    For $\lambda \in \Sigma$ Do $PM_\lambda \leftarrow 0^m$
2.    For $i \in 1 \ldots m$ Do $PM_{P_i} \leftarrow PM_{P_i} \mid 0^{m-i}10^{i-1}$

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

Fig. 2. Preprocessing the PM-table and conducting the search.

left, their first bits correspond to the difference $D[0, j] - D[0, j-1]$. This is the only phase 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. 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. Hence 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 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. 3a 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. 3b depicts our proposal: encode several patterns into the bit vectors and search for them in parallel. There are several obstacles in achieving this goal correctly, which will be discussed next.

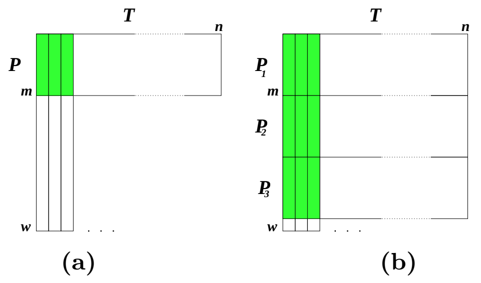

Fig. 3. For short patterns ($m < w$) Myers' algorithm (a) wastes $w - m$ bits. Our proposal (b) packs several pattern into the same computer word, and wastes only $w - rm$ bits.

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 repeatedly into the vectors without leaving any space between them. For example $D0^p[i]$ corresponds to the bit $D0[((i-1) \bmod m)+1]$ in the $D0$-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. 1 it is obvious that only the addition ("+") on line 2 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 on the other hand the correct behaviour of the algorithm is not affected: The value $VP^p[i]$ can affect only the values $D0^p[i+h]$ 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 $ZM = (01^{m-1})^r$. Figure 4 gives the code for a step.

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

```
MStep(j)
1.    $XP \leftarrow VP$ & $ZM$
2.    $D0 \leftarrow (((PM_{T_j}$ & $XP) + XP)$ ^ $XP) \mid PM_{T_j} \mid VN$
3.    $HP \leftarrow VN \mid \sim (D0 \mid VP)$
4.    $HN \leftarrow VP$ & $D0$
5.    $XP \leftarrow HP$ & $ZM$, $XN \leftarrow HN$ & $ZM$
6.    $VP \leftarrow ((XN << 1) \mid \sim (D0 \mid (XP << 1)))$
7.    $VN \leftarrow (XP << 1)$ & $D0$
```

Fig. 4. Updating the delta vectors at column j, when searching for multiple patterns.

fashion to [4]. 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 ith 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 mth bit of $MC(i)$ is set if and only if $b - MC(i) \leq k$. (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 $MC \leftarrow MC + ((HN^p \,\&\, EM) >> (m-1)) - ((HP^p \,\&\, EM) >> (m-1))$.

Property (3) means that the last bit of $MC(i)$ signals whether the ith 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. 5 gives the code to search for the patterns $P^1 \dots P^r$.

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

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

Fig. 5. Preprocessing the PM-table and conducting the search for multiple patterns.

of the (s/m)th pattern at the current text position j, clear the sth bit in OM by doing $OM \leftarrow OM \ \& \ \sim (1 << (s-1))$, 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), i.e. it holds that

$$\begin{array}{rcccl} 2^{\lfloor \log_2(x) \rfloor} & \leq & x & < & 2^{\lfloor \log_2(x) \rfloor + 1}, \\ 1 << \lfloor \log_2(x) \rfloor & \leq & x & < & 1 << (\lfloor \log_2(x) \rfloor + 1). \end{array}$$

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 e.g. to [12]. 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. 6 gives the code.

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.

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

Fig. 6. Reporting occurrences at current text position.

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 $m_1 \ldots m_r$ and the thresholds are $k_1 \ldots k_r$, we have to *and* the vertical and horizontal vectors with $ZM = 01^{m_r-1}\, 01^{m_{r-1}-1} \ldots 01^{m_1-1}$, and this fixes the problem of updating the delta vectors. With respect to the counters, the ith 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 "$>> (m-1)$", 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 few operations.

Finally, reporting the occurrences works just as before, except that the pattern number we report is no longer $(s+1)/m$ (Fig. 6). The correct pattern number can be computed efficiently e.g. 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.

Of course this is of little interest in principle, as all the copies of the pattern will report the same occurrences. However, the key idea here will be to search a *different* text segment for each pattern copy. We divide the text T into r equal-sized subtexts $T = T^1T^2 \dots T^r$. Text T^s, of length $\ell = \lceil n/r \rceil$, will be searched for the sth 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 $T_j,\ T_{j+\ell},\ T_{j+2\ell}, \dots, T_{j+(r-1)\ell}$. 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

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

We must exercise some care at the boundaries between consecutive text segments. On the one hand, processing of text segment T_s $(1 \le s < r)$ 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.

Adding up the $\lceil n/r \rceil = \lceil n/\lfloor w/m \rfloor \rceil$ bit-parallel steps required plus the n character accesses to compute PM, we obtain $O(\lceil n/\lfloor w/m \rfloor \rceil)$ complexity for $m \le w$.

6 Long Patterns and *k*-differences Problem

We have shown how to utilize the bits in computer word economically, but our methods assume that $m \le w$. 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, but replace 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 [10]. That algorithm fills the table D in column-wise order, and computes the values $D[i,j]$ in column j for only $i \le \ell_j$, where

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

The cut-off heuristic is based on the fact that the search result does not depend on cells whose value is larger than k. From the diagonal property it follows that once $D[i,j] > k$, then $D[i+h, j+h] > k$ for all $h \ge 0$ (within the bounds of D). And a consequence of this is that $D[i,j] > k$ for $i > \ell_j$.

After evaluating the current column of the matrix up to the 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 $O(k)$, and 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 (e.g. 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 [6]. 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 $b = \max\{L(k), \lceil \log(m+k) \rceil + 1\}$ 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 $\lfloor w/b \rfloor$ patterns in parallel.

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

The counters for each pattern have only b bits now, which means that the maximum pattern length is limited to $2^{b-1} - k$. 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 $\lfloor w/b \rfloor$ patterns simultaneously. The only additional modification we need is that we must add a new computer word whenever *any* of the pattern counters has accumulated k differences, 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 [7] 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 $O(n\lceil r/\lfloor w/b \rfloor \rceil)$ expected time. Finally, it is possible to apply the same scheme for single pattern search as well, resulting in $O(\lceil n/\lfloor w/b \rfloor \rceil)$ 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 $b < m$.

7 Experimental Results

We have implemented all the algorithms in C and compiled them using GCC 3.3.1 with full optimizations. The experiments were run on a Sparc Ultra 2 with 128 MB RAM that was dedicated solely for our tests. The word size of the machine is 64 bits.

In the experiments we used DNA from baker's yeast and natural language English text from the TREC collection. Each text was cut into 4 million characters for testing. The patterns were selected randomly from the texts. We compared the performance of our algorithms against previous work. The algorithms included in the experiments were:

Parallel BPM: Our parallelized single-pattern search algorithm (Section 5). We used $r = 3$ for $m = 8$ and $m = 16$, $r = 2$ for $m = 32$, and $r = 2$ and cut-off (Section 6) for $m = 64$.

Our multi-pattern algorithm: The basic multipattern algorithm (Section 4) or its cut-off version (Section 6). We determined which version to use by using experimentally computed estimates for $L(k)$.

BPM: Myers' original algorithm [6], 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.

BPD: Non-deterministic finite state automaton bit-parallelized by diagonals [1]. The complexity is $O(\lceil km/w \rceil n)$. Implemented by its original authors.

BPR: Non-deterministic finite state automaton bit-parallelized by rows [13]. The complexity is $O(\lceil m/w \rceil kn)$. We used our implementation, with hand optimized special code for $k = 1 \ldots 7$.

For each tested combination of m and k, we measured the average time per pattern when searching for 50 patterns. The set of patterns was the same for each algorithm. The results are shown in Fig. 7. Our algorithms are clear winners in most of the cases.

Our single-pattern parallel search algorithm is beaten only when $k = 1$, as **BPR** needs to do very little work in that case, or when the probability of finding occurrences becomes so high that our more complicated scheme for occurrence checking becomes very costly. At this point we would like to note, that the occurrence checking part of our single-pattern algorithm has not yet been fully optimized in practice.

Our multi-pattern algorithm is also shown to be very fast: in these tests it is worse than a single-pattern algorithm only when $w/2 < m$ and k is moderately high with relation to the alphabet size.

8 Conclusions

Bit-parallel algorithms are currently the fastest approximate string matching algorithms when Levenshtein distance is used. In particular, the algorithm of Myers [6] dominates the field when the pattern is long enough, thanks to its

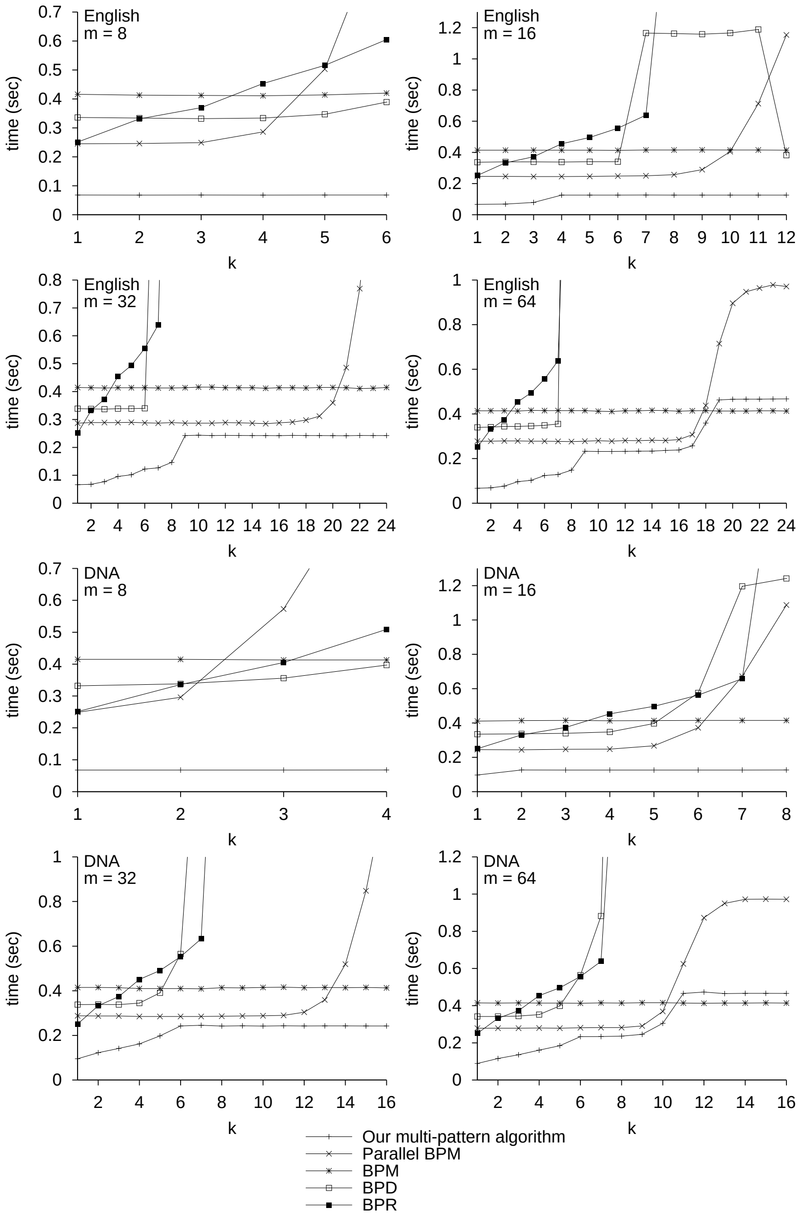

Fig. 7. The plots show the average time for searching a single pattern.

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 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 k is moderate with respect to the alphabet size.

References

1. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
2. K. Fredriksson. 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, pages 66–79, 2003.
3. H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Department of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
4. H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, pages 203–224, 2002.
5. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*
6. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
7. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
8. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7.
9. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.
10. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.
11. Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
12. H. S. Warren Jr. *Hacker's Delight.* Addison Wesley, 2003. ISBN 0-201-91465-4.
13. S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.