Row-wise tiling for the Myers' bit-parallel approximate string matching algorithm

Kimmo Fredriksson[⋆]

Department of Computer Science, PO Box 111,
University of Joensuu, FIN-80101 Joensuu
kfredrik@cs.joensuu.fi.

Abstract. Given a text $T[1..n]$ and a pattern $P[1..m]$ the classic dynamic programming algorithm for computing the edit distance between P and every location of T runs in time $O(nm)$. The bit-parallel computation of the dynamic programming matrix [6] runs in time $O(n\lceil m/w \rceil)$, where w is the number of bits in computer word. We present a new method that rearranges the bit-parallel computations, achieving time $O(\lceil n/w \rceil (m + \sigma \log_2(\sigma)) + n)$, where σ is the size of the alphabet. The algorithm is then modified to solve the k differences problem. The expected running time is $O(\lceil n/w \rceil (L(k) + \sigma \log_2(\sigma)) + R)$, where $L(k)$ depends on k, and R is the number of occurrences. The space usage is $O(\sigma + m)$. It is in practice much faster than the existing $O(n\lceil k/w \rceil)$ algorithm [6]. The new method is applicable only for small (e.g. DNA) alphabets, but this becomes *the fastest algorithm* for small m, or moderate k/m. If we want to search multiple patterns in a row, the method becomes attractive for large alphabet sizes too. We also consider applying 128-bit vector instructions for bit-parallel computations.

1 Introduction

Approximate string matching is a classical problem, with applications to text searching, computational biology, pattern recognition, etc. Given a text $T[1 \ldots n]$, a pattern $P[1 \ldots m]$, and a threshold k, we want to find all the text positions where the pattern matches the text with at most k differences. The allowed differences are the following edit operations: substitution, deletion or insertion of a character.

The first dynamic-programming-based $O(mn)$ time solution to the problem is attributed to [11]. Many faster techniques have been proposed since then, both for the worst and the average case. The first efficient and practical method is the $O(nk)$ expected time algorithm [12]. This was improved to take only $O(n\lceil k/\sqrt{\sigma} \rceil)$ time on average [2], where σ is the size of the alphabet. For large alphabets this is the fastest algorithm based on dynamic programming. For small alphabets the bit-parallel $O(n\lceil k/w \rceil)$ expected time algorithm [6], where w is

[⋆] The work was partially developed while the author was working in the Dept. of CS, Univ. of Helsinki. Partially supported by the Academy of Finland.

the number of bits in computer word (typically 32 or 64), is in practice the fastest. A few years earlier a $O(nm\log(\sigma)/w)$ algorithm was obtained [13], but this is not competitive anymore.

Besides the dynamic programming based methods, there are a myriad of filter based approaches [7]. The filters can work extremely well for small k/m and large alphabets, but the computational biology applications (searching DNA) do not fall into this category. The filtering algorithms are based on simple techniques to quickly eliminate the text positions that cannot match with k differences, and the rest of the text is verified using some slower algorithm. The filter based methods invariably use the dynamic programming based algorithms for verification.

We propose a modified version of the $O(n\lceil k/w\rceil)$ algorithm [6], that runs in $O(\lceil n/w\rceil(L(k)+\sigma\log_2(\sigma))+R)$, expected time, where $L(k)$ depends k, and R is the number of occurrences, and show that in practice this can be much faster than its predecessor, and in fact becomes the fastest algorithm for small m or large k/m.

2 Preliminaries

Let Σ be an alphabet of size σ, and the *pattern* $P[1..m]$ and the *text* $T[1..n]$ be two strings over Σ. We are interested in finding the positions j, such that the edit distance between P and a suffix of $T[1..j]$ is at most k with $0 \le k < m$, for some fixed value of k.

This problem can be solved using dynamic programming. The well-known recurrence [11] for filling the dynamic programming matrix E is as follows:

$$\begin{aligned}
E_{i,0} &= i \\
E_{0,j} &= 0 \\
E_{i,j} &= \min\{E_{i-1,j-1}+\delta(i,j), E_{i-1,j}+1, E_{i,j-1}+1\},
\end{aligned}$$

where $\delta(i,j)$ is 0, if $P[i]=T[j]$, and 1 otherwise.

The matrix can be filled in $O(nm)$ time. The matrix needs also space $O(nm)$, but it is easy to see that if the matrix is filled column-wise, then only one column need to be kept in memory at a time, reducing the space complexity to just $O(m)$.

After the matrix is filled, the values $E_{m,j} \le k$ indicate that there is an occurrence of P with at most k differences, ending at text character $T[j]$.

More efficient variation of this algorithm is given in [12], requiring only $O(nk)$ expected time. The key observation is that in the expected case it is enough to compute only the first $O(k)$ rows of each column j to guarantee that there cannot be an occurrence ending at position j.

One of the properties of the matrix E is that the adjacent (horizontal, vertical or diagonal) values can differ at most by ± 1:

$$\begin{aligned}
\Delta h_{i,j} &= E_{i,j} - E_{i,j-1} \in \{-1,0,+1\} \\
\Delta v_{i,j} &= E_{i,j} - E_{i-1,j} \in \{-1,0,+1\} \\
\Delta d_{i,j} &= E_{i,j} - E_{i-1,j-1} \in \{0,1\}
\end{aligned}$$

It is clear that these vectors fully define the matrix E, and any value $E_{i,j}$ can be obtained e.g. as follows:

$$E_{i,j} = \sum_{h=1}^{j} \Delta v_{i,h}.$$

In [6,4] it was shown how to compute the vectors $\Delta h, \Delta v, \Delta d$ bit-parallelly, in time $O(n \lceil m/w \rceil)$, where w is the number of bits in a computer word. Each vector is $\Delta h, \Delta v, \Delta d$ is now represented by the following bit-vectors:

$$\begin{aligned} hp_{i,j} &\equiv \Delta h_{i,j} = +1 \\ hn_{i,j} &\equiv \Delta h_{i,j} = -1 \\ vp_{i,j} &\equiv \Delta v_{i,j} = +1 \\ vn_{i,j} &\equiv \Delta v_{i,j} = -1 \\ d_{i,j} &\equiv \Delta d_{i,j} = 0 \end{aligned}$$

The values of these vectors can be computed very efficiently using boolean logic. The following logical equivalencies exist:

$$\begin{aligned} hp_{i,j} &\equiv vn_{i,j-1} \text{ OR NOT}(vp_{i,j-1} \text{ OR } d_{i,j}) \\ hn_{i,j} &\equiv vp_{i,j-1} \text{ AND } d_{i,j} \\ vp_{i,j} &\equiv hn_{i-1,j} \text{ OR NOT}(hp_{i-1,j} \text{ OR } d_{i,j}) \\ vn_{i,j} &\equiv hp_{i-1,j} \text{ AND } d_{i,j} \\ d_{i,j} &\equiv (P[i] = T[j]) \text{ OR } vn_{i,j-1} \text{ OR } hn_{i-1,j} \end{aligned}$$

These vectors can be computed using bit-wise logical operations, bit shifts and additions, w vector elements in $O(1)$ time, leading to a $O(n \lceil m/w \rceil)$ time algorithm. The comparison $(P[i] = T[j])$ can be implemented parallelly using preprocessed table of bit masks. For correctness and more details, refer to [6,4].

3 Rearranged bit-parallelism

For short pattens, i.e. for $m \ll w$, many bits of the computer word are effectively wasted, due to the column-wise computation of the matrix. Therefore we suggest that the matrix is computed row-wise, resulting in $O((m + \sigma \log_2(\sigma)) \lceil n/w \rceil + n)$ time, where the $\sigma \log_2(\sigma)$ term comes from preprocessing, and therefore in practice works only for small alphabets. This is relatively straight-forward. The algorithm itself does not change, only the initialization of some vectors. The same technique has been utilized before, although in different contexts, in [5, 3]. This row-wise computation can be seen as the column-wise computation if the roles of P and T are transposed, i.e. P is seen as the text, and T as the pattern. The initial boundary conditions of the dynamic programming matrix are therefore transposed as well.

The complete algorithm is shown in Alg. 1. We use C programming language like notation for the bit-wise operations: OR: `|` , AND: `&` , XOR: `^` , NOT: `~`, shift to left with zero fill: $\ll$, and shift to right with zero fill: $\gg$. The preprocessing is as in the original algorithm, besides that we preprocess T, not P.

Alg. 1 RowWise(n, eq, P, m). Row-wise computation of the dp matrix.

```
Input: $n, eq, P, m$
Output: $E_{m,j}$, $j \in \{1..n\}$

for $r \leftarrow 1$ to $\lceil n/w \rceil$ do
    $vp[r] \leftarrow 0$
    $vn[r] \leftarrow 0$
for $i \leftarrow 1$ to $m$ do
    $cp \leftarrow 1$
    $cn \leftarrow 0$
    for $r \leftarrow 1$ to $\lceil n/w \rceil$ do
        $x \leftarrow eq[P[i]][r] \mid cn$
        $d \leftarrow ((vp[r] + (x \,\&\, vp[r]))$ ^ $vp[r]) \mid x \mid vn[r]$
        $hp \leftarrow vn[r] \mid \sim(vp[r] \mid d)$
        $hn \leftarrow vp[r] \,\&\, d$
        $x \leftarrow (hp \ll 1) \mid cp$
        $vp[r] \leftarrow (hn \ll 1) \mid cn \mid \sim(x \mid d)$
        $vn[r] \leftarrow x \,\&\, d$
        $cp \leftarrow hp \gg (w-1)$
        $cn \leftarrow hn \gg (w-1)$
$d \leftarrow m$
for $r \leftarrow 1$ to $\lceil n/w \rceil$ do
    $d \leftarrow$ BlockScore$(d, (r-1)w, vp[r], vn[r])$
```

Alg. 2 BlockScore(d, pos, vp, vn). Compute scores for one block.

```
Input: $d, pos, vp, vn$
Output: $E_{m,j}$, $j \in \{pos+1..pos+w\}$

for $i \leftarrow 1$ to $w$ do
    $d \leftarrow d + (vp \,\&\, 1) - (vn \,\&\, 1)$
    $vp \leftarrow vp \gg 1$
    $vn \leftarrow vn \gg 1$
    output $pos + i$, $d$
return $d$
```

The algorithm needs $O(\lceil n/w \rceil)$ space to store the vp and vn vectors. Note however, that the order of the two nested loops can be changed, so that vp and

vn need only $O(1)$ space. In this case the carry bits cp and cn should be stored for all m rows (see Alg. 5). The eq table needs $O(\sigma)$ entries.

The computation of the matrix takes only $O(\lceil n/w \rceil m)$ time now. However, there are two problems. First, the preprocessing of T (used as the pattern) takes $O(\sigma \lceil n/w \rceil + n)$ time with the standard method. For small alphabets we can apply bit-parallelism in the preprocessing phase to bring it down to $O(\sigma \log_2(\sigma) \lceil n/w \rceil)$. The second problem is, that after computing the matrix, obtaining the scores from the last row still takes $O(n)$ time. There is not much one can do for this, as the output is of size $O(n)$. In the k-differences problem the maximum distance is restricted to be $\leq k$. In this case, the last row can be evaluated faster bit-parallelly.

4 Preprocessing

The preprocessing algorithm evaluates a bit-vector $eq(c)_j$ for each character $c \in \Sigma$. This is needed for parallel computation of Δd vectors, i.e. to perform the comparisons $P[i] = T[j]$ parallelly. The value of the bit $eq(c)_j$ is 1, iff $T[j] = c$, and 0 otherwise. These vectors are easy to compute in time $O(\min\{m, \sigma\} \lceil n/w \rceil + n)$ (or in time $O(n)$ for fixed σ). The $O(\min\{m, \sigma\} \lceil n/w \rceil)$ time is required to initialize the bit-vectors to 0. This is required only for the characters that actually occur in P. Setting the bits requires $O(n)$ additional time.

For small alphabets we can do better. First consider binary alphabets. For $\sigma = 2$ each character of T requires only one bit, and we store the string in the natural way, packing w characters to one machine word. Now, by definition, $eq(1) = T$, and $eq(0) = \text{NOT}\ eq(1)$. Hence the preprocessing is not needed, and we can use T in the place of eq, see [10].

Consider now $\sigma = 4$. This has a very important application in DNA searching. In fact, this can be seen as the main application of the algorithm, the filter algorithms do not perform very well on small alphabets and for the relatively high error levels that are typical on DNA searching. On the other hand, the filters work quite well for large alphabets and small error levels typical for applications in ASCII natural language [7].

DNA alphabet requires only two bits of storage per symbol. We keep T in compressed form, and preprocess that representation. Let the codes for the four possible DNA characters A, C, G, T be as follows: A=0, C=1, G=2, T=3. These codes are in binary 00, 01, 10, 11, respectively (least significant bit *rightmost*). However, note that real DNA may have other characters too, namely IUB/IUPAC codes that are standard degeneracy symbols for nucleotide strings. This brings the alphabet size up to 15, which requires 4 bits.

Let T be a string of bits encoding a DNA sequence using the codes given above. We interpret T as an integer. E.g. $T =$ 00 10 01 11 00 10 10 01 encodes the string "CGGATCGA". The eq vectors are easy to compute from this compressed representation. We interpret T as an unsigned integer. Now, to compute which positions have character C, for example, we use bit-wise operations to check if the low bit is 1, and the high bit is 0. This is done by shifting the bits one upon the other and using bit-wise logic: $eq'(1) = \text{NOT}(T \gg 1)$ AND T. Similarly we obtain

$$\begin{aligned}
eq'(0) &= \text{NOT } (T \gg 1)\text{AND NOT } T = 01\ 00\ 00\ 00\ 01\ 00\ 00\ 00 \\
eq'(1) &= \text{NOT } (T \gg 1)\text{AND } \qquad T = 00\ 00\ 01\ 00\ 00\ 00\ 00\ 01 \\
eq'(2) &= \qquad (T \gg 1)\text{AND NOT } T = 00\ 01\ 00\ 00\ 00\ 01\ 01\ 00 \\
eq'(3) &= \qquad (T \gg 1)\text{AND } \qquad T = 00\ 00\ 00\ 01\ 00\ 00\ 00\ 00
\end{aligned}$$

The vectors eq' are 'fat' versions of eq, each bit in eq is padded with one zero bit in eq'. We convert eq' to eq with look-up tables with 2^q precomputed entries. Each machine word packs $w/\log_2(\sigma)$ symbols, there are σ entries in eq, and processing each entry takes $\log_2(\sigma)$ bit-wise operations; hence the preprocessing time is now $O(2^q + \sigma(\log_2^2(\sigma)\lceil n/w\rceil + \lceil n/q\rceil))$.

There is even more efficient way to preprocess the DNA alphabet. If the low and high bits of w consecutive characters are stored in two different words, then the zero padding problem and the need for the look-up tables disappear. Let integer T^l store the low bits, and T^h the high bits. Now eq can be computed as follows:

$$\begin{aligned}
eq(0) &= \text{NOT } T^h\text{AND NOT } T^l = 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \\
eq(1) &= \text{NOT } T^h\text{AND } \qquad T^l = 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1 \\
eq(2) &= \qquad T^h\text{AND NOT } T^l = 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\
eq(3) &= \qquad T^h\text{AND } \qquad T^l = 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0
\end{aligned}$$

The preprocessing cost is now $O(\sigma\log_2(\sigma)\lceil n/w\rceil)$, because we do not waste bits, and the look-up table is not needed. For the rest of the paper we asssume that this latter method is used. The compression is straight-forward, and the compressed file is reduced to only 25% of its original size.

The same approach for computing eq is possible for other alphabets, and without the compression, using just the bits of the corresponding ASCII codes, but the method is more complex and slower. Alg. 3 shows the code to preprocess T in the general case, where T is coded with $\lceil\log_2(\sigma)\rceil$ separate bit vectors. The algorithm runs in $O(\lceil n/w\rceil\,\sigma\log_2(\sigma))$ time.

The method is still useful for large alphabets, if we want to search several patterns (*dictionary matching* problem). In this case, the preprocessing of the text has to be done only once. It is also possible to rearrange the search code (with or without dictionary matching) such that only $O(\sigma)$ space is required for eq table. This is achieved by processing each block of w columns for all patterns before moving to the next block. In this method the preprocessing has to be merged with the search code.

5 The *k*-differences problem

Alg. 1 can be adapted for the k differences problem. That is, we allow only at most k edit operations. In this case the distance computations of Alg. 2 can be optimized. This can be done with look-up tables [8, 3]. Consider two bit vectors v and u, each of length q. The vector v represents the increments and u the

Alg. 3 Preprocess($T^1, ..., T^{\lceil \log_2(\sigma) \rceil}$). Preprocessing eq.

Input: $T^1, ..., T^{\lceil \log_2(\sigma) \rceil}$
Output: eq

```
for $i \leftarrow 1$ to $\sigma$ do
    $eq[i] \leftarrow \sim 0$
for $j \leftarrow 1$ to $\lceil \log_2(\sigma) \rceil$ do
    $b \leftarrow \sim 0$
    for $i \leftarrow 1$ to $\sigma$ do
        $eq[i] \leftarrow eq$ & $T^j$ ^ $b$
        $b \leftarrow \sim b$
return $eq$
```

decrements as in the representation of the matrix vectors. We precompute for each possible pair of values of v, u, the total increment S and the minimum value M. Let v_i denote the ith bit of v, then:

$$S(q)_{v,u} = \sum_{i=1}^{q} v_i - u_i, \quad M(q)_{v,u} = \min \left\{ \sum_{i=1}^{j} v_i - u_i \;\middle|\; 1 \le j \le q \right\}.$$

Preprocessing of $S(q)$ and $M(q)$ takes $O(q2^{2q})$ time, and they take $O(2^{2q})$ space[1]. Using S and M it is possible to compute the distances that are $\le k$ in $O(n/q+R)$ expected time, where R is the number of occurrences. Alg. 4 shows the code. The total time of Alg. 1 with the call to **BlockScore** (Alg. 2) substituted with a call to **BlockScore-k** (Alg. 4) takes now $O(\lceil n/w \rceil\, m+\lceil n/q \rceil+R)$ expected time. So we use such q that

$$\lceil n/q \rceil = O(\lceil n/w \rceil\, m).$$

This holds for $q = O(w/m)$, which shows that very small q is enough in practice for all but very small m. This requires additional $O(q2^{2q})$ time preprocessing. It does not dominate as far as $q2^{2q} = O(\lceil n/w \rceil\, m)$, that is, for $m = \Omega(w/\log_2(n))$. For very small m we can use $q = \varepsilon w/m$ for some constant $0 < \varepsilon < 1$ to reduce the space complexity while keeping the same time complexity. The total time can therefore be made $O(\lceil n/w \rceil\, (\sigma \log_2(\sigma)+m)+R)$ including the fast preprocessing.

This can be clearly improved. The algorithm in [12] runs in $O(nk)$ expected time, and its bit-parallel version in $O(n\, \lceil k/w \rceil)$ expected time [6]. We would like to obtain $O(\lceil n/w \rceil\, k)$ expected time algorithm.

The well-known method proposed in [12] works as follows. The algorithm computes column wise the values of $E_{i,j}$ only up to row $i \le \ell_j$, where $\ell_1 = k+1$

[1] In fact, only $O(3^q)$ space would suffice, as it is not possible that both v_i and u_i equal to 1. Albeit this would permit larger q in practice, it would also require (slow) hashing to index $S(q)$ and $M(q)$.

Alg. 4 BlockScore-k(d, pos, vp, vn). Computing distances $d \le k$.

Input: d, pos, vp, vn

Output: $E_{m,j} \mid E_{m,j} \le k$, $j \in \{pos+1..pos+w\}$

```
1   $h \leftarrow (1 \ll q) - 1$
2   for $i \leftarrow 1$ to $\lceil w/q \rceil$ do
3       $v \leftarrow vp$ & $h$
4       $u \leftarrow vn$ & $h$
5       if $d + M(q)_{v,u} \le k$ then
6           compute all $q$ distances wrt $v, u$
7           output possible occurrences
8       $d \leftarrow d + S(q)_{v,u}$
9       $vp \leftarrow vp \gg q$
10      $vn \leftarrow vn \gg q$
11  return $d$
```

(i.e. the pattern mismatches), and

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

The last relevant row of column j is therefore ℓ_j. This is because the search result does not depend on the elements of the matrix whose exact values are greater than $k+1$, and the last relevant row ℓ_{j+1} for the next column $j+1$ can be at most $\ell_{j+1} \le \ell_j + 1$.

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 is $O(k)$, and hence the whole algorithm takes only $O(nk)$ time.

The problem with this approach is that since we compute w columns of E in parallel, we would need the maximum of the values $\ell_{j..j+w-1}$, but we cannot compute this from $\ell_{j-w..j-1}$. Instead, we take the following approach.

Let ℓ'_j denote the last relevant row for a *block* j of w contiguous columns, i.e. the definition is

$$\ell'_j = \max\{\ell_h \mid (j-1)w \le h \le jw - 1\}.$$

Let $\ell'_1 = k$. We first compute bit-parrallely the Δ vectors for the current block j of columns up to row ℓ'_j. During this process the score s is updated explicitly only for the first column of the current block. We then evaluate the minimum distance $s + M(w)_{vp,vn}$ of the row ℓ'_j of the block j. If $s + M(w)_{vp,vn} \le k$ we continue to the next row of the current block, and keep computing the blocks and their minimum entries $s + M(w)_{vp,vn}$ until for some row i it happens that $s + M(w)_{vp,vn} > k$. At this point it is safe to stop evaluating the rows. If i was equal to m, then there is at least one occurrence for the current block. In this case the occurrences are checked using similar method as in Alg. 4 (by using

$S(q)$ and $M(q)$). The minimum value for a given block is evaluated similarly. After the current block is computed up to row i, we compute ℓ'_{j+1}. This can be done exactly as in the original algorithm. We decrease i as long as the score for the last column of the current block is $> k$. This is easy to implement using the last bits of the Δh vectors. The complete algorithm is given in Alg. 5, and Fig. 1 illustrates the computation.

In practice we have found that it is faster to just set $\ell'_{j+1} = i$, skipping the more elaborate computation of ℓ'_{j+1}. In this case we set $\ell'_1 = k+1$, compute the Δ vectors only up to row $\ell'_j - 1$, and correspondingly require that $s + M(w)_{vp,vn} \leq k + 1$. In effect, we can decrease ℓ' values only by one, that is $\ell'_{j+1} \geq \ell'_j - 1$. In both versions it is possible that $\ell'_{j+1} = \ell'_j + w$. This simpler version is in practice faster, because the more pessimistic ℓ'_{j+1} values computed are in fact more accurate in practice. Hence the complex `if-then-else` structure of the inner loop is usually executed only once or twice.

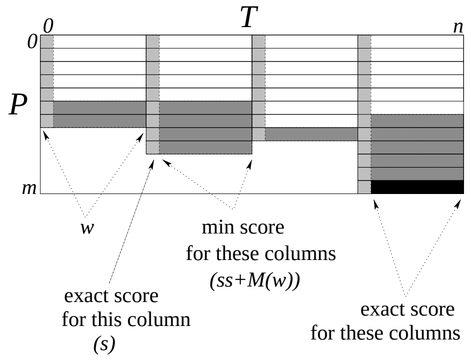

Fig. 1. Computation of the dynamic programming matrix with $w \times 1$ blocks. The bit positions shown in light gray correspond to the exact score accumulation (s) of line (21) in Alg. 5. The dark gray areas correspond to the area of minimum score computation of line (27). Finally, the black area shows the positions that need the exact score computed for score values $\leq k$ (line (35) in Alg. 5).

By using the same argument as in [12], we can see that updating ℓ takes still $O(\lceil n/w \rceil)$ total amortized time. Hence the expected running time of Alg. 5, including preprocessing, is $O(\lceil n/w \rceil\,(\sigma \log_2(\sigma) + L(k)) + \lceil n/q \rceil + R)$, where R is the number of occurrences. $L(k)$ denotes the expected value of the variable ℓ' for the simple version of updating ℓ'. The running time then becomes $O(\lceil n/w \rceil\,(\sigma \log_2(\sigma) + L(k)) + R)$ for $q = \varepsilon w / L(k)$.

For the accurate updating method and for $w = 1$ it is known that $L(k) = O(k)$ [2]. It is obvious that $L(k)$ must grow as w grows, but in Sec. 7 it is shown experimentally that at least up to $w = 128$ this growth is negligible compared to the increased parallelism. It seems clear that $L(k) = O(k + w)$, as $\ell_{j+1} \leq \ell_j + 1$,

Alg. 5 Approximate string matching allowing k errors.

Input: n, eq, P, m

Output: $E_{m,j} \;\Big|\; E_{m,j} \leq k,\ j \in \{1..n\}$

```
for $i \leftarrow 1$ to $m$ do
    $cp[i] \leftarrow 1$
    $cn[i] \leftarrow 0$
$\ell \leftarrow k$
$\ell' \leftarrow \ell$
for $r \leftarrow 1$ to $\lceil n/w \rceil$ do
    $vp \leftarrow 0,\ vn \leftarrow 0$
    $i \leftarrow 1$
    $s \leftarrow 0$
    $eq \leftarrow \textbf{Preprocess}(T^l[r], T^h[r])$
    do
        $x \leftarrow eq[P[i]] \mid cn[i]$
        $d \leftarrow ((vp + (x \,\&\, vp)) \wedge vp) \mid x \mid vn$
        $hp \leftarrow vn \mid \sim(vp \mid d)$
        $hn \leftarrow vp \,\&\, d$
        $x \leftarrow (hp \ll 1) \mid cp[i]$
        $vp \leftarrow (hn \ll 1) \mid cn[i] \mid \sim(x \mid d)$
        $vn \leftarrow x \,\&\, d$
        $cp[i] \leftarrow hp \gg (w - 1)$
        $cn[i] \leftarrow hn \gg (w - 1)$
        $s \leftarrow s + (hp \,\&\, 1) - (hn \,\&\, 1)$
        if $i < \ell$ then
            $i \leftarrow i + 1$
        else
            $ss \leftarrow s - (vp \,\&\, 1) + (vn \,\&\, 1)$
            if $i < m$ then
                if $ss + M(w)_{vp,vn} \leq k$ then
                    $i \leftarrow i + 1$
                    if $i > \ell'$ then
                        $cp[i] \leftarrow 1$
                        $cn[i] \leftarrow 0$
                else
                    break
            else
                $\textbf{BlockScore-k}(ss, (r-1)w, vp, vn)$
                break
    while $i \leq m$
    $\ell' \leftarrow i$
    $ss \leftarrow S(w)_{vp,vn}$
    while $ss > k$
        $ss \leftarrow ss - cp[i] + cn[i]$
        $i \leftarrow i - 1$
    $\ell \leftarrow i$
```

and therefore $\ell_{j+w} \leq \ell_j + w$. However, we conjecture that $L(k) = O(k + f(w))$, where $f(w) < w$.

As an implementation detail, note that the algorithm can be optimized somewhat, the line (21) can be removed, as the value of s is not needed for the first $O(L(k))$ iterations. This requires that the line (25) should be replaced with slightly more complex expression.

The preprocessing space is reduced to just $O(\sigma)$, as the preprocessing is embedded directly into Alg. 5. The *eq* values are needed only for the current block r, and the previous values can be discarded. This also improves the locality of reference, and can speed-up the algorithm in practice. We use this method in our implementation.

6 Vector instructions

Many recent processor have so called multimedia or vector instructions. These SIMD (single instruction, multiple data) instructions can be used to parallelize serial code. For example, the SSE2 instruction set found in Intel Pentium4 processors can work with 128 bits in single instruction, making it possible to compute sixteen 8 bit, eight 16 bit, four 32 bit, or two 64 bit (and for some instructions, one 128 bit) results in a single step[2]. Intel `icc` and GNU `gcc`[3] C/C++ compilers provide intrinsics (built-in functions that translate to one machine instruction) to work with 128 bit quantities. In C++ it is easy to overload standard arithmetic and bit-wise operators so that the translate to the intrinsics[4]. This allows to use exactly the same code for standard 32 bit operations and for the 128 bit operations, so that the programmer does not have to worry about the SSE2 implementation details. The only obstacle is the addition and shift instructions, which work at most in two 64 bit quantities, but this can be simulated with few 64 bits instructions. We have done just that, to provide the algorithm with $w = 128$.

7 Experimental results

We have implemented the algorithms in C/C++, compiled using `icc 7.0` with full optimizations. The experiments were run in 2GHz Pentium 4, with 512MB RAM, with Linux 2.4.

We compared the performance of the new algorithm against the original method BPM [6], ABNDM/BPA [10] (their implementation), and EXP [9] (their implementation). The implementation of ABNDM/BPA has hand optimized special versions for $k = 1..5$. Our implementation of BPM requires that $m \leq w$, and

[2] Other vector extensions are e.g. AMD's 3DNow!, DEC's MVI, Sun's VIS, Motorola's AltiVec, and MIPS's MDMX.

[3] For GCC (3.3, 3.4 prerelease) the intrinsics are currently broken, see `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=10984`.

[4] In fact, e.g. Intel provides a C++ class that does this, and comes with their C/C++ compiler.

hence is more efficient than the general method would be without this limitation. Our algorithm does not have such restriction. We experimented with Alg. 5 for $w = 32$, and $w = 128$. We used the simpler update formula for ℓ', see Sec. 5.

We experimented with randomly generated (uniform Bernoulli model) DNA of length 64Mb. The tests were run for various m and k. We also examined the performance for different w, namely for values 8, 16, 32, 64, and 128. The native word size of the machine is 32 bits, while the 64 bit data type (`unsigned long long`) is simulated by the compiler using 32 bit instructions. The 128 bit version uses the Intel SSE2 instructions, overloaded on the standard C/C++ arithmetic and bit-wise operators. The cases $w = 8$, $w = 16$ and $w = 64$ were included as a curiosity, and to study the effect of w in the average value of ℓ (the last row evaluated for a block of w columns) in Alg. 5. Fig. 2 shows the experimental values. In [1] the theoretical bound (for $w = 1$) $L \leq k/(1 - e/\sqrt{\sigma}) + O(1)$ (for $w = 1$) was proved. They also experimentally verified that the expression is very close $0.9k/(1 - 1.09/\sqrt{\sigma})$. We plot that curve in Fig. 2 too.

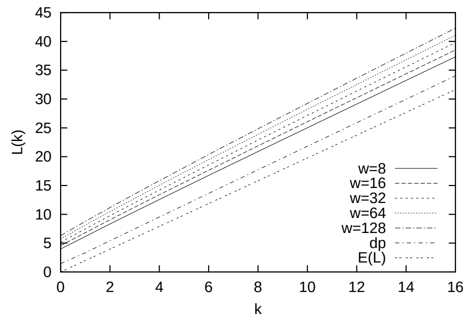

Fig. 2. The average value $L(k)$ of ℓ for different k. The pattern length was 64, and $\sigma = 4$. DP is for the $O(nk)$ expected time algorithm, and $E(L)$ denotes the predicted curve. The other curves are for our algorithm with different choices of w.

The timings are reported in Fig. 3. All the times include all preprocessing and file I/O. Our algorithms use compressed files and corresponding preprocessing method. The timings are reported for $q = 8$. Note that in the case of $m \leq 32$ we used the native 32 bit data type in implementing the Myers' algorithm. For the case $32 < m \leq 64$ we used the simulated 64 bit data type provided by the compiler. In this case the algorithm would probably be much faster if implemented using the (more complex) $O(n \lceil k/w \rceil)$ method. The implementation of ABNDM/BPA did not allow pattern lenghts of 32 or 64, and therefore they are not included in the experiments.

The results show that our new algorithm is very fast, in fact the fastest for moderate k. ABNDM/BPA could eventually become faster as the pattern length grows, as it is able to skip characters, but is only applicable for $m \leq w$.

However, the results are somewhat architecture dependent. We also compared our method against ABNDM/BPM [5], but it wasn't competitive. This was somewhat surprising, as they report excellent times in [5], but they run the experiments in different architecture (Alpha EV68). Our new algorithm is simple to implement, and has relatively few branches. The heavy pipelining of the current CPUs suffer from branching.

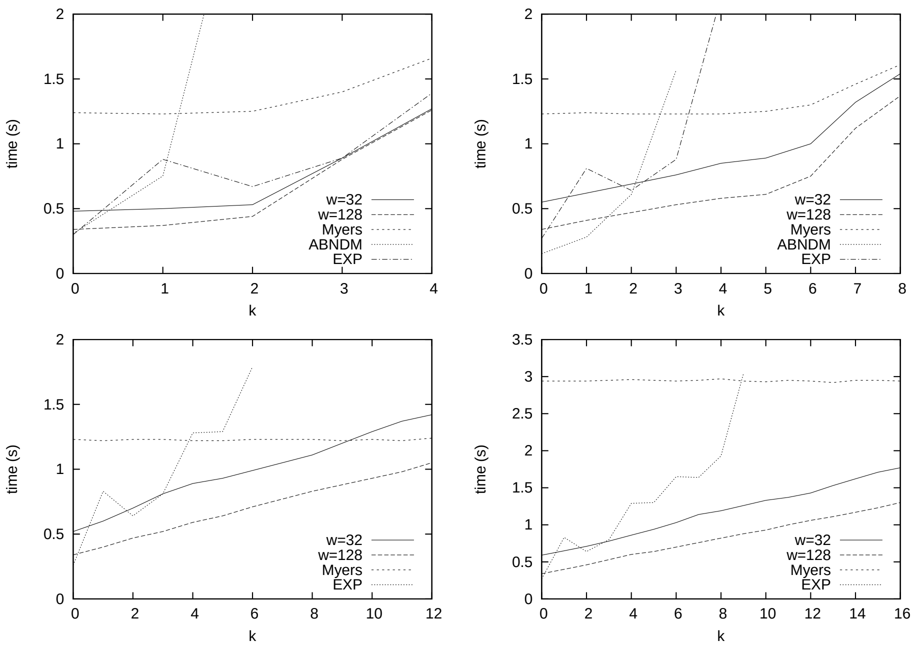

Fig. 3. Comparison of the methods for different k. From left to right, top to bottom, the pattern lengths are respectively 8, 16, 32, and 64. The times are in seconds.

In conclusion from the figures it is clear that the algorithm benefits much more on the increased parallelism than it loses for computing more of the dynamic programming matrix. The algorithm with $w = 128$ is not four times faster than with $w = 32$. There are several reasons for this. When w grows, larger portion of the matrix is evaluated. In both cases we used the same value for q (8, fixed in our current implementation), altough we should use larger q for larger w. Finally, the SIMD instructions are relatively slow, preventing the full expected speed-up. Although the SIMD instructions allow four times more bits (and larger register pool) than the integer instructions, they have also almost that much larger latency. This might improve in future.

8 Conclusions

The new arrangement of the bit-parallel computations of the dynamic programming matrix utilize the bits of computer word more economically, yielding faster approximate string matching algorithm for small patterns or for small number of allowed differences. The new algorithms work well in practice too.

References

1. R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
2. W. I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, number 664 in Lecture Notes in Computer Science, pages 175–184, Tucson, AZ, 1992. Springer-Verlag, Berlin.
3. K. Fredriksson and G. Navarro. Average-optimal multiple approximate string matching. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, LNCS 2676, pages 109–128, 2003.
4. H. Hyyrö. Extending and explaining the bit-parallel approximate string matching algorithm of Myers. Technical report A2001-10, Department of Computer and Information Sciences, University of Tampere, 2001.
5. H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 203–224, 2002.
6. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. Assoc. Comput. Mach.*, 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. Indexing text using the ziv-lempel trie. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 325–336. Springer, 2002.
9. G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.
10. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000. `http://www.jea.acm.org/2000/NavarroString`.
11. P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.
12. E. Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1–3):100–118, 1985.
13. A. H. Wright. Approximate string matching using within-word parallelism. *Softw. Pract. Exp.*, 24(4):337–362, 1994.