Average-Optimal Multiple Approximate String Matching

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

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

Abstract. We present a new algorithm for multiple approximate string matching, based on an extension of the optimal (on average) single-pattern approximate string matching algorithm of Chang and Marr. Our algorithm inherits the optimality and is also competitive in practice. We present a second algorithm that is linear time and handles higher difference ratios. We show experimentally that our algorithms are the fastest for intermediate difference ratios, an area where the only existing algorithms permitted simultaneous search for just a few patterns. Our algorithm is also resistant to the number of patterns, being effective for hundreds of patterns. Hence we fill an important gap in approximate string matching techniques, since no effective algorithms existed to search for many patterns with an intermediate difference ratio.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Given a text $T_{1...n}$, a pattern $P_{1...m}$, and a maximal number of differences permitted, k, we want to find all the text positions where the pattern matches the text up to k differences. The differences can be substituting, deleting or inserting a character. We call $\alpha = k/m$ the *difference ratio*, and σ the size of the alphabet Σ. For the average case analyses it is customary to assume a random text over a uniformly distributed alphabet.

A natural extension to the basic problem consists of *multipattern searching*, that is, searching for r patterns $P^1 \ldots P^r$ simultaneously in order to report all their occurrences with at most k differences. This has also several applications such as virus and intrusion detection, spelling applications, text retrieval under synonym or thesaurus expansion, several problems in computational biology, batch processing of single-pattern approximate searching, etc. Moreover, some single-pattern approximate search algorithms resort to multipattern searching of

* Work developed while the author was working in the Dept. of Computer Science, University of Helsinki. Supported by the Academy of Finland.
** Partially supported by Fondecyt grant 1-020831.

pattern pieces. Multidimensional search problems can also be reduced to string matching. Depending on the application, r may vary from a few to thousands of patterns. The naive approach is to perform r separate searches, so the goal is to do better.

The single-pattern problem has received a lot of attention since the sixties [8]. After the first dynamic-programming-based $O(mn)$ time solution to the problem [11], many faster techniques have been proposed, both for the worst and the average case. In 1994, Chang and Marr [3] showed that the average complexity of the problem is $O((k+\log_\sigma m)n/m)$, and gave an algorithm that achieved that average-optimal cost for $\alpha < 1/3 - O(1/\sqrt{\sigma})$.

The multipattern problem has received much less attention, not because of lack of interest but because of its difficulty. There exist algorithms that search permitting only $k = 1$ difference [6], and algorithms that handle either too few patterns or too low difference ratios [2].

Hence multiple approximate string matching is a rather undeveloped area. No algorithm exists when one searches for more than a few of patterns with intermediate difference ratios. Moreover, as the number of patterns grows, the difference ratios that can be handled get reduced.

The goal of this paper is to present an algorithm that is optimal on the average and that permits searching even for thousands of patterns with low and intermediate difference ratios, thus filling an important gap in the area. We build over an average-optimal algorithm that searches for single patterns [3] and inherit its optimality, obtaining $O(n(k + \log_\sigma(rm))/m)$ average search time. We show that the algorithm is not only theoretically appealing but also good in practice thanks to several practical improvements we introduce. Since the algorithm does not work for difference ratios beyond 1/3, we introduce a second, $O(n)$ average time variant that reaches ratios of 1/2. The algorithms are shown to be the fastest for a wide range of values of m, r and k, for small alphabets, see Sec. 6.

2 Related Work

2.1 Multiple Approximate String Matching

The naive approach to multipattern approximate searching is to perform r separate searches, one per pattern. If we use the optimal single-pattern algorithm [3], the average search time becomes $O((k + \log_\sigma m)rn/m)$ for the naive approach. On the other hand, if we use the classical $O(mn)$ algorithm [11] the time is $O(rmn)$.

Few algorithms exist for multipattern approximate searching under the k differences model. The first one, based on hashing, was presented by Muth and Manber [6]. It permits searching with $k = 1$ differences only, but is rather tolerant to the number of patterns r, which can reach the thousands without affecting much the cost of the search. The preprocessing time is $O(rm)$ and the average search time is $O(mn(1 + rm^2/M))$, where M is the size of the hash table. This adds up $O(rm + nm(1 + rm^2/M))$, which is $O(m(r+n))$ of $M = \Omega(m^2 r)$. This is basically independent of r if n is large enough.

Baeza-Yates and Navarro [2] have presented several algorithms for this problem. One of them, *partitioning into exact search*, uses the fact that, if P is cut into $k+1$ pieces, then at least one of the pieces appears inside every occurrence with no differences. Hence the algorithm splits every pattern into $k+1$ pieces and searches for the $r(k+1)$ pieces with an exact multipattern search algorithm. The preprocessing takes $O(rm)$ time. If they used an optimal multipattern exact search algorithm like MultiBDM [4], the search time would have been $O(k \log_\sigma(rm)n/m)$ on average. For practical reasons they used another algorithm, more suitable to searching for short pieces (of length $\lfloor m/(k+1) \rfloor$), albeit with worse theoretical complexity. This technique can be applied for $\alpha < 1/\log_\sigma(rm)$, a limit that gets more and more strict as m or r increase.

They also presented other algorithms that, although can handle higher difference ratios, are linear on r, which means that they give a speedup only up to a constant number c of patterns and then just divide the search into r/c groups that are searched for separately. *Superimposition* uses a standard search technique on a set of "superimposed" patterns, which means that the i-th character of the superimposition matches the i-th character of any of the superimposed patterns. Implemented over a newer bit-parallel algorithm [7], superimposition would yield average time $O(rn/(\sigma(1-\alpha)^2))$ for $\alpha < 1 - e\sqrt{r/\sigma}$ on patterns shorter than the number of bits in the computer word, w (typically $w = 32$ or 64). Different techniques are used to cope with longer patterns, but the times are worse. *Counting* extends a single-pattern algorithm that slides a window of length m over the text checking in linear time whether it shares at least $m - k$ characters with the pattern (regardless of the order). The multipattern version keeps several counters in a single computer word, achieving an average search time of $O(rn \log(m)/w)$ for $\alpha < e^{-m/\sigma}$.

2.2 The Algorithm of Chang and Marr

Chang and Marr [3] show that no approximate search algorithm for a single pattern can be faster than $O((k + \log_\sigma m)n/m)$ on the average. This is not hard to prove, and we give more details in Section 4.

In the same paper [3], Chang and Marr presented an algorithm achieving that optimal average time complexity. In the preprocessing phase they build a table D as follows. They choose a number ℓ in the range $1 \le \ell \le \lceil (m-k)/2 \rceil$, whose exact value we will consider shorty. For every string S of length ℓ (ℓ-gram), they search for S in P and store in $D[S]$ the smallest number of differences needed to match S inside P (this is a number between 0 and ℓ). Hence D requires space for σ^ℓ entries and is computed in $\sigma^\ell \ell m$ time. A numerical representation of Σ^ℓ permits constant time access to D.

The text scanning phase consists of logically dividing the text in blocks of length $b = \lceil (m-k)/2 \rceil$, which ensures that any approximate occurrence of P (which has length at least $m - k$) contains at least one whole block. Each block $T_{ib+1 \ldots ib+b}$ is processed as follows. They take the first ℓ-gram of the block,

$S^1 = T_{ib+1\ldots ib+\ell}$, and obtain $D[S^1]$. Then they take the next ℓ-gram, $S^2 = T_{ib+\ell+1\ldots ib+2\ell}$, and obtain $D[S^2]$, and so on. If, before reaching the end of the block, they have obtained $\sum_{1\le j\le t} D[S^j] > k$, then they can safely skip the block because no occurrence of P can contain the block, as merely matching those t ℓ-grams anywhere inside P requires more than k differences. If, on the other hand, they reach the end of the block without surpassing k total differences, the block must be checked. In order to check for $T_{ib+1\ldots ib+b}$ they run the classical dynamic programming algorithm over $T_{ib+1-m-k+b\ldots ib+m+k}$.

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

3 Our Algorithm

The basic idea of our algorithm is as follows. Given r search patterns $P^1 \ldots P^r$, we build the table D taking the minimum number of differences to match each ℓ-gram inside *any* of the patterns. The scanning phase is the same as in Section 2.2. If we surpass k differences inside a block we are sure that none of the patterns match, since there are t ℓ-grams inside the block that need more than k differences in order to be found inside any pattern. Otherwise, we check the patterns one by one over the block. Figure 1 gives the code. We present now several improvements over this basic idea.

```
**Search** ($T_{1\ldots n}$, $P^1_{1\ldots m} \ldots P^r_{1\ldots m}$, $k$)

1.    $\ell \leftarrow$ **Preprocess** ( )
2.    $b \leftarrow \lceil (m-k)/2 \rceil$
3.    **For** $i \in 0 \ldots \lfloor n/b \rfloor - 1$ **Do**
4.        **VerifyBlock** ( $i$, $b$ )
```

Fig. 1. High-level description of the algorithm. The input parameters are taken as global variables in the rest of the paper, to simplify the descriptions.

3.1 Optimal Choice of ℓ-grams

The basic single-pattern algorithm [3] uses the first consecutive ℓ-grams of the block in order to find more than k differences. This is simple, but not necessarily the best choice. Note that any set of non-overlapping ℓ-grams found inside the block whose total number of differences inside P exceeds k permits us discarding the block. Hence the question of using the best possible set is raised.

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

This is solved by calling M_u the maximum sum that can be obtained using ℓ-grams that start in the positions $ib+1\dots ib+u$. Initially we start with $M_u = 0$ for $-\ell < u \le 0$. Then we traverse the block computing, for increasing u values,

$$M_u \quad \leftarrow \quad \max(D[T_{ib+u\dots ib+u+\ell-1}] + M_{u-\ell}\ ,\ M_{u-1}) \qquad (1)$$

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

We compute M_u for increasing u until either (i) $M_u > k$, in which case we abandon the block, or (ii) $u > b - \ell + 1$, in which case we have to verify the block. Figure 2 gives the code.

```
CanDiscard ($i$, $b$, $D$, $\ell$)

1.    For $u \in -\ell \dots 0$ Do $M_u \leftarrow 0$
2.    For $u \in 1 \dots b-\ell+1$ Do
3.        $M_u \leftarrow \max(D[T_{ib+u\dots ib+u+\ell-1}] + M_{u-\ell}\ ,\ M_{u-1})$
4.        If $M_u > k$ Then Return TRUE
5.    Return FALSE
```

Fig. 2. Optimization technique to choose the set of overlapping ℓ-grams that maximize the sum of differences. It returns whether the block can be discarded.

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

3.2 Hierarchical Verification

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

The hierarchy is used as follows. For every internal node $[i, j]$ we have a table D computed using the minimum distances between ℓ-grams and patterns $P^i \dots P^j$. This is done by computing first the leaves (that is, each pattern separately) and then computing every cell of D in the internal node as the minimum over the corresponding cell in its two children. In order to scan the text, we use the D table of the root node, which corresponds to the full set of patterns. Every time a block has to be verified with respect to a node in the hierarchy (at first, the root node), we rescan the block considering the two children of the current node. It is possible that the block can be discarded for both children, for one, or for none. We recursively repeat the process for every child that does not permit discarding the block, see Fig. 3. If we process a leaf node and still have to verify the block, then we run dynamic programming over the corresponding single pattern.

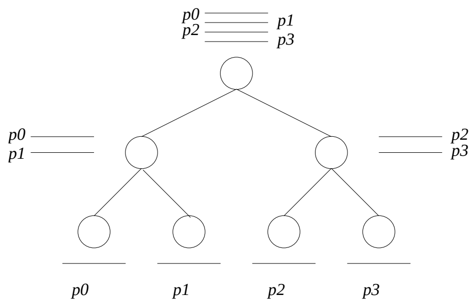

Fig. 3. Pattern hierarchy for 4 patterns.

The idea of using the hierarchy instead of plainly checking the r patterns one by one is that it is possible that the grouping of the pattern matches a block, but that none of its halves match. In this case we save verification time. The plain technique needs $O(\sigma^\ell)$ space, while hierarchical verification needs much more, $O(r\sigma^\ell)$.

Note that verification would benefit if the patterns we group together are as similar as possible, in terms of numbers of differences. A simple heuristic is to lexicographically sort the patterns before grouping them by ranges.

As a final note, we use Myers' algorithm [7] for the verification of single patterns, which makes the cost $O(m^2/w)$, where w is the number of bits in the computer word.

Figures 4 and 5 show the preprocessing and verification using hierarchical verification.

3.3 Reducing Preprocessing Time

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

```
HierarchyPreprocess ($i$, $j$, $\ell$)

1.    If $i = j$ Then $D_{i,j} \leftarrow$ PreprocessD($P^i, \ell$)
2.    Else
3.        $m \leftarrow \lfloor (i+j)/2 \rfloor$
4.        HierarchyPreprocess ($i, m, \ell$)
5.        HierarchyPreprocess ($m+1, j, \ell$)
6.        For $s \in \Sigma^\ell$ Do
7.            $D_{i,j}[s] \leftarrow \min(D_{i,m}[s], D_{m+1,j}[s])$

Preprocess ( )

8.    $\ell \leftarrow \lceil \frac{3 \log_\sigma m + \log_\sigma r}{1-c+2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)} \rceil$ // see Eq. (3)
9.    HierarchyPreprocess($1, r, \ell$)
```

Fig. 4. Preprocessing to build the hierarchy. It is initially invoked with parameters $(1, r)$ and produces global tables $D_{i,j}$ to be used by **HierarchyVerify**. The main search table is $D_{1,r}$.

```
HierarchyVerify ($i$, $j$, $b$, $s$)

1.    If NOT CanDiscard ($s$, $b$, $D_{i,j}$, $\ell$) Then
2.        If $i = j$ Then Search for $P^i$ in $T_{sb+1-m-k+b \ldots sb+m+k}$
3.        Else
4.            $m \leftarrow \lfloor (i+j)/2 \rfloor$
5.            HierarchyVerify ($i, m$)
6.            HierarchyVerify ($m+1, j$)
```

Fig. 5. Hierarchical verification. Procedure **VerifyBlock**(i, b) is then defined as **HierarchyVerify** $(1, r, b, i)$.

cessing time. In the case of hierarchical verification we pay an additional $O(r\sigma^\ell)$ time to create the D tables of the internal nodes, but this is negligible compared to the cost to compute the individual patterns.

In order to find the minimum number of differences to match an ℓ-gram S inside a pattern P, we compute the matrix $C_{i,j}$, for $0 \leq i \leq \ell$ and $0 \leq j \leq m$, as follows [11]:

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

which can be computed, for example, row-wise left to right. We need only the previous row in order to compute the current row. The minimum distance is finally $\min_{0 \leq j \leq m} C_{\ell,j}$.

We present now a method to reduce the preprocessing time to $O(rm\sigma^\ell)$, which has been used before in the context of indexed approximate string matching [10]. Instead of running the ℓ-grams one by one over a pattern P, we form a trie data structure of all the ℓ-grams. For every trie node whose path from the root spells out the string S, we compute the last row of the C matrix corresponding to searching for S inside P. For this sake we use the previous matrix row, which was computed for the parent node. Hence, if we traverse the trie using a classical depth first search recursion and compute a new matrix row at each invocation, then the execution stack contains the matrix computed up to now, so we use the row computed at the invoking process to compute the row of the invoked process. Since we work $O(m)$ at every trie node and there are $O(\sigma^\ell)$ nodes, the overall process takes $O(m\sigma^\ell)$ time. It needs just space for the stack, $O(m\ell)$. By repeating this over each pattern we obtain $O(rm\sigma^\ell)$ time.

Note finally that the trie of ℓ-grams does not need to be explicitly built, as we know that we have every possible ℓ-gram and hence can use an implicit method to traverse all them without actually storing them. Only the minima over the final rows are stored into the corresponding D entries. Figure 6 shows the code.

```
RecPreprocessD ($P$, $i$, $\ell$, $S$, $Cold$, $D$)

1.   If $i = \ell$ Then $D[S] \leftarrow \min_{0 \le j \le m} Cold_j$
2.   Else
3.       For $s \in \Sigma$ Do
4.           $Cnew_0 \leftarrow i$
5.           For $j \in 1 \ldots m$ Do
6.               If $s = P_j$ Then $Cnew_j \leftarrow Cold_{j-1}$
7.               Else $Cnew_j \leftarrow 1 + \min(Cold_{j-1}, Cold_j, Cnew_{j-1})$
8.           RecProcessD ($P, i+1, \ell, Ss, Cnew, D$)

PreprocessD ($P$, $\ell$)

9.   For $j \in 0 \ldots m$ Do $C_j \leftarrow 0$
10.  RecPreprocessD ($P, 0, \ell, \varepsilon, C, D$)
11.  Return $D$
```

Fig. 6. Preprocessing for a single table.

Again, we use Myers' algorithm [7] to compute the matrix rows, which makes the preprocessing time $O(rm\sigma^\ell/w)$. For this sake we need to modify the algorithm so that it takes the ℓ-gram as the text and P^i as the pattern. This means that the matrix is transposed, so the current "column" starts with zeros and at the i-th step its first cell has the value i. The necessary modifications are simple and are described, for example, in [5].

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

3.4 Packing Counters

Our final optimization resorts to bit-parallelism, that is, to storing several values inside the same computer word (this has been also used, for example, in the counting algorithm [2]). For this sake we will denote the bitwise *and* operation as "&", the *or* as "|", and the bit complementation as "∼". Shifting i positions to the left (right) is represented as "$<< i$" ("$>> i$"), where the bits that fall are discarded and the new bits that enter are zero. We can also perform arithmetic operations over the computer words. We use exponentiation to denote bit repetition, e.g. $0^3 1 = 0001$, and write the most significant bit as the leftmost bit.

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

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

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

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

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

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

We have also to consider how this is combined with the optimization algorithm of Section 3.1, since the best choice to maximize one counter may not be the best choice to maximize another. The solution is to pack also the different values of M_u in a single computer word. The operation of Eq. (1) can be perfectly done in parallel for several counters, as long as we replace the sum by the above technique to avoid overflows. The only obstacle is the maximum, which as far as we know has never been used in a bit-parallel scenario. We do that now.

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

Fig. 7 shows the bit-parallel version of the counter accummulation, and Fig. 8 shows an example of pattern hierarhy.

4 Analysis

We analyze our algorithm by following the analysis of the corresponding single pattern algorithm [3]. Two useful lemmas shown there follow (we have written them in a way more convenient for us).

Lemma 1 [3] The probability that two random ℓ-grams have a common subsequence of length $(1-c)\ell$ is at most $a\sigma^{-d\ell}/\ell$, for constants $a = (1+o(1))/(2\pi c(1-$

```
CanDiscard (i, b, D, ℓ)

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

Fig. 7. The bit-parallel version of **CanDiscard**. It requires that D is preprocessed by packing the values of A different patterns in the same way. Lines 1–6 can in fact be done once at preprocessing time.

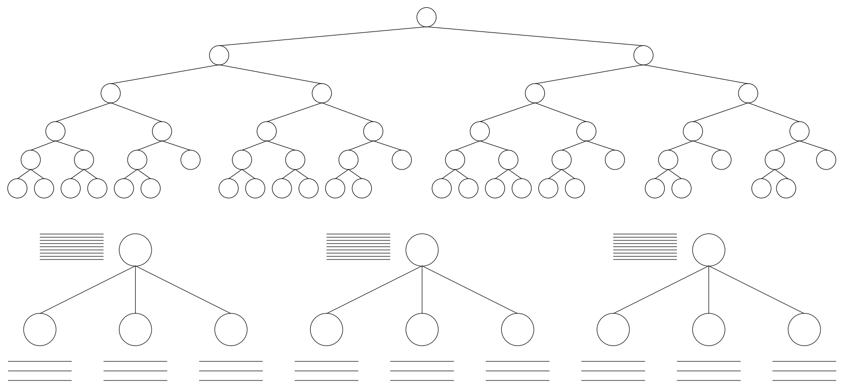

Fig. 8. Top: basic pattern hierarchy for 27 patterns. Bottom: pattern hierarchy with bit-parallel counters (27 patterns).

c)) and $d = 1 - c + 2c \log_\sigma c + 2(1-c)\log_\sigma(1-c)$. The probability decreases exponentially for $d > 0$, which surely holds if $c < 1 - e/\sqrt{\sigma}$.

Lemma 2 [3] If S is an ℓ-gram that matches inside a given string P (larger than ℓ) with less than $c\ell$ differences, then S has a common subsequence of length $\ell - c\ell$ with some ℓ-gram of P.

We measure the amount of work in terms of inspected characters. For a given text block, if there is a single ℓ-gram inside the block that matches inside any pattern P^i with less than $c\ell$ differences, we pessimistically assume that we verify the whole block. Otherwise, after considering $1+\lceil k/(c\ell)\rceil$ non-overlapping ℓ-grams, we abandon the block without verifying it. For the latter to be correct, it must hold $k/m = \alpha < c/(c+2)(1+O(1/m))$, since otherwise we reach the end of the block (of length $(m-k)/2$) before considering those $1+\lceil k/(c\ell)\rceil$ ℓ-grams.

Given Lemmas 1 and 2, the probability that a given ℓ-gram matches with less than $c\ell$ differences inside some P^i is at most that of having a common subsequence of length $\ell - c\ell$ with some ℓ-gram of some P^i. The probability of this is $mra\sigma^{-d\ell}/\ell$. Consequently, the probability that any ℓ-gram in the current text block matches is $m^2ra\sigma^{-d\ell}/\ell^2$, since there are m/ℓ ℓ-grams. (We assume for the analysis that we do not use the optimization of Section 3.1; this is pessimistic for every possible text block.)

Hence, with probability $m^2ra\sigma^{-d\ell}/\ell^2$ we verify the block, and otherwise we do not. In the first case we pay $O(m^2r)$ time if we use plain verification (Section 3.2, we see the case of hierarhical verification later) and dynamic programming. In the second case we pay the number of characters inspected in order to process $1+\lceil k/(c\ell)\rceil$ ℓ-grams, that is, $\ell + k/c$. Hence the average cost is upper bounded by

$$O\left(\frac{n}{m}\ \left(\frac{am^4r^2}{\ell^2}\sigma^{-d\ell}\ +\ \ell+\frac{k}{c}\right)\right)$$

The first part is the cost of verifications, and we have to make it negligible compared to the second part, that is, we have to ensure that verifications are rare enough. A sufficient condition on ℓ is

$$\ell\ \geq\ \frac{4\log_\sigma m + 2\log_\sigma r}{d}\ =\ \frac{4\log_\sigma m + 2\log_\sigma r}{1-c+2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)}$$

(in fact a slightly better, but more complicated, bound can be derived).

Note that we are free to choose any constant $2\alpha/(1-\alpha) < c < 1-e/\sqrt{\sigma}$. If we let c approach $1-e/\sqrt{\sigma}$, the value of ℓ goes to infinity and so does our preprocessing cost. If we let c approach $2\alpha/(1-\alpha)$, ℓ gets as small as possible but our search cost becomes $O(n)$. Having properly chosen c and ℓ, our algorithm is on average

$$O\left(\frac{n(k+\log_\sigma(rm))}{m}\right) \qquad (2)$$

character inspections. We remark that this is true as long as $2\alpha/(1-\alpha) < 1-e/\sqrt{\sigma}$, that is, $\alpha < 1/3 - O(1/\sqrt{\sigma})$, as otherwise the whole algorithm reduces to dynamic programming.

Recall that our preprocessing cost is $O(mr\sigma^\ell/w)$. Given the value of ℓ, this is $O(m^5r^3\sigma^{O(1)}/w)$. The space with plain verification is $\sigma^\ell = m^4r^2\sigma^{O(1)}$ integers.

As a practical consideration, we have that since σ^ℓ must fit in memory, we must be able to hold $\ell\log_2\sigma$ bits in a single computer word, so we can read a whole ℓ-gram in a single computer instruction. The number of instructions executed then becomes $O(n(1+k/\log M)/m)$, where M is the amount of memory we spend on a D table. Note that this is not true if we use the optimization method of Section 3.1, although we are not able to analyze the benefit that this method produces, on the other hand.

The fact that we perform the verification using Myers' algorithm [7] changes its cost to $O(rm^2/w)$, and this permits reducing ℓ a bit in practice, but the overall complexity does not change.

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

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

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

$$O\left(\frac{n}{m}\ \left(\frac{am^3r(m+\log r)}{\ell^2}\sigma^{-d\ell}\ +\ \ell + \frac{k}{c}\right)\right)$$

which is clearly better than the cost with plain verification. The condition on ℓ to obtain the same search time of Eq. (2) is now

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

which is smaller and hence requires less preprocessing effort. This time the preprocessing cost is $O(m^4r^2(m+\log r)\sigma^{O(1)}/w)$, smaller than with plain verification. The space requirement of hierarchical verification, however, is $2r\sigma^\ell = 2m^3r^2(m+\log_2 r)\sigma^{O(1)}$, which is larger than with plain verification.

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

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

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

To summarize, we have shown that we are able to perform, on average, $O(n(k + \log_\sigma(rm))/m)$ character inspections whenever $\alpha < 1 - e/\sqrt{\sigma}$. This requires a preprocessing time of roughly $O(m^4 r^2 (m + \log_{w/\log k} r) \sigma^{O(1)}/w)$ and an extra space of $O(m^3 r^2 (m + \log_{w/\log k} r) \sigma^{O(1)} \log(k)/w)$ by using the best techniques. The number of machine instructions for the search can be made $O(n(1 + k/\log M)/m)$ provided we use M memory for a single D table.

It has been shown that, for a single pattern, $O(n(k + \log_\sigma m)/m)$ is optimal [3]. This uses two facts. The first is that it is necessary to inspect at least $k+1$ characters in order to skip a given text window of length m, so we need at least $\Omega(kn/m)$ character inspections. The second is that the $\Omega(n \log_\sigma(m)/m)$ lower bound of Yao [14] for exact string mathcing applies to approximate searching too, as exact searching is included in the approximate search problem. When searching for r patterns, this lower bound becomes $\Omega(n \log_\sigma(rm)/m)$, as we show in the Appendix A. Hence our algorithm is optimal.

5 A Slower Algorithm for Higher Difference Ratios

A weakness of the algorithm is that it cannot cope with difference ratios beyond 1/3. This is due in part to the use of text blocks of length $(m-k)/2$. A different alternative to fixed-position blocks is the use of a sliding window of t ℓ-grams, where $t = \lfloor (m-k+1)/\ell \rfloor - 1$. If we consider text blocks for the form $T_{i\ell+1 \ldots i\ell+t\ell}$, we are sure that every occurrence (whose minimum length is $m-k$) contains a complete block. Then, if the ℓ-grams inside the window add up more than k differences, we can move to the next block.

The main difference is that blocks overlap with each other by $t-1$ ℓ-grams, so we should be able to update our difference counter from one text block to the next in constant time. This is rather easy, although it does not permit anymore the use of the optimization technique of Section 3.1. The result is an algorithm that takes $O(n)$ time for $\alpha < 1/2 - O(1/\sqrt{\sigma})$. Figure 9 shows this algorithm.

6 Experimental Results

We have implemented the algorithms in C, compiled using `gcc 3.2.1` with full optimizations. The experiments were run in 2GHz Pentium 4, with 512MB RAM, with Linux 2.4.

We ran experiments for alphabet sizes $\sigma = 4$ (DNA), $\sigma = 20$ (protein) and $\sigma = 256$ (ASCII text). The test data for DNA and protein alphabets was randomly generated. The texts were 64MB characters for DNA, and 16MB characters for protein, and the patterns were 64 characters. The texts were stored used only 2 (DNA) and 5 bits (protein) per character, which allowed $O(1)$ time access to the ℓ-grams.

```
**Search** $(T_{1\ldots n},\ P^1_{1\ldots m} \ldots P^r_{1\ldots m},\ k)$

1.   $\ell \leftarrow$ **Preprocess** ( )
2.   $t \leftarrow \lfloor (m-k+1)/\ell \rfloor - 1$
3.   $b \leftarrow t\ell$
4.   $M \leftarrow 0$
5.   **For** $i \in 0 \ldots t-2$ **Do** $M \leftarrow M + D[T_{i\ell+1 \ldots i\ell+\ell}]$
6.   **For** $i \in t-1 \ldots \lfloor n/\ell \rfloor$-1 **Do**
7.       $M \leftarrow M + D[T_{i\ell+1 \ldots i\ell+\ell}]$
8.       **If** $M \leq k$ **Then** Verify text block
9.       $M \leftarrow M - D[T_{(i-t+1)\ell+1 \ldots (i-t+1)\ell+\ell}]$
```

Fig. 9. High-level description of the slower algorithm. The verification of a text block can also be done hierarchically.

Table 1. Preprocessing times in seconds for various number of patterns, and for various ℓ-gram lenghts. The pattern lenghts are $m = 64$ for DNA and protein, and $m = 16$ for ASCII.

DNA	1	8	32	64	protein	1	64	256	1024	ASCII	1	64	256	1024
4	0.00	0.00	0.00	0.00	1	0.00	0.01	0.01	0.02	1	0.00	0.01	0.01	0.06
6	0.01	0.01	0.04	0.08	2	0.00	0.01	0.03	0.07	2	0.01	0.59	2.60	10.65
8	0.02	0.15	0.58	1.17	3	0.01	0.09	0.40	1.55	3	4.26			
10	0.38	3.02	12.00	24.19	4	0.04	6.09							

Table 1 gives the preprocessing times for various alphabets, number of patterns and ℓ-grams. The preprocessing timings are for the basic algorithms, without the bit-parallel counters technique, which requires slightly more time. The maximum values in practice are $\ell \leq 8$ for DNA, $\ell \leq 3$ for protein, and $\ell \leq 2$ for ASCII. The search times were measured for these maximum values.

Figs. 10, 11, and 12 give the search times for the DNA, protein, and ASCII alphabets. The abreaviations in the figures are as follows. SL: the basic sublinear time algorithm, SLO: SL with the optimal choice of ℓ-grams, L: the basic linear time filtering algorithm, SLC: SL with bit-parallel counters, SLCO: SLC with the optimal choice of ℓ-grams, LC: L with bit-parallel counters. All filters use the hierarchical verification. For comparison, Fig. 13 gives timings for the exact pattern partitioning algorithm given in [1]. This algorithm beats the new algorithms for large rm, σ, k. Fig. 14 illustrates.

Optimal choice of ℓ-grams helps only sometimes, but is usually slower due to its complexity. The linear time filtering algorithms quickly become faster than the sublinear algorithms for large rm, k. The bit-parallel counters speed-up the search for large rm. The performance of the algorithms collapse when the error ratio grows past a certain limit, and this collapse is very sharp. Before that limit, the new algorithms are very efficient.

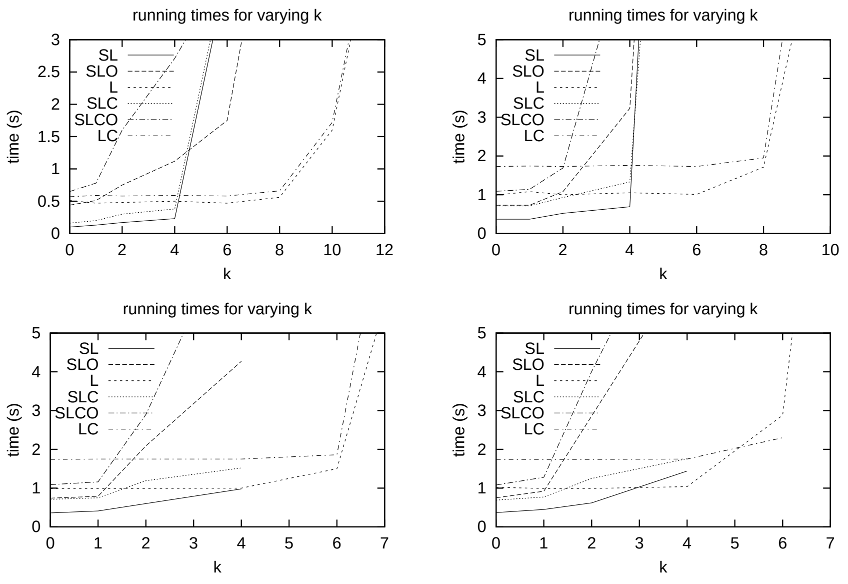

Fig. 10. Search times in seconds. Parameters are: $\sigma = 4$, $m = 64$, and $\ell = 8$. The figures are for, from left to right, top to bottom: $r = 1$, $r = 8$, $r = 32$, and $r = 64$.

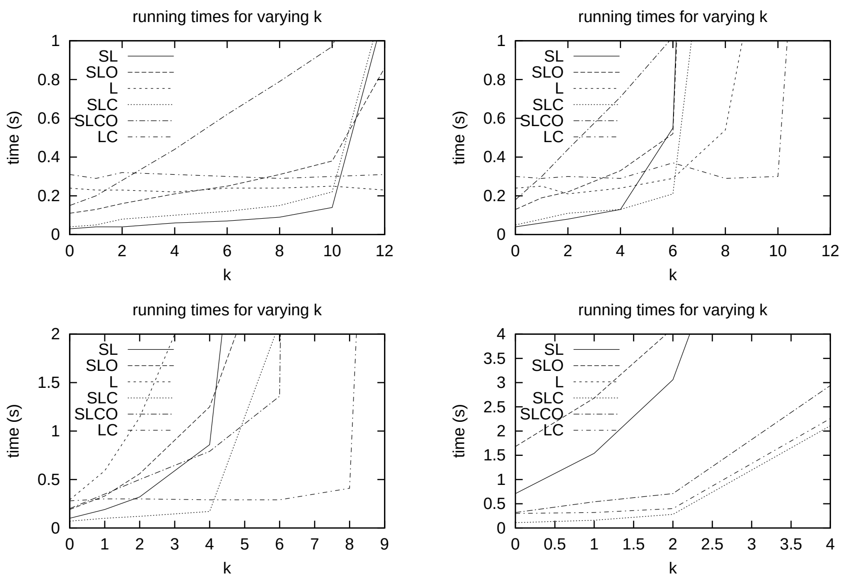

Fig. 11. Search times in seconds. Parameters are: $\sigma = 20$, $m = 64$, and $\ell = 3$. The figures are for, from left to right, top to bottom: $r = 1$, $r = 64$, $r = 256$, and $r = 1024$.

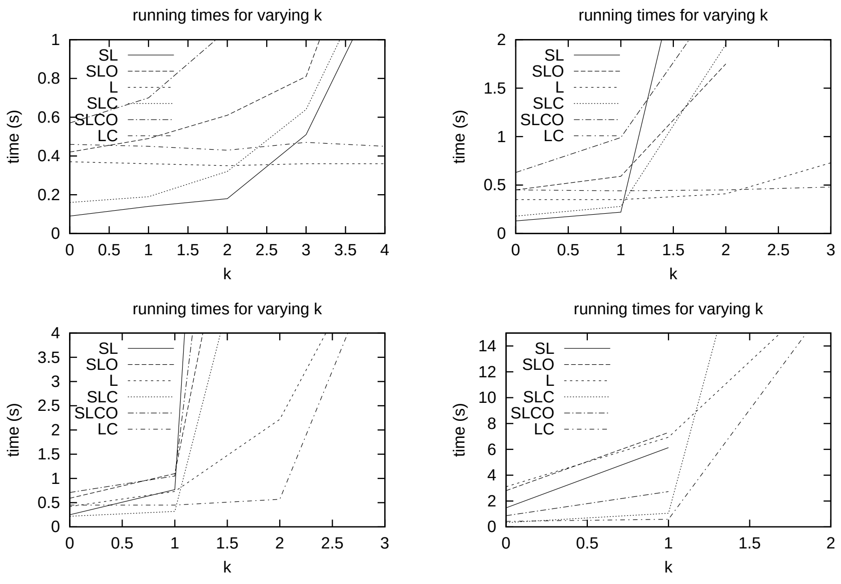

Fig. 12. Search times in seconds. Parameters are: $\sigma = 64$ (ASCII), $m = 64$, and $\ell = 3$. The figures are for, from left to right, top to bottom: $r = 1$, $r = 64$, $r = 256$, and $r = 1024$.

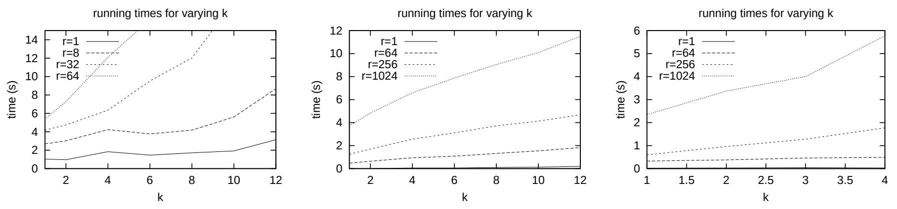

Fig. 13. Search times in seconds for exact pattern partitioning algorithm, for $\sigma = 4$ (DNA), $\sigma = 20$ (protein), and $\sigma = 64$ (ASCII) alphabets.

7 Conclusions

Multiple approximate string matching is an important problem that arises in several applications, and for which the current state of the art is in a very

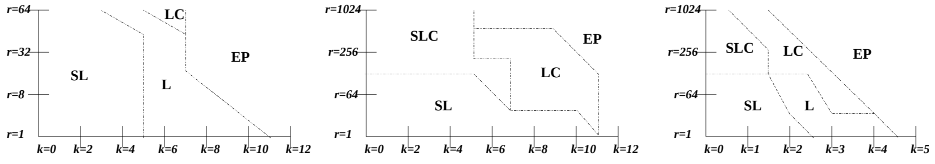

Fig. 14. Areas where each algorithm performs best.

primitive stage. Nontrivial solutions exist only for the case of very low difference ratios or very few patterns.

We have presented a new algorithm to improve this situation. Our algorithm is not only optimal on average, but also practical. A second algorithm we present is slower but handles higher difference ratios. We have shown that they perform well in handling large numbers of patterns and intermediate difference ratios. They are indeed the best alternatives for reasonably small alphabets.

The algorithms do not induce any order on the ℓ-grams, but they can appear in any order, as long as their total distance is at most k. The filtering can be still improved by requiring that the ℓ-grams from the pattern must appear in approximately same order in the text. This approach was used in [12]. The same method can be applied for multiple patterns as well.

There are several ways we plan to try in order to reduce preprocessing time and memory usage. A first one is lazy evaluations of the table cells. Instead of fully computing the D tables of size σ^ℓ for each pattern, we compute the cells only for the text ℓ-grams as they appear. If a given table cell is not yet computed, we compute it on the fly for all the patterns. This gives a preprocessing cost that is $O(rm\sigma^\ell(1-e^{-n/\sigma^\ell}))$ on the average (using Myers' algorithm for the ℓ-grams inside the patterns, as $\lceil \ell/w \rceil = 1$). This, however, is advantageous only for very long ℓ-grams, namely $\ell + \Theta(\log\log\ell) > \log_\sigma n$.

Another possibility is to compute D only for those ℓ-grams that appear in a pattern with at most ℓ' differences, and assume that all the others appear with $\ell'+1$ differences. This reduces the effectivity at search time but, by storing the relevant ℓ-grams in a hash table, requires $O(rm(\sigma\ell)^{\ell'})$ space and preprocessing time (either for plain or hierarchical verification), since the number of strings at distance ℓ' to an ℓ-gram is $O((\sigma\ell)^{\ell'})$ [13]. With respect to plain verification, the space is reduced for $\ell' < (\ell - \log_\sigma(rm))/(1+\log_\sigma \ell)$, and with respect to hierarchical verification, for $\ell' < (\ell - \log_\sigma m)/(1+\log_\sigma \ell)$. These values are reasonable.

It is also possible to improve the verification performance. A simple strategy is to sort the patterns before grouping by ranges in order to achieve some clustering in the groups. This could be handled with an algorithm designed for hierarchical clustering. This clustering could be done taking a distance defined as the number of differences necessary to convert one pattern into the other, or any other reasonable measure of similarity (Hamming distance, longest common subsequence, etc.).

Indexing consists of preprocessing the text to build a data structure (index) on it that can be used later for faster querying [9]. In general, we find that methods designed for indexed approximate string matching can be adapted to (non-indexed) multiple approximate string matching. The idea is to index the pattern set and use the text somehow as the pattern, in order to "search for the text" inside the structure of the patterns. Our present ideas are close to approximate q-gram methods, and several other techiques can be adapted too. We are currently pursuing this line.

References

1. R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In F. Dehne et al., editor, *Proceedings of the 5th Annual Workshop on Algorithms and Data Structures (WADS'97)*, pages 174–184, 1997.
2. R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. *Random Structures and Algorithms (RSA)*, 20:23–49, 2002.
3. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 259–273, 1994.
4. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.
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. R. Muth and U. Manber. Approximate multiple string search. In *Proc. 7th Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 75–86, 1996.
7. E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
8. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
9. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001. Special issue on Managing Text Natively and in DBMSs.
10. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. 11th Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 350–363, 2000.
11. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.
12. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 50–63, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.
13. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
14. A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal of Computing*, 8(3):368–387, 1979.

A Lower Bound for Multipattern Matching

We extend the classical proof of Yao [14] to the case of searching for several patterns. Let us assume that we search for r random patterns of length m in a text of length n. Random patterns means that they are independently generated sequences where each character is chosen from the set Σ with uniform probability. To simplify matters, we will not assume that the patterns are necessarily different from each other, they are simply r randomly generated sequences. We prove that the lower bound of the problem on the average is $\Omega(n \log_\sigma(rm)/m)$, where $\sigma = |\Sigma|$. The bound refers to the number of character inspections made.

We use the same trick of dividing the text in blocks of length $2m-1$, and assume that we just have to search for the presence of the patterns inside each block (which is an optimistic assumption). Since no information gathered inside one block can be used to search the other, we can regard each block in isolation. So the cost is at least $n/(2m-1)$ times the cost to search a single block. Hence what we have to prove is that we have to work $\Omega(\log_\sigma(rm))$ inside a given block.

Inside a given block $B_{1\ldots 2m-1}$, each of the r patterns can match in m different positions (starting at position 1 to m). Each possible match position of each pattern will be called a *candidate* and identified by the pair (t,i), where $t \in 1\ldots r$ is the pattern number and $i \in 1\ldots m$ is the starting position inside the block. Hence there are rm candidates.

We have to examine enough characters to ensure that we have found every match inside the block. We will perform a sequence of accesses (block character reads) inside the block, at positions $i_1,\ i_2\ldots i_k$ until the information we have gathered is enough to know that we found every pattern occurrence. Which is the same, we have to "rule out" all the rm candidates, or report those candidates that have not been ruled out after considering their m positions.

Note that each candidate has to be ruled out independently of the rest. Moreover, the only way to rule out a candidate (t,i) is to perform an access i_j such that $B_{i_j} \neq P^t_{i_j-i+1}$.

Given an access i_j to block B, the probability to rule out a candidate (t,i) with the access is at most $1-1/\sigma$: even assuming that the area covered by the candidate includes i_j (that is, $i \leq i_j < i+m$) and that the candidate has not been already outruled by a previous access, there is a probability of $1/\sigma$ that $B_{i_j} = P^t_{i_j-i+1}$ and hence we cannot rule out (t,i). This means that the probability that a given access does not rule out a given candidate is $\geq 1/\sigma$. Note that the way we have bounded the probability permits us considering every access independently of the others. Consequently, the probability of *not* ruling out a given candidate after k accesses is at least $1/\sigma^k$.

Since every candidate has to be ruled out independently of the others, a sequence of k accesses leaves at least rm/σ^k candidates not ruled out, on average. Each individual candidate can be directly verified by examining $\sigma/(\sigma-1)$ characters on average. Hence, our average cost is at least

$$k \ + \ \frac{rm}{\sigma^{k-1}(\sigma-1)}$$

The optimum is to keep examining characters until the average cost to directly verify the candidates equals the cost we would pay if we kept examining characters, and then switch to direct verification. This corresponds to minimizing the above formula. The optimum is

$$k^* \ = \ \log_\sigma\left(\frac{rm\sigma\ln\sigma}{\sigma-1}\right)$$

and hence the lower bound on the average cost per block is

$$\frac{1+\ln\left(\frac{rm\sigma\ln\sigma}{\sigma-1}\right)}{\ln\sigma} \ = \ \Theta(\log_\sigma(rm))$$

which proves our claim.