A New Filtration Method Based on the Locality Property for Approximate String Matching

Chia Wei Lu and R. C. T. Lee
Department of Computer Science
National Tsing Hua University, Hsinchu City, Taiwan, ROC
d9762807@oz.nthu.edu.tw, rctlee@ncnu.edu.tw

Abstract

In this paper, we consider the approximate string matching problem. We give a method to eliminate candidate locations in text T as there can be no substring S ending at those locations such that the edit distance between S and pattern P is smaller than or equal to a specified error bound k. Our method is simple to implement. Experimental results show that our method is effective. For instance, for a DNA type data, when the text is 10M characters long, patterns are 10 characters long and $k=1$, in average, 99% of locations are eliminated by our method. We have also performed comparisons with other algorithms, and the results showed that our algorithm can eliminate more parts than most of the others.

1 Introduction

For the approximate string matching problem, we first define the edit distance [17] which measures the similarity between two strings. The edit distance between two strings A and B, denoted $ED(A,B)$, is the minimum number of insertion, deletion or substitution operations needed to transform B into A. Edit distance can be computed by using dynamic programming method [22]. In this paper, we are interested in the approximate string matching problem defined as follows: Given a text string $T=t_1t_2...t_n$, a pattern string $P=p_1p_2...p_m$ and a error bound k, let $de_i = \min_{1\le j\le i} ED(P,t_j...t_i)$; the problem is to find all i such that $de_i \le k$. The problem is also known as the k *differences problem* [13].

In general, there are two approaches to solve our problem. The first approach, called non-Filtration Approach, is to check every position of T based upon the dynamic programming method. The first algorithm [18] based on non-Filtration Approach has a running time of O(mn). Some improved the time-complexity by using the properties of the dynamic programming matrix [9, 20] and reduced the running time to O(kn). Some use bit-parallelism in a computer word to reduce the number of operations which can solve our problem in $O(nk\lceil m/w\rceil)$ time-complexity [8, 10, 23].

Another approach, named Filtration Approach, uses an efficient filtration algorithm to eliminate uninteresting parts of the text T, and then checks for possible solutions at the locations which are not eliminated by using a non-filtration algorithm. The basic idea of the filtration approach can be generalized into two steps as follows.

Step 1: Consider a window of T ending at t_i. Compute a filtration function F. For some i, if the computation of F indicates that $de_i > k$, ignore the position i; else, mark i.

Step 2: For each i which is marked in Step 1, compute de_i and check if $de_i \le k$ by using a non-filtration algorithm.

In general, computing de_i is much slower than F. The key of filtration approach is the filtration function. We measure the effectiveness of the filtration function F by counting the number of positions which can be eliminated. Recent research [7, 13] shows that k *differences problem* can be efficiently solved by Filtration Approach algorithms if many locations can be eliminated. Therefore, it is meaningful to investigate the Filtration Approach because it will make the algorithms very efficient in average case time-complexity. We refer the readers to Section 8 in [13] for more details about the filtration algorithms. In this paper, we shall give an efficient method to filter out many locations. In the following, we categorize the existing Filtration algorithms into three different types.

(1) The Partial Exact Matching

The idea is to search some partial part of P with no mismatch instead of to search P approximately. Suppose we have a pattern P and we want to find a substring S in T such that $ED(S,P)\le k$. For k reasonably small, there must exist a substring in P which exactly appears in S. This approach therefore does an exact string matching first. Different algorithms select different substrings in P to do the exact string matching, as shown below.

There are many algorithms using this idea. Some algorithms use the fact that by dividing P into $(k+1)$ non-overlapping pieces, then at least one of the pieces must appear in the string S, where S is

a substring of T and $ED(S,P) \le k$. The filtration step is first to divide P into $(k+1)$ non-overlapping and approximately equal length of pieces. Then, for each window $W = T(i-m+1, i)$, check whether any of the pieces appears in W exactly. If none of the pieces appears in W, ignore this window W; otherwise, check it by using a non-filtration algorithm. This idea was used in many algorithms [1, 2, 11, 14, 15 and 23].

Another algorithm [3] finds the maximal substring of P on T starting from some locations. A maximal substring $S = t_i t_{i+1} ... t_j$ is a substring of P while $t_i t_{i+1} ... t_j t_{j+1}$ is not. It works as follows: traverse the text from the beginning and consider the text as the form $T = S_1 y_1 S_2 y_2 ...$ where S_i is a maximal substring and y_i is a character. For each i, if the size of $S_i y_i S_{i+1} y_{i+1} ... S_{i+k+1} y_{i+k+1}$ is less than $m-k$, it can be proved that there is no solution starting from the positions inside $S_i y_i$, and we then can skip $S_i y_i$. Otherwise, we check the region $S_i y_i S_{i+1} y_{i+1} ... S_{i+k+1} y_{i+k+1}$. Note that $m-k$ is the minimum length of a possible solution. This algorithm is a linear expected time algorithm, denoted *LET*. In [3, 4], the authors also gave an algorithm, called *SET*, which runs in sublinear expected time. *SET* is similar to *LET* except that the text is split into blocks with equal length $(m-k)/2$. It is obvious that any answer must contain at least one block. *SET* processes the text starting from the beginning of each block and finds $k+1$ maximal substring, $S_1 y_1 S_2 y_2 ... S_{k+1} y_{k+1}$, in each. If the ending position, denoted x, is inside this block, then there is no answer containing this block and we can skip this block; otherwise, check the substring of T starting from $(m+3k)/2$ characters to the left of the block and ending at x.

(2) Bad W-suffix

Consider a window W of length $m-k$ in T. We call *W-suffix* to be a suffix of the string W. If the distances between a *W-suffix* and every substring of P are larger than k, we call this *W-suffix* to be a *Bad W-suffix*. For example, if $P = acaac$, $W = agga$, and $k = 1$, there is a *Bad W-suffix*, '*gga*', whose edit distances with every substring of P are larger than $k = 1$. It is not possible that an answer contains a *Bad W-suffix*. Thus, if there exists a *Bad W-suffix* in W, we than can safely shift the window to the right and not to contain the whole *Bad W-suffix*. More precisely, for a window $W = T(i-m+k+1, i)$, if there is a *Bad W-suffix* $= T(j, i)$, we can prove that there is no answer starting from the positions in the region i to j, and we can safely shift the window to the position $j+1$. This is due to the fact that any solution that starts inside the region i to j must contain the *Bad W-suffix* because the length of a solution is at least $m-k$ and the size of W is $m-k$. For example, if $P = acaac$, $W = T(1,4) = agga$, and $k = 1$, we can find that '*gga*' is a *Bad W-suffix*, and then we can shift the window to the position 3. Thus the next window is $W = T(3,6)$.

The algorithms using this rule first try to find whether there exists a *Bad W-suffix* in the window $W = T(i-m+k+1, i)$. If there exists, we prefer to use the shortest one and shift the window without any checking. Note that the smallest *Bad W-suffix* can make us have a largest shift. Otherwise, we check the substring $T(i-m-k+1, i)$ against P by applying a non-filtration algorithm.

To find the *Bad W-suffix* in a window, there are several algorithms. The algorithm in [16] constructs a nondeterministic suffix automaton which recognizes every reverse prefix of P allowing at most k errors. Using this automaton, we read the characters of W from the right to the left, and it will stop reading if no pattern substring matches what was read with at most k errors. That is, it stops when we find a *Bad W-suffix*. Naturally, this *Bad W-suffix* is the shortest one.

Another filtration algorithm is proposed by [19]. This algorithm also considers the window with length $m-k$. For each window, it examines each character of window from the right to the left to see if the character appears in its corresponding characters in P. The corresponding characters are defined as follows. For a window $W = w_1 w_2 ... w_{m-k}$, the corresponding characters of w_i are p_{i-k}, p_{i-k+1}, .., and p_{i+k}. If a character of W does not appear in its corresponding characters, this character must be an error and we call it a *bad character*. Thus, this algorithm examines each character of W from the right to the left and stops when it finds $k+1$ bad characters. If it finds $k+1$ bad characters, this means that we have found a *Bad W-suffix*.

Before introducing another algorithm in [6] used to find *Bad W-suffix*, we define the *l-gram* first. An *l-gram* is a string with length *l*. This algorithm first constructs a table $D: \Sigma^l \to \mathrm{N}$ recording, for every possible *l*-gram, the edit distance of *l*-gram with its most similar substring of P. For each W, we read the non-overlapping *l*-grams of W from the right to the left and sum up their total value in D. If the total value exceeds k, it stops and this means that we find a *Bad W-suffix*. Note that this *Bad W-suffix* may not be the shortest one.

(3) Counting *q*-gram

This rule only concerns the contents of the two strings and ignores the order of the contents. For example, $x = abaaab$ and $y = bbaab$. We know the there are four '*a*'s and two '*b*'s in x and two '*a*'s and three '*b*'s in y. The difference between

the number of 'a' in x and that in y is 2 and 1 for 'b'. Thus, we must pay at least two edit operations to make the numbers of 'a' and 'b' in x and y to be equal. For example, delete one 'a' in x and substitute one 'a' in x by 'b'. In the following, we define q-gram which illustrates, in some way, the contents of a string. Let Σ be a finite alphabet and Σ^q be the set of all strings of length q over Σ. Let $x = x_1 x_2 ... x_{|x|}$ be a string over the Σ. v is a q-gram of x if $v = x_i x_{i+1} ... x_{i+q-1}$ and $(i+q-1) \le |x|$ for some i. Let $G(x)[v]$ denote the total number of the occurrences of v in x. Then, the q-gram distance [21] between two strings x and y is defined as follows.

$$D_q(x,y) = \sum_{v \in \Sigma^q} |G(x)[v] - G(y)[v]|.$$

For example, let $x = abaaab$ and $y = bbaab$ be strings over the alphabet $\{a,b\}$. Then, $G(x)[a] = 4$, $G(y)[a] = 2$, $G(x)[b] = 2$, and $G(y)[b] = 3$. The 1-gram distance between x and y is $D_1(x,y) = |4-2| + |2-3| = 2+1 = 3$. In this example, there are five and four 2-grams in x and y respectively. We have $G(x)[aa] = 2$, $G(x)[ab] = 2$, $G(x)[ba] = 1$, $G(y)[aa] = 1$, $G(y)[ab] = 1$, $G(y)[ba] = 1$, and $G(y)[bb] = 1$. The 2-gram distance between x and y is $D_2(x,y) = |2-1| + |2-1| + |1-1| + |0-1| = 1+1+0+1 = 3$.

Let $d_i(j) = D_q(P, t_j ... t_i)$, for some $q > 0$. The relationship between q-gram distance $d_i(j)$ and edit distance de_i is given in [21] as follows.

Theorem 1 [21] For $1 \le i \le n$, $d_i(i-m+1) \le 2q de_i$.

This shows that q-gram distance is a lower bound of $2q de_i$ and can be used as a filter. If $d_i(i-m+1)/(2q) > k$ for some i, then we have $de_i > k$, and we can ignore the position i. [21] also gives an efficient algorithm to compute $d_i(i-m+1)$ for all i in linear time. It marks all i such that $d_i(i-m+1)/(2q) \le k$ and then evaluates de_i only for those marked i's. The k differences problem can be solved in time O(min($n + rk^2, kn$)), where r is the number of indexes i such that $d_i(i-m+1)/(2q) \le k$.

For example, if $P = abcd$, $T(3,6) = aaba$, $q = 1$ and $k = 1$. We first find that $d_6(3) = D_1(P, t_3 ... t_6) = 4$. We have $d_6(3)/(2 \cdot 1) = 2 > k$. Then we know that de_6 must be larger than 2 and skip this position.

However, consider if $P = abcdef$ and a window $T(8,13) = ceafbd$ of T. Let $q = 1$ and $k = 1$. We have $d_{13}(8) = D_1(P, t_8 ... t_{13}) = 0$, and $d_{13}(8)/(2 \cdot 1) = 0 < k$. Using the algorithm [21], we need to check the position 8. But in this case, if we consider the order of characters, it is quite obvious that the de_{13} would be very large.

Another disadvantage of algorithm [21] is that if q is set too large, the value of $d_i(i-m+1)/(2q)$ would be small and cause that many positions are needed to check. For example, if $P = abcd$, $T(3,6) = aaba$, $q = 2$ and $k = 1$. We have $d_6(3) = D_2(P, t_3 ... t_6) = 3$ and $d_6(3)/(2 \cdot 2) = 3/4 < k$.

This idea also is used in [12]

In the next section, we give our filtration mechanism which combines the Partial Exact Matching Rule and the Counting q-gram Rule. Our filtration uses a special property, called the locality property which is used implicitly in [19].

2 Our Algorithm: A New Distance Function for Filtration Based upon the Locality Property

Our algorithm is based upon the following ideas:

1. We partition P into $(k+1)$ non-overlapping pieces P^0, P^1, …, and P^k. That is, for a pattern $P = p_1 p_2 ... p_m$, $P^j = p_{jl+1} p_{jl+2} \cdots p_{(j+1)l}$ for $0 \le j < k$, and $P^k = p_{kl+1} p_{kl+2} ... p_m$, where $l = \lfloor m/(k+1) \rfloor$. Let us assume that there is a substring S of T whose edit distance from P is $\le k$. Then there must be at least one piece of P exactly appearing in S. This is the Partial Exact Matching rule which was introduced in Section 1. In other words, for a window W with appropriate size, if no P^i appears exactly in W, we may ignore this window.
2. To decide whether there exists one piece of P in W, we shall use a special q-gram checking function which will be defined below. When we apply this function, we shall utilize a special property, called the Location Property.

Q-gram Checking Function: The q-gram checking function $DL_q(A,B)$ between two strings A and B is defined as follows.

$$DL_q(A,B) = \sum_{v \in \Sigma^q} \max(G(A)[v] - G(B)[v], 0).$$

The value of $DL_q(A,B)$ is equal to the number of q-grams of A which are not in B. For the case when the string A appears in the string B, it is obvious that $G(A)[v] \le G(B)[v]$ for all $v \in \Sigma^q$, and we have $DL_q(A,B) = 0$. It can be easily proved that if $DL_q(A,B) \ne 0$, the string A does not appear in B. Thus we have the following lemma.

Lemma 1 If String A exactly appears in string B, $DL_q(A,B) = 0$.

The above lemma means that we can use this checking function to determine whether string A

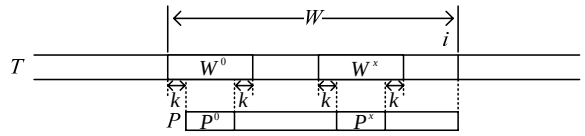

Fig. 1. The pieces of P and their corresponding substrings in a window W.

exactly appears in string B of not. We merely compute $DL_q(A,B)$ and check whether it is equal to 0 or not. For example, if $A = aaccg$ and $B = aagacgcg$, there is a 2-gram of A, 'cc', which is not in B. We have $DL_2(A,B) = 1 \neq 0$, and we can conclude that A does not appear in B.

In our filtration step, we partition P into $k+1$ pieces P^0, P^1, …, and P^k. We determine whether any P^i exactly appears in a window $W = w_1 w_2 ... w_{m+k} = T(i-m-k+1, i)$, we do not apply the above checking function directly. We shall utilize a special property which can be explained by considering Fig. 1.

As shown in Fig. 1, if P^i appears exactly in W, it does not appear in some arbitrary location. Instead, it must appear in the corresponding segment W^i of W. For example, if $P = abaccb$ and $S = aaaacb$, and we have $ED(P,S) = 2$. We partition P into three substrings, namely 'ab', 'ac' and 'cb'. It should be obvious that 'ab' must appear at the beginning, 'ac' in the middle and 'cb' in the end of S. That is, we may produce $k+1$ segments from W into and we only have to check whether P^i appears exactly in its corresponding segment W^i in W or not. The $(k+1)$ segments W^0, W^1, …, and W^k of W are defined as follows. $W^j = w_{jl+1} w_{jl+2} \cdots w_{(j+1)l+2k+1}$ for $0 \le j < k$, and $W^k = w_{k \cdot l+1} w_{k \cdot l+2} ... w_{m+k}$, where $l = \lfloor m/(k+1) \rfloor$. For example, if the window $W = aactgtccaa$, $m = 8$, and $k = 2$, we have $l = \lfloor 8/(2+1) \rfloor = 2$, and $W^0 = aactgt$, $W^1 = ctgtcc$, and $W^2 = gtccaa$.

Lemma 2 For a window $W = T(i-m-k+1, i)$, if no P^j appears in W^j, then no substring in W has edit distance with respect to P less than or equal to k.

Proof: We prove by contradiction. Suppose that no piece of P appears in its corresponding segment W^j and $de_i \le k$. Let S be the substring of W such that $ED(P,S) \le k$. Based on [23], we know that at least one piece of P appears in S. If some piece P^j appears exactly in S and not in its corresponding segment, then a part of it appears outside of W^j and we must need $(k+1)$ insertions or deletions to transform S into P. Thus, $ED(P,S) > k$. Therefore, there is no S existing such that $ED(P,S) \le k$, and $de_i > k$.

We shall call the above lemma the **Bad Piece Rule** for filtration of approximate string matching.

Theorem 2 If $de_i \le k$ for some i and $W = T(i-m-k+1, i)$, then there is at least one piece P^x of P such that $DL_q(P^x, W^x) = 0$.

Proof: We prove by contradiction. Suppose for every piece P^x of P, $DL_q(P^x, W^x) \neq 0$. Then no piece P^x appears in its corresponding string W^x. By Lemma 2, we have de_i is larger than k.

Theorem 2 also implies the following theorem:

Theorem 3: If for some i, $W = T(i-m-k+1, i)$, and there is no one piece P^x of P such that $DL_q(P^x, W^x) = 0$, then $de_i > k$.

To use the Theorem 3 as a filtration, we need to compute the distance functions $DL_q(P^j, W^j)$ for $0 \le j < k$. Let $q = 1$ and $k = 1$. For example, if $P = abcdef$ and a window $W = T(8,15) = fceafbd$ of T. We divide P into $(k+1) = 3$ parts, $P^0 = ab$, $P^1 = cd$, and $P^2 = ef$. And the corresponding $(k+1)$ parts of W are $W^0 = fcea$, $W^1 = eafb$, and $W^2 = fbd$. We have $DL_1(P^0, W^0) = 1 > 0$, $DL_1(P^1, W^1) = 2 > 0$, and $DL_1(P^2, W^2) = 1 > 0$, and by Theorem 3, we can conclude that $de_{15} > k$. This means that there is no substring of T ending at position 15 whose edit distance with respect to P is less than or equal to k and we can therefore skip this position.

We shall show that each $DL_q(P^j, W^j)$ can be updated efficiently when the window is shifted one step to the next window. For each W^j, construct an array $GW(j)[0:\alpha^q]$ to store the number of q-grams in W^j for $0 \le j \le k$ and an array $GW[0:\alpha^q]$ for $W_m = w_{k+1} w_{k+2} ... w_{m+k}$. For the first window $W = T(1, m+k)$, we compute the $GW[0:\alpha^q]$, $GW(j)[0:\alpha^q]$'s, $D_q(P, W_m)$ and $DL_q(P^j, W^j)$'s directly. After the computation concerning with the first window is done, we shift the window one step to the right and update the values of them. In a new window, for each W^x or W_m, there are only two q-grams changed, one old q-gram Q_1 and one new q-gram Q_2 as shown in Fig. 2.

To update the values of $D_q(P, W_m)$ and $DL_q(P^j, W^j)$'s, we use the following formulas. Let $GP(j)[0:\alpha^q]$ be the number of q-grams in P^j for $0 \le j < k$ and $GP[0:\alpha^q]$ for P which can be computed in preprocessing.

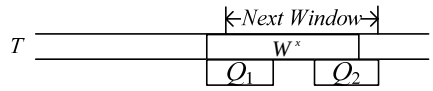

Figure 2. The two q-grams needed to update when shift the window one step.

$$(1)\ \begin{cases} \text{if } GP(j)[Q_1] >= GW(j)[Q_1], \\ \quad DL_q(P^j, W^j) = DL_q(P^j, W^j) + 1. \\ \text{if } GP(j)[Q_2] > GW(j)[Q_2], \\ \quad DL_q(P^j, W^j) = DL_q(P^j, W^j) - 1. \end{cases}$$

$$(2)\ \begin{cases} \text{if } GP[Q_1] < GW[Q_1], \\ \quad D_q(P, W_m) = D_q(P, W_m) - 1. \\ \text{otherwise}, \\ \quad D_q(P, W_m) = D_q(P, W_m) + 1. \end{cases}$$

$$(3)\ \begin{cases} \text{if } GP[Q_2] > GW[Q_2], \\ \quad D_q(P, W_m) = D_q(P, W_m) - 1. \\ \text{otherwise}, \\ \quad D_q(P, W_m) = D_q(P, W_m) + 1. \end{cases}$$

After updating the values of $D_q(P, W_m)$ and $DL_q(P^j, W^j)$'s, we also need to update the profiles of $GW(j)$'s and $GW(m)$ by decreasing one count of Q_1 and increasing one count of Q_2 in each W^j's and W_m.

There is another point which we must pay attention to. Suppose that our filtration process decides that position i needs to be checked. We do not immediately perform a dynamic programming procedure to the substring of T starting from $i-m-k+1$ and ending at i. Instead, we look at Position $i+1$. If Position $i+1$ also needs to be checked, we can easily see that it will be more efficient to apply the dynamic programming procedure to the substring starting from $i-m-k+1$ and ending at $i+1$. We use an array $Mark[1:n]$ to record the positions which can be filtered and those which has to be checked. If position i is needed to check, set $Mark[i]=1$; otherwise, $Mark[i]=0$. Initially, set $Mark[m+k-1]=1$, and $Mark[i]=0$ for $m+k \le i \le n$.

In the following, we present our algorithm to solve the *k differences problem*.

Our Algorithm

Input: A text T, a pattern P, an error bound k, and q.
Output: All the positions i such that $de_i \le k$.

```
1.  Preprocess( P, k, GP(0…k)[0:Σ^q], GP[0:Σ^q] )
2.  W ← T(1, m+k), l ← ⌊m/(k+1)⌋
3.  For j ∈ 0…k−1 Do
4.      ComputeProfiles( w_{jl+1} w_{jl+2} … w_{(j+1)l+2k+1}, GW(j)[0:Σ^q] )
5.  ComputeProfiles( w_{kl} w_{kl+1} … w_{m+k}, GW(k)[0:Σ^q] )
6.  ComputeProfiles( w_{k+1} w_{k+2} … w_{m+k}, GW[0:Σ^q] )
7.  Mark[m+k−1] ← 1, Mark[m+k] ← 0
8.  D_q ← ComputeDq( GP[0:Σ^q], GW[0:Σ^q] )
9.  For j ∈ 0…k Do
10.     DL_q(j) ← ComputeDL_q( GP(j)[0:Σ^q], GW(j)[0:Σ^q] )
11.     If DL_q(j) ≠ 0, Mark[m+k] ← 1
12. If D_q / 2q > k, Mark[m+k] ← 0
13. For i ∈ m+k+1…n Do
14.     W ← T(i−m−k+1, i)
15.     Update( GP(0…k)[0:Σ^q], GP[0:Σ^q], GW(0…k)[0:Σ^q], GW[0:Σ^q],
        DL_q(0…k), D_q, W, t_{i−m−k} )
16.     Mark[i] ← 0
17.     For j ∈ 0…k Do
18.         If DL_q(j) ≠ 0, Mark[i] ← 1
19.     If D_q / 2q > k, Mark[i] ← 0
20. CheckPhase( P, T, Mark[1…n] )
```

```
Preprocess( P, k, GP(0…k)[0:Σ^q], GP[0:Σ^q] )
1.  l ← ⌊m/(k+1)⌋
2.  For j ∈ 0…k−1 Do
3.      ComputeProfiles( p_{jl+1} p_{jl+2} … p_{(j+1)l}, GP(j)[0:Σ^q] )
```

4. *ComputeProfiles*($p_{kl+1}p_{kl+2}\cdots p_m$, $GP(k)[0:\Sigma^q]$)
5. *ComputeProfiles*(P , $GP[0:\Sigma^q]$)

ComputeProfiles(S , $G[0:\Sigma^q]$)
1. For $i \in 0\ldots|\Sigma|^q$ Do
2. $G[i] \leftarrow 0$
3. For $i \in 1\ldots|S|-q+1$ Do
4. $G[qgramValue(S(i,i+q-1))] \leftarrow G[qgramValue(S(i,i+q-1))]+1$

qgramValue(S)
1. For every character $a \in \Sigma$, if it is j th largest in lexicographic order, encode it by j, $En(a)=j$.
2. $value \leftarrow 0$
3. For $i \in 1\ldots|S|$ Do
4. $value \leftarrow value \cdot |\Sigma| + En(s_i)$
5. Return $value$

ComputeDL$_q$($G_1[0:\Sigma^q], G_2[0:\Sigma^q]$)
1. $DL_q \leftarrow 0$
2. For $i \in 0\ldots|\Sigma|^q$ Do
3. If $G_1[i]-G_2[i]>0$, $DL_q \leftarrow DL_q + (G_1[i]-G_2[i])$
4. Return DL_q

Update($GP(0\ldots k)[0:\Sigma^q], GP[0:\Sigma^q], GW(0\ldots k)[0:\Sigma^q], GW[0:\Sigma^q], DL_q(0\ldots k)$, D_q, W, x)
1. $VQ_1 \leftarrow qgramvalue(x \circ w_1w_2\ldots w_{q-1})$ and $VQ_2 \leftarrow qgramvalue(w_{2k-q+2}\ldots w_{2k+2}w_{2k+1})$
2. For $j \in 0\ldots k-1$ Do
3. If $GP(j)[VQ_1] \geq GW(j)[VQ_1]$, $DL_q(j) \leftarrow DL_q(j)+1$.
4. If $GP(j)[VQ_2] > GW(j)[VQ_2]$, $DL_q(j) \leftarrow DL_q(j)-1$.
5. $GW(j)[VQ_1] \leftarrow GW(j)[VQ_1]-1$, $GW(j)[VQ_2] \leftarrow GW(j)[VQ_2]+1$
6. $VQ_1 \leftarrow qgramvalue(w_{jl+1}w_{jl+2}\ldots w_{jl+q-1})$, $VQ_2 \leftarrow qgramvalue(w_{(j+1)l+2k-q+2}\ldots w_{(j+1)l+2k+1})$
7. $VQ_1 \leftarrow qgramvalue(w_{kl}w_{kl+1}\ldots w_{kl+q-1})$, $VQ_2 \leftarrow qgramvalue(w_{m+k-q+1}\ldots w_{m+k})$
8. If $GP(k)[VQ_1] \geq GW(k)[VQ_1]$, $DL_q(k) \leftarrow DL_q(k)+1$.
9. If $GP(k)[VQ_2] > GW(k)[VQ_2]$, $DL_q(k) \leftarrow DL_q(k)-1$.
10. $VQ_1 \leftarrow qgramvalue(w_{k+1}w_{k+2}\ldots w_{k+q-1})$, $VQ_2 \leftarrow qgramvalue(w_{m+k-q+1}\ldots w_{m+k})$
11. If $GP[VQ_1] < GW[VQ_1]$, $D_q \leftarrow D_q-1$; otherwise, $D_q \leftarrow D_q+1$.
12. If $GP[VQ_2] > GW[VQ_2]$, $D_q \leftarrow D_q-1$; otherwise, $D_q \leftarrow D_q+1$.
13. $GW[VQ_1] \leftarrow GW[VQ_1]-1$, $GW[VQ_2] \leftarrow GW[VQ_2]+1$

CheckPhase(P , T , $Mark[1\ldots n]$)
1. For $i \in m+k-1\ldots n$ Do
2. If $Mark[i]=1$, then
3. For $j \in i-m-k+1\ldots i$ Do
4. $Mark[j] \leftarrow 1$
5. Apply [9] Algorithm on those substrings $T(i,j)$ in T where $Mark[i\ldots j]=1$

3 Experiments

In our experiments, we tested our algorithm and compared with other filtration algorithms. The algorithms are:

CL90_LET: The linear expected time algorithm in [3].
CL90_SET: The sublinear expected time algorithm in [3].
CM94: Chang and Marr algorithm [5].
FN04: Fredriksson and Navarro algorithm [6].
NR00: Navarro and Raffinot algorithm [16].
WM92: Wu and Manber algorithm [23].
U92: Ukkonen algorithm [21].

We implemented these algorithms using C and compared the efficiency on the percentage of

positions which can be filtered.

We also tested our algorithm combining *NR00*. That is, we do not check the positions which pass our filtration algorithm immediately. Instead, we apply another filtration algorithm, *NR00*, so that we may filter more positions which are not possible answers.

In this experiment, we generated P with size $m = \{10, 20, 50\}$ and T with size $n = 10\text{M}$ randomly by using an alphabet $\Sigma = \{a, c, g, t\}$. We tested the performance on different error bounds k, $1 \le k \le 10$. Many algorithms [5, 6, 21] have a parameter q and different q-grams will have different efficiencies. We only show the best performance of some q for each algorithm, $1 \le q \le 5$. Each result is the average of 10 experiments with different P's and one T. The results are shown in Table 1-3.

Table 1. The percentage of positions needed to check for *m*=10

k	1	2	3
CL90_LET	48.71	98.97	100
CL90_SET	99.11	100	100
CM94	100	100	100
FN04	0.87	33.02	97.76
NR00	0.98	43.5	99.89
WM92	2.32	47.95	98.47
U92	1.48	49.43	93.51
Our	0.47	30.29	91.94
Our+NR00	0.23	18.08	90.76

Table 2. The percentage of positions needed to check for *m*=20

k	1	2	3	4	5
CL90_LET	0.02	2.68	52.38	98.36	100
CL90_SET	9.6	88.87	100	100	100
CM94	8.86	70.27	100	100	100
FN04	0	0	0.89	26.42	91.62
NR00	0	0	0.1	4.7	59.03
WM92	0	1.74	9.66	42.23	94.09
U92	0	0.03	5.46	54.54	95.86
Our	0	0.01	2.02	32.25	90.74
Our+NR00	0	0	0.05	2.87	49.03

Table 3. The percentage of position needed to check for *m*=50

k	1	2	3	4	5
CL90_LET	0	0	0	0	0
CL90_SET	0	0	0.16	8.94	84.79
CM94	0	0.02	0.38	4.42	82.93
FN04	0	0	0	0	0
NR00	0	0	0	0	0
WM92	0	0	0	0.03	0.55
U92_Qgram	0	0	0	0	0
Our	0	0	0	0	0
Our+NR00	0	0	0	0	0
k	6	7	8	9	10
CL90_LET	0.02	1.32	23.64	80.09	98.89
CL90_SET	99.7	100	100	100	100
CM94	98.76	99.99	100	100	100
FN04	0	0	0.01	0.45	11.59
NR00	0	0	0	0	0
WM92	2.61	11.8	44.07	48.57	94.76
U92_Qgram	0	0.21	6.22	42.72	91.55
Our	0	0.09	3.29	24.65	83.98
Our+NR00	0	0	0	0	0

4 Concluding Remarks and Future Research

We have proposed a method to eliminate candidates in a text string for approximate string matching. We plan to work on the average case time complexity analysis of our algorithm in the future. We also plan to extend our idea to tackle the exact string matching problem.

References

[1] Baeza-Yates, R. and Navarro, G. 1999. Faster approximate string matching. Algorithmica 23, 2, 127–158. Preliminary versions in Proceedings of CPM '96 (LNCS, vol. 1075, 1996) and in Proceedings of WSP'96, Carleton Univ. Press, 1996.

[2] Baeza-Yates, R. and Perleberg, C. 1996. Fast and practical approximate pattern matching. Information Processing Letters 59, 21–27.

[3] Chang, W. I. and Lawler, E. L., Approximate String Matching in Sublinear Expected Time, in: Proceedings of the

ACM-SIAM 31st Annual Symposium on Foundations of Computer Science, 1990, pp. 116-124.

[4] Chang, W. and Lawler, E. 1994. Sublinear approximate string matching and biological applications. Algorithmica 12, 4/5, 327–344. Preliminary version in FOCS ’90.

[5] W. Chang, T. Marr, Approximate string matching and local similarity, in: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, in: LNCS, vol. 807, Springer-Verlag, 1994, pp. 259–273.

[6] Fredriksson, K., Navarro, G., Average-Optimal Multiple Approximate String Matching, ACM Journal of Experimental Algorithmics, Vol. 9, No. 1.4, 2004, pp.1-47.

[7] Giegerich, R., Kurtz, S., Hischke, F., and Ohlebusch, E. 1997. A general technique to improve filter algorithms for approximate string matching. In Proceedings of the 4th South American Workshop on String Processing (WSP ’97). Carleton Univ. Press. 38–52. Preliminary version as Tech. Rep. 96-01, Universit¨at Bielefeld, Germany, 1996.

[8] Hyyrö, H. and Navarro, G., Bit-parallel Witnesses and their Applications to Approximate String Matching, Algorithmica, Vol. 41, No. 3, 2005, pp.203-231.

[9] Landau, G. and Vishkin, U., Fast Parallel and Serial Approximate String Matching, Journal of Algorithms, Vol. 10, 1989, pp.157-169.

[10] Myers, G., A fast bit-vector algorithm for approximate pattern matching based on dynamic programming, In Proc. CPM’98, LNCS, Vol. 1448, 1998, pp.1-13.

[11] Navarro, G. and Baeza-Yates, R. 1998a. Improving an algorithm for approximate pattern matching. Tech. Rep. TR/DCC-98-5, Dept. of Computer Science, Univ. of Chile. Algorithmica, to appear. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/dexp.ps.gz.

[12] Navarro, G. 1997a. Multiple approximate string matching by counting. In Proceedings of the 4th South American Workshop on String Processing (WSP ’97). Carleton Univ. Press, 125–139.

[13] Navarro, G., A Guided Tour to Approximate String Matching, ACM Computing Surveys, Vol. 33, 2001, pp.31-88.

[14] Navarro, G. and Baeza-Yates, R., Very fast and simple approximate string matching, Information Processing Letters, Vol. 72, 1999, pp.65-70.

[15] Navarro, G. and Baeza-Yates, R., A Hybrid Indexing Method for Approximate String Matching, Journal of Discrete Algorithms, Vol.1, No.1, 2000, pp.205-239.

[16] Navarro, G. and Raffinot, M. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM J. Exp. Algor. 5, 4. Previous version in Proceedings of CPM ’98. Lecture Notes in Computer Science, Springer-Verlag, New York.

[17] Needleman, S. B. and Wunsch, C. D., A general method applicable to the search for similarities in the aminoacid sequence of two proteins, Journal of Molecular Biology, Vol. 48, 1970, pp.443-453.

[18] Sellers, P. H., String Matching with Errors, Journal of Algorithms, Vol. 20, No. 1, 1980, pp.359-373.

[19] Tarhio, J. and Ukkonen, E., Approximate Boyer-Moore String Matching, SIAM Journal on Computing, Vol. 22, No. 2, 1993, pp.243-260.

[20] Ukkonen, E., Finding approximate patterns in strings, J. of Algorithms, Vol. 6, 1985, pp.132-137.

[21] Ukkonen, E., Approximate string matching with *q*-grams and maximal matches, Theoretical Computer Science, Vol. 92, 1992, pp.191-211.

[22] Wagner, R.A. and Fisher, M.J., The string-to-string correction problem, J. ACM, Vol. 21, 1974, pp. 168-173.

[23] Wu, S. and Manber, U., Fast Text Searching: Allowing Errors, Communications of the ACM, Vol. 35, 1992, pp.83-91.