Approximate String Matching using Compressed Suffix Arrays*

Trinh N.D. Huynh[1], Wing-Kai Hon[2], Tak-Wah Lam[2**], and Wing-Kin Sung[1***]

[1] School of Computing, National University of Singapore, Singapore,
{huynhngo,ksung}@comp.nus.edu.sg
[2] Department of Computer Science, The University of Hong Kong, Hong Kong,
{wkhon,twlam}@csis.hku.hk

Abstract. Let T be a text of length n and P be a pattern of length m, both strings over a fixed finite alphabet A. The k-difference (k-mismatch, respectively) problem is to find all occurrences of P in T that have edit distance (Hamming distance, respectively) at most k from P. In this paper we investigate a well-studied case in which T is fixed and preprocessed into an indexing data structure so that any pattern query can be answered faster. We give a solution using an $O(n \log n)$ bits indexing data structure with $O(|A|^k m^k \cdot \max(k, \log n) + occ)$ query time, where occ is the number of occurrences. The best previous result requires $O(n \log n)$ bits indexing data structure and gives $O(|A|^k m^{k+2} + occ)$ query time. Our solution also allows us to exploit compressed suffix arrays to reduce the indexing space to $O(n)$ bits, while increasing the query time by an $O(\log n)$ factor only.

1 Introduction

Let T be a text of length n and P be a pattern of length m, both strings over a fixed finite alphabet A. The string matching problem is to find all occurrences of P in T which satisfy some criteria. Depending on the criteria, we have three different problems. (1) The exact string matching problem requires us to find all exact occurrences of P in T; (2) The k-difference problem is to find all occurrences of P in T that have edit distance at most k from P; and (3) The k-mismatch problem is to find all occurrences of P in T that have Hamming distance at most k from P. Edit distance between two strings is defined to be the number of character insertions, deletions and replacements to convert one string to another. When only character replacements are allowed, we have Hamming distance. These problems are well-studied. They find applications in many areas including computational biology, signal processing, text retrieval, handwriting recognition, pattern recognition, etc.

In the past, most of the research are on the online version of the string matching problem. This version of the problem assumes both the text T and the pattern P are not known in advance. For exact online matching, well-known algorithms include Boyer-Moore [1] and Knuth-Morris-Pratt [2] algorithms. For approximate online matching,

* An extended abstract version of this paper was presented at the Fifteenth Annual Combinatorial Pattern Matching Symposium (CPM 2004) [Lecture Notes in Comput. Sci., vol 3109].
** This work was supported in part by the Hong Kong RGC Grant HKU/7042/02E.
*** This work was supported in part by the NUS Academic Research Grant R-252-000-119-112.

the problem can be solved by standard dynamic programming in $O(mn)$ time. Landau and Vishkin[3] improved the time complexity to $O(kn)$. Other results include Baeza-Yates and Navarro [5] and Amir et al [4]. We refer to Navarro [6] for a survey.

Recently, people start to consider the offline version of the problem, which assumes the text T is given in advance and we can preprocess it to build an indexing data structures so that any pattern query can be answered faster. One of the motivating applications of the offline version of the problem is the DNA sequence searching. This application requires us to find DNA subsequences (like genes, promoter consensus sequences) over some known DNA genome sequences like the human genome. Since the genome sequence is very long, people would like to preprocess it to accelerate pattern queries.

In the literature, there are a number of indexing data structures for the exact offline pattern matching problem. Suffix trees [7] and suffix arrays [8] are some well-known solutions. For a text T, building a suffix tree takes $O(n)$ time. After that, exact occurrences of a pattern P can be located in $O(m + occ)$ time where occ is the number of occurrences. For suffix arrays, construction and searching take $O(n)$ time and $O(m + \log n + occ)$ time, respectively. Both data structures require $O(n \log n)$ bits space, though suffix arrays are associated with a smaller constant. Recently, two compressed versions of suffix arrays have been devised, which are compressed suffix arrays (CSA) [9] and FM-index [10]. Both of them occupy only $O(n)$ bits space, yet still supporting exact pattern searching efficiently.

Besides exact matching, the k-difference and the k-mismatch problems are also important since the text and the pattern may contain "errors" (for example, in gene hunting, there may be some mutations in a gene). Jokinen and Ukkonen [25] were the first to treat the approximate offline matching problem in which the text T can be preprocessed. Since then, many different approaches have been proposed. (Please refer to Navarro et al [26] for a brief survey.) Some techniques are fast in average [11–15, 21]. However, they incur a query time complexity depending on n, i.e., in the worst case, they are inefficient even if the pattern is very short and k is as small as one. The first solution which has query time complexity depends only on m and k is proposed by Ukkonen [20]. There are several interesting results focusing on one single error (k=1) [16–19]. Cobbs [16] gave an indexing data structure using $O(n \log n)$ bits space and having $O(m^2 + occ)$ query time for $k = 1$. More recently, Amir et al [17] proposed a result with $O(n \log^2 n)$ preprocessing time, $O(n \log^3 n)$ bits indexing space and $O(m \log n \log \log n + occ)$ query time and Buchsbaum et al [18] proposed another indexing data structure which uses $O(n \log^2 n)$ bits space and can be preprocessed in $O(n \log n)$ time. After building the index, every query can be solved in $O(m \log \log n + occ)$ time.

For $k \geq 1$, Cobbs [16] proposed an $O(n \log n)$-bit indexing data structure in which every query takes $O(m^{k+2}|A|^k + occ)$ time. Cole, Gottlieb, and Lewenstein [19] proposed an indexing data structure with query times of $O(\frac{(c \log n)^k \log \log n}{k!} + m + occ)$ and $O(\frac{(c \log n)^k \log \log n}{k!} + m + 3^k \cdot occ)$ for the k-mismatch and k-difference problems, respectively, where $c > 1$ is some constant. However, their solutions use $O(n \frac{(d \log n)^k}{k!} \log n)$ bits space and $O(n \frac{(d \log n)^k}{k!})$ preprocessing time, where $d > 1$ is some constant. Therefore, their solutions are impractical when n is large.

The contribution of this paper is a faster solution for the k-difference and k-mismatch problems. We assume that $|A|$ is constant. For the special case when $k = 1$, we show in this paper that a suffix array plus an inverse suffix array can give a simple solution that uses $O(n \log n)$-bit space and $O(m \log n + occ)$ query time. We combine the techniques of forward searching and backward searching on suffix arrays data structure to achieve this time bound. Furthermore, this solution allows us to exploit compressed suffix arrays to reduce the space to $O(n)$ bits, while increasing the query time by an $O(\log n)$ factor only. Though this solution is not the fastest solution in literature for $k = 1$, it uses optimal space [1]. Moreover, our indexing data structure can be constructed in $O(n)$ time, and hence our solution requires only linear preprocessing time.

We also show how to extend our algorithm for k-difference and k-mismatch problems for $k \geq 1$. Our solution takes $O(m^k|A|^k \cdot \max(k, \log n) + occ)$ or $O(m^k|A|^k \cdot \max(k, \log^2 n) + occ \log n)$ query time, when using $O(n \log n)$ bits or $O(n)$ bits indexing data structure, respectively.

The structure of this paper is as follows: In the next section we formally define the k-difference and k-mismatch problems and give an introduction about suffix arrays and compressed suffix arrays data structure. Then, Section 3 shows our solution for the problems with $k = 1$. Next, Section 4 extends the solution and solves the problems for $k \geq 1$. Finally, Section 5 concludes the paper.

2 Preliminaries

2.1 Definition of the problem

Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$ be strings over an alphabet A. The edit distance between A and B, denoted by $dist(A, B)$, is the minimum number of character deletions, replacements, and insertions to convert A to B. The Hamming distance between A and B is the minimum number of character replacements to convert A to B.

Distance $dist(X, Y)$ can be evaluated in time $O(mn)$ by using a very simple form of dynamic programming [29]. The method evaluates an $(m + 1) \times (n + 1)$ table e such that $e(i, j) = dist(x_1 \cdots x_i, y_1 \cdots y_j)$. Entries in one column are used to evaluate entries in the next column. Hence $dist(X, Y) = e(m, n)$.

The $editing\ trace$ from X to Y is any sequence $\mathcal{T} = \tau_1 \tau_2 \cdots \tau_q$ of character operations applied on positions in X to get Y, ordered from the rightmost position in X to the leftmost position, where each τ_i is either ***d (delete)***, r_c ***(replace with a character*** c***)***, ***u (unchange)***, or i_c ***(insert a character*** c***)***. For example, a trace from $aaaaa$ to $aaacb$ can be $r_b i_c duuu$, in which the r_b-operation is to replaced the rightmost character "a" by "b", the i_c-operation is to insert "c", the d-operation is to delete the second rightmost "a", and the remaining three u-operations are to keep the remaining "a" unchanged. There is exactly one operation (u,d, or r) for each character in X, whereas there may be zero or more i-operation. The cost $c(\mathcal{T})$ of $\mathcal{T}$ is the number of d, r, or i operations in $\mathcal{T}$. Thus $dist(X, Y)$ is the minimum possible cost of a trace from X to Y.

We place an order on the operations as follows: $u < d < r_* < i_*$, where $*$ denotes any characters in A. Given two character c and c' in A, where $c < c'$, we let $r_c < r_{c'}$ and

[1] It is non-trivial to change the data-structures of [17–19] to linear space data structure.

$i_c < i_{c'}$. A trace $\mathcal{T}$ from X to Y is said to be *smaller* than another different trace $\mathcal{T}'$ also from X to Y if either (1) $c(\mathcal{T}) < c(\mathcal{T}')$, or (2) $c(\mathcal{T}) = c(\mathcal{T}')$ and $\mathcal{T}$ is lexicographically smaller than $\mathcal{T}'$.

We have the following results from [20].

Lemma 1. [20] *Given a pattern P of length m over an alphabet A. The number of different traces of cost less than or equal to k that can be applied on P is $O(m^k|A|^k)$.*

k-difference problem Consider a text T of length n and a pattern P of length m, both strings over a fixed finite alphabet A. The *k-difference* problem is to find all positions j such that the edit distance between P and some substring starting at j in T is less than or equal to k.

k-mismatch problem Consider a text T of length n and a pattern P of length m, both strings over a fixed finite alphabet A. The *k-mismatch* problem is to find all positions j such that the Hamming distance between P and some substring starting at j in T is less than or equal to k.

In this paper we focus on the offline version of the problems, in which T is fixed and can be preprocessed to accelerate pattern queries.

2.2 Suffix arrays and inverse suffix arrays

Let $T[0..n] = t_0 t_1 \cdots t_n$ be a text of length n over an alphabet A where $t_n = \$$ is a special symbol that is not in A and smaller than any other symbol in A. The j-th suffix of T is defined as $T[j..n] = t_j \cdots t_n$ and is denoted by T_j.

The *suffix array* $SA[0..n]$ of T is an array of integers j that represent suffixes T_j and the integers are sorted in lexicographic order of the corresponding suffixes. We have that $SA[0] = n$.

Together with suffix arrays, we also use *inverse suffix arrays* to support searching in our algorithm. The inverse suffix array of T is denoted as $SA^{-1}[0..n]$, that is, $SA^{-1}[i]$ equals the number of suffixes which are lexicographically smaller than T_i.

The sizes of SA and SA^{-1} are $O(n \log n)$ bits. Both data structures can be constructed in linear time [23, 27, 28].

In this paper, an interval $[st..ed]$ is called the range of the suffix array of T corresponding to a string P if $[st..ed]$ is the largest interval such that P is a prefix of every suffix T_j for $j = SA[st], SA[st+1], \ldots, SA[ed]$. We write $[st..ed] = range(T, P)$.

We have the following lemma based on the forward searching technique on suffix arrays.

Lemma 2. [24] *Given a text T together with its suffix array, assume $[s..e] = range(T, P)$. Then, for any character c, the interval $[s'..e'] = range(T, Pc)$ can be computed in $O(\log n)$ time.*

Hence occurrences of a pattern P in T can be found forwardly in $O(m \log n)$ time by applying the above lemma m times.

We have another lemma.

Lemma 3. *Given the interval $[st_1..ed_1] = range(T, P_1)$ and the interval $[st_2..ed_2] = range(T, P_2)$, we can find the interval $[st..ed] = range(T, P_1P_2)$ in $O(\log n)$ time using the suffix array and the inverse suffix array of T.*

Proof. To find the interval $[st..ed]$, we have to find the smallest st and the largest ed such that both $T_{SA[st]}$ and $T_{SA[ed]}$ have P_1P_2 as their prefixes. So $[st..ed]$ is a subinterval of $[st_1..ed_1]$.

Let the length of P_1 be m_1. By the definition of suffix arrays, the lexicographic orders of $T_{SA[st_1]}, T_{SA[st_1+1]}, \ldots, T_{SA[ed_1]}$ are increasing. Since they share the same prefix P_1, the lexicographic orders of $T_{SA[st_1]+m_1}, T_{SA[st_1+1]+m_1}, \ldots, T_{SA[ed_1]+m_1}$ are also increasing. Thus $SA^{-1}[SA[st_1] + m_1] < SA^{-1}[SA[st_1 + 1] + m_1] < \ldots < SA^{-1}[SA[ed_1] + m_1]$.

To find st and ed, we find the smallest st such that $st_2 \leq SA^{-1}[SA[st]+m_1] \leq ed_2$ and the largest ed such that $st_2 \leq SA^{-1}[SA[ed] + m_1] \leq ed_2$. This can be done by binary search on the interval $[st_1..ed_1]$ and make $O(\log n)$ calls to the suffix array and the inverse suffix array. □

Assuming we have an array C such that for any c in A, $C[c]$ stores the total number of occurrences of all characters c' in T, where $c' \leq c$. This gives us the following backward searching technique.

Lemma 4. *Given the suffix array and the inverse suffix array of T, assume $[s..e] = range(T, P)$. For any character c, assume we have in advance the array C, we can find the interval $[s'..e'] = range(T, cP)$ in $O(\log n)$ time.*

Proof. The lemma follows directly from Lemma 3 where $P_1 = c$ and $P_2 = P$. Note that $range(T, P_1) = [C[c'] + 1..C[c]]$ where c' is a character alphabetically just smaller than c in the alphabet A. □

Hence occurrences of a pattern P in T can be found backwardly in $O(m \log n)$ time by applying the above lemma m times.

2.3 Compressed suffix array

The data structure compressed suffix array (CSA) $\Psi[1..n]$ [9] is a compressed version of a suffix array and has the size of the same order as the text itself. It is defined as

$$\Psi[i] = SA^{-1}[SA[i] + 1]$$

A compressed suffix array can be constructed in $O(n)$ time and can be stored in $O(n)$ bit space while every $\Psi[i]$ can be accessed in constant time [23]. We have the following result from Sadakane and Shibuya [22].

Lemma 5. [22] *We can store a compressed suffix array together with a supporting data structure in $O(H_0 n)$ bits space so that, for every i, $SA[i]$ and $SA^{-1}[i]$ can be evaluated in $O(\log n)$ time, where H_0 is the order-0 entropy of the text.*

By Lemma 5, we can simulate operations done on suffix arrays and inverse suffix arrays using compressed suffix arrays with the time complexity slowed down by a factor of $O(\log n)$. Thus, for compressed suffix arrays, we also have the same results for Lemmas 2, 3 and 4, but in time $O(\log^2 n)$.[2]

[2] Backward search (Lemma 4) actually can be done in $O(1)$ time using compressed suffix arrays [23].

3 The k-difference and k-mismatch problems with $k = 1$

For the k-difference problem where $k = 1$, Lemma 3 gives us an idea how to solve it. For $k = 1$, there is at most one "error" between the pattern $P[1..m]$ and any of its occurrences in $T[0..n]$. An error may be a character insertion, replacement or deletion. We can try to put the error at each position in the pattern to form an edited pattern P' which has edit distance 1 from P and check if P' is in T. Normally, checking if P' is in T requires $O(|P'|)$ time. Based on Lemmas 3 and 4, such checking can be done in $O(\log n)$ time as follows. We let P'_L and P'_R be the portions of the pattern P to the left and to the right of the error, respectively. Assume we know in advance the interval $[s..e] = range(T, P'_L)$ and the interval $[s'..e'] = range(T, P'_R)$. By using Lemma 2, we can find in $O(\log n)$ time the interval $[s''..e'']$ which is the range of the suffix array of T corresponding to P'_L appended with the error. Then, given $[s''..e'']$ and $[s'..e']$, by Lemma 3, we can find the interval $[st..ed]$ which is the range of the suffix array of T corresponding to the whole edited pattern P' using $O(\log n)$ time.

The algorithm is shown in Fig. 1. We first construct $F_{st}[1..m+1]$ and $F_{ed}[1..m+1]$ which are arrays such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$. Here we assume $P[m+1..m]$ is an empty string and $F_{st}[m+1] = 0$, $F_{ed}[m+1] = n$. F_{st} and F_{ed} are used later as the range of the suffix array of T corresponding to P'_R.

Then, in Step 3, the loop will iterate $m+1$ times for $i = 1$ to $m+1$. In iteration i, the algorithm first forms an edited pattern P' by considering an error at position i and check if P' exists in the text T. The algorithm maintains an invariant that $[s'..e'] = range(T, P'_L)$, where P'_L is the portion of the pattern just left to the error. Using the array F_{st} and F_{ed}, we can compute the interval $[s..e] = range(T, P'_R)$. By the above idea and together with Lemmas 3 and 4, we can get the interval $[st..ed] = range(T, P')$.

Then we have the following theorem.

Theorem 1. *After an $O(n)$ time preprocessing on the text T, an $O(n \log n)$-bit data-structure can be constructed such that the k-difference (k-mismatch) problem with $k = 1$ can be solved in $O(|A| m \log n + occ)$ time.*

Proof. We use the suffix array and the inverse suffix array as the index of T. Their size is of $O(n \log n)$ bits.

We will do a time analysis based on the algorithm in Fig. 1. In the first step we build two arrays F_{st} and F_{ed}. These arrays can be found using backward searching, $F_{st}[i]$ and $F_{ed}[i]$ are updated from $F_{st}[i+1]$ and $F_{ed}[i+1]$ (using Lemma 4) where $F_{st}[m+1] = 0$ and $F_{ed}[m+1] = n$. This can be done in $O(m \log n)$ time.

The algorithm iterates i from 1 to $m+1$. At each iteration, it assumes the error is at position i. The interval $[s'..e']$ is the range of the suffix array of T corresponding to the first half of the pattern just left to the error, that is, $P[1..i-1]$. In the beginning, $[s'..e']$ is set to $[0..n]$. At each step i, there are 3 cases:

1. The character at position i is deleted from the pattern ($1 \le i \le m$). The pattern becomes $P' = P[1..i-1]P[i+1..m]$. We already have $[F_{st}[i+1]..F_{ed}[i+1]] = range(T, P[i+1..m])$ and $[s'..e'] = range(T, P[1..i-1])$, we find $[st..ed] = range(T, P')$. By Lemma 3, this takes $O(\log n)$ time.

1. Construct $F_{st}[1..m+1]$ and $F_{ed}[1..m+1]$, such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$. If the interval $[F_{st}[1]..F_{ed}[1]]$ is valid, then P has exact occurrences in T, we report occurrences in this interval.
2. $s' := 0, e' := n$
3. For $i :=$ 1 to $m+1$
   (a) (deletion at $i \leq m$, ignored for 1-mismatch problem)
      i. $P' = P[1..i-1]P[i+1..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$ and the interval $[F_{st}[i+1]..F_{ed}[i+1]] = range(T, P[i+1..m])$, find $[st..ed] = range(T, P')$.
      iii. Report $[st..ed]$ if exist.
   (b) (replacement at $i \leq m$) for each $c \neq P[i]$ in A
      i. $P' = P[1..i-1]cP[i+1..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$, find $[s''..e''] = range(T, P[1..i-1]c)$.
      iii. Given the interval $[s''..e''] = range(P[1..i-1]c)$ and the interval $[F_{st}[i+1]..F_{ed}[i+1]] = range(T, P[i+1..m])$, find $[st..ed] = range(T, P')$.
      iv. Report $[st..ed]$ if exist.
   (c) (insertion at i, ignored for 1-mismatch problem) for each c in A
      i. $P' = P[1..i-1]cP[i..m]$
      ii. Given the interval $[s'..e'] = range(T, P[1..i-1])$, find $[s''..e''] = range(T, P[1..i-1]c)$.
      iii. Given the interval $[s''..e''] = range(P[1..i-1]c)$ and the interval $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$, find $[st..ed] = range(T, P')$.
      iv. Report $[st..ed]$ if exist.
   (d) Given $[s'..e'] = range(T, P[1..i-1])$, find $[s..e] = range(T, P[1..i])$. If not exist, exit out of the loop.
      $s' := s, e' := e$

Fig. 1. Algorithm for 1-difference and 1-mismatch problems.

2. The character at position i is replaced by each character c in A $(1 \leq i \leq m)$. The pattern becomes $P' = P[1..i-1]cP[i+1..m]$. Having the interval $[s'..e'] = range(T, P[1..i-1])$, we find the interval $[s''..e''] = range(T, P[1..i-1]c)$ in $O(\log n)$ time using Lemma 2. Then we find $[st..ed] = range(T, P')$ from $[s''..e'']$ and $[F_{st}[i+1]..F_{ed}[i+1]]$ using Lemma 3. This takes $O(|A| \log n)$ time.
3. Each symbol c in A is inserted to the pattern at position i. The pattern becomes $P' = P[1..i-1]cP[i..m]$ $(1 \leq i \leq m+1)$. Having the interval $[s'..e'] = range(T, P[1..i-1])$, we find the interval $[s''..e''] = range(T, P[1..i-1]c)$ in $O(\log n)$ time using Lemma 2. Then we find $[st..ed]$ for P' from $[s''..e'']$ and $[F_{st}[i]..F_{ed}[i]]$ using Lemma 3. This takes $O(|A| \log n)$ time.

Reporting occurrences takes $O(occ)$ time. (Actually, this is not true, because an occurrence may be reported multiple times. For example, when $T = aaaa$ and $P = aa$, deleting either the first or the second character of P leads to the same $P' = a$, and

the same set of occurrences is reported each time. Let us ignore this problem now. In Section 4, we will generalize the solution in this section and fix this problem.)

Finally $[s'..e']$ is updated to be the range of the suffix array corresponding to $P[1..i]$ using forward searching (Lemma 2) in $O(\log n)$ time. For an error to appear at i, $P[1..i-1]$ must exist somewhere in T. If $[s'..e']$ does not exist, we exit out of the loop and stop the algorithm.

So the time complexity of the algorithm is $O(m \log n + m(\log n + |A| \log n + |A| \log n + \log n) + occ)$ which is $O(|A|m \log n + occ)$. □

By using compressed suffix array instead of suffix array and inverse suffix array to index T, we get the following corollary.

Corollary 1. *After an $O(n)$ time preprocessing on the text T, an $O(H_0 n)$-bit data-structure can be constructed such that the k-difference (k-mismatch) problem with $k = 1$ can be solved in $O(|A|m \log^2 n + occ \log n)$ time.*

Proof. We use compressed suffix array data structure to index the text T, then from Lemma 5, we achieve the $O(H_0 n)$ bits space. We use the same algorithm as in Theorem 1. As compressed suffix array incurs the penalty of $O(\log n)$ for every access to suffix array and inverse suffix array (Lemma 5) and for every time we use Lemma 2, 3 or 4 and report occurrences, thus from the time analysis in Theorem 1, the time complexity becomes $O(m \log^2 n + m(\log^2 n + |A| \log^2 n + |A| \log^2 n + \log^2 n) + occ \log n)$, which equals $O(|A|m \log^2 n + occ \log n)$. □

4 The k-difference and k-mismatch problems with $k \geq 1$

This section generalizes Theorem 1 for $k \geq 1$. The idea is the same as in Theorem 1.

We also use both the suffix array and the inverse suffix array as the indexing data structure. Instead of just finding 1 error, we try to locate k errors in the pattern P to get an approximation P^* of P which has edit distance at most k from P and report occurrences of P^* in T. By recursion we locate k errors from left to right one by one, with the first error at the leftmost and the k^{th} error at the rightmost.

The algorithm is shown in Fig. 2. It has two main phases. The first phase is to construct $F_{st}[1..m+1]$ and $F_{ed}[1..m+1]$, where $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$ for every $i = 1, 2, \ldots, m+1$. These arrays are used to speed up the search in the second phase.

The second phase executes a recursive function $kapproximate$, which takes five parameters: an interval $[s'..e']$, i, k', a pattern P', and a trace $\mathcal{T}$. The parameters to $kapproximate$ satisfy: (a) P' is an approximation of the prefix $P[1..i-1]$ of P such that $dist(P', P[1..i-1]) \leq k'$; (b) $\mathcal{T}$ is a trace from $P[1..i-1]$ to P'; and (c) $[s'..e'] = range(T, P')$. When $kapproximate$ is called, it recursively introduces at most $k - k'$ errors to $P[i..m]$ to get an approximation having edit distance at most $k - k'$ from $P[i..m]$ and appends this approximation to P' to get P^*, which is an approximation of distance at most k from P. Then, it reports occurrences of P^* in T. In other word, it is used to locate all approximate occurrences of $P'P[i..m]$ in T such that there are at

I Construct $F_{st}[1..m+1]$ and $F_{ed}[1..m+1]$ such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$.
II Call $kapproximate([0..n], 1, 0, \emptyset, \emptyset)$ to find approximate occurrences of P.

$kapproximate([s'..e'], i, k', P', \mathcal{T})$
begin

1. Given $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$ and $[s'..e'] = range(T, P')$, by Lemma 3 find $[st..ed] = range(T, P'P[i..m])$.
2. Report occurrences of $P^* = P'P[i..m]$ in $[st..ed]$ if the interval exists.
3. If $(k' = k)$ then return.
4. For j := i to $m+1$
   (a) /* delete at j, ignored for k-mismatch problem */
   Call $kapproximate([s'..e'], j+1, k'+1, P', d\mathcal{T})$.
   (b) /* replace at j */
   for each $c \neq P[j]$ in A
   i. Given $[s'..e'] = range(T, P')$, by Lemma 2 find $[s''..e''] = range(T, P'c)$.
   ii. Call $kapproximate([s''..e''], j+1, k'+1, P'c, r_c\mathcal{T})$.
   (c) /* insert at j, ignored for k-mismatch problem */
   for each c in A
   i. Given $[s'..e'] = range(T, P')$, by Lemma 2 find $[s''..e''] = range(T, P'c)$.
   ii. Call $kapproximate([s''..e''], j, k'+1, P'c, i_c\mathcal{T})$.
   (d) /* unchange at j */
   Given $[s'..e'] = range(T, P')$, by Lemma 2 find $[s''..e''] = range(T, P'P[j])$.
   $s' := s''; e' := e''; P' := P'P[j]; \mathcal{T} := u\mathcal{T}$;

end

Fig. 2. Algorithm for k-difference and k-mismatch problem

most $k - k'$ errors all in $P[i..m]$. To find all approximate occurrences of P with at most k errors, the second phase executes $kapproximate$ with $[s'..e'] = [0..n]$, $i = 1$, $k' = 0$, $P' = \emptyset$, and $\mathcal{T} = \emptyset$, where $\emptyset$ denotes empty string.

Here, the parameters P' and $\mathcal{T}$ are for the sake of explanation only. In an actual implementation, we do not need them.

$kapproximate([s'..e'], i, k', P', \mathcal{T})$ can be subdivided into two parts. The first part (Steps 1-2 in Fig. 2) finds and reports the interval $[st..ed] = range(T, P^*)$, where $P^* = P'P[i..m]$. Given $[s'..e']$ and $[F_{st}[i]..F_{ed}[i]]$, by Lemma 3, $[st..ed]$ can be found in $O(\log n)$ time.

The second part (Step 4 in Fig. 2) tries to introduce one more error into $P[i..m]$ and call recursively $kapproximate$ to generate all approximations of $P[i..m]$ that have at least one and at most $k - k'$ errors. This part is a loop which iterates j from i to $m+1$. For each j, the algorithm introduces an error at position j. Then, recursively call $kapproximate$ to generate and report more approximations.

From the discussion, we have the following lemma.

Lemma 6. *After an $O(n)$ time preprocessing on the text T, an $O(n \log n)$-bit data structure can be constructed such that the algorithm shown in Fig. 2 solves the k-difference (k-mismatch) problem in $O(|A|^k m^k \log n + outputtime)$ time, where $outputtime$ is the time spent to report occurrences of P^* at Step 2.*

Proof. We use suffix array and its inverse, which occupy $O(n \log n)$ bits of memory, as the indexing data structure of T.

Now we do the time analysis of the algorithm in Fig. 2. In the first phase we build two arrays F_{st} and F_{ed}. By Lemma 2, they can be constructed in $O(m \log n)$ time.

In the second phase, to analyze the time taken by the function $kapproximate$, we measure the total time taken at each step in the function. First of all, we count the number of times the function is called during the recursive algorithm. For each call $kapproximate([s'..e'], i, k', P', \mathcal{T})$, if we let $P^* = P'P[i..m]$ and $\mathcal{T}^* = uu \cdots u\mathcal{T}$ ($\mathcal{T}$ prefixed with $m - i + 1$ u's), then $\mathcal{T}^*$ is the trace from P to P^* that is of cost less than or equal to k.

It is easy to see that different calls to $kapproximate$ correspond to different traces $\mathcal{T}^*$. From Lemma 1, the number of different traces $\mathcal{T}^*$ is $O(m^k|A|^k)$. Therefore there are $O(m^k|A|^k)$ calls to the function $kapproximate$.

Now we evaluate the time taken by Step 1. By Lemma 3, each execution of the step takes $O(\log n)$ time. In total, Step 1 takes $O(|A|^k m^k \log n)$ time throughout the algorithm.

Next, we consider Step 4c. In each call to $kapproximate$, this step is executed at most $m - i + 2$ times. By Lemma 2, each execution takes $O(|A| \log n)$ time. We note that this step is run only if $\mathcal{T}^*$ is of cost less than k, otherwise the algorithm would return at Step 3. From Lemma 1, the number of traces of cost less than k that can be applied on P is $O(|A|^{k-1}m^{k-1})$. Hence, the total time taken by Step 4c throughout the algorithm is $O(|A|^k m^k \log n)$.

The same analysis goes for other Steps 4a, 4b, and 4d. Each of these step totally takes $O(|A|^k m^k \log n)$ time throughout the algorithm.

In total, the running time of the algorithm is $O(|A|^k m^k \log n + outputtime)$. □

The algorithm shown in Fig. 2 may report an approximate occurrence of P in T multiple times. We observe that this happens when one of the following two cases happens: (1) when there are more than one different trace to generate an approximation P^* of P, the algorithm reports the same set of occurrences of P^* for each trace; and (2) when there are two different approximations P_1^* and P_2^* of P, in which P_2^* is a prefix of P_1^*, the set of occurrences of P_1^* is a subset of that of P_2^*, and hence the algorithm reports each occurrence of P_1^* more than one time. Thus, in order to avoid reporting an occurrence multiple times, below discussion improves the algorithm so that it only reports occurrences of a generated pattern P^* when:

(A) the trace we just found to generate P^* is the smallest trace to generate P^* from P; and
(B) P^* contains no prefix P' such that the edit distance between P and P' is less than k.

Then, it is clear that the algorithm's output time is linear with the number of occurrences.

Let $D(i, j, l)$ be the edit distance between the suffix $P[i..m]$ of P and the substring $P[j..l]$ of P, where $1 \leq i \leq m$, $i - 2k \leq j \leq i + 2k$ and $max(j, m - 3k) \leq l < m$. Thus, the table $D(i, j, l)$ has $O(mk^2)$ entries. By a simple form of dynamic programming (see [29]), all entries can be evaluated in $O(mk^2)$ time. Let $E(i, j)$, where $1 \leq i \leq m$ and $i - 2k \leq j \leq i + 2k$, be the minimum value of all entries $D(i, j, l)$, for $max(j, m - 3k) \leq l < m$. Thus, $E(i, j)$ is the minimum edit distance between the suffix $P[i..m]$ of P and some substring starting at position j in P (excluding $P[j..m]$). We can see that having the table D, all $E(i, j)$ entries can be found in $O(mk^2)$ time. We precompute the table E prior to calling the function $kapproximate$ described in Fig. 2. Its usage is explained later.

During the execution of the function $kapproximate$, we generate an approximation of P by appending characters onto it one by one (at Steps 4b, 4c, and 4d). During the process, an $m \times |P'|$ table A is maintained such that $A(i, j) = dist(P[1..i], P'[1..j])$. When we append a character to P' (append a new column to A), the entries in the new column can be evaluated easily using entries in the previous column. Because the entries $A(1, |P'|), \ldots, A(|P'| - k - 1, |P'|)$, and $A(|P'| + k + 1, |P'|), \ldots, A(m, |P'|)$ are always larger than k, we will not interested in and evaluate these entries. Hence, there are only $O(k)$ entries to update: $A(|P'| - k, |P'|), \ldots, A(|P'| + k, |P'|)$. So the cost of maintaining the table A is $O(k)$ time whenever we append a new character to P' at Steps 4b, 4c, and 4d.

At the beginning of each call to $kapproximate$, if we let $P^* = P'P[i..m]$ and $\mathcal{T}^* = uu \cdots u\mathcal{T}$ ($\mathcal{T}$ prefixed with $m - i + 1$ u's), then $\mathcal{T}^*$ is the trace from P to P^*. Thus $\mathcal{T}^*$ is the smallest trace from P to P^* if and only if $\mathcal{T}$ is the smallest trace from $P[1..i - 1]$ to P'.

Maintaining (A) In order to ensure the constraint (A), we need to make sure that $\mathcal{T}$ is the smallest trace from $P[1..i - 1]$ to P'.

To do so, we make the parameters to function $kapproximate$ satisfying an invariant that $\mathcal{T}$ is the smallest trace from $P[1..i - 1]$ to P'. Whenever $\mathcal{T}$ is updated at Steps 4a, 4b, and 4c, and 4d, $\mathcal{T}$ is prefixed with some new operation τ before calling $kapproximate$. We need to do the following checking:

1. $c(\mathcal{T})$ is minimum, and
2. there is no other trace $\mathcal{T}'$ also from $P[1..j]$ to P' such that $c(\mathcal{T}')$ is also minimum and $\mathcal{T}'$ is lexicographically smaller $\mathcal{T}$.

Checking (1) is easy and can be done in $O(1)$ time: a minimum $c(\mathcal{T})$ is equivalent to $c(\mathcal{T}) = A(j, |P'|)$, since $\mathcal{T}$ is a trace from $P[1..j]$ to P'.

For checking (2), since we maintain the invariant, we only need to check that there is no trace $\mathcal{T}'$ from $P[1..j]$ to P' with minimum cost that begins with an operation smaller than τ. We will explain how to do this by examples.

For example, if $\tau = r$ (the most recent operation in $\mathcal{T}$ is replace) and we want to check that there is no trace with minimum cost beginning with d (delete), which is smaller than r. To do so, we check if $A(j - 1, |P'|) + 1 = A(j, |P'|)$. If yes, this means there exists a minimum cost trace from $P[1..j]$ to P' that begins with d, and therefore $\mathcal{T}$ is not the smallest. Otherwise, there is no such trace.

Similarly, checking if there is no trace with minimum cost that begins with u (unchange), which is also smaller than r, is equivalent to checking $A(j-1, |P'|-1) \neq A(j, |P'|)$.

Generalizing this checking for other operations is straightforward. It is easy to see that given the table A, such checking can be done in $O(1)$ time (when $\mathcal{T}$ is updated at Steps 4a, 4b, 4c, and 4d). So the constraint (A) is satisfied.

Maintaining (B) To ensure the constraint (B), that is, $P^* = P'P[i..m]$ contains no prefix of distance $\leq k$ from P, we need to check that (1) P' contains no prefix of distance $\leq k$ from P; and (2) P^* contains no prefix of the form $P'\alpha$, where $\alpha \neq P[i..m]$, that is of distance $\leq k$ from P.

By checking if $A(m, |P'|) \leq k$ every time P' is updated at Steps 4b, 4c, and 4d, we can always check if P' is of edit distance $\leq k$ from P. If it is so, we stop appending character to P' (otherwise, the approximation of P we are going to generate will contain a prefix of distance $\leq k$ from P) and stop continuing the current trace to generate P^*.

Before we report occurrences of $P'P[i..m]$ in Step 2, we need to check that it does not contain any prefix of the form form $P'\alpha$, where $\alpha \neq P[i..m]$ and $\alpha \neq \emptyset$, that is of distance $\leq k$ from P. Hence, $\alpha = P[i..l]$ for some $i \leq l < m$. We observe that if $dist(P'\alpha, P) \leq k$, there is some x such that $dist(P', P[1..x-1]) + dist(\alpha, P[x..m]) \leq k$, which is equivalent to $A(x-1, |P'|) + dist(\alpha, P[x..m]) \leq k$. Hence, we need to check that there are no such x and α.

Suppose that there are such x and α. Since $dist(P[1..x-1], P') \leq k$, we have $|P'| - k \leq x - 1 \leq |P'| + k$. Also, since $dist(P[1..i-1], P') \leq k$, we have $i - 1 - k \leq |P'| \leq i - 1 + k$. Therefore, $i - 2k \leq x \leq i + 2k$.

Also, since $dist(\alpha = P[i..l], P[x..m]) \leq k$, we have $l - i \geq m - x - k$. Since $x \leq i + 2k$, we have $l - i \geq m - i - 3k$, or equivalently, $l \geq m - 3k$. Therefore, this enables us to use the table E described above, and we have $E(x, i) \leq dist(P[i..l], P[x..m]) = dist(\alpha, P[x..m])$.

Hence, for each x in $[i - 2k, i + 2k]$, we check if $A(x-1, |P'|) + E(x, i) \leq k$. If yes, there are such x and α. The total time taken for this checking, given tables A and E, is $O(k)$.

From the discussion, we have the following theorem.

Theorem 2. *After an $O(n)$ time preprocessing on the text T, an $O(n \log n)$-bit data-structure can be constructed such that the k-difference (k-mismatch) problem can be solved in $O(|A|^k m^k \cdot \max(k, \log n) + occ)$ time.*

Proof. The time to construct the table E is $O(mk^2)$ and the time taken by each step in the function $kapproximate$ now becomes $O(|A|^k m^k \cdot \max(k, \log n) + outputtime)$. Since we have avoid reporting duplicate outputs, now we have $outputtime = O(occ)$. From a similar analysis of Lemma 6 and from the above discussion, the theorem follows. □

To achieve linear indexing space, we replace suffix array and inverse suffix array by compressed suffix array to index T and use the same algorithm. Then we have the following result.

Corollary 2. *After an $O(n)$ time preprocessing on the text T, an $O(H_0 n)$-bit data-structure can be constructed such that the k-difference (k-mismatch) problem can be solved in $O(|A|^k m^k \cdot \max(k, \log^2 n) + occ \log n)$ time.*

Proof. We use compressed suffix array data structure to index the text T, then from Lemma 5, we achieve the $O(n)$ bits space. We use the same algorithm as in Theorem 2. As compressed suffix array incurs the penalty of $O(\log n)$ for every access to suffix array and inverse suffix array (Lemma 5) every time we use Lemma 2, 3 or 4 and report occurrences, thus from the time analysis in Theorem 2, the time complexity follows.

□

5 Conclusion

We have described our algorithms for the k-difference and k-mismatch problems which use $O(n \log n)$-bit and $O(n)$-bit indexing space and each query takes $O(|A|^k m^k \cdot \max(k, \log n) + occ)$ and $O(|A|^k m^k \cdot \max(k, \log^2 n) + occ \log n)$ time, respectively. For $k = 1$, although this is not the fastest result in literature, it uses optimal indexing space. For $k \geq 1$, it is the fastest result in literature that uses a moderate amount of indexing space for large n. Another advantage of our solution over other approaches is that our data structures require only linear preprocessing time.As a future work, we would like to improve the query time. More specifically, for $k = 1$, is it possible to achieve linear query time with $\Omega(n \log n)$ indexing space?

References

1. R. Boyer and S. Moore. A Fast String Matching Algorithm. CACM, 20(1977), 762–772
2. D. E. Knuth, J. Morris, V. Pratt. Fast Pattern Matching in Strings. SIAM Journal on Computing 6 (1977), 323–350.
3. G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. J. Algorithms,10:157–169, 1989.
4. A. Amir, M. Lewenstein, Ely. Porat. Faster Algorithms for String Matching with k Mismatches. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 794–803, 2000
5. R. A. Baeza-Yates, G. Navarro. A Faster Algorithm for Approximate String Matching. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching (CPM'96)*, pages 1–23.
6. G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1): 31-88, 2001.
7. E. M. MCreight. A Space Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
8. U. Manber and G. Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
9. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.
10. P. Ferragina, G. Manzini. Opportunistic Data Structures with Applications In *Proc. 41st IEEE Symp. Foundations of Computer Science (FOCS'00)*, pages 390-398, 2000.

11. F. Shi. Fast Approximate String Matching with q-Blocks Sequences. In *Proceedings of the 3rd South American Workshop on String Processing (WSP'96)*, Carleton University Press, 1996.
12. R. A. Baeza-Yates, G. Navarro. A Practical Index for Text Retrieval Allowing Errors. In *CLEI*, volume 1, pages 273–282, November 1997.
13. G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio. Indexing Text with Approximate q-Grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in LNCS, Springer, 2000.
14. G. Navarro, R. A. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. J. of Discrete Algorithms, 1(1):205–239, 2000, 18.
15. E. Sutinen, J. Tarhio. Filtration with q-Samples in Approximate String Matching. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching (CPM'96)*, pages 50–63.
16. A. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proc. 6th Ann. Symp. on Combinatorial Pattern Matching (CPM'95)*, LNCS 807, pages 41–54, 1995
17. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and Dictionary Matching with One Error. In *Proc. 6th WADS*, volume 1663 of LNCS, pages 181-92. Springer-Verlag, 1999.
18. A.L. Buchsbaum, M.T. Goodrich, J. Westbrook. Range Searching Over Tree Cross Products. In *ESA 2000*: pages 120-131
19. R. Cole, L.A. Gottlieb, M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 91–100, 2004.
20. E. Ukkonen. Approximate Matching over Suffix Trees. In *Proc. Combinatorial Pattern Matching 1993*, volume 4, pages 228–242. Springer-Verlag, June 1993.
21. G. Navarro, R. A. Baeza-Yates. A New Indexing Method for Approximate String Matching. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching (CPM'99)*, pages 163–185.
22. K. Sadakane, T. Shibuya. Indexing Huge Genome Sequences for Solving Various Problems. *Genome Informatics* 12:175–183, 2001
23. W. K. Hon, K. Sadakane, W. K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proc. of IEEE Symposium on Foundations of Computer Science*, 2003.
24. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, 1997.
25. P. Jokinen, E. Ukkonen. Two Algorithms for Approximate String Matching in Static Texts. In *Proc. MFCS'91*, Lect. Notes in Computer Science 520 (Springer-Verlag 1991), pages 240–248.
26. G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19-27, 2001
27. D.K. Kim, J.S. Sim, H. Park, K. Park. Linear-Time Construction of Suffix Arrays. In *CPM 2003*: 186-199
28. P. Ko, S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *CPM 2003*: 200-210
29. R.A. Wagner, M.J. Fischer. The String-to-String Correction Problem. J. ACM, 21:168-173, 1974.