On the String Matching with k Differences in DNA Databases

Yangjun Chen
Dept. Applied Computer Science
University of Winnipeg, Canada
y.chen@uwinnipeg.ca

Hoang Hai Nguyen
Dept. Applied Computer Science
University of Winnipeg, Canada
nhhaidee@gmail.com

ABSTRACT

In this paper, we discuss an efficient and effective index mechanism for the string matching with k differences, by which we will find all the substrings of a target string y of length n that align with a pattern string x of length m with not more than k insertions, deletions, and mismatches. A typical application is the searching of a DNA database, where the size of a genome sequence in the database is much larger than that of a pattern. For example, n is often on the order of millions or billions while m is just a hundred or a thousand. The main idea of our method is to transform y to a BWT-array as an index, denoted as $BWT(y)$, and search x against it. The time complexity of our method is bounded by O($k \cdot |T|$), where T is a tree structure dynamically generated during a search of $BWT(y)$. The average value of $|T|$ is bounded by O($|\Sigma|^{2k}$), where Σ is an alphabet from which we take symbols to make up target and pattern strings. This time complexity is better than previous strategies when $k \leq$ O($\log_{|\Sigma|} n$). The general working process consists of two steps. In the first step, x is decomposed into a series of l small subpatterns, and $BWT(y)$ is utilized to speed up the process to figure out all the occurrences of such subpatterns with $\lfloor k/l \rfloor$ differences. In the second step, all the found occurrences in the first step will be rechecked to see whether they really match x, but with k differences. Extensive experiments have been conducted, which show that our method for this problem is promising.

PVLDB Reference Format:
Yangjun Chen and Hoang Hai Nguyen. On the String Matching with k Differences in DNA Databases. PVLDB, 14(6): 903-915, 2021.
doi:10.14778/3447689.3447695

1 INTRODUCTION

By the string matching with k differences, we mean a problem to find all the occurrences of a pattern string $x = x_1x_2 \ldots x_m$ in a target string $y = y_1y_2 \ldots y_n$ with at most k differences, where x_i, $y_j \in \Sigma$, a given alphabet. In general, we distinguish among three kinds of differences:

(1) A character of the pattern corresponds to a different character of the target. In this case we say that there is a mismatch between the two characters;
(2) A character of the target corresponds to "no character" in the pattern (an insertion); and
(3) A character of the pattern corresponds to "no character" in the target (a deletion).

The number of such differences between x and a substring of y: $y_i \ldots y_j$ is called the *Edit Distance*. Then, the problem is to find all i, j such that the *Edit Distance* between x and $y_i \ldots y_j$ is $\leq k$. As an example, consider a target y = *abcdefghi*, a pattern x = *bpdqegh*, and k = 3. There is an occurrence with three differences that starts at the second location of the target (i.e., $y_2 \ldots y_7$ = *bcdefgh*), as shown below.

```
  b p d q e   g h
a b c d   e f g h i
```

In the above alignment, we can see three differences: p to c, q to nothing, and nothing to f, in positions 2, 4, and 6 of the pattern, respectively.

This problem becomes of paramount importance with the advent of DNA databases, where to support the biological research, we need to locate all the appearances of a read (a short DNA sequence obtained by sequencing [38]) in a genome (a massive DNA sequence) for disease diagnosis or some other purposes. Due to polymorphisms or mutations among individuals or even sequencing errors, the read may disagree in some positions at any of its occurrences in the genome. In the past several decades, different methods have been proposed, such as the algorithms discussed in [5, 13, 14, 31, 45, 46, 50, 53, 58]. Most of them are based on the computation of *Levenshtein distance* between $\boldsymbol{x}_i = x_1x_2 \ldots x_i$ and $\boldsymbol{y}_j = y_1y_2 \ldots y_j$ for $i = 1, \ldots, m$, $j = 1, \ldots, n$:

$$d_{i,j} = min\{d_{i-1,j} + w(x_i, \phi), d_{i,j-1} + w(\phi, y_j), \\ d_{i-1,j-1} + w(x_i, y_j)\}, \tag{1}$$

where ϕ represents an empty character, and $w(x_i, y_j)$ the cost to transform x_i into y_j; and use the dynamic programming paradigm to calculate the above formula as scanning y from left to right.

In this paper, we discuss a new method based on the so-called *Burrows-Wheeler* transformation [8, 11, 36], by which y is transformed to an array, called a *BWT-array*, denoted as $BWT(y)$, to replace the scanning of y by searching $BWT(y)$ with the following advantages:

- The positions with the same character in y will be clustered together and the scanning of y character by character will be changed to subset (of characters) by subset. In this sense, y is folded and becomes shorter. Searching such a "folded" and shorter string, we are able to save much time for doing the task.
- All the folded strings searched during the computation can be represented as a tree structure T and we speed up the working process by cutting off a lot of branches in T in two ways: (i) establishing suffix trees over patterns and use them to recognize useless branches when searching T; (ii) identifying similar paths to avoid repeated work.

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3421424.3421427

Theoretically, by using our method, the time complexity can be reduced to O($k \cdot |T|$) with O($n + km$) space requirements. On average, $|T|$ is bounded by O($|\Sigma|^{2k}$). When $k \leq$ O($\log_{|\Sigma|} n$), this running time is better than any existing strategy for this problem. Together with pattern partition, our method can achieve more than 1000-fold improvements.

The organization of the rest of this paper is as follow. First, in Section 2, we summarize all the symbols and notations used throughout the paper. Then, we review the related work in Section 3. Next, in Section 4, we review the classic dynamic programming solution and the *BWT*-transformation as a discussion background. Section 5 is devoted to the description of our algorithm. In Section 6, we describe a pattern partitioning method, which is important to solving problems with large k values. In section 7, we report our experiment results. Finally, a short conclusion is set forth in Section 8.

2 NOTATIONS

In this section, we summarize all the symbols and notations used thoughout the paper, in the following table.

Table 1: Symbols and notations

$\boldsymbol{x}_i$	$\boldsymbol{x}_i = x_1 \ldots x_i$, a prefix of the pattern x
$\bar{x}$	reverse of x, i.e., $\bar{x} = z_1 \ldots z_m = x_m \ldots x_1$
$\boldsymbol{y}_j$	$\boldsymbol{y}_j = y_1 \ldots y_j$, a prefix of the target y
$BWT(y)$	BWT-array of y
$<e, [\alpha, \beta]>$	a range (segment) from rank α to β in F_e
$L_{e,[\alpha,\beta]}$	a range in $L = BWT(y)$, corresponding to $<e, [\alpha, \beta]>$ in F_e
$D[^*, j]$, D_j	the jth column (D-vector) of matrix D
D_v	a D-vector associated with node v
T	search tree, created during the computation
T_P	suffix tree created over reversed pattern $\bar{x}$
$\boldsymbol{s}_v$	a string along the path from *root* to v in T
B_v	shortest suffix of $\boldsymbol{s}_v$ such that there exists a largest i such that the distance between $\boldsymbol{z}_i = z_1 \ldots z_i$ (a prefix of $\bar{x}$) and $\boldsymbol{s}_v$ equals the distance between $\boldsymbol{z}_i$ and B_v
$A_v[i]$	vector used to store the distance between $\boldsymbol{z}_i$ and the shortest suffix of $\boldsymbol{s}_v$

3 RELATED WORK

By the inexact matching, we are required to determine all the substrings in a target having a distance of at most k from a pattern. In terms of different distance functions, we generally differentiate two kinds of inexact matches: string matching with k *differences* (or say k *errors*) and string matching with k *mismatches*. A third kind of inexact matching is with *Don't Care*, or *wild-card* symbols which match any single symbol, including another *Don't Care*.

In the following, we will mainly review some of the noteworthy methods for solving these problems.

k differences As mentioned in the previous section, when the distance function is the Levenshtein distance, the problem is known as the string matching with k differences [13].

Beginning in the early 1980's, genetics and DNA sequence analysis research provided the impetus for advances in the string matching with k differences. The very first O($m{\cdot}n$)-time algorithm was given by Sellers in 1980 [53]. In the same year, Masek and Paterson [43] proposed another O($m{\cdot}n$)-time algorithm with O($n + m$) space requirements. Later, this time complexity has been improved by different researchers. In 1985, Ukkonen proposed a cut-off method based on the diagonalwise monotonicity of dynamic programming matrices with its average time complexity bounded by O($k{\cdot}n$) [58]. The algorithm presented by Landau and Vishkin was similar to the Ukkonen's, but changed the Ukkonen's columnwise computation of values to the diagonalwise [30, 31]. This is asymptotically faster, but less practical since a suffix tree has to be constructed online over a combined string of the pattern and the target. The time complexity of their algorithm is bounded by O($k{\cdot}n$). In 1990 and 1993, Chang and Lawler proposed two k-difference algorithms involving the use of suffix trees over pattern strings to build up the so-called matching statistics [14]. The running time of the first algorithm is bounded by O($k{\cdot}n$) and the second has the same worst-case time as the first, but with a sublinear expected time complexity. The algorithm discussed in [28] is for an extended k-difference problem, by which the 'swap' of characters is used as one of the edit operations. Its time complexity is also bounded by O($k{\cdot}n$). Lastly, the algorithms discussed in [45, 46, 50, 59] are all index-based. In [59], a suffix tree is constructed over target strings and used as an index. Its time complexity is bounded by O($m{\cdot}min(n, m^{k+1}|\Sigma|^{k+1}) + n$) with O($m{\cdot}min(n, m^{k+1}|\Sigma|^{k+1})$) space requirements. In [45], each substring of length log n in y is viewed as a radix-$|\Sigma|$ number, and its first characer is kept in a list of length n, used as an index. It requires O($k{\cdot}n^{f(\varepsilon)}$) time with O($n + m$) space requirements, where $\varepsilon = k/m$ and $f(\varepsilon)$ is a concave function. When ε is small enough, $f(\varepsilon) < 1$. In [46], the index is in essence a set of *inverted lists* (or say inverted files), by which each different substring s of length q (called a q-gram) in y is associated with a list of the form $\{p_1, \ldots, p_j\}$ with each p_i ($i = 1, \ldots, j$) indicating a position where s appears. When a pattern arrives, it will be divided into $k + 1$ pieces and for each piece all the matching q-grams will be found and all the positions in their associated inverted lists will be checked to find all the occurrences of x with k differences using an existing online algorithm. In addition, for constructing inverted lists, the hash technique is used [27]. In the worst case, the running time is bounded by O($k{\cdot}n$) with O($n + m$) space requirements. The method discussed in [50] is similar to [46]. The main difference consists in that $BWT(y)$ is utilized to recognize all the same q-grams when creating inverted lists. So, it has the same computatinal complexities as [46].

From the above description, we can see that no strategy up to now is able to break the bottleneck of O($k{\cdot}n$) running time. But our method runs in O($k{\cdot}|\Sigma|^{2k}$) time and is the first one to bring down O($k{\cdot}n$) by orders of magnitude.

Besides the algorithms described above, where the entire target string is available from the very beginning, there are many streaming algorithms for the same problem [1, 2, 10, 12, 19, 54], but for a different situation that it is the pattern available at the beginning while the target comes in a stream, one character at a time, such as telecommunication and Internet traffic, as well as establishing firewall to block virus and malware. Since no index over target strings can be constructed, the technique developed is quite different.

k mismatches When the distance function is the *Hamming* distance, the problem is known as the string matching with k mismatches [4, 33]. By the Hamming distance, we count the number of differences between x and the corresponding substring s in y. In comparison with k differences, the string matching with k mismatchings is less error tolerant. Also, many algorithms have been proposed for this problem, such as those online strategies reported in [4–6, 24, 32, 33, 47, 56, 57]. Among them, all the methods discussed in [5, 32, 56, 57] need $O(m \cdot n)$ time, while in all the other methods, either *mismatching information* or *periocity* in patterns is utilized in different ways to speed up computation. In [6], a method called the *shift-add* is discussed, by which the mismatching information is represented as bit strings. When patterns are so small that $m \cdot \log m$ is smaller than the system word-size, the method is efficient. For large patterns, however, the multiple-precision arithmetic operations are required for preprocessing x, and the running time then becomes quadratic. In the methods discussed in [24] and [33], the mismatching information is also precomputed, but stored in a table. The time complexity of their algorithms are bounded by $O(k \cdot n + m \cdot \log m)$. The methods discussed in [4, 47] work differently. Instead of mismatching information, they use the periodicity within a pattern, and achieve a better running time $O(n \cdot \log k)$. The method proposed in [48] is for a spacial case, where only the mismatching at borders is considered. Finally, the algorithms discussed in [36] and [16, 17] are both index-based, by which y is transformed to $BWT(y)$, used as an index. In [16, 17], the mismatch information for x is employed while in [36] not. The time complexity of [16, 17] is reduced to $O(k \cdot n' + n + m \cdot \log m)$, where n' is the number of leaf nodes of a *mismatching tree*, produced during a search of $BWT(y)$. In the case of $m \geq 2(k + 1)$, the average value of n' is bounded by $O((1 + 1/|\Sigma|)^{k+1})$. The running time of [36] is bounded by $O(m \cdot n'' + n)$, where n'' is also the number of leaf nodes of a tree produced during the search of $BWT(y)$, but bounded by $O(n)$ in the worst case. If m is large, it can be worse than all those online methods discussed in [4, 24, 33, 47]. A third index-based method is based on a Brute Force searching of the suffix tree over y [20]. Its time complexity is bounded by $O(m + n + (c \log n)k/k!)$, where c is a very large constant. When n is large and k is larger than a certain constant, this method can also be worse than an online strategy. Finally, in [37], the mismatching algorithms are used to evaluate the mapping qualities of reads to a genome sequence.

don't care As a different kind of inexact matching, the string matching with *don't cares* (*wild cards*) has been a third active research topic for decades. A wild card matches any character and we may have wild-cards in x, in y, or in both of them. Due to *don't cares*, the 'match' relation is no longer transitive, which precludes straightforward adaption of the *shift* information used by Knuth-Morris-Pratt [29] and Boyer-Moore [9]. In general, we need quadratic time to solve the problem [51]. According to [41], however, using the suffix array for y as an index the searching time can be reduced to $O(\log n)$ for some patterns, which contain only a sequence of consecutive *don't cares*.

Finally, we point out that since Knuth-Morris-Pratt [29], many efficient algorithms for the exact string matching have been proposed such as those described in [3, 9, 21–23, 26, 34, 39, 42, 44, 52, 55, 59, 61]. However, none of them can be extended or modified for the string matching with k differences since the most important mechanism for the exact string matching, the so-called *failure links* built up for patterns to avoid repeated access of characters in targets cannot be established for the inexact matching.

4 BACKGROUND

In this section, we briefly describe two techniques, based on which our method is established. One is *dynamic programming*, which is a basic paradigm used for solving the string matching with k differences [53]. The other is the so-called Burrows-Wheeler (BWT) transformation originally proposed for text compression [11, 49, 63], but can also be employed for solving the classical string matching problem [15, 18, 36].

4.1 Dynamic programming

The string matching with k differences can be easily solved by using dynamic programming with no preprocessing. Let $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_n$. Denote by $\boldsymbol{x}_i = x_1 x_2 \ldots x_i$ a prefix of x and by $\boldsymbol{y}_j = y_1 y_2 \ldots y_j$ a prefix of y. We represent the Levenshtein distance between $\boldsymbol{x}_i$ and $\boldsymbol{y}_j$ by $D(i, j)$, which can be computed by constructing a $(m + 1) \times (n + 1)$ matrix D according to the following formulae:

$$D(0, j) = j \quad (0 \leq j \leq n);\ D(i, 0) = 0 \quad (0 \leq i \leq m);$$

$$D(i, j) = min \begin{cases} D(i-1, j) + w(x_i, \phi) \\ D(i-1, j-1) + \delta(x_i, y_j) \\ D(i, j-1) + w(\phi, y_j) \end{cases} \tag{2}$$

where $\delta(x_i, y_j) = 0$ if $x_i = y_j$; otherwise, $\delta(x_i, y_j) = w(x_i, y_j)$, representing the cost to change x_i to y_j [58].

We can design a simple process to compute the entries in D column by column when scanning y from left to right, which involves comparisons between the current target character and every pattern character. For example, for $x = gcaca$ and $y = acatatg$, we will generate a matrix as illustrated in Table 2. By the computation, we set $w(x_i, y_j) = 0$ if $x_i = y_j$; otherwise, $w(x_i, y_j) = 1$. Once the final value in a column j (i.e., $D[m, j]$) has been evaluated, it is compared with k. If, and only if, it does not exceed k, then at the current target position j at least one approximate occurrence of the pattern with no more than k differences is found. For instance, if k is set to be 2, checking the last row of Table 2, we will find two occurrences of x in y, ending at positions 3, and 5, respectively.

In the subsequent discussion, we will refer to the columns of D as the D-vectors , and use D_j to refer to the jth vector in D. That is, $D_j = D[^*, j]$.

4.2 BWT and string matching

The Burrows-Wheeler (BWT) array is closely related to the suffix array [42]. To know what it is, let us consider a string $y = gtataca$. We show all its suffixes in column 1 of Table 3, sorted suffixes in column 2, and the corresponding suffix array SA_y in column 3, which contains the positions of all the sorted suffixes' first character in y.

Table 2: Matrix D

	j	0	1	2	3	4	5	6	7
i			$a$	c	$a$	t	$a$	t	g
0		0	0	0	0	0	0	0	0
1	g	1	1	1	1	1	1	1	0
2	c	2	2	1	2	2	2	2	1
3	$a$	3	2	2	1	2	2	3	2
4	c	4	2	2	2	2	3	3	3
5	$a$	5	4	3	**2**	3	**2**	3	4

Table 3: Suffixes, suffix array, BWT array, and ranks

suffix	sorted suffix	SA_y	r_F	F	sorted rotations	L	r_L
$gtataca\$$	$\$$	7	-	$\$$	$\$g_1t_1a_1t_2a_2c_1a_3$	$a$	1
$tataca\$$	$a\$$	6	1	$a$	$a_3\$g_1t_1a_1t_2a_2c_1$	c	1
$ataca\$$	$aca\$$	4	2	$a$	$a_2c_1a_3\$g_1t_1a_1t_2$	t	1
$taca\$$	$ataca\$$	2	3	$a$	$a_1t_2a_2c_1a_3\$g_1t_1$	t	2
$aca\$$	$ca\$$	5	1	c	$c_1a_3\$g_1t_1a_1t_2a_2$	$a$	2
$ca\$$	$gtataca\$$	0	1	g	$g_1t_1a_1t_2a_2c_1a_3\$$	$\$$	-
$a\$$	$taca\$$	3	1	t	$t_2a_2c_1a_3\$g_1t_1a_1$	$a$	3
$\$$	$tataca\$$	1	2	t	$t_1a_1t_2a_2c_1a_3\$g_1$	g	1

Here, we assume that y terminates with a special character \$, which does not appear elsewhere in y and is alphabetically prior to all other characters. In the case of DNA sequences, we have $\$ < a < c < g < t$.

By using the suffix array SA_y of y, its *BWT-array*, denoted as $BWT(y) = L$, is defined as follows

$$\begin{cases} L[i] = \$, & \text{if } SA_y[i] = 0; \\ \\ L[i] = y_{SA_y[i]-1}, & \text{otherwise.} \end{cases} \qquad (3)$$

By applying (3) to $y = gtataca$ and the corresponding suffix array, we immediately get $BWT(y)$ as shown in the 7th column of Table 3. This transformation process is referred to as the *BWT transformation*.

The generation of $BWT(y)$ can also be described in a different way, which is a little tedious, but enables us to observe why it can be used to speed up the string matching.

To produce $BWT(y)$, we first rotate y consecutively to create $|y|$ different strings, sort them lexicographically. Then, write them stacked vertically as shown in the 6th column of Table 3, where each character is subscripted to represent its position in the original y. (That is, we rewrite y as $g_1t_1a_1t_2a_2c_1a_3$.) For example, a_2 represents the second appearance of $a$ in y; and t_1 the first appearance of t in y. In the same way, we can check all the other appearances of different characters.

By a close check of these sorted strings, we can remark the following, which is important to the string matching:

- L is identical to the array made up of the last characters of those sorted strings. (To see this, compare column 7 and the last characters of the sorted strings in column 6.)
- Denote by F the array made up of the first characters of the sorted strings (which is identical to the 5th column of Table 3.) When ranking the elements e in both F and L in such a way that if e is the ith appearance of a certain character it will be assigned i, the same element will get the same number in both arrays. For example, in F the rank of a_3, denoted as $r_F(a_3)$, is 1 (showing that a_3 is the first appearance of $a$ in F). Its rank in L, $r_L(a_3)$ is also 1. (To see this, compare column 4 and column 8.) This property: $r_F(e) = r_L(e)$ holds for all the elements e in y.
- The whole F can be divided into 5 segments with each containing only the same characters, denoted as $F_\$$, F_a, F_c, F_g, and F_t, respectively.

In addition, to describe how the string matching can be conducted based on the above facts, we need yet another two notations:

- $<e, [\alpha, \beta]>$ ($e \in \{a, c, g, t\}$, $\alpha \leq \beta$ are two ranks), representing a range (or say, a subsegment) in F_e. For example, $<t, [1, 2]>$ represents $F[6 .. 7]$.
- $L_{<e,[\alpha,\beta]>}$, representing a range in L corresponding to $<e, [\alpha, \beta]>$. For example, in the F and L shown in Table 3, we have $L_{<a,[1,3]>} = L[1 .. 3]$, $L_{<t,[1,2]>} = L[6 .. 7]$, $L_{<a,[2,3]>} = L[2 .. 3]$, and so on. (Note that F and L are the arrays respectively made up of the first and last characters of the sorted strings shown in column 6.)

During the search of $L = BWT(y)$, in each step, two operations will be performed: (i) searching within a certain $L_{<e,[\alpha,\beta]>}$ to find the first and the last appearance of some character e' (represented by their ranks α' and β', respectively); (ii) in terms of $<e', (\alpha', \beta')>$, figure out $L_{<e',[\alpha',\beta']>}$. Since the second operation corresponds to a step of scanning in y, but in the reverse direction, we need to work on the pattern x in the reverse order (referred to as a *backward search*). That is, we will search $\bar{x}$ (reverse of x) against $BWT(y)$. We denote such a searching step by $S(e', <e, [\alpha, \beta]>)$ defined below:

$$S(e', < e, [\alpha, \beta] >) = \begin{cases} < e', [\alpha', \beta'] >, & \text{if } e' \in L_{<e,[\alpha,\beta]>}; \\ \\ \phi, & \text{otherwise.} \end{cases} \qquad (4)$$

The following example helps for illustration.

Example 4.1. Let $x = tata$ and $y = gtataca$. To find all the occurrences of x in y, we will search $\bar{x}$ against L and F generated for y (shown in Table 3) as below.

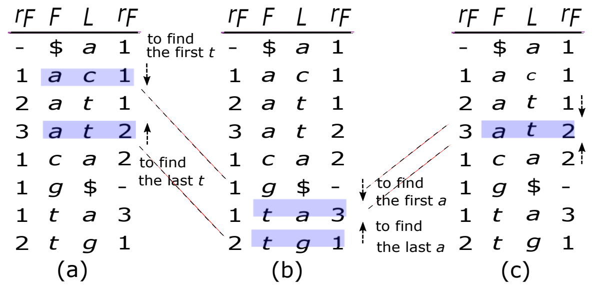


Figure 1: Illustration of backward search.

Step 1: Check $x_4 = a$ in x, and then figure out $F_a = F[1 .. 3] = <a, [1, 3]>$. Corresponding to $<a, [1, 3]>$, we can find $L_{<a,[1,3]>} = L[1 .. 3]$.

Step 2: Check $x_3 = t$. Call $S(t, <a, [1, 3]>)$, by which $L_{<a,[1,3]>} = L[1 .. 3]$ will be searched to find the first and last appearances of t

within $L[1 .. 3]$ (see Fig. 1(a) for illustration). We will get <t, [1, 2]>, corresponding to which we can then find $L_{<t,[1,2]>} = L[6 .. 7]$.

Step 3: Check $x_2 = a$. Call $S(a$, <t, [1, 2]>), by which $L_{<t,[1,2]>} = L[6 .. 7]$ will be searched to find the first and last appearance of a (see Fig. 1(b)). This time, we will get <a, [3, 3]>, and $L_{<a,[3,3]>} = L[3 .. 3]$.

Step 4: Check $x_1 = t$. Call $S(t$, <a, [3, 3]>), by which $L_{<a,[3,3]>} = L[3 .. 3]$ will be searched to find the first and last appearances of t (see Fig. 1(c)). So, we will get <t, [2, 2]> = $F[7 .. 7]$. Since now we have exhausted all the characters in x and $F[7 .. 7]$ contains only one element, one occurrence of x in y is found. It is the substring beginning at t_1 in y (note that $r_F(t_1) = 2$. See column 6 in Table 3.)

Let $\bar{x} = z_1 \ldots z_m$. The general working process can be represented as a sequence of the following form:

<z_1, [α_1, β_1]>, <z_2, [α_2, β_2]>, . . . , <z_m, [α_m, β_m]>,

where <z_1, [α_1, β_1]> = F_{z_1}, and <z_i, [α_i, β_i]> = $S(z_i$, <z_{i-1}, [α_{i-1}, β_{i-1}]>) for $i \in \{2, \ldots, m\}$.

We call such a sequence a *searching sequence*. For example, the working process described in Example 1 can be represented as a searching sequence shown below:

<a, [1, 3]>, <t, [1, 2]>, <a, [3, 3]>, <t, [2, 2]>.

Denote by c_i the cost of $S(z_i$, <z_{i-1}, [α_{i-1}, β_{i-1}]>). Then, the time complexity of this process is obviously bounded by $O(\sum_{i=1}^{m} c_i)$. However, by using a technique *rankAll* discussed in [11, 17, 25], we can reduce $O(c_i)$ to $O(1)$ for $i = 1, \ldots, m$, at cost of more space usage. (Concretely, $|\Sigma|$ times the space will be used.) This time complexity even beats the quantum string matching [40].

5 ALGORITHM DESCRIPTION

In this section, we discuss our algorithm based on the BWT transformation. First, we give a basic algorithm for the string matching with k differences in Section 5.1. Then, we describe how to improve this algorithm in Section 5.2. Finally, we analyze the computational complexities in Section 5.3.

5.1 Basic algorithm

Different from the evaluation of an exact string matching, to find all the occurrences of $x = x_1 \ldots x_m$ in $y = y_1 \ldots y_n$ with k differences, a tree, instead of a single sequence, will be dynamically created. In such a tree, each path

$$v_0 \rightarrow v_1, \ldots \rightarrow v_l,$$

corresponds to a searching sequence with each v_j ($0 \leq j \leq l$) being labeled with a pair of the form <e_j, [α_j, β_j]> as described in Section 4.2. In addition, for each v_j ($0 \leq j \leq l$), a D-vector will be generated by checking e_j against every character z_i in $\bar{x} = z_1 \ldots z_m$ ($= x_m \ldots x_1$) in the same way as dynamic programming. First, we will set $D_{v_0} = \langle 0, 1, ..., m\rangle^{\top}$ for v_0. Then, for each node v_j ($1 \leq j \leq l$) along the path, D_{v_j} will be calculated from $D_{v_{j-1}}$ by using formulae (5), which are almost the same as formulae (2). Only the inital values are set differently. It is because by formulae (5), we compute the distance between $z_1 \ldots z_i$ ($i = 1, \ldots, m$) and a substring of $\bar{y}$ along a *root-to-leaf* path in the tree (instead of a prefix of $\bar{y}$). More importantly, we have multiple choices at each step to match characters in a pattern. That is why the working process has to be represented by a tree, rather than a single path.

$$\begin{cases} D_{v_j}[0] = D_{v_{j-1}}[0] + 1; \\ \\ D_{v_j}[i] = min\{D_{v_j}[i-1] + w(z_i, \phi), D_{v_{j-1}}[i] + w(\phi, e_j), \\ \qquad\qquad D_{v_{j-1}}[i-1] + \delta(z_i, e_j)\}, \text{ for } \ i > 0. \end{cases} \tag{5}$$

So, we have the concept of *search trees* defined below.

Definition 5.1. (*Search trees*) A search tree (S-tree for short) T with respect to x and y is a tree structure to represent the search of $\bar{x}$ against $BWT(y)$. In T, each node is labeled with a pair <e, [α, β]> and there is an edge from v (= <e, [α, β]>) to u (= <e', [α', β']>) if $S(e', v) = u$. In addition, a special node is designated as the *root*, labeled with <-, [1, $|L|$]>, representing the whole BWT-array $L = BWT(y)$.

As an example, consider $x = acacg$, $y = gtataca$. Assume that $k = 2$. To find all the occurrences of x in y with up to 2 differences, a search tree T as shown in Fig. 2 will be created when checking $\bar{x} = gcaca$ against $BWT(y) = L$ and F (remember that they are shown in column 7 and 5 of Table 3, respectively.)

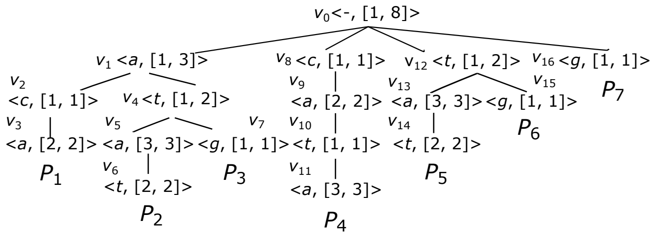


Figure 2: A search tree.

In Fig. 2, v_0 is the root, representing the whole L. Then, to create all its children, we will search L to find the first and last appearance of e for each $e \in \Sigma$. As with dynamic programming, each time a node v is created, its D-vector will be computed. Recall that the D-vector for v_0 is $\langle 0, 1, \ldots, m\rangle^{\top}$. Subsequently, for any node u different of the root, its D-vector D_u is calculated from D_v, the D-vector of its parent v, in terms of formulas (5). For example, by exploring path $P_1 = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$ in Fig. 2, we will generate a searching sequence as below:

<-, [1, 8]>, <a, [1, 3]>, <c, [1, 1]>, <a, [2, 2]>.

All their D-vectors (and also all the other nodes' vectors in the tree) are shown in Fig. 3, in which we use D_i to represent D_{v_i} for simplicity.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}
-	*a*	*c*	*a*	*t*	*a*	*t*	*g*	c	*a*	*t*	*a*	*t*	*a*	*t*	*g*	*g*
0	1	2	3	2	3	4	3	1	2	3	4	1	2	3	2	1
1	1	2	3	2	3	4	2	1	2	3	4	1	2	3	1	0
2	2	1	2	2	3	4	3	1	2	3	4	2	2	3	2	1
3	2	2	1	3	2	3	3	2	1	2	3	3	2	3	3	2
4	3	2	2	3	3	3	4	3	2	2	3	4	3	3	4	3
5	4	3	**2**	4	3	4	4	4	3	3	**2**	5	4	4	5	4

Figure 3: D-vectors.

In essence, T is generated in the depth-first fashion, but the expansion of a path stops at a node v (= <e, [α, β]>) when one of four conditions is satisfied:

(1) The last entry in v's D-vector, i.e., $D_v[m]$, is $\leq k$. It shows that some occurrences of the pattern with k or less than k diffrences have been found. We will then backtrack to explore another path;
(2) $L_v = L_{<e,[\alpha,\beta]>}$ contains only one element '\$', showing that the path can not be further explored;
(3) All the entries in D_v are $> k$, showing that no answer can be found in the subtree rooted at v;
(4) The length of the path from *root* to v is equal to $m + k$ since the edit distance between x and any string of length $> m + k$ must be $> k$.

For example, for v_3 (on P_1 shown in Fig. 2) we have $D_3[5] = 2$ (see Fig. 3), which shows that some occurrences of the pattern with 2 diffrences have been found. In fact, only one occurrence is found since the interval [2, 2] (in the pair <a, [2, 2]>) associated with v_3 contains only one value. So, the search along P_1 stops and we backtrack to generate another child v_4 of v_1 and proceed continually. The same analysis applies to v_{11} (on P_4). In addition, we can observe that three paths P_3, P_6 and P_7 stop at a node v such that L_v contains only '\$', and two paths P_2 and P_5 terminate at a node with all the entries in its D-vector $> k = 2$.

In terms of the above discussion, we give the following formal description of our basic algorithm, in which two extra notations are used:

s_v - a string made up of the characters along the path from *root* to node v;

stack - a stack used to control the generation of T in the depth-first fashion, in which each entry is a pair of the form (v, u) with v being the parent of u. For *root*, its parent is set *empty*, represented as Φ.

Algorithm 1: *stringM*($\bar{x}$, $BWT(y)$, k)

```
Input  : $\bar{x}$ - reverse of $x$, $BWT(y)$ - BWT-array of $y$, $k$ - number of errors
Output : marking all occurrences of $x$ in $y$
1  create root of T; push(stack, ($\Phi$, root));
2  while stack is not empty do
3      (v, u) := pop(stack);
4      if $|s_u| \leq |x| + k$ then
5          let $u = <e, [\alpha, \beta]>$;
6          compute $D_u$ from $D_v$ by using formulas ( 5);
7          if not all entries in $D_u > k$ then
8              if $D_u[|x|] \leq k$ then
9                  mark the corresponding occurrences;
10             end
11             else
12                 for each $e' \in \Sigma$ do
13                     $w := S(e', <e, [\alpha, \beta]>)$;
14                     push(stack, (u, w));
15                 end
16             end
17         end
18     end
19 end
```

In the above algorithm, we first create the *root* of T, and put it into *stack* (see line 1). Next, we go into a **while**-loop. At each iteration of the **while**-loop, we will pop the top element $u = <e, [\alpha, \beta]>$ out of *stack*. If $|s_u| \leq |x| + k$, compute D_u from its parent's vector D_v (see line 6. But for *root*, its D-vector D_{root} is simply set to $<0, 1, \ldots, m>^{\top}$.) If in D_u there are some entries $\leq k$, we will check whether $D_u[|x|] \leq k$. If it is the case, we will mark the corresponding occurrences (see line 9). Otherwise, we will search $L_{<e,[\alpha,\beta]>}$ to find $w = <e', [\alpha', \beta']>$ for each $e' \in \Sigma$ by calling $S(\ )$, where α' and β' are respectively the ranks of the first and last appearances of e' (see lines 11 - 12). Then, push it into *stack* (see line 13). Otherwise, nothing will be done. This process continues until *stack* becomes empty.

Example 5.2. In this example, we run the above algorithm on $\bar{x}$ = *gcaca* against *BWT(y)* = L and F shown in Table 3 with $k = 2$, where y = *gtataca*\$; and demonstrate its first seven steps. The complete tree generated is shown in Fig. 2.

Step 1: create root v_0 =<-, [1, 8]>, push (Φ, v_0) into *stack*. (See Fig. 4(a).)

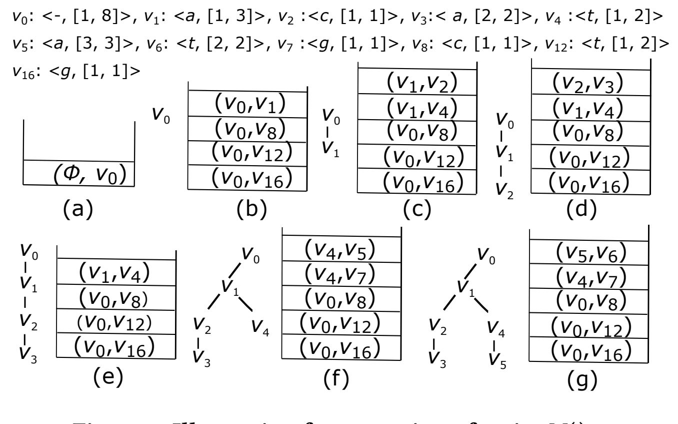
Step 2: Pop out the top element (Φ, v_0). Generate $D_0 = <0, 1, 2, 3, 4, 5>^{\top}$ for v_0. For each $e' \in \Sigma$ $S(e', <-, [1, 8]>)$ will be executed, by which v_1 = <a, [1, 3]>, v_8 = <c, [1, 1]>, v_{12} = <t, [1, 2]>, and v_{16} = <g, [1, 1]> will be generated. Then, all (v_0, v_i) (i = 1, 8, 12, 16) will be pushed into *stack*. (See Fig. 4(b).)

Step 3: Pop out the top element (v_0, v_1). v_1 = <a, [1, 3]>. Generate $D_1 = <1, 1, 2, 2, 3, 4>^{\top}$ (see Fig. 3.) For each $e' \in \Sigma$ $S(e', <a, [1, 3]>)$ will be executed, by which v_2 = <c, [1, 1]> and v_4 = <t, [1, 2]> will be generated and pushed into *stack*. (See Fig. 4(c).)

Step 4: Pop out the top element (v_1, v_2). v_2 = <c, [1, 1]>. Generate $D_2 = <2, 2, 1, 2, 2, 3>^{\top}$. For each $e' \in \Sigma$ $S(e', <c, [1, 1]>)$ will be executed, by which v_3 = <a, [2, 2]> will be created and pushed into *stack*. (See Fig. 4(d).)

Step 5: Pop out the top element (v_2, v_3). v_3 = <a, [2, 2]>. Generate $D_3 = <3, 3, 2, 1, 2, 2>^{\top}$. Since $D_{v_3}[5] = 2$, an occurrence of the pattern has been found. Nothing will be pushed into *stack* and we backstrack. (See Fig. 4(e).)

Step 6: Pop out the top element (v_1, v_4). v_4 = <t, [1, 2]>. Generate $D_4 = <2, 2, 2, 3, 3, 4>^{\top}$. For each $e' \in \Sigma$ $S(e', <t, [1, 2]>)$ will be executed, by which v_5 = <a, [3, 3]> and v_7 = <g, [1, 1]> will be created and pushed into *stack*. (See Fig. 4(f).)



v_0: <-, [1, 8]>, v_1: <a, [1, 3]>, v_2 :<c, [1, 1]>, v_3:< a, [2, 2]>, v_4 :<t, [1, 2]>
v_5: <a, [3, 3]>, v_6: <t, [2, 2]>, v_7 :<g, [1, 1]>, v_8: <c, [1, 1]>, v_{12}: <t, [1, 2]>
v_{16}: <g, [1, 1]>
(Φ, v_0)
(v_0,v_1)
(v_0,v_8)
(v_0,v_{12})
(v_0,v_{16})
(v_1,v_2)
(v_1,v_4)
(v_2,v_3)
(v_4,v_5)
(v_4,v_7)
(v_5,v_6)
(a)
(b)
(c)
(d)
(e)
(f)
(g)

Figure 4: Illustration for execution of *stringM*( ).

Step 7: Pop out the top element (v_4, v_5). v_5 = <a, [3, 3]>. Generate D_5 = <3, 3, 3, 2, 3, 3>$^\top$. For each $e' \in \Sigma$ $S(e'$, <a, [3, 3]>) will be executed, by which v_6 = <t, [2, 2]> will be created and pushed into *stack*. (See Fig. 4(g).)

5.2 Improvements

The basic algorithm described above can be greatly improved by cutting off tree branches in two different ways: one is to establish a suffix tree T_P over $\bar{x}$ and replace searching some parts of T with searching few paths in T_P, but with no need to calculating D-vectors; the other is to recognize similar paths, besides the stopping conditions listed in 5.1, to cut off branches to avoid some repeated work.

5.2.1 Searching suffix trees over patterns to replace searching part of T . Let v be the node currently created. Assume that there exists some $0 < j < m$ such that $D_v[j] = k$ while all the other entries in $D_v > k$. Then, adding a suffix of $\bar{x}$: $\boldsymbol{z}^j = z_{j+1} \dots z_m$ gives a string $\boldsymbol{s}_v \circ \boldsymbol{z}^j$ whose distance from $\bar{x}$ is k, where '$\circ$' represents the concatenation operation over substrings. If such a substring can be found in y, it is exactly an occurrence of x in y. Thus, we need only to check whether it is possible to explore a single path (in T) labeled with $\boldsymbol{z}^j$. For example, in the tree shown in Fig. 2, the D-vector of v_4 is <2, 2, 2, 3, 3, 4>$^\top$ (see Fig. 3), and then the possible extentions are the suffixes *caca* and *aca* of $\bar{x}$ = *gcaca*. Therefore, the subtree rooted at v_7 (labeled with <g, [1, 1]>) should not be explored since $g \notin \{a, c\}$ (note that c is the first character of *caca* while a is the first character of *aca*.) In fact, we will only explore the subpath starting with 'a' (path P_2) 'since 'c' cannot be found when searching $L_{<t,[1,2]>}$. Even this path (P_2) will break off when we meet 't' which does not appear in *aca*. Assume that $j_1, \dots, j_h$ ($h > 0$) are all the entries in the D-vector D_v for the current node v such that $D_v[j_i] = k$ ($1 \le i \le h$) while all the other entries $> k$. Then, we can organize all the suffixes of $\bar{x}$: $\boldsymbol{z}^{j_1}, \dots, \boldsymbol{z}^{j_h}$ into a tree (forest) and use it to control the exploration of $T[v]$ in such a way that each searched path is exactly along a certain $\boldsymbol{z}^{j_i}$ ($1 \le i \le h$) if any. More importantly, in this process, no D-vectors need to be calculated. For efficiency, however, it is better to construct a suffix tree T_P for the whole $\bar{x}$ when x arrives. Then, we can associate T_P with an extra array C of length m, in which each entry $C[i]$ is a pointer to a leaf node of T_P, locating the suffix starting at position i of $\bar{x}$, (as illustrated in Fig. 5, where we show the suffix tree for $\bar{x}$ = *gcaca* and its associated array.)

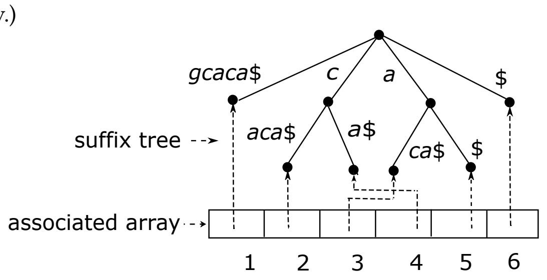


Figure 5: Illustration for the suffix tree over a pattern.

Therefore, each time we encounter a node v with some entries in D_v eqaul to k and all the other entries $> k$, we will explore T_P as follows.

- Let $D_v[j_1], \dots, D_v[j_h]$ be all those entries equal to k. We will search T_P bottom-up, starting from all those leaf nodes pointed to by $C[j_1], \dots, C[j_h]$, respectively. In this process, each encountered edge will be marked.
- Search T_P once again, but top-down and access only the marked edges. Assume that v is labeled with a pair <z_j, $[\alpha_j, \beta_j]$> for some j. Then, corresponding to the suffix of $\bar{x}$: $z_{j+1} \dots z_m$ along a path in T_P, a search sequence will be generated, as shown below:

  <z_{j+1}, $[\alpha_{j+1}, \beta_{j+1}]$>, . . . , <$z_{m'}$, $[\alpha_{m'}, \beta_{m'}]$>,

  where $m' \le m$, and <z_i, $[\alpha_i, \beta_i]$> = $S(z_i$, <z_{i-1}, $[\alpha_{i-1}, \beta_{i-1}]$>) for $i \in \{j+1, \dots, m'\}$.

Such a path stops when we have exhausted the whole suffix, or at a node <z_i, $[\alpha_i, \beta_i]$> such that $S(z_{i+1}$, <z_i, $[\alpha_i, \beta_i]$>) returns ϕ.

5.2.2 Recognizing similar paths. The second improvement utilizes in a crucial way a technique originally used by Ukkonen in [59] for handling *suffix links*. But we have changed it for our own purpose.

Let v be a node in T. Let $\boldsymbol{s}_v = e_1 \dots e_j$. Denote by B_v the shortest suffix of $\boldsymbol{s}_v$ satisfying a condition that there exists a largest i such that the distance between $\boldsymbol{z}_i = z_1 \dots z_i$ (a prefix of $\bar{x}$) and B_v is $\le k$ and equals the distance between $\boldsymbol{z}_i$ and $\boldsymbol{s}_v$. Then we check whether $|B_v| = |\boldsymbol{s}_v|$. If it is the case, continue. Otherwise, we stop the current path and backtrack. Two questions need to be answered. First, how to figure out B_v? Second, why can we use this condition to terminate a path expansion? To compute B_v, we need to trace the generation of each entry in D_v. In terms of formula (5), each entry $D_v[i]$ ($1 \le i \le m$) is produced possibly in one of three ways: (I) $D_v[i] = D_v[i - 1] + 1$, (II) $D_v[i] = D_u[i - 1] + \delta(z_i, e_j)$, and (III) $D_v[i] = D_u[i] + 1$, where u stands for the parent of v. But 7 cases should be recognized. In case (1), $D_v[i]$ is produced either in (I), in (II) or in (III). That is, all the three expressions $D_v[i - 1] + 1$, $D_u[i - 1] + \delta(z_i, e_j)$, and $D_u[i] + 1$ evaluate to the same value. In case (2), only the first two evaluate to the same value while the third is larger than them. In case (3), the first and the third are the same while the second is larger. In the same way, we can recognize all the remaining four cases. We summarize the 7 cases in Table 4.

Table 4: 7 cases

	I	II	III
1	✓	✓	✓
2	✓	✓	
3	✓		✓
4	✓		
5		✓	✓
6		✓	
7			✓

Table 5: Matrix A

i	j	0	1 a	2 c	3 a	4 t
1	g	0	0	0	0	0
2	c	0	0	1	0	0
3	a	0	1	1	2	3
4	c	0	1	2	2	3
5	a	0	1	2	3	3

In terms of these 7 possibilities, we present the following reccurrence to calculate the length of the shortest suffix of s_v whose distance from $\boldsymbol{z}_i = z_1 \dots z_i$ is $D_v[i]$. Let $P = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_l = v$ be the path from *root* (= v_0) to v (= v_l). We have

$A_{v_0}[i] = 0$ ($1 \le i \le m$);

$A_{v_j}[1] = 1$ if $e_j = z_1$; otherwise, $A_{v_j}[1] = 0$ ($j > 0$);

$$A_{v_j}[i] = \begin{cases} A_{v_j}[i-1], & \text{if case (1), (2), (3), or (4);} \\ A_{v_{j-1}}[i-1] + 1, & \text{if case (5) or (6);} \quad (\text{for } i > 1,\ j > 0) \\ A_{v_{j-1}}[i] + 1, & \text{if case (7).} \end{cases} \tag{6}$$

Special attention should be paid to case (1), (2), (3) and (4) in Table 4. In these cases, $A_{v_j}[i]$ is set to be $A_{v_j}[i$ - 1] although $D_{v_j}[i]$ is also possibly equal to $D_{v_{j-1}}[i$ - 1] + $\delta(z_i, e_l)$, and/or $D_{v_{j-1}}[i]$ + 1. This shows why the equation correctly computes the value. It is because if $A_{v_j}[i$ - 1] correctly records the length of the shortest suffix of $\boldsymbol{s}_{v_j}$ such that its distance from $\boldsymbol{z}_{i-1}$ is equal to $D_{v_j}[i$ - 1], then $A_{v_j}[i] = A_{v_j}[i$ - 1] if $D_{v_j}[i] = D_{v_j}[i$ - 1] + 1. The correctness of case (5) and (6), as well as case (7) can be easily checked.

Using $A_v[i]$, B_v can be computed as follows.

Let $s_v = e_1 \dots e_l$. We have

$$B_v = e_{l-\alpha(v)+1} \dots e_l, \tag{7}$$

where $\alpha(v)$ equals $A_v[i]$ for the largest position i in $\bar{x}$ such that $D_v[i] \le k$. For instance, for $\bar{x}$ = *gcaca*, and $\boldsymbol{s}_v$ = *acat* (along a tree path $v_0 \to v_1 \to \dots \to v_4 = v$), we have $\alpha(v_4) = A_{v_4}[4] = 3$ (see A(4, 4) in Table 5) since $D_{v_4}[4] = 2$ (by formulae (5), D_{v_4} = <4, 4, 3, 2, 2, 3>$^{\top}$) and $i = 4$ is the largest position in $\bar{x}$ = *gcaca* such that $D_{v_4}[4] \le 2$. Thus, B_v = *cat* $\ne \boldsymbol{s}_v$ = *acat*.

If $\alpha(v) = 0$, $B_v = e_{l-\alpha(v)+1} \dots e_l = e_{l+1} \dots e_l$, which is defined to be an empty string, represented as ϕ.

We now discuss why we can terminate a path at a node v if $|B_v| \ne |\boldsymbol{s}_v|$, even if none of the stopping conditions given in 5.1 is satisfied.

The reason for this is that a path P starting from a child of the root, but labeled with a substring identical to B_v, has aleady been explored, or will be explored. To see this property, consider an ancestor u of v such that the string on the path P from u to v equals B_v. Let <e_0, [α_0, β_0]> be the pair labeling u. There must be a child w of the root, whose label is <e_0, [α'_0, β'_0]> such that [α'_0, β'_0] $\supseteq$ [α_0, β_0]. Next, we check the child u_1 of u on P. Since [α'_0, β'_0] $\supseteq$ [α_0, β_0], there must be a child w_1 of w, whose interval contains u_1's interval. In this way, we can check all the other descendants till v on P, and prove the above claim by a simple induction. See Fig. 6 for illustration. It remains to expain that cutting off such a path at

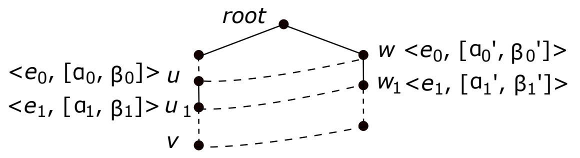


Figure 6: Illustration for B_v and path similarity.

v in the case of $|B_v| \ne |\boldsymbol{s}_v|$ will not lead to loss of answers. For this, we show another important property of B_v.

Lemma 5.3. *Let u, v be two nodes in T, not related by the ancestor/descendant relationship. If $s_u = B_v$, then $D_v[i] = D_u[i]$ ($1 \le i \le m$) if $D_v[i] \le k$.*

Proof. The proof is similar to Theorem 1 in [59]. □

See Fig. 7 for a better understanding of Lemma 5.3.

In Fig. 7(a), we show the D-matrix D for $\boldsymbol{s}_v$ = *acat* checked against $\bar{x}$ = *gcaca* with k = 2. For $\boldsymbol{s}_v$, B_v = *cat* $\ne$ *acat* = $\boldsymbol{s}_v$ as discussed above. In Fig. 7(b), we show the D-matrix D' for s_u = *cat* checked against $\bar{x}$. In these matrices, the arrows are used to intuitively indicate how a value is computed. From them, we can not only observe all the equal entries ≤ 2 in both D and D', but also trace their generation. In Fig. 7(c), we show only those paths starting from an entry ≤ 2 in D_4 (the last column in D). For each of them we have a same path in D' shown in Fig. 7(b).

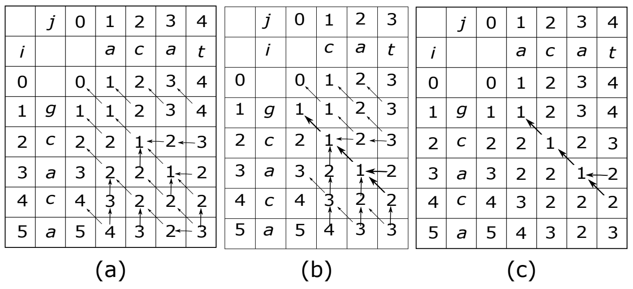


Figure 7: Illustration for trace of D-values' generation.

From Lemma 5.3, we can see that by extending $\boldsymbol{s}_u$ we can regain the answers which get lost due to cutting off $\boldsymbol{s}_v$. So, we have the following proposition.

Proposition 5.4. *During the execution of stringM($\bar{x}$, BWT(y), k), cutting off $\boldsymbol{s}_v$ if $|\boldsymbol{s}_v| \ne |B_v|$ will not cause loss of answers.*

Proof. The proof can be derived from Lemma 5.3 and the similarity between the path from *root* to v and the path from *root* to u for v and u referred to in Lemma 5.3. □

5.3 Computational complexities

Now we analyze the time complexity. By using the technique *rankAll* [11], the cost of $S(e'$, <e, [α, β]>) is O(1). Thus, the time complexity of *stringM*($\bar{x}$, *BWT*(y), k) is bounded by O($m \cdot |T|$). It is because for each node v in T, we need O(m) time to compute D_v. However, as observed by several earlier papers [14, 45, 58], this running time can be reduced to O($k \cdot |T|$) by using the diagonalwise monotonicity (i.e., in a dynamic programming matrix, the values along a diagonal never decrease.)

To estimate $|T|$, we need to count the number of the *root-to-leaf* paths generated during the execution of *stringM*($\bar{x}$, *BWT*(y), k). We notice the following facts:

- The probability of a character $\in \Sigma$ appearing at a certain position in y is $\frac{1}{|\Sigma|}$. Thus, the probability of a substring of length j of x appearing in y is $(\frac{1}{|\Sigma|})^j$.
- At a given position p in x, the following operations can be carried out to change it to another string.
  1. Delete the character at position p. There is only one way to do it.
  2. Insert j characters around position p. That is, insert $0 \le i \le j$ characters before position p while the remaining j - i characters after it. In this way, we can possibly change x to $|\Sigma|^j$ different strings each corresponding to the insertion of j characters.
  3. Substitute a nonindentical character for the pth character in x and then insert j - 1 new characters around the pth position, by which we can possibly change x to $(|\Sigma|$ - 1$)|\Sigma|^{j-1}$ different strings.

Thus, any $\boldsymbol{s}_v$ in T that is k (or less) different from $\bar{x}$ can be represented as an *operation* sequence: $o_1 \dots o_l$ (for some $l \le k$) with each o_i ($1 \le i \le l$) being a pair [p, q], where $p \in \{1, \dots, m\}$, and q is a symbol 'd', representing 'deleting' the pth character in $\bar{x}$; or a sequence of characters: $c_1 \dots c_j$, inserted around position p, or c_1

is used to substitue for the pth character while all the remaining j - 1 characters inserted. Applying such a sequence to $\bar{x}$, we will change $\bar{x}$ to s_v.

Denote by $\Gamma(m, k)$ the number of all such sequences. Let n' be the estimated number of leaf nodes in T. Then, we have

$$n' \leq \sum_{j=1}^{k} \frac{1}{|\Sigma|^{m-j}} \Gamma(m, j) \tag{8}$$

where $\frac{1}{|\Sigma|^{m-j}}$ is the probability that m - j characters of x appears in y. To estimate $\Gamma(m, k)$, we establish the following recursive equation.

$$\Gamma(m,k) = \begin{cases} 0, & \text{if } m = 0,\ k > 0; \\ 1, & \text{if } k = 0; \\ 2|\Sigma|, & \text{if } m = 1,\ k = 1; \\ (2|\Sigma| - 1)|\Sigma|^{k-1}, & \text{if } m = 1,\ k > 1; \\ \Gamma(m-1,k) + \Gamma(m-1,k-1) + & \text{Otherwise.} \\ \quad \frac{1}{k} \sum_{j=1}^{k} |\Sigma|^j \Gamma(m-1,k-j) + & \\ \quad \frac{1}{k}(|\Sigma| - 1) \sum_{j=1}^{k} |\Sigma|^{j-1} \Gamma(m-1,k-j), & \end{cases} \tag{9}$$

When m = 0, and k > 0, there is no sequence corresponding to this case and therefore $\Gamma(0, k)$ = 0. When m > 0, and k = 0, there is only one empty sequence. So $\Gamma(m, 0)$ = 1. $\Gamma(1, 1) = 2|\Sigma|$ since in this case, we may have $2|\Sigma|$ sequences of length 1: a sequence containing one deletion; $|\Sigma|$ - 1 sequences each containing one substitution; and $|\Sigma|$ sequences each containing one insertion. When m = 1, k > 1, a deletion cannot be performed since if the unique character is removed the other k - 1 operations cannot be conducted. Therefore, in this case, we can divide the sequences into two parts. One part contains $(\Sigma| - 1)|\Sigma|^{k-1}$ sequences each containing one sustitution followed by k - 1 insertions. Another part contains $|\Sigma|^k$ sequences each containing k insertions.

Finally, for the general cases of m > 1 and k > 1, we can categorize all the operation sequences into 4 classes.

- Class 1 contains all those sequences with p > 1 in the first element $[p, q]$. The size of this class is $\Gamma(m$ - 1, $k)$.
- Class 2 contains all those sequences whose first operation is to delete the first character of $\bar{x}$. The size of this class is $\Gamma(m$ - 1, k - 1).
- Class 3 contains all those sequences whose first operation is to insert j characters directly around the first position of $\bar{x}$. The average size of this class is
$$\frac{1}{k} \sum_{j=1}^{k} |\Sigma|^j \Gamma(m-1, k-j).$$
- Class 4 contains all those sequences whose first operation is to substitute the first character of $\bar{x}$ followed by j - 1 insertions. The average size of this class is
$$\frac{1}{k}(|\Sigma| - 1) \sum_{j=1}^{k} |\Sigma|^{j-1} \Gamma(m-1, k-j).$$

Adding all the above expressions together, we get equation (9). Concerning the estimation of $\Gamma(m, j)$, we have the following proposition. It can be proven by a simple induction.

Proposition 5.5. *For $m \geq 0, j \geq 0$, we have $\Gamma(m, j) \leq O(|\Sigma|^{m+j})$.*

Based on this proposition, we immediately get the upper bound on the time complexity of our algorithm.

Proposition 5.6. *The average time complexity of stringM($\bar{x}$, $BWT(y)$, k) is bounded by $O(k \cdot |\Sigma|^{2k})$.*

Proof. Let n' be the number of leaf nodes of a tree T generated during the execution of stringM($\bar{x}$, $BWT(y)$, k). Then, we have

$$\begin{aligned} n' \quad &\leq \sum_{j=1}^{k} \frac{1}{|\Sigma|^{m-j}} \Gamma(m, j) \leq \sum_{j=1}^{k} \frac{1}{|\Sigma|^{m-j}} |\Sigma|^{m+j} \\ &\leq \mathrm{O}(|\Sigma|^{2k}). \end{aligned} \tag{10}$$

If the average outdegree of nodes in T is > 1, $|T|$ is bounded by $\mathrm{O}(|\Sigma|^{2k})$. So, $\mathrm{O}(k \cdot |T|) \leq \mathrm{O}(k \cdot |\Sigma|^{2k})$. If the average outdegree of nodes in T is 1, $|T|$ is then bounded by $\mathrm{O}(m + k)$ since we can control the generation of T so that the length of each path in T is bounded by $\mathrm{O}(m + k)$. It is because the edit distance between x and any string of length $\geq m + k + 1$ must be > k. Thus, we have $\mathrm{O}(k \cdot |T|) \leq \mathrm{O}(k \cdot |\Sigma|^{2k})$. □

To deduct the space overhead, we can keep the D-vectors only for the nodes along the current path. In addition, using the diagonalwise monotonicity and a techniques described in [14], we only need to calculate $\mathrm{O}(k)$ entries in a D-vector. Thus, the overall space requirement, in addition to the index, is bounded by $\mathrm{O}(k \cdot (m + k))$.

6 PARTITION OF PATTERN STRINGS

From the above discussion, we have seen that the average running time of Algorithm *stringM*( ) is bounded by $\mathrm{O}(k \cdot |\Sigma|^{2k})$. It must be efficient for sufficiently small k values. To extend this strategy for larger distances, we adopt a two-step method as described below.

- In the first step, we partition the pattern $x = x_1 \ldots x_m$ evenly into l segements, denoted as $x = P_1 P_2 \ldots P_l$ with each $P_i = x_{(i-1)r+1} \ldots x_{ir}$ for $1 \leq i \leq l$ - 1, and $P_l = x_{(l-1)r+1} \ldots x_m$, where $r = \lceil m/l \rceil$. Then, we run *stringM* ( ) on each $\bar{P}_i$ against $BWT(y)$ with $k' = \lfloor k/l \rfloor$ differences in turn to find all the occurrences of P_i ($1 \leq i \leq l$) in y. Each occurrence is represented by (i, j), indicating that P_i matches a segment ending at position j in y with k' differences.
- In the second step, for each occurrence (i, j) found in the first step, the substring of the target: $s_{i,j} = y_{j-ir+1-k} \cdots y_{j-ir+1+m+k}$ will be again closely checked against x with k differences by using a classical algorithm [14]. The length of $s_{i,j}$ is $m + 2k$ (see Fig. 8 for illustration.)

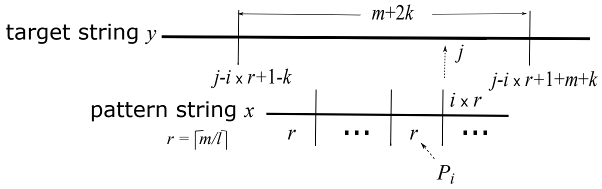


Figure 8: Illustration for occurrence of P_i in $s_{i.j}$.

This method works based on an observation that if x matches a segment s of y with less than k differences, then at least one of the P_i's matches s with $\lfloor k/l \rfloor$ differences. It is because if no P_i is present in s with $\lfloor k/l \rfloor$ or less than $\lfloor k/l \rfloor$ differences, then when we transform x to s, by which each P_i will be transformed into a subsegment s_i of s, the number of needed operations is larger than $l \cdot (\lfloor k/l \rfloor + 1) > l \cdot (k/l) = k$. It is a contradiction.

Thus, the first step is just like a filter to discard all those subsegments where a match cannot possibly occur. This idea was first suggested in [62] and has been used in several algorithms [7, 45]. It can somewhow improve the performance. But the worst-case time complexities of these algorithms are almost unchanged since the dominant cost for searching long target strings remains the same, and the benefit gain by checking small patterns is overshadowed by the multiple checkings of divided subpatterns. It is only by our method that this idea becomes significant. It is because by our method the size of T is almost independent of the length of target strings; but heavily depends on k and $|\Sigma|$. In terms of the above discussion, the time complexity of the first step is bounded by

$$\sum_{i=1}^{l} \lfloor \frac{k}{l} \rfloor \cdot |T_i| \leq l \lfloor \frac{k}{l} \rfloor |\Sigma|^{2\frac{k}{l}} = O(k \cdot |\Sigma|^{2\frac{k}{l}}),$$

where T_i is the tree generated when running *stringM*( ) on $\bar{P}_i$ and $BWT(y)$ with $\lfloor k/l \rfloor$ differences.

Let h be the number of occurrences found in the first step. By using dynamic programming, the time for the second step is bounded by $O(m \cdot h \cdot (m + 2k)) = O(h \cdot m^2)$ since for each occurrence a checking of x against a segment of length $m + 2k$ in y will be conducted. As mentioned in 5.3, this time complexity can be improved to $O(h \cdot k \cdot m)$ by using a more sophiscated method [14].

7 EXPERIMENTS

In our experiments, we have tested altogether 7 strategies:

- Ukkonen's onlline method (*u-o* for short, [58]),
- Chang-Lawer's first method (*ch-1* for short, [14]),
- Chang-Lawer's second method (*ch-2* for short, [14]),
- Ukkonen's index-based method *(u-i* for short, [59]),
- Myers's index-based method (*m-i* for short, [45]),
- Peri-Culpepper's index-based method (*pc-i* for short, [50]), and
- ours, discussed in this paper.

The first three are all online strategies while the remaining four are all index-based, All codes are written in C++ and compiled by GNU g++ compiler version 5.4.0 with compiler option '-O2'. All tests run on a 64-bit Ubuntu OS with a single core of Intel Xeon E5-2637 @3.50Ghz. The system memory is of 64 GB. For time measurements, we used the Unix time commands. In addition, the suffix trees for patterns (in the Chang-Lawler's method and ours), as well as for reference sequences (in the Ukkonen's index-based method) are constructed by using the algorithm described in [60]. To construct the suffix arrays and the BWT-arrays, we used a code found in the *libdivsufsort* library (https://github.com/y-256/libdivsufsort) with the compact factor for the *rankAll* array $\beta = 4$ and the compact factor for the suffix array $\gamma = 16$.

7.1 Data sets

For the experiments, three genome and two protein sequences were downloaded from ensemble.org (ftp://ftp.ense mbl.org/pub/release-93/) and SMS (https://www.bioinfor matics.org/sms2/random_protein.html), respectively. The size of the alphabet for the protein sequenses is 20, much larger than that of DNA sequences. The patterns which were tested against different genome and protein sequences were generated by using the *wgsim* tool which is part of the *SAMtools* package [35]. In Table 6, we show all the tested sequences, as well as the time spent for constructing their BWT-arrays.

Table 6: Genome and protein sequences, as well as time for constructing their BWT-arrays.

reference sequences*	num. characters	time (s)
Gorilla	3,063,403,506	406.817
Danio Rerio (ZebraFish)	1,373,472,378	173.142
Gorilla Chr1	228,908,641	25.03
Protein-1	144,000,000	15.92
Protein-2	30,000,000	3.04

*The first three are genome sequences while the last two are protein sequences.

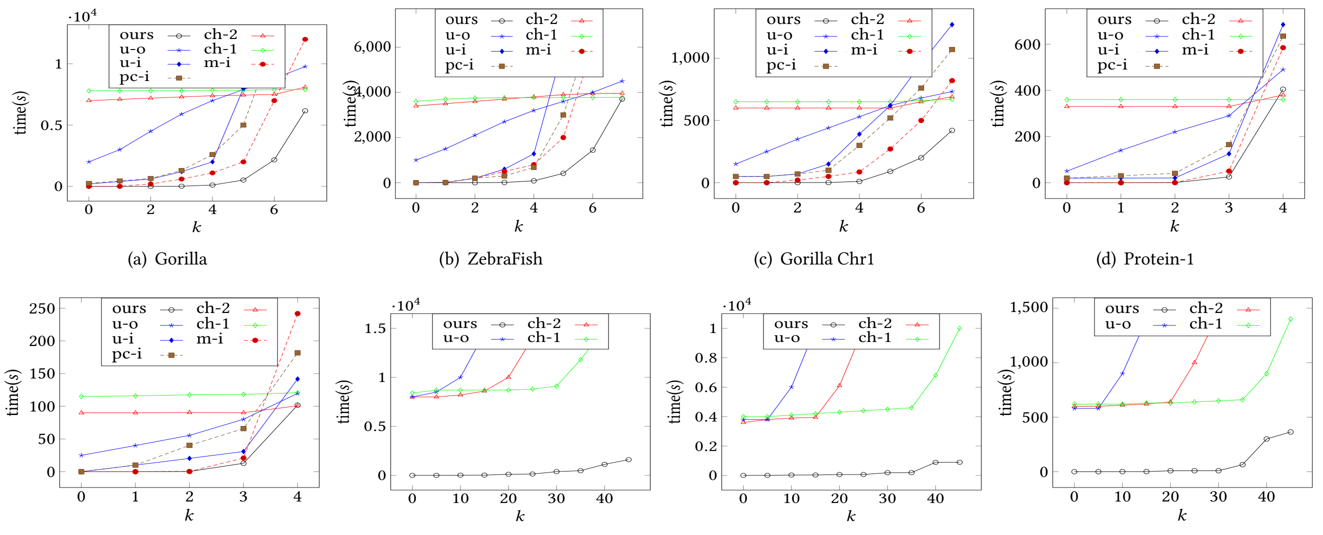
7.2 Test 1: on string matching with small k

We first report the experiments on the string matching with small number of differences. In this test, we run all the algorithms on 100 sequences of length 100 characters against all the 5 sequenses listed in Table 6 with k set from 1 to 7. In Fig. 9(a), we show the test result on the Gorilla genome. From this, we can see that our method uniformly outperforms all the other methods for small k (≤ 6). The smaller the value of k, the larger the discrepancies between ours and all the tested online methods. For example, when $k = 6$, our method is 4 times better than the Chang-Lawer's second method. When $k = 4$, ours is almost 65 times faster than this method. Especially, for $k = 1$, ours can be up to 70,000 times better. The observation on the index-based methods is just opposite. Specifically, as k goes up, the differences between ours and all the other three index-based increases. The Ukkonen's algorithm works not so well since during its execution, almost the whole suffix tree over a genome has to be searched and some parts will be repeatedly accessed. Although the suffix links can be used to save time, the searching of a whole suffix tree has veiled this advantage. Its theorectical time complexity is bounded by $O(m \cdot min\{n, m^{k+1}|\sum|^{k+1}\} + n)$ [59]. During the execution of the Myers's, a big set called the *neighbourhood* needs to be created, which contains a large number of sequences whose distance is $\leq k$ from the pattern. Theoretically, its size is bounded by $O(|\Sigma|^{m+k})$. Although the set is somehow reduced by imposing a constraint that any sequence in the set cannot be a prefix of another, its theoretical bound remains unchanged [45]. The Peri-Culpepper's method is slightly worse than Myers's since no constraints are used to reduce the size of such neighbourhoohds. Obviously, all the three methods are much worse than ours. To have a deeper insight into the behaviour of our algorithm, we give the number of nodes generated during the execution of our algorithm in Table 7. From this, we can see that the size of the trees generated is much smaller than the number of characters in the original genome due to the fact that by our method the sequence is 'folded' and becomes much shorter, and the branch cut during the searching of T.

Table 7: Nubmer of nodes in T

k	1	2	3	4	5	6	7
$\|T\|$	1.4K	25K	278.5K	2M	10M	39.72M	92M

The Chang-Lawer's second algorithm is a little bit better than their first one. In addition, although the Ukkonen's online algorithm is much better than the Chang-Lawer's for small k, its performance



(a) Gorilla (b) ZebraFish (c) Gorilla Chr1 (d) Protein-1

(e) Protein-2 (f) Gorilla (g) ZebraFish (h) Gorilla Chr1

Figure 9: Test results.

quickly deteriorates as k increases. In Fig. 9(b), and Fig. 9(c), we show the test results on the Zebrafish and Gorilla Chr1, respectively. From these figures, we can see that for $k \leq 6$, our method is consistantly better than the others. But for $k = 7$, the performance of our method becomes comparable to or even worse than the others. It is because for shorter genome sequences the number of nodes created by our method can be larger than their lengths. However, our method is still much better than the other three index-based algorithms. In Fig. 9(d), and Fig. 9(e), we show the test results on the two protein sequences, from which we can see that when $k \geq 4$, the performance of our method degrades. This shortage of our method can be effectively removed by splitting ptterns into small ones, as demonstrated by the test reported in the next subsection.

7.3 Test 2: on string matching with large k

From the previous subsection, we can see that for small k the performance of our method is superior to all the others. However, as k goes up, the performance of ours, just like the other three index-based methods, degrades greatly. In the opposite, all the online methods seem not so sensitive to the value of k, but to the size of genomes. So, for large k, we have to do the pattern partition and utilize the two-step method described in Section 6. It is what is done in this test, in which the threshold is set to be 6 for genome sequences. That is, if k is ≥ 6, the pattern will be divided into l parts such that $\lfloor \frac{k}{l} \rfloor < 6$. We choose threshold 6 since when $k = 6$ the performance of our method becomes worse. For the same reason we choose the threshold 3 for protein sequences. Again, we have tested 100 patterns against the 5 different sequences listed in Table 6, but each of them is of length 300 characters.

For this test, we first notice that all the other three index-based methods are not included. It is because for large k, they have to resort to the pattern partition just like ours. Otherwise, their performance must be much worse than any online algorithm. Doing the pattern partition, howevere, their performance is definitely worse than ours, even with quickly inceased discrepancies. For the same reason, if any online method works in two steps using the pattern partition, it will also definitely be inferior to ours. Only one question remains, will they have better performance than ours if the pattern division is not conducted for them? The figures shown in Fig. 9(f) - (h) and Fig. 10(a) - (b) give the answer. In these figures, we demonstrate the test results of checking different patterns against different target sequences listed in Table 6, respectively.

As expected, we can see a huge difference between ours and all the online methods from these figures. The reason is twofold. First, each subquery can be evaluated super fast in the first step by our method. Second, the number of suviving segments is small in comparison with the original sequences, as exhibited in Table 8 and Table 9.

Table 8: Two-step execution details on Gorilla genome

k	20	25	30	35	40	45
step-1	23.1	23.1	173.2	172.9	1187.0	1187.5
step-2	97.41	122.1	263.4	311.7	493.22	565.58
num.	30.5K	30.5K	52.9K	52.7K	69.5K	69.3K
size	353	364	388	399	428	439

Table 9: Two-step execution details on ZebraFish genome

k	20	25	30	35	40	45
step-1	58.73	58.57	187.5	187.6	953.0	952.4
step-2	18.41	21.48	34.24	38.85	60.33	60.70
num.	3638	3633	4709	4702	6376	6365
size	423	433	444	455	463	474

In Table 8, we show the details of our method's execution on the Gorilla genome with l (the number of subpatterns) being set to 10, including the time (in seconds) of Step 1 and Step 2 and the numbers of segments rechecked in Step 2, as well as the average length of such segments. In Table 9, we display the details on the

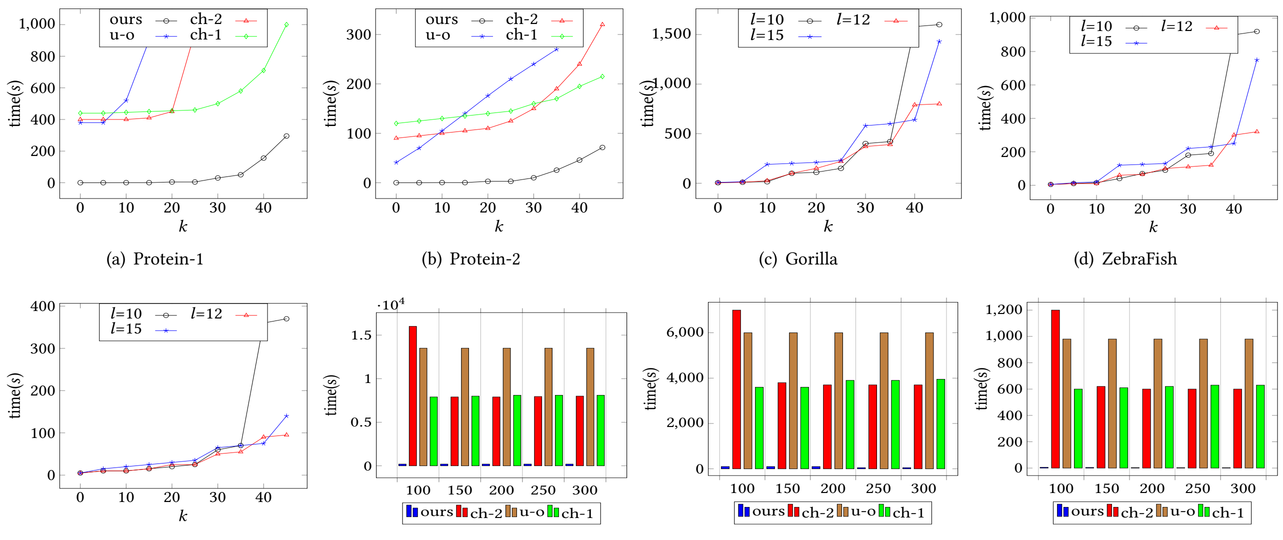

(a) Protein-1
(b) Protein-2
(c) Gorilla
(d) ZebraFish
(e) Gorilla Chr1
(f) Gorilla
(g) ZebraFish
(h) Gorilla Chr1

Figure 10: Test results

ZebraFish genome, also with l set to 10. From these two tables, we can see that both Step 1 and Step 2 can be carried out very quickly by our method.

7.4 Test 3: on numbers of subpatterns

In this test, we focus on the impact of the number of subpatterns l to performance. The setup for this test is the same as Test 2, but with varying values of $l = 10, 12, 15$. However, we report the test results only for the Gorilla, Zebrafish, Gorilla Chr1 and show them in Fig. 10(c), 10(d), 10(e), respectively. From these results, we can observe that when k is relatively small ($k \leq 30$), whichever l is chosen will not much impact the performance. However, for large k (≥ 40), choosing larger values of l is obviously beneficial. Concretely, we can achieve more than 2-3 times faster running time by setting l to 12 or 15 than by setting l to 10. The reason for this is that larger l means smaller $k' = \lfloor k/l \rfloor$ and the first step can thus be done very rapidly. On the other hand, from Table 10, we can see that the number of surviving segments to be checked in Step 2 are comparable to each other for different l values. This implies that the difference in performance mainly consists in the execution time of Step 1. Anyway, for all the three tested genomes, the performance goes down as k goes up.

Table 10: Nubmer of segments checked in Step 2.

k	20	25	30	35	40	45
l=10	30.5K	30.5K	52.9K	52.7K	69.5K	69K
l=12	28.7K	56.2K	56.0K	55.8K	60.5K	61.4K
l=15	30.5K	30.5K	52.9K	52.7	69.5K	69.3K

7.5 Test 4: on varying length of patterns

In this test, we check the performance of our method and all the online methods on the length of patterns. For this purpose, we set a fixed value of $k = 10$ and $l = 5$ for our method, but varying pattern lengths from 100 to 300 character. The test results are exhibited in Fig. 10(f) - 10(h) and Fig. 11(a) - 11(b), respectively, for all the five different sequences. From these figures, we can see that all their performance is not much changed for different pattern lengths, but consistantly demonstrates a big difference between ours and all the tested online methods. All of them, including ours, are almost pattern-length independent.

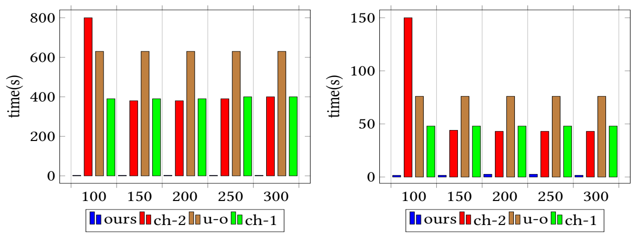

(a) Protein-1
(b) Protein-2

Figure 11: Test results

8 CONCLUSIONS

In this paper, a new index-based method is discussed for solving the string matching with k differences. This is highly important to searching genome and protein sequences in modern DNA databases, as well as genetics research, especially, for very long sequences. The average time complexity of our method is bounded by $O(k \cdot |\Sigma|^{2k})$. Together with pattern partition, our method can achieve more than 1000-fold improvement than the existing methods. In addition, the index itself can be constructed very fast with only linear time and space required. Therefore, it can also be considered as a competetive on-line strategy, better than any existing one. Extensive experiments are conducted, showing that our method is not only a theoretically, but also a practically efficient method. We believe that our algorithm as a tool will be very useful for the biological research.

REFERENCES

[1] A. Abboud and et al. 2016. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proc. of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA*. 375–388.

[2] A. Abboud and A. Rubinstein. 2018. Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, Cambridge, MA, USA*. 35:1–35:14.

[3] A.V. Aho and M.J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Communication of the ACM* 23, 1 (June 1975), 333–340.

[4] A. Amir, M. Lewenstein, and E. Porat. 2004. Faster algorithms for string matching with k mismatches. *Journal of Algorithms* 50, 2 (Feb. 2004), 257–275.

[5] R.A. Baeza-Yates and G.H. Gonnet. 1989. A new approach to text searching. In *N.J. Belkin and C.J. van Rijsbergen (eds.) SIGIR 89, Proc. 12th Annual Intl. ACM Conf. on Research and Development in Information Retrieval.* 168–175.

[6] R.A. Baeza-Yates and G.H. Gonnet. 1992. A new approach in text searching. In *Communication of the ACM*, Vol. 35. 74–82.

[7] R.A. Baeza-Yates and C.H. Perleberg. 1992. Fast and practical approximate string matching. In *A. Apostolico, M. Crocchemore, Z. Galil, U. Manber (eds.) Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 644.* Springer-Verlag, Berlin, 185–192.

[8] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. 2013. Space-Efficient Construction of the Burrows-Wheeler Transform. In *16th Interl. Symposium on String Processing and Information Retrieval.*

[9] R.S. Boyer and J.S. Moore. 1977. A fast string searching algorithm. *Communication of the ACM* 20, 10 (Oct. 1977), 762–772.

[10] K. Bringmann and M. Künnemann. 2015. Quadratic conditional lower bounds for string problems and dynamic time warping. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA*. 79–97.

[11] M. Burrows and D.J. Wheeler. 1994. A Block-sorting Lossless Data Compression Algorithm. *Systems Research Center* (May 10, 1994).

[12] D. Chakraborty, D. Das, and M. Koucky. 2018. Approximate Online Pattern Matching in Sub-linear Time. *arXiv:1810.03551v1 [cs.DS]* (2018).

[13] W.L. Chang and J. Lampe. 1992. Theoretical and empirical comparisons of approximate string matching algorithms. In *A. Apostolico, M. Crocchemore, Z. Galil, U. Manber (eds.) Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 644.* Springer-Verlag, Berlin, 175–184.

[14] W.I. Chang and E.L. Lawler. 1994. Sublinear Approximate String-Matching and Biological Applications. *Algorithmica* 12, 4 (1994), 327–344.

[15] Y. Chen and Y. Wu. 2016. On the Massive String Matching Problem. In *Proc. ICNC-FSKD 2016.* IEEE, Changsha, China.

[16] Y. Chen and Y. Wu. 2017. Mismatching Trees and BWT Arrays: A New Way for String Matching with k-Mismatches. In *ICDE'17.* San Diego, USA, 339–410.

[17] Y. Chen and Y. Wu. 2018. On the String matching with k mismatches. *Theoretical Computer Science* 726 (May 2018), 5–29.

[18] Y. Chen, Y. Wu, and J. Xie. 2016. An Efficient Algorithm to Search Reads in DNA Databases. In *Proc. 8th International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA 2016).* Lisbon, Portugal, 23–34.

[19] R. Clifford and B. Sach. 2010. Online approximate matching with non-local distances. In *Combinatorial Pattern Matching, Proc. 19th Annual Symposium, New York, NY, USA*. 101–111.

[20] R. Cole, L. Gottlieb, and M. Lewenstein. 2004. Dictionary Matching and Indexing with Errors and Don't Cares. In *STOC'04.* 91–100.

[21] B. Commentz-Walter. 1979. A String Matching Algorithm Fast on the Average. In *Proc. 6th Colloquium on Automata, Languages and Programming.* 118–132.

[22] M. Crochemore and et al. 1999. Fast practical multi-pattern matching. *Inform. Process. Lett.* 71 (1999), 107–113.

[23] Z. Galil. 1977. On improving the worst case running time of the Boyer-Moore string searching algorithm. *Communication of the ACM* 22, 9 (1977), 505–508.

[24] Z. Galil and R. Giancarlo. 1986. Improved string matching with k mismatches. *ACM SIGACT News* 17, 4 (1986), 52–54.

[25] R. Grossi, A. Gupta, and J. Vitte. 2003. High-order entropy-compressed text indexe. In *Proc.14th ACM-SIAM Symposium on Discrete Algorithms.* Oregon, USA, 814–850.

[26] M.C. Harrison. 1971. Implementation of the substring test by hashing. *Communication of the ACM* 14, 12 (1971), 777–779.

[27] R.L. Karp and M.O. Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31, 2 (March 1987), 249–260.

[28] D.K. Kim, J.S. Lee, K. Park, and Y. Cho. 1999. Efficient Algorithms for Approximate String Matching with Swaps. *Journal of Complexity* 15 (1999), 128–147.

[29] D.E. Knuth, J.H. Morris, and V.R. Pratt. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6, 2 (June 1977), 323–350.

[30] G.M. Landau. 2018. Efficient String Matching With K Differences. In *Classic Reprint Series.*

[31] G.M. Landau and U. Vishikin. 1988. Fast string matching with k differences. *J. Comput. System Sci.* 37, 1 (1988), 63–78.

[32] G.M. Landau and U. Vishkin. 1985. Efficient string matching with k mismatches. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science.* 126–136.

[33] G.M. Landau and U. Vishkin. 1986. Efficient string matching with k mismatches. *Theoretical Computer Scienc* 43 (1986), 239–249.

[34] T. Lecroq. 1990. A variation on the Boyer-Moore algorithm. In *J.R. Gilbert and R. Karlssion (eds.) SWAT 90, Proc. 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science.* Springer Verlag, 243–260.

[35] H. Li. 2011. wgsim: a small tool for simulating sequence reads from a reference genome. https://github.com/lh3/wgsim/.

[36] H. Li and R. Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler Transform. *Bioinformatics* 25, 14 (2009), 1754–1760.

[37] H. Li and et al. 2008. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.* 18 (2008), 1851–1858.

[38] H. Li and N. Homer. 2010. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics* 11, 5 (doi:10.1093/bib/bbq015, 2010), 473–483.

[39] H. Lin and et al. 2008. ZOOM! Zillions of oligos mapped. *Bioinformatics* 24 (2008), 2431–2437.

[40] T-S. Lin, C-Y. Lu, and S-Y. Kuo. 2010. Quantum switching and quantum string matching. In *10th IEEE International Conference on Nanotechnology.* DOI:10.1109/NANO.2010.5697866.

[41] U. Manber and R.A. Baeza-Yates. 1991. An algorithm for string matching with a sequence of don't cares. *Inform. Process. Lett.* 37 (Feb. 1991), 133–136.

[42] U. Manber and E.W. Myers. 1990. A block-sorting lossless data compression algorithm. In *Proc. the 1st Annual ACM-SIAM Symposium on Discrete Algorithms.* SIAM, Philadelphia, PA, 319–327.

[43] W.J. Masek and M.S. Paterson. 1980. A fast algorithm computing string edit distances. *J. Comput. System Sci.* 20, 1 (1980), 18–31.

[44] E.M. McCreight. 1976. A space economical suffix tree construction algorithm. *J. ACM* 23, 2 (April 1976), 262–272.

[45] E.W. Myers. 1994. A Sublinear Algorithm for Approximate Keyword Searching. *Algorithmica* 12 (1994), 345–374.

[46] G. Navarro and R. Baeza-Yates. 1998. A Practical q-Gram Index for Text Retrieval Allowing Errors. *CLEI Electron. J.* 2, 1 (1998).

[47] M. Nicolas and S. Rajasekarian. 2013. On string matching with k mismatches. In *https://arxiv.org/pdf/1307.1406.*

[48] N. Ben Nsira, M. Elloumi, and T. Lecroq. 2017. On-line String Matching in Highly Similar DNA Sequences. *Math. Comput. Sci.* 11, 2 (2017), 113–126.

[49] D. Okanohara and K. Sadakane. 2009. A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In *16th Interl. Symposium on String Processing and Information Retrieval.* Finland.

[50] M. Petri and J.S. Culpepper. 2012. Efficient Indexing Algorithms for Approximate Pattern Matching in Text. In *ADCS'12.* Otago, Dunedin, NZ.

[51] R.Y. Pinter. 1985. Efficient string matching with don't care patterns. In *A. Apostolico and Z. Galil (eds.) Combinatorial Algorithms on Words, NATO ASI Series,* Vol. F12. Springer-Verlag, Berlin, 11–29.

[52] C. Ryu, T. Lecroq, and K. Park. 2020. Fast string matching for DNA sequences. *Theor. Comput. Sci.* 812 (2020), 137–148.

[53] P.H. Sellers. 1980. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms* 1 (1980), 359–373.

[54] A. T. Starikovskaya. 2017. Communication and streaming complexity of approximate pattern matching. In *Combinatorial Pattern Matching, Proc. 19th Annual Symposium, Warsaw, Poland.* 13:1–13:11.

[55] M. Tahir, M. Sardaraz, and Ataul A. Ikram. 2017. EPMA: Efficient pattern matching algorithm for DNA sequences. *Expert Syst. Appl.* 80 (2017), 162–170.

[56] J. Tarhio and E. Ukkonen. 1989. Boyer-Moore approach to approximate string matching. In *J.R. Gilbert and R. Karlssion (eds.) SWAT 90, Proc. 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, Vol. 447.* Springer-Verlag, Berlin, 348–359.

[57] J. Tarhio and E. Ukkonen. 1990. Approximate Boyer-Moore String Matching. *SIAM J. Comput.* 22, 2 (1990), 243–260.

[58] E. Ukkonen. 1985. Algorithms for approximate string matching. *Information and Control* 64 (1985), 100–118.

[59] E. Ukkonen. 1993. Approximate String-Matching over Suffix Trees. In *Proc of the .4th Annual Symposium on Combinatorial Pattern Matching.* Springer-Verlag, 228–242.

[60] E. Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica* 14, 3 (1995), 249–260.

[61] P. Weiner. 1973. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory.* 1–11.

[62] S. Wu and U. Manber. 1992. Fast text searching allowing errors. *CACM* 35, 10 (Oct. 1992), 83–91.

[63] D. Zhang, Q. Liu, Y. Wu, Y. Li, and L. Xiao. 2013. Compression and Indexing Based on BWT: A Survey. In *10th Web Information System and Application Conference.* Yangzhou, China, 61–64.