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 \( O(k \cdot \log n) \), 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 \( \Sigma \) is an alphabet from which we take symbols to make up target and pattern strings. This time complexity is better than previous strategies when \( k \leq \frac{\log |\Sigma|}{n} \). 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 \( k/l \) 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:

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

By the string matching with \( k \) differences, we mean a problem to find all the occurrences of a pattern string \( x = x_1 x_2 \ldots x_m \) in a target string \( y = y_1 y_2 \ldots y_n \) 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 target 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 pattern corresponds to “no character” in the pattern (an insertion); and

3. A character of the target 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 \( y = abdefghi \), 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., \( y_2 \ldots y_7 = bcddefgh \)), as shown below.

\[
\begin{array}{ccccccc}
  b & p & d & q & e & g & h \\
  a & b & c & d & e & f & g & h
\end{array}
\]

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 \( x_i = x_1 x_2 \ldots x_i \) and \( y_j = y_1 y_2 \ldots y_j \) for \( i = 1, \ldots, m, j = 1, \ldots, n \):

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

\[
d_{i-1,j-1} + w(x_i, y_j),
\]

where \( \phi \) 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.

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 follows. 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 throughout the paper, in the following table.

<table>
<thead>
<tr>
<th>Table 1: Symbols and notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
</tr>
<tr>
<td>$\hat{x}$</td>
</tr>
<tr>
<td>$y_j$</td>
</tr>
<tr>
<td>$BWT(y)$</td>
</tr>
<tr>
<td>$&lt;e, [\alpha, \beta]&gt;$</td>
</tr>
<tr>
<td>$L_{e, [\alpha, \beta]}$</td>
</tr>
<tr>
<td>$D^{(\ast, j)}$, $D_j$</td>
</tr>
<tr>
<td>$D_v$</td>
</tr>
<tr>
<td>$T$</td>
</tr>
<tr>
<td>$T_p$</td>
</tr>
<tr>
<td>$s_o$</td>
</tr>
<tr>
<td>$B_v$</td>
</tr>
<tr>
<td>$A_0[\ell]$</td>
</tr>
</tbody>
</table>

## 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 $O(mn)$ time algorithm was given by Sellers in 1980 [53]. In the same year, Masek and Paterson [43] proposed another $O(mn)$-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 diagonally monotonicity of dynamic programming matrices with its average time complexity bounded by $O(kn)$ [58]. The algorithm presented by Landau and Vishkin was similar to the Ukkonen’s, but changed the Ukkonen’s column-wise computation of values to the diagonal-wise [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 $O(kn)$. 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 $O(kn)$ 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 $O(kn)$. 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 $O(m \cdot \min(n, kn^{k+1}) + n)$ with $O(m \cdot \min(n, kn^{k+1} + 1))$ space requirements. In [45], each substring of length $n$ in $y$ is viewed as a radix-$|\Sigma|$ number, and its first character is kept in a list of length $n$, used as an index. It requires $O(kn^2\varepsilon)$ time with $O(n + m)$ space requirements, where $\varepsilon = k/m$ and $f(\varepsilon)$ is a concave function. When $\varepsilon$ 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 $p_j (i = 1, \ldots, j)$ 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 $O(kn)$ 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 computational complexities as [46].

From the above description, we can see that no strategy up to now is able to break the bottleneck of $O(kn)$ running time. But our method runs in $O(kn^2k^k)$ time and is the first one to bring down $O(kn)$ 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 mismatches 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 · n) time, while in all the other methods, either mismatching information or periodicity 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 · 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 + m) · 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 · log k). The method proposed in [48] is for a special 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 · n′ + n · m · 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 ≥ 2(k + 1), the average value of n′ is bounded by O((1 + 1/2k)k). The running time of [36] is bounded by O(m · 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 a new, precise notion of 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 = x₁...xₙ and y = y₁...yₙ. Denote by xᵢ = xᵢ₁xᵢ₂...xᵢₖ a prefix of x and by yⱼ = yⱼ₁yⱼ₂...yⱼₖ a prefix of y. We represent the Levenshtein distance between xᵢ and yⱼ by D(i, j), which can be computed by constructing a (m + 1) × (n + 1) matrix D according to the following formulae:

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

\[
D(i, j) = \begin{cases} 
D(i - 1, j) + w(xᵢ, ⌀) \\
D(i - 1, j - 1) + δ(xᵢ, yⱼ) \\
D(i, j - 1) + w(⋆, yⱼ)
\end{cases}
\]

where δ(xᵢ, yⱼ) = 0 if xᵢ = yⱼ; otherwise, δ(xᵢ, yⱼ) = w(xᵢ, yⱼ), representing the cost to change xᵢ to yⱼ [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 = acatag, we will generate a matrix as illustrated in Table 2. By the computation, we set w(xᵢ, yⱼ) = 0 if xᵢ = yⱼ; otherwise, w(xᵢ, yⱼ) = 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 Dⱼ to refer to the jth vector in D. That is, Dⱼ = 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...
Table 2: Matrix D

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>t</td>
<td>a</td>
<td>t</td>
<td>g</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>suffix</th>
<th>sorted suffix</th>
<th>SA_y</th>
<th>r_F</th>
<th>sorted rotations</th>
<th>L</th>
<th>r_L</th>
</tr>
</thead>
<tbody>
<tr>
<td>gataca$</td>
<td>$</td>
<td>7</td>
<td>-</td>
<td>$</td>
<td>$</td>
<td>1</td>
</tr>
<tr>
<td>tataca$</td>
<td>a$</td>
<td>6</td>
<td>1</td>
<td>a</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>atac$</td>
<td>aca$</td>
<td>4</td>
<td>2</td>
<td>a</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>taca$</td>
<td>ataca$</td>
<td>2</td>
<td>3</td>
<td>a</td>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>ca$</td>
<td>gataca$</td>
<td>1</td>
<td>0</td>
<td>g</td>
<td>g</td>
<td>-</td>
</tr>
<tr>
<td>a$</td>
<td>taca$</td>
<td>3</td>
<td>1</td>
<td>t</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>$</td>
<td>tataca$</td>
<td>1</td>
<td>2</td>
<td>t</td>
<td>1</td>
<td>a</td>
</tr>
</tbody>
</table>

In addition, to describe how the string matching can be conducted based on the above facts, we need yet another two notations:
- \(<e, [a, \beta]>, (e \in \{a, c, g, t\}, \alpha \leq \beta \) are two ranks, representing a range (or say, a subsegment) in \(F_e\). For example, \(<t, [1, 2]>, \) represents \(F[6..7]\).
- \(L_{<e, [a, \beta]>}, \) representing a range in \(L\) corresponding to \(<e, [a, \beta]>, \)

During the search of \(L = BWT(y)\), in each step, two operations will be performed: (i) searching within a certain \(L_{<e, [a, \beta]>}, \) to find the first and the last appearance of some character \(e'\) (represented by their ranks \(a' \) and \(\beta'\), respectively); (ii) in terms of \(e'\), \((a', \beta')>\), figure out \(L_{<e', [a', \beta>]}, \) 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 \(x\) (reverse of \(x\)) against \(BWT(y)\). We denote such a searching step by \(S(e', <e, [a, \beta]>)\) defined below:

\[ S(e', <e, [a, \beta]>) = \left\{ \begin{array}{ll} <e', [a', \beta']> & \text{if } e' \in L_{<e, [a, \beta]>}; \\ \phi & \text{otherwise}. \end{array} \right. \]  

The following example helps for illustration.

**Example 4.1.** Let \(x = tata\) and \(y = gataca\). To find all the occurrences of \(x\) in \(y\), we will search \(x\) against \(L\) and \(F\) generated for \(y\)

<table>
<thead>
<tr>
<th>(F)</th>
<th>(L)</th>
<th>(r_F)</th>
<th>(F)</th>
<th>(L)</th>
<th>(r_F)</th>
<th>(F)</th>
<th>(L)</th>
<th>(r_F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- $</td>
<td>a</td>
<td>1</td>
<td>- $</td>
<td>a</td>
<td>1</td>
<td>- $</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>1</td>
<td>a</td>
<td>c</td>
<td>1</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>t</td>
<td>2</td>
<td>a</td>
<td>t</td>
<td>2</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>t</td>
<td>2</td>
<td>3</td>
<td>a</td>
<td>t</td>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>2</td>
<td>1</td>
<td>c</td>
<td>2</td>
<td>1</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>a</td>
<td>3</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>t</td>
<td>1</td>
<td>3</td>
<td>t</td>
<td>1</td>
<td>3</td>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>g</td>
<td>1</td>
<td>2</td>
<td>g</td>
<td>1</td>
<td>2</td>
<td>g</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 1:** Illustration of backward search.

**Step 1:** Check \(x_1 = a\) 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 \(x_2 = t\). 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 \(L_{<a,[3,3]>} = L[6..7]\) 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 \(f(t_1) = 2\). See column 6 in Table 3).

Let \(x = z_1 \ldots z_m\). The general working process can be represented as a sequence of the following form:

\[
<z_1, [\alpha_1, \beta_1]>, <z_2, [\alpha_2, \beta_2]>, \ldots, <z_m, [\alpha_m, \beta_m]>,
\]

where \(<z_1, [\alpha_1, \beta_1] = F_0, z_2, [\alpha_2, \beta_2] = S(z_1, \langle z_1, \ldots, [\alpha_{i-1}, \beta_{i-1}]\rangle)\) 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 \(c_j\) the cost of \(S(z_i, \langle z_{i-1}, \ldots, [\alpha_{i-1}, \beta_{i-1}]\rangle)\). 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 \(O(c_j)\) to \(O(1)\) for \(i = 1, \ldots, m\), at cost of more space usage. (Concretely, \(2\) 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 \(v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_l\)

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

\[
\begin{align*}
D_{v_j}[0] &= D_{v_{j-1}}[0] + 1; \\
D_{v_j}[i] &= \min\{D_{v_j}[i - 1] + w(z_i, \phi, D_{v_{j-1}}[i]) + w(\phi, e_j), \\
&\quad D_{v_{j-1}}[i - 1] + \delta(z_j, e_j)\}, \quad i > 0.
\end{align*}
\]

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 \(x\) against \(BWT(y)\). In \(T\), each node is labeled with a pair \(<e, [\alpha, \beta]>\) and there is an edge from \(v = <e, [\alpha, \beta]>\) to \(u = <e', [\alpha', \beta']>\) 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 \(x = acacga\), \(y = gatataca\). 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 \(x = gacga\) 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.](image)

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 \(<0, 1, \ldots, m>\). Subsequently, for any node \(v\) different of the root, its D-vector \(D_v\) is calculated from \(D_v\), the D-vector of its parent \(v\), in terms of formulae (5). For example, by exploring path \(P_1 = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \in V_2\), we will generate a searching sequence as below:

\[
<-,[1,8]>, <-,[1,3]>, <-,[1,1]>, <-,[2,2]>
\]

All their D-vectors (and also all the other nodes’ vectors in the tree) are shown in Fig. 3, in which we use \(D_0\) to represent \(D_{v_0}\) for simplicity.

![Figure 3: D-vectors.](image)

In essence, \(T\) is generated in the depth-first fashion, but the expansion of a path stops at a node \(v = <e, [\alpha, \beta]>\) 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 $k$ differences have been found. We will then backtrack to explore another path;
(2) $L_v = L_{<, [\alpha, \beta]}$ contains only one element ‘$\ast$’, 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 $v_1$ (on $P_1$ shown in Fig. 2) we have $D_v[5] = 2$ (see Fig. 3), which shows that some occurrences of the pattern with 2 differences 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 $v_2$ of $v_1$ and proceed continually.

The same analysis applies to $v_1$ on $P_3$.

In addition, we can observe that three paths $P_3, P_4$, and $P_5$ stop at a node $v$ such that $L_v$ contains only ‘$\ast$’, 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, which are 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 $(o, u)$ with $v$ being the parent of $u$. For root, its parent is set empty, represented as $\Phi$.

**Algorithm 1: stringM(**$x$, $BWT(y)$, $k$**)

**Input**: $x$ - 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. $(o, u)$ := pop(stack);
4. if $|s_o| \leq |x| + k$ 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

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 $|s_o| \leq |x| + k$, 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 $<0, 1, \ldots, m^+>^T$. If in $D_v$ 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 $L_{<, [\alpha, \beta]}$ to find $w = <e', [\alpha', \beta']>$ for each $e' \in \Sigma$ by calling $S()$, where $\alpha'$ and $\beta'$ 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 $x = gcaca$ against $BWT(y) = L$ and $F$ shown in Table 3 with $k = 2$, where $y = gataca$. We demonstrate its first seven steps. The complete tree generated is shown in Fig. 2.

Step 1: create root $v_0 = <\ast, [1, 8]>$, push $(\Phi, v_0)$ into stack. (See Fig. 4(a)).

Step 2: Pop out the top element $(\Phi, v_0)$. Generate $D_0 = <\ast, 1, 2, 3, 4, 5>^T$ for $v_0$. For each $e' \in \Sigma S(e', <\ast, [1, 8]>)$ will be executed, by which $v_1 = <a, [1, 3]>$, $v_2 = <\ast, [1, 1]>$, $v_8 = <<, [1, 1]>$, and $v_{12} = <<, [1, 2]>$. Otherwise, we will search and proceed continually.

Step 3: Pop out the top element $(v_0, v_1)$. $v_1 = <a, [1, 3]>$. Generate $D_1 = <\ast, 1, 2, 3, 4, 5>^T$ (see Fig. 3). For each $e' \in \Sigma S(e', <a, [1, 3]>)$ will be executed, by which $v_2 = <\ast, [1, 1]>$. If $v_8 = <<, [1, 1]>$ will be generated. Then, all $(a, v_1) (i = 1, 8, 12, 16)$ will be pushed into stack. (See Fig. 4(b)).

Step 4: Pop out the top element $(v_1, v_2)$. $v_2 = <<, [1, 1]>$. Generate $D_2 = <\ast, 2, 1, 2, 3, 4>^T$. For each $e' \in \Sigma S(e', <a, [1, 1]>$ will be executed, by which $v_3 = <a, [2, 2]>$ will be created and pushed into stack. (See Fig. 4(c)).

Step 5: Pop out the top element $(v_2, v_3)$. $v_3 = <a, [2, 2]>$. Generate $D_3 = <\ast, 3, 2, 1, 2, 2>^T$. Since $D_0[5] = 2$, an occurrence of the pattern has been found. Nothing will be pushed into stack and we backtrack. (See Fig. 4(e)).

Step 6: Pop out the top element $(v_3, v_4)$. $v_4 = <\ast, [1, 2]>$. Generate $D_4 = <\ast, 2, 2, 3, 4, 5>^T$. For each $e' \in \Sigma S(e', <\ast, [1, 2]>$ will be executed, by which $v_5 = <\ast, [3, 3]$ and $v_7 = <g, [1, 1]$ will be created and pushed into stack. (See Fig. 4(f)).

---

**Figure 4: Illustration for execution of stringM().**
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 \( T \) 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 \( D_v \) are larger than \( k \). Then, adding a suffix of \( x: z^j = z_{j+1} \ldots z_m \) gives a string \( s_x^j \) whose distance from \( x \) is \( k \), where ‘o’ represents the concatenation operation over substrings. If such a substring can be found in \( y \), then we can replace the search of \( z^j \) in \( T \) by that of \( v_z^j \) in \( T_P \). For example, in the tree shown in Fig. 2, the \( D \)-vector of \( v_4 \) is \( <2, 2, 2, 3, 3, 4, 5, 6> \). Thus, we need only to search for \( v_4 \) if it is possible to explore a single path (in \( T_P \) labeled with \( z^j \)). In fact, we can only explore the path starting with \( a \) (path \( P_2 \)) since ‘c’ cannot be found when searching \( L_{<t, 1, 2>} \). This path \( P_2 \) will break off when we meet ‘t’ which does not appear in \( a \). Assume that \( j_1, \ldots, j_h \) \((h > 0)\) are all the entries in the \( D \)-vector \( D_v \) for the current node \( v \) such that \( D_v[j] = k \) \((1 \leq i \leq h)\) while all the other entries > \( k \). Then, we can organize all the suffixes of \( x: z^i = z_{i+1} \ldots z_m \) 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 \( z^h \) \((1 \leq i \leq h)\) 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 \( x \) 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 \( x \). (As illustrated in Fig. 5, where we show the suffix tree for \( x = gcaca \) and its associated array.)

![Suffix tree and associated array](image)

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

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

- Let \( D_v[j_1], \ldots, D_v[j_h] \) be all those entries equal to \( k \). We will search \( T_P \) bottom-up, starting from all those leaf nodes pointed to by \( C[j_1], \ldots, C[j_h] \), 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 \( <j, [α_j, β_j]> \) for some \( j \). Then, corresponding to the suffix of \( x: z_{j+1} \ldots z_m \) along a path in \( T_P \), a search sequence will be generated, as shown below:

\[
<\bar{z}_j = [α_j, β_j], s_m, \ldots, s_{m'}>, \quad \alpha_{m'} > 0.
\]

Such a path stops when we have exhausted the whole suffix, or at a node \( <z_i, [α_i, β_i]> \) such that \( S(z_i, z_{i-1}, [α_{i-1}, β_{i-1}]) \) returns \( φ \).

5.2.2 Recognizing similar paths. The second improvement utilizes 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 \( s_x = e_1 \ldots e_j \). Denote by \( B_v \) the shortest suffix of \( s_x \) satisfying a condition that there exists a largest \( i \) such that the distance between \( z_i = z_{i+1} \ldots z_m \), a prefix of \( x \) and \( B_v \) is \( ≤ k \) equals the distance between \( z_i \) and \( s_x \). Then, we check whether \( B_v = [s_x] \). 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 (5), each entry \( D_v[i] \) \((1 ≤ i < m)\) is produced possibly in one of three ways:

1. \( D_v[i] = D_v[i - 1] + 1 \) (II),
2. \( D_v[i] = D_u[i - 1] + δ(z_i, e_j) \), and (III)
3. \( D_v[i] = D_u[i] + 1 \), 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 \( D_v[i - 1] + 1, D_u[i - 1] + δ(z_i, e_j) \), 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>
<thead>
<tr>
<th>Case</th>
<th>I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

In terms of these 7 possibilities, we present the following recurrence to calculate the length of the shortest suffix of \( s_x \) whose distance from \( z_j = z_{j+1} \ldots z_m \) is \( D_v[i] \). Let \( P = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j = v \) be the path from root \( (=v_0) \) to \( v (=v_j) \). We have

\[
A_{v_0}[i] = 0 \quad (1 ≤ i ≤ m);
\]

\[
A_{v_j}[1] = 1 \quad \text{if } e_j = z_j; \quad \text{otherwise, } A_{v_j}[1] = 0 \quad (j > 0);
\]

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

(6)
Special attention should be paid to case (1), (2), (3) and (4) in Table 4. In these cases, $A_0[i]$ is set to be $A_0[i - 1]$ although $D_0[i]$ is also possibly equal to $D_{0,i-1}[i + 1] + \delta(z_i, \epsilon)$, and/or $D_{0,i-1}[i + 1]$. This shows why the equation correctly computes the value. It is because if $A_0[i - 1]$ correctly records the length of the shortest suffix of $s_y$ such that its distance from $z_{i-1}$ is equal to $D_{0,i-1}[i - 1]$, then $A_0[i] = A_0[i - 1] + 1$ if $D_{0,i}[i] = D_{0,i-1}[i + 1] + 1$. The correctness of case (5) and (6), as well as case (7) can be easily checked.

Using $A_0[i]$, $B_0$ can be computed as follows.

Let $s_i = \epsilon_1 \ldots \epsilon_l$. We have

$$B_0 = \epsilon_{l-\alpha(v)} + \ldots + \epsilon_l,$$

(7) where $\alpha(v)$ equals $A_0[i]$ for the largest position $i$ in $\tilde{x}$ such that $D_{0,i}[i] \leq k$. For instance, for $\tilde{x} = gcaca$, and $s_0 = acat$, the we have $\alpha(v) = A_0[4] = 3$ (see A(4, 4) in Table 5) since $D_{0,4}[4] = 2$ (by formulae (5), $D_{0,4} = <4, 4, 3, 2, 2, 3>$) and $i = 4$ is the largest position in $\tilde{x} = gcaca$ such that $D_{0,i}[4] \leq 2$. Thus, $B_0 = \epsilon = s_0 = acat$.

If $\alpha(v) > 0$, $B_0 = \epsilon_{l-\alpha(v)} + \ldots + \epsilon_l = \epsilon_{l-1}s_0 \ldots \epsilon_l$, which is defined to be an empty string, represented as $\phi$.

We now discuss why we can terminate a path at a node $v$ if $|B_0| \neq |s_0|$, 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 $B_0$, has already 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 $B_0$. Let $u = \langle a_0, b_0 \rangle$ be the pair labeling $u$. There must be a child $w$ of the root, whose label is $\langle a_0', b_0' \rangle$ such that $[a_0', b_0'] \supseteq [a_0, b_0]$. Next, we check the child $u_1$ of $u$ on $P$. Since $[a'_0, b'_0] \supseteq [a_0, b_0]$, 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 $v$, and prove the above claim by a simple induction. See Fig. 6 for illustration. It remains to expat that cutting off such a path at the end.

Figure 6: Illustration for $B_0$ and path similarity.

$v$ in the case of $|B_0| \neq |s_0|$ will not lead to loss of answers. For this, we show another important property of $B_0$.

**Lemma 5.3.** Let $u, v$ be two nodes in $T$, not related by the ancestor-descendant relationship. If $s_u = B_0$, then $D_0[i] = D_u[i]$ (1 $\leq i \leq m$) if $D_0[i] \leq k$.

**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 $s_0 = acat$ checked against $\tilde{x} = gcaca$ with $k = 2$. For $s_0, B_0 = \epsilon = \epsilon_0 = s_0$, as discussed above. In Fig. 7(b), we show the $D$-matrix $D'$ for $s_0 = cat$ checked against $\tilde{x}$. 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 $\leq 2$ in both $D$ and $D'$, but also trace their generation. In Fig. 7(c), we show only those paths starting from an entry $\leq 2$ in $D_k$ (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 $s_0$, we can regain the answers which get lost due to cutting off $s_0$. So, we have the following proposition.

**Proposition 5.4.** During the execution of $stringM(\tilde{x}, BWT(y), k)$, cutting off $s_0$ if $|s_0| \neq |B_0|$ 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 of rankAll [11], the cost of $ST'(\epsilon, [\alpha, \beta])$ is $O(1)$. Thus, the time complexity of $stringM(\tilde{x}, BWT(y), k)$ is bounded by $O(m \cdot |T|)$. It is because for each node $v$ in $T$, we need $O(m)$ time to compute $D_0$. 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 $stringM(\tilde{x}, BWT(y), k)$. We notice the following facts:

- The probability of a character $c \in \Sigma$ 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 $\left(\frac{1}{|\Sigma|}\right)^j$.

- 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 $0 \leq i \leq j$ 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 nonidentical character for the $p$th character in $x$ and then insert $j - 1$ new characters around the $p$th position, by which we can possibly change $x$ to $(|\Sigma| - 1)|\Sigma|^{j-1}$ different strings.

Thus, any $\bar{x}$ in $T$ that is $k$ (or less) different from $\tilde{x}$ can be represented as an operation sequence: $o_1 \ldots o_q$ (for some $l \leq k$) with each $o_i (1 \leq i \leq l)$ being a pair $[p, q]$, where $p \in \{1, \ldots, m\}$, and $q$ is a symbol 'd', representing 'deleting' the $p$th character in $\tilde{x}$; or a sequence of characters: $c_1 \ldots c_j$, inserted around position $p$, or $c_i$
Adding all the above expressions together, we get equation (9). It can be proven by a simple induction.

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 $x$. 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 $x$. The average size of this class is

$$\sum_{j=1}^{k} \frac{1}{k} |\Sigma|^{i} \Gamma(m - 1, k - j).$$

- Class 4 contains all those sequences whose first operation is to substitute the first character of $x$ followed by $j - 1$ insertions. The average size of this class is

$$\sum_{j=1}^{k} |\Sigma|^{i} \Gamma(m - 1, k - j).$$

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 5.1**. The average time complexity of $\text{stringM}(\bar{y}, \text{BW}(y), k)$ is bounded by $O(k|\Sigma|^{2k})$.

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

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

(10)

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

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 technique described in [14], we only need to calculate $O(k)$ entries in a $D$-vector. Thus, the overall space requirement, in addition to the index, is bounded by $O(k(m + k))$.

6 PARTITION OF PATTERN STRINGS

From the above discussion, we have seen that the average running time of Algorithm $\text{stringM}()$ is bounded by $O(k|\Sigma|^{2k})$. 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 $l$ segments, denoted as $x = P_1 \ldots P_l$ with each $P_i = x_{l(i-1) \ldots l_i}$, for $1 \leq l \leq l - 1$, and $P_l = x_{l(l-1) \ldots l_m}$, where $r = [m/l]$. Then, we run $\text{stringM}()$ on each $P_i$ against $\text{BW}(y)$ with $k' = [k/l]$ repetitions to find out 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: $s_{i,j} = y_{j - (r+1)+1} \ldots y_{j - (r+1)+1+m+k}$ 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 $s_{j,i}$**

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 $|k|/l$ differences. It is because if no $P_i$ is present in $s$ with $|k/l|$ or less than $|k/l|$ differences, then when we transform $x$ to $s$, by which each $P_i$ will be transformed into a subsegment $s_{j,i}$ of $x$, the number of needed operations is larger than $1(|k/l| + 1) > l(k/l) = k$. 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 somewhat 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

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

where $T_i$ is the tree generated when running $stringM(\cdot)$ on $P_i$ and $BWT(y)$ with $|k/1|$ 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 $O(m \cdot h(m + 2k)) = O(hm^2)$ 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 $O(k \cdot m)$ by using a more sophisticated method [14].

7 EXPERIMENTS

In our experiments, we have tested altogether 7 strategies:

- Ukkonen’s online method ($u$-o for short, [58]),
- Chang-Lawer’s first method (ce-1 for short, [14]),
- Chang-Lawer’s second method (ce-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 64 GB. For time measurements, we used the Unix time commands. In addition, the suffix trees for patterns (in the Chang-Lawer’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.ensembl.org/pub/release-93/) and SMS (https://www.bioinformatics.org/sms2/random_protein.html), respectively. The size of the alphabet for the protein sequences 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.

<table>
<thead>
<tr>
<th>reference sequences*</th>
<th>num. characters</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gorilla</td>
<td>3,063,403,506</td>
<td>406.817</td>
</tr>
<tr>
<td>Danio Rerio (Zebrafish)</td>
<td>1,373,472,378</td>
<td>173.142</td>
</tr>
<tr>
<td>Gorilla Chr1</td>
<td>228,908,641</td>
<td>25.03</td>
</tr>
<tr>
<td>Protein-1</td>
<td>144,000,000</td>
<td>15.92</td>
</tr>
<tr>
<td>Protein-2</td>
<td>30,000,000</td>
<td>3.04</td>
</tr>
</tbody>
</table>

*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 sequences 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 $k$ ($\leq 6$). 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 theoretical time complexity is bounded by $O(m \cdot \min(n, m + 1) \cdot |\Sigma|^{k+1} + n)$ [59]. During the execution of the Myers’, 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 $O(|\Sigma|^{m+1})$. 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 neighbourhodds. 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: Number of nodes in $T$

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.4K</td>
<td>25K</td>
<td>278.5K</td>
<td>2M</td>
<td>10M</td>
<td>39.72M</td>
<td>92M</td>
</tr>
</tbody>
</table>

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
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 consistently 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 patterns 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 $\geq 6$, the pattern will be divided into $l$ parts such that $\left\lfloor \frac{k}{l} \right\rfloor < 6$. 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, however, their performance is definitely worse than ours, even with quickly increased 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 surviving 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

<table>
<thead>
<tr>
<th>$k$</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>step-1</td>
<td>23.1</td>
<td>23.1</td>
<td>173.2</td>
<td>172.9</td>
<td>1187.0</td>
<td>1187.5</td>
</tr>
<tr>
<td>step-2</td>
<td>97.41</td>
<td>122.1</td>
<td>263.4</td>
<td>311.7</td>
<td>493.22</td>
<td>565.58</td>
</tr>
<tr>
<td>num.</td>
<td>30.5K</td>
<td>30.5K</td>
<td>52.9K</td>
<td>52.7K</td>
<td>69.5K</td>
<td>69.3K</td>
</tr>
<tr>
<td>size</td>
<td>353</td>
<td>364</td>
<td>388</td>
<td>399</td>
<td>428</td>
<td>439</td>
</tr>
</tbody>
</table>

### Table 9: Two-step execution details on Zebrafish genome

<table>
<thead>
<tr>
<th>$k$</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>step-1</td>
<td>58.73</td>
<td>58.57</td>
<td>187.5</td>
<td>187.6</td>
<td>953.0</td>
<td>952.4</td>
</tr>
<tr>
<td>step-2</td>
<td>18.41</td>
<td>21.48</td>
<td>34.24</td>
<td>38.85</td>
<td>60.33</td>
<td>60.70</td>
</tr>
<tr>
<td>num.</td>
<td>3638</td>
<td>3633</td>
<td>4709</td>
<td>4702</td>
<td>6376</td>
<td>6365</td>
</tr>
<tr>
<td>size</td>
<td>423</td>
<td>433</td>
<td>444</td>
<td>455</td>
<td>463</td>
<td>474</td>
</tr>
</tbody>
</table>

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
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 \( k \) (\( \geq 40 \)), 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 \( k'=[k/l] \) 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: Number of segments checked in Step 2.

<table>
<thead>
<tr>
<th>( k )</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l=10 )</td>
<td>30.5K</td>
<td>30.5K</td>
<td>52.9K</td>
<td>52.7K</td>
<td>69.5K</td>
<td>69K</td>
</tr>
<tr>
<td>( l=12 )</td>
<td>28.7K</td>
<td>56.2K</td>
<td>56.9K</td>
<td>55.8K</td>
<td>60.5K</td>
<td>61.4K</td>
</tr>
<tr>
<td>( l=15 )</td>
<td>30.5K</td>
<td>30.5K</td>
<td>52.9K</td>
<td>52.7</td>
<td>69.5K</td>
<td>69.3K</td>
</tr>
</tbody>
</table>

### 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 consistently demonstrates a big difference between ours and all the tested online methods. All of them, including ours, are almost pattern-length independent.

### 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. Also, extensive experiments are conducted, showing that our method is not only a theoretically, but also a practically efficient strategy. We believe that our algorithm as a tool will be very useful for the biological research.