Title: BWT Arrays and Mismatching Trees: A New Way for String Matching with k Mismatches

Article Type: Regular Paper (10 - 40 pages)

Section/Category: A - Algorithms, automata, complexity and games

Keywords: string matching
BWT transformation

Corresponding Author: Dr. Yangjun Chen,

Corresponding Author's Institution: University of Winnipeg

First Author: Yangjun Chen

Order of Authors: Yangjun Chen; Yujia Wu
BWT Arrays and Mismatching Trees: A New Way for String Matching with $k$ Mismatches

YANGJUN CHEN and YUJIA WU
The University of Winnipeg

In this paper, we discuss an efficient and effective index mechanism to do the string matching with $k$ mismatches, by which we will find all the substrings in a target string $s$ having at most $k$ positions different from a pattern string $r$. The main idea is to transform $s$ to a BWT-array as an index, denoted as $\text{BWT}(s)$, and search $r$ against it. During the process, the precomputed mismatch information of $r$ will be utilized to speed up the $\text{BWT}(s)$’s navigation. In this way, the time complexity can be reduced to $O(kn' + n + m\log m)$, where $m = |r|$, $n = |s|$, and $n'$ is the number of leaf nodes of a tree structure, called a mismatching tree, produced during a search of $\text{BWT}(s)$. In the case of $m \geq 2(k + 1)$, the average value of $n'$ is bounded by $O\left(\left(1 + \frac{1}{|\Sigma|}\right)^{k+1}\right)$, where $\Sigma$ is an alphabet from which we take symbols to make up target and pattern strings. Extensive experiments have been conducted, which show that our method for this problem is promising.

Categories and Subject Descriptors: F.2.2 [Analysis of algorithms and Problem Complexity]: Non-numerical Algorithms and Problems Pattern matching; computation on discrete structures

General Terms: Databases, Algorithms, Performance

Additional Key Words and Phrases: String matching; DNA sequences; tries; BWT-transformation

1. INTRODUCTION

By the string matching with $k$ mismatches, we mean a problem to find all the occurrences of a pattern string $r$ in a target string $s$ with each occurrence having up to $k$ positions different between $r$ and $s$. This problem is important for DNA databases to support the biological research, where we need to locate all the appearances of a read (a short DNA sequence) in a genome (a very long NDA 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.

As an example, consider a target $s = ccacacagaagc$, and a pattern $r = aaaaacacaac$. Assume that $k = 4$. Let us see whether there is an occurrence of $r$ with $k$ mismatches that starts at the third position in $s$.

```
 a a a a a c a a a c
 c c a c a c a g a a g c c
```

At only four locations $s$ and $r$ have different characters, implying an occurrence of $r$ starting at the third position of $s$.
prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1539-9087/2010/03-ART39 $15.00

Note that the case $k = 0$ is the extensively studied string matching problem.

This topic has received much attention in the research community and many efficient algorithms have been proposed, such as [2, 15, 21, 29, 35, 44]. Among them, [21] and [29] are two on-line algorithms (no indexes) with the worst-case time complexities bounded by $O(kn + mlogn)$, where $n = |s|$ and $m = |r|$. By these two methods, the mismatching information among substrings of $r$ is used to speed up the working process. The methods discussed in [2] and [44] are also on-line strategies, but with a slightly better time complexity $O(n\sqrt{k} logk)$ by utilizing the periodicity within $r$. Only the algorithms discussed in [15, 35] are index-based. By the method discussed in [15], a (compressed) suffix tree over $s$ is created. Then, a brute-force tree searching is conducted to find all the possible string matchings with $k$ mismatches. Its time complexity is bounded by $O(m + n + (clogn)/k!)$, where $c$ is a very large constant. For DNA databases, this time complexity can be much worse than $O(nk)$ since $n$ tends to be very large and $k$ is often set to be larger than 10. By the method discussed in [35], $s$ is transformed to a BWT-array (denoted BWT(s)) as an index [10, 31]. In comparison with suffix trees, BWT(s) uses much less space [20]. However, the time complexity of [35] is bounded by $O(mn' + n)$, where $n'$ is the number of leaf nodes of a tree (forest) produced during the search of BWT(s). Again, this time requirement can also be much worse than the best on-line algorithm for large patterns. Thus, simply indexing $s$ is not always helpful for $k$ mismatches. The reason for this is that in both the above index-based methods neither mismatching information nor periodicity within $r$ is employed, leading to a lot of redundancy, which shadows the benefits brought by indexes. However, to use such information efficiently and effectively in an indexing environment is very challenging since in this case $s$ will no longer be scanned character by character and the auxiliary information extracted from $r$ cannot be simply integrated into an index searching process.

In this paper, we address this issue, and propose a new method for the $k$-mismatch problem, based on a BWT-transformation, but with the mismatching information within $r$ being effectively utilized.

Specifically, two techniques are introduced, which will be combined with a BWT-array scanning:

- An efficient method to calculate the mismatches between $r[i .. m]$ and $r[j .. m]$ ($i, j \in \{1, ..., m\}$, $i \neq j$), where $r[i .. m]$ represents a substring of $r$ starting from position $i$ and ending at position $m$. The mismatches between them is stored in an array $R$ such that if $R[p] = q$ then we have $r[i + q \cdot 1] \neq r[j + q \cdot 1]$ and it is their $p$th mismatch.
- A new tree (forest) structure $D$ to store the mismatches between $r$ and different segments of $s$. In $D$, each node $v$ stores an integer $i$, indicating that there are some positions $i_1, i_2, ..., i_l$ in $s$ such that $s[i_q + i \cdot 1] \neq r[i]$ ($q = 1, ..., l$). If $v$ is at the $p$th level of $D$, it also shows that it is the $p$th mismatch between each $s[i_q .. i_q + i \cdot 1]$ and $r$.

By using these two techniques, the time complexity can be reduced to $O(kn' + n)$. In the case of $m \geq 2(k + 1)$, the average value of $n'$ is bounded by $O\left(\left.\left(1 + \frac{1}{|\Sigma|}\right)^{k+1}\right\rangle\right)$, where $\Sigma$ is the alphabet. Our experiment over a DNA database shows that $n' \ll n$.

The remainder of the paper is organized as follows. In Section 2, we summarize the notations used throughout the paper. In Section 3, we review the related work. In Section 4, we briefly describe how a BWT-transformation can be used to speed up string matches. Section 5 is devoted to the discussion of our algorithm to find all the occurrences of $r$ in $s$, but up to $k$ mismatches. In Section 6, the time complexity of the algorithm is analyzed. Section 7 reports the test results. Finally, we conclude with a short summary and a brief discussion on the future work in Section 8.
2. Notations

Throughout the paper, a bunch of symbols and notations are used, which are summarized in the following table for reference.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>a pattern string</td>
</tr>
<tr>
<td>(r[i])</td>
<td>(i)-th character in (r)</td>
</tr>
<tr>
<td>(r[i..j])</td>
<td>substring from (i)-th to (j)-th position in (r)</td>
</tr>
<tr>
<td>(m)</td>
<td>(m =</td>
</tr>
<tr>
<td>(s)</td>
<td>a target string</td>
</tr>
<tr>
<td>(n)</td>
<td>(n =</td>
</tr>
<tr>
<td>(\Sigma)</td>
<td>an alphabet</td>
</tr>
<tr>
<td>(x, y, z, a, b, c)</td>
<td>characters in a string</td>
</tr>
<tr>
<td>(\bar{s})</td>
<td>reverse of (s)</td>
</tr>
<tr>
<td>(L = \text{BWT}(s))</td>
<td>BWT array for (s)</td>
</tr>
<tr>
<td>(F)</td>
<td>an array, in which the same characters are clustered</td>
</tr>
<tr>
<td>(rk_L(e))</td>
<td>rank of an element in (L)</td>
</tr>
<tr>
<td>(rk_F(e))</td>
<td>rank of an element in (F)</td>
</tr>
<tr>
<td>(T)</td>
<td>search tree, which is constructed when searching (s) to find all occurrences of (r) in (s)</td>
</tr>
<tr>
<td>(P)</td>
<td>a path in (T)</td>
</tr>
<tr>
<td>(P[i])</td>
<td>the (i)-th node on (P)</td>
</tr>
<tr>
<td>(D)</td>
<td>match tree, whose nodes are created to represent mismatching information</td>
</tr>
<tr>
<td>(u, v, w)</td>
<td>nodes in (T)</td>
</tr>
<tr>
<td>(\pi = \langle x, [\alpha, \beta]\rangle)</td>
<td>a pair representing a segment of (F)</td>
</tr>
<tr>
<td>(L_x)</td>
<td>a segment in (L), corresponding to (\pi)</td>
</tr>
<tr>
<td>(x[])</td>
<td>an array created for (x \in \Sigma)</td>
</tr>
<tr>
<td>(A_x[])</td>
<td>an compact version of (x[])</td>
</tr>
<tr>
<td>(\text{search}(z, \pi))</td>
<td>a procedure to find the first and last appearance of (z) within a segment corresponding to (\pi)</td>
</tr>
<tr>
<td>(\text{search}(\pi))</td>
<td>An improved version of (\text{search}(z, \pi))</td>
</tr>
<tr>
<td>(Y_a, Y_c, Y_e, Y_t)</td>
<td>starting position of (a)-segment, (c)-segment, (g)-segment, (t)-segment in (F)</td>
</tr>
<tr>
<td>(Z_a, Z_c, Z_e, Z_t)</td>
<td>ending position of (a)-segment, (c)-segment, (g)-segment, (t)-segment in (F)</td>
</tr>
<tr>
<td>(sL(P)())</td>
<td>a procedure to search (L) from a certain position upwards to a higher position</td>
</tr>
<tr>
<td>(sDown())</td>
<td>a procedure to search (L) from a certain position downwards to a lower position</td>
</tr>
<tr>
<td>(Q)</td>
<td>A queue used to search a subtree of (D) in the breadth-first manner</td>
</tr>
<tr>
<td>(R_u)</td>
<td>An array storing the mismatching information between (r[i..m]) and (r[i..m])</td>
</tr>
</tbody>
</table>

3. RELATED WORK

The string matching problem has always been one of the main focuses in computer science. A huge number of algorithms have been proposed. Roughly speaking, all of them can be divided into two categories: exact matching and inexact matching. By the former, all the occurrences of a pattern string \(r\) in a target string \(s\) will be searched. By the latter, a best alignment between \(r\) and \(s\) (i.e., a correspondence with the highest score) is searched in terms of a given distance function or a score matrix \(M\), which is established to indicate the relevance between different characters.

- Exact matching

  The first interesting algorithm for this problem is the famous Knuth-Morris-Pratt’s algorithm \([27]\), which scans both \(r\) and \(s\) from left to right and uses an auxiliary next-table (for \(r\)) containing the so-called shift information (or say, failure function values) to indicate how far to shift the pattern from right to left when the current character in \(r\) fails to match the current character in \(s\). Its time complexity is bounded by \(O(m + n)\), where \(m = |r|\) and \(n = |s|\). (By the shift information, we mean a largest integer \(j\) associated with a position \(i\) in \(r\)
such that \( r[1..j] = r[i - j + 1..i] \). Thus, if the current character from the target does not match \( r[i + 1] \), we will compare \( r[j + 1] \) with the character next to the current one at a next step.) The Boyer-Moore’s approach [9] works a little bit better than the Knuth-Morris-Pratt’s. In addition to the next-table, a skip-table skip (also for \( r \)) is kept, in which each entry \( \text{skip}[w] \) is a smallest integer \( j \) such that \( r[m - j] = w \). (Here, we notice that the entries in skip are indexed by characters \( w \) in the alphabet \( \Sigma \).) For a large alphabet and small pattern, the expected number of character comparisons is about \( nm \), and is \( O(m + n) \) in the worst case. These two methods have sparked a series of subsequent research on this problem [1, 32, 50, 51]. Especially, the idea of the ‘shift information’ has also been adopted by Aho and Corasick [1] for multiple pattern matches, by which \( s \) is searched for an occurrence of any one of a set of \( l \) patterns: \( \{ r_1, r_2, \ldots, r_l \} \). Their algorithm needs only \( O(\sum_{i=1}^{l} |r_i| + n) \) time. This method has been slightly improved in different ways. In [55], Commentz-Walter combines the Boyer-Moore’s technique into the Aho-Corasick’s algorithm. In [56], Wu and Mamber extend the Boyer-Moore’s algorithm to currently search multiple pattern strings. Instead of using bad character heuristics to compute shift values, they utilize a character block containing 2 or 3 characters. In addition, hash tables are created to link the blocks and the related patterns. In [57], a concept of superalphabets is introduced, in which each (super) character corresponds to a set of \( q \)-grams (each being a substring from a certain pattern and represented as a bit string, called a signature, generated by using a hash function). In this way, a super automaton can be created, in which each transition is labeled with a super character. \( s \) will also be handled as a sequence of \( q \)-grams and searched in the same way as the Aho and Corasick’s algorithm. The main problem of this method is the false positive and a very time-consuming verification process is needed. In [12], Crochemore et al. combine the directed acyclic word graphs into the Aho-Corasick’s algorithm. If the total length of all patterns is polynomial with respect to the shortest length \( m' \) of a pattern, the average number of comparisons is \( O((n/m')\log m') \).

However, all the improved algorithms have the same worst-case time complexity as the Aho-Corasick’s.

In situations where a fixed string \( s \) is to be searched repeatedly, it is worthwhile constructing an index over \( s \), such as suffix trees [42, 53], suffix arrays [40], and more recently the BWT-transformation [10, 13, 31, 35, 36]. A suffix tree is in fact a trie structure [26] over all the suffixes of \( s \); and by using the Weiner’s algorithm [53] it can be built in \( O(n) \) time. However, in comparison with the BWT-transformation, a suffix tree needs much more space. Especially, for DNA sequences the BWT-transformation works highly efficiently due to the small alphabet \( \Sigma \) of DNA strings. By the BWT, the smaller \( \Sigma \) is, the less space will be occupied by the corresponding indexes. According to a survey done by Li and Homer [37] on sequence alignment algorithms for next-generation sequencing, the average space required for each character is 12 - 17 bytes for suffix trees while only 0.5 - 2 bytes for the BWT. The experiments reported in [16] also confirm this distinction. For example, the file size of chromosome 1 of human is 270 Mb. But its suffix tree is of 26 Gb in size while its BWT needs only 390 Mb – 1 Gb for different compression rates of auxiliary arrays, completely handleable on PC or laptop machines.

By the hash-table-based algorithms [25], short substrings called ‘seeds’ will be first extracted from a pattern \( r \) and a signature (a bit string) for each of them will be created. The search of a target string \( s \) is similar to that of the Brute Force searching, but rather than directly comparing the pattern at successive positions in \( s \), their respective signatures are compared. Then, stick each matching seed together to form a complete alignment. Its expected time is \( O(m + n) \), but in the worst case, which is extremely unlikely, it takes \( O(mn) \) time. The hash technique has also been extensively used in the DNA sequence research [23, 33, 38, 47]. However, almost all experiments show that they are generally inferior to the suffix tree and the BWT index in both running time and space requirements.
- **Inexact matching**

By the inexact matching, we will find, for a certain pattern \( r \) and an integer \( k \), all the substrings \( s' \) of \( s \) such that \( d(s', r) \leq k \), where \( d \) is a distance function. In terms of different distance functions, we distinguish between two kinds of inexact matches: string matching with \( k \) mismatches and string matching with \( k \) errors. A third kind of inexact matching is that involving Don’t Care, or wild-card symbols which match any single symbol, including another Don’t Care.

**k mismatches** When the distance function is the *Hamming* distance, the problem is known as the string matching with \( k \) mismatches [2, 30]. By the Hamming distance, the number of differences between \( r \) and the corresponding substring \( s' \) is counted. There are a lot of algorithms proposed for this problem, such as [2, 4, 5, 22, 29, 30, 44, 50, 51]. They are all online algorithms. Except those discussed in [2, 5, 22, 29, 30, 44], all the other methods have the worst-case time complexity \( O(mn) \). In [5], a method called the *shift-add* is discussed, by which the mismatches are represented by bit strings and the bit string ‘shift’ and bit-wise ‘and’ operations are used to check string matches. But this method is efficient only for small string patterns (i.e., when \( m \log m \) is smaller than the system word-size). For large patterns, multiple-precision arithmetic operations are required in the manipulation of \( s \) and the pre-process of \( r \), and the running time then becomes quadratic. The methods discussed in [22] and [30], however, need only \( O(kn + m \log m) \) time, by which the mismatching arrays for \( r \) are precomputed and exploited to speed up the search of \( s \). The methods discussed in [2, 44] work slightly better, by which the periodicity within \( r \) is utilized. Their time complexity is bounded by \( O(n \sqrt{k} \log k) \). The algorithm discussed in [35] is index-based, by which \( s \) is transformed to a BWT-array, used as an index. Its time complexity is bounded by \( O(mn' + n) \), where \( n' \) is the number of leaf nodes of a tree produced during the search of \( \text{BWT}(s) \). If \( m \) is large, it can be worse than all those on-line methods discussed in [2, 22, 30, 44]. Another index-based method is based on a brute-force searching of suffix trees [14]. Its time complexity is bounded by \( O(m + n + (\log n)^2/k!) \), where \( c \) is a very large constant. It can also be worse than an online algorithm when \( n \) is large and \( k \) is larger than a certain constant.

**k errors** When the distance function is the *Levenshtein* distance, the problem is known as the string matching with \( k \) errors [6]. By the Levenshtein distance, we have

\[
d_{i,j} = \min\{d_{i-1,j} + w(r_i, \phi), \quad d_{i,j-1} + w(\phi, s'_j)\}.
\]

where \( d_{i,j} \) represents the distance between \( r[1 .. i] \) and \( s'[1 .. j] \), \( r_i \) \((s'_j)\) the \( i \)th character in \( r \) (resp. \( j \)th character in \( s' \)), \( \phi \) an empty character, and \( w(r_i, s'_j) \) the cost to transform \( r_i \) into \( s'_j \).

Also, many algorithms have been proposed for this problem [6, 11, 19, 52]. They are all some kinds of variants of the *dynamic programming* paradigm [18] with the worst-case time complexity bounded by \( O(mn) \). However, by the algorithm discussed in [11], the expected time can reach \( O(\sqrt{k}n) \).

**don’t care** As a different kind of inexact matching, the string matching with Don’t-Cares has been a third active research topic for decades, by which we may have wild-cards in \( r \), in \( s \), or in both of them. A wild card matches any character. Due to this property, the ‘match’ relation is no longer transitive, which precludes straightforward adaption of the shift information used by Knuth-Morris-Pratt and Boyer-Moore. Therefore, all the methods proposed to solve this problem seem not so skilful and in general need a quadratic time [45]. Using a suffix array as the index, however, the searching time can be reduced to \( O(\log n) \) for some patterns, which contain only a sequence of consecutive Don’t Cares [41].

## 4. BWT-TRANSFORMATION

In this section, we give a brief description of the BWT transformation to provide a discussion background.
4.1 BWT and String searching

We use $s$ to denote a string that we would like to transform. Assume that $s$ terminates with a special character $\$, which does not appear elsewhere in $s$ and is alphabetically prior to all other characters. In the case of DNA sequences with $\Sigma = \{a, c, g, t\}$, we have $\ < a < c < g < t$. As an example, consider $s = \text{ccagaca}\$. We can rotate $s$ consecutively to create eight different strings, and put them in a matrix as illustrated in Fig. 1(a).

\[
\begin{align*}
\text{c c a g a c a } & \quad \text{c c a g a a } \\
\text{c a g a c a } & \quad \text{a g a c a c } \\
\text{a g a c a c } & \quad \text{g a c a a g } \\
\text{a c a g a c } & \quad \text{c a a g a g } \\
\text{a c a a g a c } & \quad \text{c c a a g a } \\
\text{c c a a g c } & \quad \text{g a c a c a a } \\
\text{c c a a g c a } & \quad \text{c c a a g c a a } \\
\text{c c a a g c a a } & \quad \text{c c a a g c a a a } \\
\end{align*}
\]

(a)

Now we sort the rows of the matrix alphabetically, and get another matrix, as demonstrated in Fig. 1(b), which is called the Burrow-Wheeler Matrix [10, 31] and denoted as $\text{BWM}(s)$. Especially, the last column $L$ of $\text{BWM}(s)$, read from top to bottom, is called the $\text{BWT}$-transformation (or the $\text{BWT}$-array) and denoted as $\text{BWT}(s)$. So for $s = \text{ccagaca}\$, we have $\text{BWT}(s) = \text{acgacacag}$ (see Fig. 1(c)). The first column is referred to as $F$.

Special attention should be paid to Fig. 1(b) and 1(c). In both of them, for ease of explanation, the position of a character in $s$ is represented by its subscript. (That is, we rewrite $s$ as $c_1c_2a_1a_2c_3c_4a_3\$.) For example, $a_2$ representing the second appearance of $a$ in $s$; and $c_1$ the first appearance of $c$ in $s$. In the same way, we can check all the other appearances of different characters.

When ranking the elements $e$ in both $F$ and $L$ in such a way that if $e$ is the $i$th appearance of a certain character it will be assigned $i$, the same element will get the same number in these two columns. For example, in $F$ the rank of $a_3$, denoted as $\text{rk}_F(a_3)$, is 1, showing that $a_3$ is the first appearance of $a$ in $F$. Its rank in $L$, $\text{rk}_L(a_3)$ is also 1. (See Fig. 2(a) for illustration.) We can check all the other elements and find that this property, called the rank correspondence, holds for all the elements. That is, for any element $e$ in $s$, we always have

\[
\text{rk}_F(e) = \text{rk}_L(e)
\]

(1)

According to this property, a string searching can be very efficiently conducted. To see this, let us consider a pattern string $r = \text{aca}$ and try to find all its occurrences in $s = \text{ccagaca}\$.

First, we notice that we can store $F$ as $|\Sigma| + 1$ intervals, such as $F_1 = F[1 .. 1]$, $F_2 = F[2 .. 4]$, $F_3 = F[5 .. 7]$, $F_4 = F[8 .. 8]$, and $F_5 = \varnothing$ for the above example (see Fig. 1(c).) We can also represent a segment within an $F_i$ (with $x \in \Sigma$) as a pair of the form $<x, [a, b]>$, where $a \leq b$ are two ranks of $x$. Thus, we have $F_0 = F[2 .. 4] = <a, [1, 3]>$, $F_2 = F[5 .. 7] = <c, [1, 3]>$, and $F_8 = F[8 .. 8] = <g, [1, 1]>$.

We will also use $Y_x$ and $Z_x$ to represent the positions of the first and the last element of $F_x$ in $F$, respectively. For example, $Y_0$ is 2 and $Z_0$ is 4. Then, for a given $<x, [a, b]>$, its segment in $F$ can be easily determined: $F[Y_x + a - 1 .. Y_x + b - 1]$. In addition, we can use $L_x$ to represent a range in $L$ corresponding to a pair $x = <x, [a, b]>$. For example, in Fig. 1(c), $L_{<a, [1, 3]>} = L[2 .. 4]$, $L_{<a, [2, 3]>} = L[3 .. 4]$, $L_{<b, [1, 2]>} = L[5 .. 6]$, and so on.

Finally, we use a procedure $\text{search}(z, \pi)$ to search $L_z$ to find the first and the last rank of $z$ (denoted as $\alpha'$ and $\beta'$, respectively) within $L_z$, and return $<z, [\alpha', \beta']>$ as the result:
\[ search(z, π) = \begin{cases} <z, [α', β']>, & \text{if } z \text{ appears in } L_π^c \\ φ, & \text{otherwise.} \end{cases} \] (2)

To locate \( r \) in \( s \), we work on the characters in \( r \) in the reverse order (referred to as a \textit{backward search}). That is, we will search \( ρ \) (reverse of \( r \)) against \( \text{BWT}(s) \), as shown below.

Step 1: Check \( r[3] = a \) in the pattern string \( r \), and then figure out \( F_a = F[2 .. 4] = <a, [1, 3]> \). (See Fig. 2(a) for illustration.)

Step 2: Check \( r[2] = c \). Call \( search(c, L_{\text{lo}}, [1, 3]) \). It will search \( L_{\text{lo}}[1, 3] = L[2 .. 4] \) to find a range bounded by the first and last rank of \( c \). Concretely, we will find \( rk_L(c_1) = 1 \) and \( rk_L(c_2) = 2 \). So, \( search(c, L_{\text{lo}}, [1, 3]) \) returns \( <c, [1, 2]> \). It is \( F[5 .. 6] \) since \( Y_c + 1 - 1 = 5 \) and \( Y_c + 2 - 1 = 6 \) (see Fig. 2(b)).

Step 3: Check \( r[3] = a \). Call \( search(a, L_{<}, [1, 2]) \). Notice that \( L_{<} = L[5 .. 6] \). So, \( search(a, L_{<}, [1, 2]) \) returns \( <a, [2, 2]> \). It is \( F[2 .. 2] \). Since now we have exhausted all the characters in \( r \) and \( F[2 .. 2] \) contains only one element, one occurrence of \( r \) in \( s \) is found. It is represented by \( a_2 \) in \( s \) (see Fig. 2(c)). (In general, let \( l \) be the number of entries in the segment (in \( F \)) found in the last step of such a process. Then, there are \( l \) occurrences of \( r \) in \( s \) with each indicated by an entry in that segment.)

The above working process can be represented as a sequence of three pairs: \( <a, [1, 3]> \), \( <c, [1, 2]> \), \( <a, [2, 2]> \). In general, for \( r = c_1 \ldots c_m \), its search against \( \text{BWT}(s) \) can always be represented as a sequence of pairs (with each made up of a character and an interval):

\[ <x_1, [α_1, β_1]>, \ldots, <x_m, [α_m, β_m]>, \]

where \( <x_i, [α_i, β_i] > = F_{x_i} \), and \( <x_i, [α_i, β_i] > = search(x_i, L_{\text{lo}}, [α_{i-1}, β_{i-1}]) \) for \( 1 < i \leq m \). We call such a sequence as a \textit{search sequence}. Thus, the time used for this process is bounded by \( O(∑^{m}_i τ_i) \), where \( τ_i \) is the time for an execution of \( search(x_i, L_{\text{lo}}, [α_{i-1}, β_{i-1}]) \). However, this time complexity can be reduced to \( O(m) \) by using the so-called \textit{rankAll} method (with more space to be used [31]) and the multi-character checking to be discussed in Section 7.

From the above discussion, we can observe a most important property of the \textit{BWT} transformation, by which we check, in each step, a subset of characters (represented by a subsegment of \( F \)) from a target string \( s \) while by any on-line strategy only one character from \( s \) is checked in each step.

**4.2 Construction of BWT arrays**

A BWT-array can be constructed in terms of a relationship to the suffix arrays [10, 20, 31, 48].

![Figure 2. Sample trace](image-url)
As mentioned above, a string \( s = a_1a_2 \ldots a_n \) is always ended with $ (i.e., a_i \in \Sigma \text{ for } i = 1, \ldots, n-1, \text{ and } a_n = $). Let \( s[i] = a_i \) (\( i = 1, 2, \ldots, n \)) be the \( i \)-th character of \( s \), \( s[i..j] = a_i \ldots a_j \) a substring and \( s[i..n] \) a suffix of \( s \). Suffix array \( H \) of \( s \) is a permutation of the integers \( 1, \ldots, n \) such that \( H[i] \) is the start position of the \( i \)-th smallest suffix. The relationship between \( H \) and the BWT-array \( L \) can be determined by the following formulas:

\[
\begin{align*}
L[i] &= $, \quad \text{if } H[i] = 0; \\
L[i] &= s[H[i] - 1], \quad \text{otherwise}
\end{align*}
\]

(3)

Since a suffix array can be generated in \( O(n) \) time [40], \( L \) can then be created in a linear time. However, most algorithms for constructing suffix arrays require at least \( O(n\log n) \) bits of working space, which is prohibitively high and amounts to 12 GB for the human genome. Recently, Hon et al. [54] proposed a space-economical algorithm that uses \( n \) bits of working space and requires only \(< 1 \) GB memory at peak time for constructing \( L \) of the human genome. We use this for our purpose.

5. STRING MATCHING WITH \( K \) MISMATCHES

5.1 Basic working process

By the string matching with \( k \) mismatches, we allow up to \( k \) characters in a pattern \( r \) to match different characters in a target \( s \). By using the BWT as an index, for finding all such string matches, a tree structure will be generated, in which each path corresponds to a search sequence discussed in the previous section. It is due to the possibility that a position in \( r \) may be matched to different characters in \( s \) and we need to call \text{search()} \ multiple times to do this task, leading to a tree representation.

Definition 1 (search tree) Let \( r \) be a pattern string and \( s \) be a target string. A search tree \( T \) (\( S \)-tree for short) is a tree structure to represent the search of \( r \) against \( \text{BWT}(s) \) (which is equivalent to the search of \( r \) against \( BWT(s) \)). In \( T \), each node is a pair of the form \( <x, [\alpha, \beta]> \), and there is an edge from \( v (= <x, [\alpha, \beta]> \) to \( u (= <x', [\alpha', \beta']>) \) if \( \text{search}(x, L_v) = u \).

As an example, consider the case where \( r = acac \), \( s = acagacc \) and \( k = 2 \). To find all occurrences of \( r \) in \( s \) with up to two mismatches, a search tree \( T \) shown in Fig. 3 will be created.

![Figure 3. Search for string matching with 2 mismatches](image-url)

In Fig. 3, \( v_0 \) is a virtual root, representing the whole \( L \), and ‘virtually’ corresponds to the virtual starting character \( r[0] = 'a' \). By exploring paths \( P_1 = v_1 \rightarrow v_4 \rightarrow v_{10} \rightarrow v_{16} \rightarrow v_{21} \) and \( P_5 = v_1 \rightarrow v_5 \rightarrow v_{12} \rightarrow v_{18} \rightarrow v_{22} \), we will find two occurrences of \( r \) with 2 mismatches: \( s[1..5] (=
\[ a_{1c1}a_{2g}a_{3c} \text{ and } s[3..7] (= a_{2g}a_{3c}a_{3c}) \text{ while by any of other paths } P_2, P_4, P_5, P_6 \text{ and } P_7 \text{ no string matching with at most 2 mismatches can be found.}

A node \( \langle a, [a] \rangle \) in such a tree is called a matching node if it corresponds to a same character in \( r \). Otherwise, it is called a mismatching node. For example, node \( v_4 = \langle c, [1, 2] \rangle \) is a matching node since it corresponds to \( r[2] = c \) while \( v_1 = \langle g, [1, 1] \rangle \) is a mismatching node since it corresponds to \( r[4] = c \).

For a path \( P_i \), we can store all its mismatching positions in an array \( B_i \) of length \( k + 1 \) such that \( B_i[j] = j \) if \( P_i[j] \neq r[j] \) and this is the \( j \)th mismatch between \( P_i \) and \( r \), where \( P_i[j] \) is the \( j \)th character appearing on \( P_i \). If the number of mismatches, \( k' \), say, between \( P_i \) and \( r \) is less than \( k + 1 \), then the default value \( \infty \) onwards, i.e.,

\[ B_i[k' + 1] = B_i[k' + 2] = \ldots = B_i[k + 1] = \infty. \]

We call \( B_i \) a mismatch array. For instance, in Fig. 3, for \( P_4 \), we have \( B_1 = [4, 5, \infty] \), indicating that at position 1, we have the first mismatch \( P_1[4] = g \neq r[4] = c \) and at position 5 we have the second mismatch \( P_1[5] = a \neq r[5] = c \). For the same reason, we have \( B_2 = [3, 4, \infty], B_3 = [2, \infty, \infty], B_4 = [1, 2, 3], B_5 = [1, 3, \infty], B_6 = [1, 2, \infty], \) and \( B_7 = [1, 2, 3] \).

These data structures can be easily created by maintaining and manipulating a temporary array \( B \) of length \( k + 1 \) to record the mismatches between the current path \( P \) and \( r \). Initially, each entry of \( B \) is set to be \( \infty \) and an index variable \( i \) pointing to the first entry of \( B \). Each time a mismatch is met, its position is stored in \( B[i] \) and then \( i \) is increased by 1. Each time \( r \) is exhausted or \( B \) becomes full (i.e., each entry is set a value not equal to \( \infty \)), we will store \( B \) as an \( B_i \) and associate it with the leaf node of the corresponding \( P_i \). Then, ‘backtrack’ to the lowest ancestor of the current node, which has at least a branch not yet explored, to search a new path. For instance, when we check \( v_{21} \), \( r \) is exhausted and the current value of \( B \) is \([4, 5, \infty] \). We will store \( B \) in \( B_1 \) (the array associated with the leaf node \( v_{21} \) of \( P_1 \)) and ‘backtrack’ to \( v_4 \) to explore a new path \( P_2 \). At the same time, all those values in \( B \), which are set after \( v_4 \), will be reset to \( \infty \), i.e., \( B \) will be changed to \([2, \infty, \infty] \).

Now we consider another path \( P_3 \). The search along it will stop at \( v_{13} \) since when reaching it \( B \) becomes full \((B = [1, 2, 3]) \). Therefore, the search will not be continued, and \( v_{16}, v_{23} \) will not be created.

It is essentially a brute-force search to check all the possible occurrences of \( r \) in \( s \). Denote by \( n' \) the number of leaf nodes in \( T \). The time used by this process is bounded by \( O(mn') \).

In fact, it is the main process discussed in [35]. The only difference is that in [35] a simple heuristics is used, which precomputes, for each position \( i \) in \( r \), a number \( \mu(i) \) of consecutive, disjoint substrings in \( r[i..m] \), which do not appear in \( s \). For example, in Fig. 3, \( \mu(1) = 1 \) since in \( r[1..5] = acacc \) \( r[1..4] = acacc \) does not occur in \( s = acagacc \). But \( \mu(3) = 0 \) since any substring in \( r[1..3] = acc \) does appear in \( s \). Assume that the number of mismatches between \( r[1..i-1] \) and \( P[1..i-1] \) (the current path) is \( l \). Then, if \( k - l < \mu(i) \), we can immediately stop exploring the subtree rooted at \( P[i-1] \) as no satisfactory answers can be found by exploring it.

The time required to establish such a heuristics is \( O(m) \) by using \( BWT(s) \) [35]. However, the theoretic time complexity of this method is still \( O(mn') \). Even in practice, this heuristics is not quite helpful since \( \mu(i) \) delivers only the information related to \( r[i..m] \) and the whole \( s \), rather than the information related to \( r[i..m] \) and the relevant substrings of \( s \), to which it will be compared. To see this, pay attention to part of the tree marked grey in Fig. 3. Since \( \mu(3) = 0 \), the search along \( P_1 \) will be continued. But no answer can be found. The heuristics here is in fact useless since it is not about \( r[3..5] \) and \( s[5..7] \), which is to be checked in the subsequent computation.

5.2 Mismatching information

Searching \( S \)-trees is an improvement over scanning strings, but it often happens that there are repetitive traversals of similar subtrees due to the multiple appearances of a same pair. (For example, node \( v_{16} \) is exactly the same as node \( v_5 \).) However, such repeated appearance of
pairs cannot be simply removed since they may be aligned to different positions in \( r \). For example, the first appearance of \(<g, [1, 1]>\) (\( v_{16} \) in Fig. 3) is compared to \( r[4] \) while its second appearance of it (\( v_b \)) is to \( r[2] \). Hence, we cannot use the result computed for \( v_{16} \) (when \(<g, [1, 1]>\) is first met) as the result for \( v_b \).

However, if we have stored the mismatching information \( R \) between substrings of \( r \), like \( r[4 .. 5] \) and \( r[2 .. 3] \), in some way, the mismatches along \( P_3 \) can be (partially) derived from \( R \) and \( B_i \) (the mismatches recorded for \( P_i \)), instead of simply exploring \( P_3 \) again in a way done for \( P_1 \). To do so, for each pair \( i, j \in \{1, ..., m\} \), we need to maintain a data structure \( R_{ij} \) containing the positions of the first \( k + 1 \) mismatches between \( r[i .. m - q + i] \) and \( r[j .. m - q + j] \), where \( q = \max(i, j) \), such that if \( R_{ij}[l] = f (\neq \infty) \) then \( r[i + f - 1] \neq r[j + f - 1] \) or one of them does not exist, and it is the \( l \)th mismatch between them.

Clearly, this task requires \( O(km^2) \) time and space.

For this reason, we will precompute only part of \( R \), instead of \( R_{ij} \) for all \( i, j \in \{1, ..., m\} \). Specifically, \( R_{i2}, ..., R_{im} \) for \( r \) will be pre-constructed in a way as described in [30], giving the positions of the mismatches between the pattern and itself at various relative shifts. That is, each \( R_{ii} \) (\( 2 \leq i \leq m \)) contains the positions within \( r \) of the first \( 2k + 1 \) mismatches between the substring \( r[1 .. m - i] \) and \( r[i + 1 .. m] \), i.e., the overlapping portions of the two copies of pattern \( r \) for a relative shift of \( i \). Thus, if \( R_{ij}[l] = f \), then \( r[l] \neq r[i + f - 1] \) or one of them does not exist, which is the \( l \)th mismatch between \( r[1 .. m - i] \) and \( r[i + 1 .. m] \). (See Fig. 4(a) for illustration.)

In Fig. 4(b), we show the running pattern \( r = acacc \) and all the possible right-to-left shifts: 
\[ r_1 = r[1 .. 5] = acacc, r_2 = r[2 .. 5] = cacc, r_3 = r[3 .. 5] = acc, \] and so on. In Fig. 4(c), we give \( R_{12} \) for \( r_1 \). In an \( R_{ii} \) (\( i = 2, ..., 5 \)), if the number of mismatches, \( k' \), say, between \( r[1 .. m - i] \) and \( r[i + 1 .. m] \) is less than \( 2k + 1 \), then the default value \( \infty \) onwards, i.e.,

\[ R_{ii}[k' + 1] = R_{ii}[k' + 2] = ... = R_{ii}[2k + 1] = \infty. \]

We will also use \( \delta(R_{ii}) \) to represent the number of all those entries in \( R_{ii} \), which are not \( \infty \). Trivially, \( R_{11} \) is set to be \( \{\infty, ..., \infty\} \) and therefore \( \delta(R_{11}) = 0 \).

Using the algorithm of [30], \( R_{12}, ..., R_{1m} \) can be constructed in \( O(\m log m) \) time, just before the process for the string matching gets started. In addition, we need to keep \( 2k + 1 \), rather than \( k + 1 \) mismatches in each \( R_{ij} \) (\( i = 2, ..., m \)), since for generating an \( R_{ij} \), up to \( 2k + 1 \) mismatches in some \( R_{ij} \) with \( i < j \) are needed to get an efficient algorithm (see [30] for detailed discussion.)

Each time we meet a node \( u \) (compared to a certain \( r[j] \)), which is the same as an already encountered one \( v \) (compared to an \( r[i] \)), we need to derive dynamically the relevant mismatches, \( R_{ij} \), between \( r[i .. m - q + i] \) and \( r[j .. m - q + j] \) from \( R_{ii} \) and \( R_{ij} \), as well as \( r \), to compute mismatching information for some new paths (to avoid exploring them by using \text{search}(). (A node \( <x, [\alpha, \beta]> \) is said to be the same as another node \( <x', [\alpha', \beta'] > \) if \( x = x' \), \( \alpha = \alpha' \) and \( \beta = \beta' \).) For this purpose, we design a general algorithm to create \( R_{ij} \) efficiently.

- Let \( \omega, \omega_1 \) and \( \omega_2 \) be three strings. Let \( A_1 \) and \( A_2 \) be two arrays containing all the positions of mismatches between \( \omega \) and \( \omega_1 \), and \( \omega \) and \( \omega_2 \), respectively.
• Create a new array \( A \) such that if \( A[i] = j (\neq \infty) \), then \( \omega_1[j] \neq \omega_2[j] \), or one of them does not exist. It is the \( i \)-th mismatch between them.

The algorithm works in a way similar to the \textit{sort-merge-join}, but with a substantial difference in handling a case when an entry in \( A_1 \) is checked against an equal entry in \( A_2 \). In the algorithm, two index variables \( p \) and \( q \) are used to scan \( A_1 \) and \( A_2 \), respectively. The result is stored in \( A \).

1. If one of \( A_2 \) and \( A_1 \) is \( \phi \), set \( A \) to \( \phi \) and return. (\( \phi \) means that no information is provided.)
2. \( p := 1; q := 1; l := 1; \)
3. If \( A_2[q] < A_1[p] \), then \( \{A[l] := A_2[q]; q := q + 1; l := l + 1;\} \)
4. If \( A_1[p] < A_2[q] \), then \( \{A[l] := A_1[p]; p := p + 1; l := l + 1;\} \)
5. If \( A_1[p] = A_2[q] \), then \( \{\text{if } \omega_1[p] \neq \omega_2[q], \text{then } \{A[l] := q; l := l + 1;\} \}
6. If \( p > |A_1| \), \( q > |A_2| \), or both \( A_1[p] \) and \( A_2[q] \) are \( \infty \), stop (if \( A_1 \) (or \( A_2 \)) has some remaining elements, which are not \( \infty \), first append them to the rear of \( A \), and then stop.)
7. Otherwise, go to (3).

We denote this process as \textit{merge}(\( A_1, A_2, \omega_1, \omega_2 \)). As an example, let us consider the case where \( A_1 = R_{12} = [1, 2, 3, \infty, \infty] \), \( A_2 = R_{13} = [3, \infty, \infty, \infty, \infty] \), \( \omega_1 = r[2 .. 4] = \text{cacc} \) and \( \omega_2 = r[3 .. 5] = \text{acc} \), and demonstrate the execution of \textit{merge}(\( A_1, A_2, \omega_1, \omega_2 \)) in Fig. 5. The result is \( A = [1, 2, \infty, \infty, \infty] \), showing the mismatches between these two substrings.

![Fig. 5: Illustration for \textit{merge}( )](image)

In step 4: \( p = 4, q = 2, l = 3 \). We compare \( A_1[4] \) and \( A_2[2] \), and find that both \( A_1[4] \) and \( A_2[2] \) are \( \infty \). Stop.

Obviously, the running time of this process is bounded by \( O(k) \).

**Proposition 1** Let \( A \) be the result of \textit{merge}(\( A_1, A_2, \omega_1, \omega_2 \)) with \( A_1, A_2, \omega_1, \omega_2 \) defined as above. Let \( k' \) be the number of mismatches between \( \omega_1 \) and \( \omega_2 \). Then, \( A[i] \) must be the position of the \( i \)-th mismatch between \( \omega_1 \) and \( \omega_2 \), or \( \infty \), depending on whether \( i \) is \( \leq k' \).

**Proof.** Consider \( \omega_2[j] \). Position \( j \) may satisfy either, neither, or both of the following conditions:
i) $j$ corresponds to the $l$th mismatch between $o_1$ and $o_2$ for some $l$, i.e., $o_1[j] \neq o_2[j]$ and $A_2[l] = j$.

ii) $j$ corresponds to the $f$th mismatch between $o_1$ and $o_3$ for some $f$, i.e., $o_1[j] \neq o_3[j]$ and $A_3[f] = j$.

If (i) holds, but (ii) not, (2) in $merge(A_1, A_2, o_1, o_2)$ will be executed. Since in this case, we have $o_1[j] \neq o_2[j]$ and $o_1[j] = o_1[j]$, (2) is correct.

If (ii) holds, but (i) not, (3) will be executed. Since in this case, we have $o_3[j] = o_3[j]$ and $o_1[j] = o_2[j]$, (3) is also correct.

If both (i) and (ii) hold, no conclusion concerning $o_1[j]$ and $o_2[j]$ can be drawn and we need to compare them. In this case, (4) is executed. If neither (i) nor (ii) is satisfied, we must have $o_1[j] = o_2[j]$ or $o_1[j] = o_3[j]$. So $o_2[j] = o_1[j]$, i.e., we have a matching at $j$. □

5.3 Main idea: mismatching information derivation

Now we are ready to present the main idea of our algorithm, which is similar to the generation of an $S$-tree described in Subsection 5.1. However, each time we meet a node $u$ (compared to a position in $r$, say, $r[j]$), which is the same as a previous one $v$ (compared to a different position in $r$, say, $r[i]$), we will not explore $T[u]$ (the subtree rooted at $u$), but do the following operations to derive the relevant mismatching information:

First, we will create $R_i$ by executing $merge(R_{ii}, R_{ij}, r[i..m-q+i], r[j..m-q+j])$, where $q = \max(i, j)$. Then, we will created a set of mismatch arrays for all the sub-paths in $T[u]$, which start at $u$ and end at a leaf node, by doing two steps explained below.

- For each path $P_i$ going through $v$, figure out a sub-array of $B_i$, denoted as $B_i^j$, containing only those values in $B_i$, which are larger than or equal to $i$. Moreover, each value in it will be decreased by $i - 1$. (For example, for $B_4 = [1, 2, 3, 4]$, we have $B_4^1 = [1, 2, 3, 4]$, $B_4^2 = [1, 2]$, $B_4^3 = [2]$, $B_4^4 = B_4^5 = \phi$.)

- Create the mismatch arrays for all the paths going through $u$ by executing $merge(B_i^j, R_{ij}, P_i[i..m_i], r[j..m])$ for each $P_i$, where $m_i = |P_i|$.

We denote this process as $mi$-creation($u, v, j, i$).

As an example, consider $v_3$ (in Fig. 3, labeled $<g, [1, 1]>$) and compared to $r[1] = a$, which is the same as $v_5$ (compared to $r[2] = c$). By executing $mi$-creation($v_3, v_5, 1, 2$), the following operations will be performed, to avoid repeated access of the corresponding subtree (i.e., part of $P_7$ shown in Fig. 6(a)):

1. Create $R_{12}$:
   
   $R_{12} = [1, 2, 3, 4, \infty], R_{11} = [\infty, \infty, \infty, \infty, \infty]$.

2. Create part of mismatch information for $P_7$:
   
   $B_3 = [2, \infty, \infty], B_3^2 = [1, \infty, \infty]. P_3[2..5] = gacc, r[1..4]) = acac.$

In general, we will distinguish between two cases:

(i) $i < j$. This case can be illustrated in Fig. 6(b). In this case, the mismatching information for the new paths can be completely derived.

(ii) $i > j$. This case can be illustrated in Fig. 6(c), in which only part of mismatching information for the new paths can be derived. Thus, after the execution of $merge(\ )$, we have to continue to extend the corresponding paths.

Therefore, among different appearances of a certain node $v$, we should always use the one compared to $r[i]$ with $i$ being the least to derive as much mismatching information as possible for the to be created paths.
Finally, we notice that it is not necessary for us to consider the case $i = j$ since the same node will never appear at the same level more than once. The following lemma is easy to prove.

**Lemma 1** In an $S$-tree $T$, if two nodes are with the same pair, then they must appear at two different levels. □

![Diagram](image_url)

**5.4 Algorithm Description**

The main idea presented in the previous subsection can be dramatically improved. Instead of keeping a $B_i$ for each $P_i$, we can maintain a general tree structure, called a *mismatch tree*, to store the mismatching information for all the created paths. First, we define two simple concepts related to $S$-trees.

**Definition 2** *(match path)* A sub-path in an $S$-tree $T$ is called a match path if each node on it is a matching node in $T$.

**Definition 3** *(maximal match sub-path)* A maximal match sub-path (MM-path for short) in an $S$-tree $T$ is a match sub-path such that the parent of its first node in $T$ is a mismatching node and its last node is a leaf node or has only mismatching nodes as its children.

For example, edge $v_1 \rightarrow v_4 \rightarrow v_{10}$ in $T$ shown in Fig. 3 is a MM-path. Path $v_{12} \rightarrow v_{18} \rightarrow v_{22}$ is another one. The node $v_7$ alone is also a MM-path in $T$.

Based on the above concepts, we define another important concept, the so-called mismatch trees.

**Definition 4** *(mismatch trees)* A mismatch tree $D$ (M-tree for short) for a given $S$-tree $T$, is a tree, in which for each mismatching node $<x, [a, b]>$(compared to $r[i]$ for some $i$) in $T$ we have a node of the form $<x, i>$, and for each MM-path a node of the form $<., 0>$. There is an edge from $u$ to $u'$ if one of the following three conditions is satisfied:

- $u$ is of the form $<x, i>$ corresponding to a pair $<x, [a, b]>$(compared to $r[i]$), which is the parent of the first node of an MM-path represented by $u'$ in $T$; or

- Both $u$ and $u'$ correspond to mismatching nodes, compared to two consecutive positions in $r$, respectively; or

- $u$ is of the form $<., 0>$ and $u'$ corresponds to a mismatching node which is a child of a node on the MM-path represented by $u$.

Without causing confusion, we will also call $<., 0>$ in $D$ a **matching node**, and $<x, i>$ a **mismatching node**.

For example, for $T$ shown in Fig. 3, we have its $M$-tree shown in Fig. 7, in which $u_0$ is a virtual root corresponding to the virtual root of the $S$-tree shown in Fig. 3. Its value is set to be $<., 0>$ since it will be handled as a matching node. Then, each path in the $M$-tree corresponds to a $B_i$. For instance, path $u_0 \rightarrow u_1 \rightarrow u_4 \rightarrow u_{11}$ corresponds to $B_1 = [4, 5, \infty]$ if all the matching nodes on the path are ignored. For the same reason, $u_0 \rightarrow u_1 \rightarrow u_5 \rightarrow u_{12}$ corresponds to $B_2 = [3, 4, \infty]$.
In addition, we can store all the different nodes \( v = <x, [\alpha, \beta]> \) in \( T \) in a hash table with each entry associated with a pointer to a node in the corresponding M-tree \( D \), described as follows.

- If \( v \) is a mismatching node (compared to \( r[i] \) for some \( i \in \{1, \ldots, m\} \)), a node \( u = <x, i> \) will be created in \( D \) and a pointer (associated with \( v \)) to \( u \) will be generated.
- If \( v \) is a matching node (compared to \( r[j] \)), a node \( u = <[\alpha, \beta], 0> \) will be created in \( D \) and a pointer to \( u \) will be generated. If the parent \( u' \) of \( u \) itself is a matching node of the form \( <-, 0> \), \( u \) will be merged into its parent. That is, \( v \) will be linked to \( u' \) while \( u' \) itself will not be generated.

For instance, when \( <a, [1, 3]> \) (\( v_1 \) in \( T \) shown in Fig. 3) is created, it is compared to \( r[1] = a \) and we have a matching. For this, a node \( u_1 = <-, 0> \) in the M-tree \( D \) will be created. At the same time, we will insert \( <a, [1, 3]> \) into the hash table and produce a pointer associated with it to \( u_1 \) (see Fig. 7 for illustration). When \( <c, [1, 2]> \) (\( v_4 \) in \( T \) shown in Fig. 3) is created, it is compared to \( r[2] = c \) and we have a second matching node. But no node is generated for it in \( D \). We only need to create a link from \( <c, [1, 2]> \) (in the hash table) to \( u_1 \). However, when \( <a, [1, 3]> \) (\( v_{21} \) in \( T \) shown in Fig. 3) is created, it is compared to \( r[5] = c \) and we will create a mismatching node \( u_{11} = <a, 5> \) in \( D \) since \( a \neq c \). Again, we will insert \( <a, [1, 3]> \) into the hash table and produce a pointer associated with it to \( u_{11} \) (see Fig. 7).

In order to generate \( D \), we will use a stack \( S \) to control the process, in which each entry is a quadruple \( (v, j, k, u) \), where

\[ v \] – a node inserted into the hash table.

\[ j \] – an integer to indicate that \( v \) is compared to \( r[j] \).

\[ k \] – the number of mismatches between the path and \( r[0 \ldots j] \) (recall that \( r[0] = 't' \)).

\[ u \] – the parent of a node in \( D \) to be created for \( v \).

In this way, the parent/child link between \( u \) and the node to be created for \( v \) can be easily established, as described below.

- Each time an entry \( e = (v, j, k, u) \) with \( v = <x, [\alpha, \beta]> \) is popped out from \( S \), we will check whether \( x = r[j] \).
  - i) If \( x \neq r[j] \), we will generate a node \( u' = <x, j> \) (if \( x \neq $ \)) and link it to \( u \) as a child.
  - ii) If \( x = r[j] \), we will check whether \( u \) is a node of the form \( <-, 0> \). If it is not the case or \( u \) is the root of \( D \), generate a node \( u' = <-, 0> \). Otherwise, set \( u' \) to be \( u \).
  - iii) Using search() to find all the children of \( u \): \( v_1, \ldots, v_l \). Let \( v_i = <y_i, [\alpha, \beta]> \) \((i = 1, \ldots, l)\). Push each \((v_i, j + 1, k', u')\) into \( S \) with \( k' \) being \( k \) or \( k + 1 \), depending on whether \( y_i = r[j + 1] \). If \( y_i = r[j + 1] \), \( k' = k + 1 \). Otherwise, \( k' = k \).

Obviously, in the case that \( x = $, j = |r| \), or \( k > k + 1 \), (iii) will not be conducted.

Finally, note that in this process it is not necessary to keep \( T \), but insert all the nodes (of \( T \)) in the hash table as discussed above.
Example 2 In this example, we run the above process on \( r = accc \) and \( L = BWT(s) \) \((s = ccagaca)\) shown in Fig. 1(c) with \( k = 2 \), and show its first 8 steps. The tree created is shown in Fig. 7.

Step 1: Create the root, \( v_0 = <-, [1, 8]> \). Push \( (v_0, 0, 0, \phi) \) into \( S \), where \( \phi \) is used to represent the parent of the root \( D \). See Fig. 8(a).

Step 2: Pop out the top element \((v_0, 0, 0, \phi)\) from \( S \). Create the root \( u_0 \) of \( D \), which is set to be a child of \( \phi \). Push \( <v_3, 1, 1, u_0> \), \( <v_2, 1, 1, u_0> \), \( <v_1, 1, 0, u_0> \) into \( S \), where \( v_3, v_2, \) and \( v_1 \) are three children of \( v_0 \). See Fig. 8(b).

Step 3: Pop out \((v_1, 1, 0, u_0)\) from \( S \). \( v_1 = <a, [1, 3]> \). Since \( r[1] = a \), a matching node \( u_1 = <-, 0> \) will be created and set to be a child of \( u_0 \). Then, push two entries \( <v_5, 2, 1, u_1> \) and \( <v_4, 2, 0, u_1> \) into \( S \), where \( v_4 \) is the child of \( v_1 \). See Fig. 8(e).

Step 4: Pop out \((v_4, 2, 0, u_1)\) from \( S \). \( v_4 = <-, [1, 2]> \). Since \( r[2] = c \), we would create a matching node \( <-, 0> \). However, its parent \( u_1 \) itself is a matching node. So, this new matching node will be merged into \( u_1 \). Push \( <v_{13}, 3, 1, u_1> \) and \( <v_{10}, 3, 0, u_1> \) into \( S \), where \( v_{10} \) and \( v_{11} \) are children of \( v_4 \). See Fig. 8(d).

Step 5: Pop out \((v_{10}, 3, 0, u_1)\) from \( S \). \( v_{10} = <a, [2, 2]> \). \( r[3] = a \). However, no new node is created since \( u_1 \) is a matching node. Push \((v_{16}, 4, 1, u_1)\) into \( S \), where \( v_{16} \) is the only child of \( v_{10} \). See Fig. 8(e).

Step 6: Pop out \((v_{16}, 4, 1, u_1)\) from \( S \). \( v_{16} = <g, [1, 1]> \). \( r[4] = c \neq g \). A mismatching node \( u_4 = <g, 4> \) will be generated. Push \((v_{21}, 5, 2, u_4)\) into \( S \). See Fig. 8(f).

Step 7: Pop out \((v_{21}, 5, 2, u_4)\) from \( S \). \( v_{21} = <a, [3, 3]> \). \( r[5] = c \neq a \). A mismatching node \( u_{11} = <-, 5> \) will be generated. No new entries will be pushed into \( S \) since now \( j = |r| \). See Fig. 8(g).

Step 8: Pop out \((v_{11}, 3, 1, u_1)\) from \( S \). \( v_{11} = <c, [3, 3]> \). \( r[3] = a \neq c \). A mismatching node \( u_5 = <c, 3> \) will be generated. Push \((v_{17}, 4, 1, u_5)\) into \( S \). See Fig. 8(h).

From the above sample trace, we can see that \( D \) can be easily generated. In the following, we will discuss how to extend this process to a general algorithm for our task.

As with the basic process, each time a node \( v = <x, [a, \beta]> \) (compared to \( r[j] \)) is encountered, which is the same as a previous one \( v' = <x', [a', \beta'] \) (compared to \( r[j] \)), we will not create a subtree in \( T \) in a way as for \( v' \), but create a new node \( u \) for \( v \) in \( D \) and then go
along the link associated with $v'$ to find the corresponding nodes $u'$ in $D$ and search $D[u']$ in the breadth-first manner to generate a subtree rooted at $u$ in $D$ by simulating the merge operation discussed in Subsection 5.2 and 5.3. In other words, $D[u]$ (to be created) corresponds to the mismatch arrays for all the paths going through $v$ in $T$, which will not be actually produced. See Fig. 9 for illustration.

![Fig. 9: Illustration for generation of subtrees in $T'$](image)

For this purpose, we introduce a third kind of nodes of the form $<\cdot, \infty>$ into $D$ to represent symbol $\infty$ in mismatching arrays. For example, for $k = 2$, the $M$-tree shown in Fig. 7 will be changed to the tree shown in Fig. 10. (In the modified tree, symbol $\$ is not included since it does not appear in any pattern string.)

![Fig. 10: A modified mismatch tree](image)

To search $D[u']$ breadth-first, a queue data structure $Q$ is used to control the search of $D[u']$ and at the same time generate $D[u]$. In $Q$, each entry $e$ is a triplet $(w, \gamma, h)$ with $w$ being a node in $D[u']$, $\gamma$ an entry in $R_\gamma$, and $h$ is the number of mismatching nodes on the path from the root to the node to be created in $D[u]$. Initially, put $(u', R_\gamma[1], h')$ into $Q$, where $h'$ is the number of mismatching nodes on the path from the root to $u$. In the process, when $e$ is dequeued from $Q$ (taken out from the front), we will make the following operations (simulating the steps in $merge()$):

1. Let $e = (w, R_\gamma[l], h)$. Assume that $w = <z, f>$ and $R_\gamma[l] = val$.
   - If $<z, f>$ is equal to $<\cdot, 0>$, then create a copy of $<\cdot, 0>$ added to $D[u]$. Let $u_1, \ldots, u_g$ be the children of $w$. We will enqueue (append at the end) $(u_1, R_\gamma[l], h), \ldots, (u_g, R_\gamma[l], h)$ into $Q$ in turn.
   - If $z$ is a mismatching node, do (2), (3), or (4).
   - If $<z, f>$ is equal to $<\cdot, \infty>$, do (5).

2. If $f < i + val - 1$, add $<z, j + f - i + 1>$ to $D[u]$. If $h < k + 1$, enqueue $(u_i, R_\gamma[l], h + 1), \ldots, (u_g, R_\gamma[l], h + 1)$ into $Q$.

3. If $f > i + val - 1$ (and $f \neq \infty$), we will scan $R_\gamma$ starting from $R_\gamma[l]$ until we meet the largest $l' \leq k - h + l$ such that $f > i + R_\gamma[l'] - 1$. For each $R_\gamma[q]$ ($l \leq q \leq l'$), we create a new node $<r[i$
+ $R_0[q] \cdot 1$, $j + R_0[q] \cdot 1 >$ added to $D[u]$. If $l' < k - h + l$, add $<, \infty>$ to $D[u]$, and enqueue $<w, R_0[l + 1], h + l' - l + 1 >$ into $Q$.

4. If $f = i + val - 1$, we will distinguish between two subcases: $z \neq r[j + val - 1]$ and $z = r[j + val - 1]$. If $z \neq r[j + val - 1]$, we have a mismatch and a copy of $w$ will be generated and added to $D[u]$. If $h < k + 1$, enqueue $(u_1, R_0[l + 1], h + 1, \ldots, (u_{eg}, R_0[l + 1], h + 1)$ into $Q$. If $z = r[j + val - 1]$, create a node $<, 0>$ added to $D[u]$. (If its parent is also $<, 0>$, it will be merged into its parent.) Also enqueue $<u_1, R_0[l + 1], h, \ldots, <u_{eg}, R_0[l + 1], h >$ into $Q$.

5. If $w = <, \infty>$, scan $R_0$ starting from $R_0[l]$ until we find the largest $l' \leq k - h + l$ such that $R_0[l'] \neq \infty$. For each $R_0[q] (l \leq q \leq l')$, we create a new node $<r[i + R_0[q] \cdot 1], j + R_0[q] - 1 >$ added to $D[u]$. If $l' < k - h + l$, add $<, \infty >$ to $D[u]$, and enqueue $<w, R_0[l' + 1], h + l' - l + 1 >$ into $Q$.

   In the above process, (2) corresponds to step 3 in $merge()$, (3) to step 4 in $merge()$, and (4) to step 5 in $merge()$.

   In (2), we handle the case when $f < i + val - 1$. In this case, we must have $r[f] = r[j + f \cdot i]$. Then, by the following simple inference:

   \[ P[f] \neq r[f], r[f] = r[j + f \cdot i] \Rightarrow P[f] \neq r[j + f \cdot i] \]  

   (see Fig. 11(a) for illustration),

   we know that a mismatching node should be added to $D[u]$. Here, $P$ stands for a path starting from $v'$ in $T$ corresponding to a path starting from $u'$ in $D$, and $P[f]$ for the $f$th node on $P$. See Fig. 11(a) for illustration.

   In (3), we handle the case that $f > i + val - 1$. In this case, we have, for each $i' \in \{i + val - 1, \ldots, f\}$ with $R_0[q] = i' (l \leq q \leq l')$,

   \[ P[i'] = r[i'], r[i'] \neq r[j + i' \cdot i] \Rightarrow P[i'] \neq r[j + i' \cdot i] \]  

   (see Fig. 11(b) for illustration).

   Thus, for each $R_0[q] (l \leq q \leq l')$, a mismatching node will be created and added to $D[u]$.

In the above description, we ignored the technical details on how $D[u]$ is constructed for simplicity. However, in the presence of $D[u']$, it is easy to do such a task by manipulating links between nodes and their respective parents.

Denote the above process by node-creation($w, \gamma, i, j, R_0$). We have the following proposition.

**Proposition 2** node-creation($w, \gamma, i, j, R_0$) create nodes in $D[u]$ correctly.

**Proof.** The correctness of node-creation($w, \gamma, i, j, R_0$) can be derived from Proposition 1. □

**Example 1** As an example, consider Fig. 3 and Fig. 10 once again. Assume that $D[u_0]$ (shown in Fig. 10) has been created. Here, $u_0 = <g, 2>$ is generated when we meet $<g, [1, 1]>$ (compared to $r[2] = c$). When we meet $<g, [1, 1]>$ (compared to $r[1] = a$) again, we generate the corresponding subtree (see $u_0$ in Fig. 10) by the derivation of mismatching information.

Step 1: $i = 2, j = 1$. Generate $R_{21} = [1, 2, 3, 4]$.

Step 2: Enqueue $<g, 2>, R_{21}[1] = 1, 0 >$ into $Q$.

Step 3: $(g, 2), 1, 0 >$ : dequeue($Q$). $h = 0$. 

Fig. 11: Illustration for generation of nodes in $D$
\( f = 2 = R_{21}[1] + i - 1 = 1 + 2 - 1 = 2 \). (3) will be executed, by which we will compare \( g \) and \( r[j + R_{21}[1] - 1] = r[1] = a \). Since \( g \neq a \), a mismatching node \( <g, 1> \) (\( u_3 \) in Fig. 1) will be generated. Enqueue \((u_{13}, R_{21}[2], h + 1) = (<-, \infty>, 2, 1)\).

Step 4: \((<-, \infty>, 2, 1) \Rightarrow \text{deque}(Q), h = 1\).

(5) will be carried out, by which we will find \( R_{21}[2] \) and \( R_{21}[3] \) (i.e., \( l = 2 \), \( l' = 3 \)), and for each of them a mismatching node will be constructed. That is, two nodes: \(<r[i + R_0[2] - 1], j + R_0[2] - 1 > = <r[3], 2> = <a, 2> \) and \(<r[i + R_0[3] - 1], j + R_0[3] - 1 > = <r[4], 3> = <c, 3> \) will be added to \( D \). (See the subtree rooted at \( u_3 \) in Fig. 10.) Since \( l' = 3 \) is not smaller than \( k - h + l = 2 - 1 + 2 = 3 \). We will not add \(<-, \infty> \) as the last node of the newly generated path. □

Finally, if \( i > j \), \( D[u] \) needs to be extended in some cases, which can be done in a way similar to the extension of mismatch arrays as discussed in Subsection 5.3.

The following is the formal description of the working process.

**Algorithm A(L, r, k)**

```plaintext
begin
1. create root of T; push(S, (root, 0, 0, φ));
2. while S is not empty do {
3. (v, j, k, u) := pop(S); let \( v = <x, α, β> \);
4. if \( v \) is same as an existing \( v' \) (compared to \( r[i] \)) then {
5. \( q := \max(i, j); \)
6. \( R_q := \text{merge}(R_{11}, R_{1j}, r[i..m-q+i], r[j..m-q+j]); \)
7. \( \text{enqueue}(Q, (p(v'), R_0[1])); \)
8. while \( Q \) is not empty do {
9. \( (w, γ) := \text{deque}(Q); \text{node-creation}(w, γ, i, j, R_0);) \}
10. else {
11. if \( x ≠ r[j] \) then create \( u' = <x, j> \) and make it a child of \( u \);
12. else if \( u \) is \(<-, 0> \) then \( u' := u \);
13. else create \( u' = <-, 0> \) and make it a child of \( u \);
14. \( p(v) := u' \); (*associate with \( v \) a pointer to \( u'\)*)
15. if \( j < |r| \) and \( k ≤ k \) then {
16. for each \( y ∈ \Sigma \) within \( L_v \) do {
17. \( w := \text{search}(y, L_v); \)
18. if \( w ≠ φ \) then {
19. if \( y = r[j + 1] \) then push(S, (w, j + 1, k, u'));
20. if \( y ≠ r[j + 1] \) and \( k < k \) then {push(S, (w, j + 1, k + 1, u'))}
21. } \}
22. } \}
23. } \}
end
```

If we ignore lines 3 – 9 in the above algorithm, it is almost a depth-first search of a tree. Each time an entry \((v, j, k, u)\) is popped out from \( S \) (see line 4), it will be checked whether \( v \) is the same as a previous one \( v' \) (compared to \( r[i] \)). (See line 4.) If it is not the case, a node \( u' \) for \( v \) will be created in \( D \) (see lines 11 – 14). Then, all the children of \( v \) will be found by using the procedure \( \text{search}() \) (see line 17) and pushed into \( S \) (see lines 18, and 19.) Otherwise, we will first create \( R_0 \) by executing \( \text{merge}(R_{11}, R_{1j}, r[i..m-q+i], r[j..m-q+j]); \) where \( q = \max(i, j) \). (see lines 5 – 6.) Then, we create a subtree in \( D \) by executing a series of node-creation operations (see lines 8 – 9.)

Concerning the correctness of the algorithm, we have the following proposition.

**Proposition 3** Let \( L \) be a BWT-array for the reverse \( \bar{s} \) of a target string \( s \), and \( r \) a pattern. Algorithm \( A(L, r, k) \) will generate a mismatching tree \( D \), in which each root-to-leaf path represents an occurrence of \( r \) in \( s \) having up to \( k \) positions different between \( r \) and \( s \).

**Proof.** In the execution of \( A(L, r, k) \), two data structures will be generated: a hash table and a matching tree \( D \), in which some subtrees in \( D \) are derived by using the mismatching
information over \( r \). Replacing each matching node in \( D \) with the corresponding maximum matching path and each mismatching node \(<x, i>\) with the corresponding pair \(<x, [a, b]>\) (compared to \( r[i] \)), we will get an \( S \)-tree, in which each path corresponds to a search sequence discussed in Section III. Thus, in \( D \) each root-to-leaf path represents an occurrence of \( r \) in \( s \) having up to \( k \) positions different between \( r \) and \( s \). □

6. ANALYSIS OF TIME COMPLEXITY

The time complexity of the algorithm mainly consists of three parts: the cost for generating the mismatching information over \( r \) which is bounded by \( O(m \log m) \); the cost for generating the \( M \)-tree and maintaining the hash table, which is bounded by \( O(kn') \), where \( n' \) is the number of the \( M \)-tree's leaf nodes; and the cost for checking the characters in \( s \) against the characters in \( r \), which is bounded by \( O(n) \) if the cost of search( ) can be reduced to \( O(1) \).

So, to know the time complexity of our method, we need to make an estimation of \( n' \).

Let \( M = |\Sigma| \). It is reasonable to assume that the probability of a character \( \in \Sigma \) appearing in \( s \) is \( p = \frac{1}{M} \). Let \( q = 1 - p \). The probability that a path in \( D \) contains \( i \) mismatching nodes is

\[
\left(\frac{k+1}{i}\right)p^{m}q^{i} = \left(\frac{k+1}{i}\right)\left(\frac{1}{M}\right)^{m-i} \left(1 - \frac{1}{M}\right)^{i}.
\]

The number of such paths is obviously bounded by \( |\Sigma|^{i} \).

Therefore, the average number of leaf nodes in \( D \) generally is

\[
\sum_{i=0}^{k+1}(\frac{k+1}{i})\left(\frac{1}{M}\right)^{m-i} \left(1 - \frac{1}{M}\right)^{i} M^{i}
\]

(5)

If \( m \geq 2(k + 1) \), we have

\[
\sum_{i=0}^{k+1}(\frac{k+1}{i})\left(\frac{1}{M}\right)^{m-i} \left(1 - \frac{1}{M}\right)^{i} \leq \sum_{i=0}^{k+1}(\frac{1}{M})^{i} \left(\frac{1}{M}\right)^{m-1} \left(1 - \frac{1}{M}\right)^{i} = \left(1 + \frac{1}{M}\right)^{k+1}.
\]

If \( m < 2(k + 1) \), we have \( \delta = \frac{M - k - 1}{k + 1} < 1 \).

\[
\sum_{i=0}^{k+1}(\frac{k+1}{i})\left(\frac{1}{M}\right)^{m-i} \left(1 - \frac{1}{M}\right)^{i} = \sum_{i=0}^{k+1}(\frac{1}{M})^{i} \left(\frac{M - 1}{M-1}\right)^{i} M^{i-\delta}
\]

\[
\leq \sum_{i=0}^{k+1}(\frac{k+1}{i})\left(\frac{1}{M}\right)^{m-i} \left(M^{1-\delta} - \frac{1}{M^{\delta}}\right)^{i}.
\]

However, in practice, \( m \) is often \( \geq 2(k + 1) \). Especially, in DNA databases, the length \( m \) of a read (pattern string) is normally larger than 100 while \( k \) is set to be between 10 and 20.

Thus, in such scenarios, the number of leaf nodes of \( D \) is bounded by \( O\left(1 + \frac{1}{|\Sigma|}\right)^{k+1} \).
According to the above analysis, the following proposition is easy to prove.

**Proposition 4** If \( m \geq 2(k + 1) \), the average time of our algorithm for the string matching with \( k \) mismatches is bounded by \( O(k \left(1 + \frac{1}{|\Sigma|}\right)^{k+1} + n \log m) \) if the execution of \( \text{search}( ) \) requires only a constant time, where \( n = |s| \), and \( m = |r| \).

**Proof.** Since for all the same nodes inserted into the hash table the array \( L \) is accessed only once, the total cost of searching \( L \) (for constructing the whole \( D \)) is bounded by \( O(n) \) if the cost of \( \text{search}( ) \) is \( O(1) \). Taking the cost for creating \( D \), as well as the cost for generating mismatching information over \( r \) into account, the whole cost is bounded by \( O(k \left(1 + \frac{1}{|\Sigma|}\right)^{k+1} + n \log m) \). \( \square \)

In the next section, we discuss a method for implementing \( \text{search}( ) \), by which the cost of \( \text{search}( ) \) is reduced to \( O(1) \).

**7. FURTHER IMPROVEMENTS**

The operation \( \text{search}( ) \) discussed in 4.1 can be greatly improved in two ways. One is to use the so-called \( \text{rankAll} \) mechanism, by which \( |\Sigma| \) arrays are arranged each for a character \( x \in \Sigma \) [31]. With help of such data structures, the cost of \( \text{search}( ) \) can be reduced to \( O(1) \). In the second way, we will rearrange the search of a segment of \( L \) when we visit a node \( v \) in \( T \) to do the so-called multi-character checking to further decrease the searching cost of \( L \).

In the following, we will discuss these two methods in great detail.

**7.1 RankAll**

By the rankAll, we will arrange \( |\Sigma| \) arrays each for a character \( x \in \Sigma \) such that \( x[i] \) (the \( i \)th entry in the array for \( x \)) is the number of appearances of \( x \) within \( L[1 .. i] \). For instance, in Fig. 12(a) \( a[1] \) is 1 since \( L[1 .. 1] \) contains only one \( a \). But \( a[5] \) is 2 since \( L[1 .. 5] \) contains two \( a \)'s.

<table>
<thead>
<tr>
<th>( j )</th>
<th>( F )</th>
<th>( L )</th>
<th>( a )</th>
<th>( c )</th>
<th>( g )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( S )</td>
<td>( a_1 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( a_1 )</td>
<td>( c_1 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( a_2 )</td>
<td>( g_1 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>( a_1 )</td>
<td>( c_2 )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>( c_1 )</td>
<td>( a_2 )</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>( c_1 )</td>
<td>( c_1 )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>( e_1 )</td>
<td>( S )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>( g_1 )</td>
<td>( a_1 )</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

For each \( \sigma = 4 \) values in \( L \), a \( \text{rankAll} \) value is stored.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( A_3 )</th>
<th>( A_4 )</th>
<th>( A_5 )</th>
<th>( A_6 )</th>
<th>( A_7 )</th>
<th>( A_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>(a)</td>
</tr>
</tbody>
</table>

![Fig. 12: LF-mapping and rank-correspondence](image)

Now, instead of scanning a certain segment \( L[\alpha .. \beta] \) (\( \alpha \leq \beta \)) to find a subrange for a certain \( x \in \Sigma \), we can simply look up the array for \( x \) to see whether \( x[\alpha .. 1] = x[\beta] \). If it is the case, then \( x \) does not occur in \( L[\alpha .. \beta] \). Otherwise, \( [x[\alpha .. 1] + 1, x[\beta]] \) should be the found range. For example, to find the first and the last appearance of \( c \) in \( L[2 .. 5] \), we only need to find \( c[2 - 1] = c[1] = 0 \) and \( c[5] = 2 \). So the corresponding range is \( [c[2 - 1] + 1, c[5]] = [1, 2] \).

In this way, the searching of \( L \) can be saved and we need only a constant time to determine a subrange for a character encountered during a pattern searching.

The problem of this method is its high space requirement, which can be mitigated by replacing \( x[ ] \) with a compact array \( A_x \) for each \( x \in \Sigma \), in which, rather than for each \( L[i] \) (\( i \in \)
{(1, ..., n)}, only for some entries in L the number of their appearances will be stored. For example, we can divide L into a set of buckets of the same size and only for each bucket a value will be stored in $A$. Obviously, doing so, more search will be required. In practice, the size $\sigma$ of a bucket (referred to as a compact factor) can be set to different values. For example, we can set $\sigma = 4$, indicating that for each four contiguous elements in L a group of $|\Sigma|$ integers (each in an $A$) will be stored. That is, we will not store all the values in Fig. 12(a), but only store $\\alpha[4], a[4], c[4], g[4], t[4], \$[8], a[8], c[8], g[8], t[8]$ in the corresponding compact arrays, as shown in Fig. 12(b). However, each $x[j]$ for $x \in \Sigma$ can be easily derived from $A$ by using the following formulas:

$$x[j] = A_s[i] + \rho,$$

where $i = \lfloor j/\sigma \rfloor$ and $\rho$ is the number of $x$'s appearances within $L[i \cdot \sigma + 1 .. j]$, and

$$x[j] = A_s[i'] \cdot \rho',$$

where $i' = \lceil j/\sigma \rceil$ and $\rho'$ is the number of $x$'s appearances within $L[j + 1 .. i' \cdot \sigma]$.

Thus, we need two procedures: sDown(L, j, $\sigma$, x) and sUp(L, j, $\sigma$, x) to find $\rho$ and $\rho'$, respectively. In terms of whether $j \cdot \sigma \leq i' \cdot \sigma - j$, we will call sDown(L, j, $\sigma$, x) or sUp(L, j, $\sigma$, x) so that fewer entries in L will be scanned to find $x[j]$.

Finally, we notice that the column for $\$ can always be divided into two parts. In the first part, each entry is 0 while in the second part each entry is 1 (see Fig. 12(a)). So we can simply keep a number to indicate where it is divided, instead of storing the whole column.

7.2 Multiple character checking

From the above discussion, we can see that each time we meet a node $v = <x, [\alpha, \beta]> \in T$, we have to call sDown() or sUp() to generate each of its children. That is, for each of its children, L will be searched in some way to determine the corresponding intervals. Instead of searching the segment for each child separately, however, we can manage to search the segment only once for all the children of $v$. To this end, we will use integers to represent characters in $\Sigma$. For example, we can use 1, 2, 3, 4, 5 to represent $a, c, g, t, \$ in a DNA sequence. In addition, a counter $C_i$ is associated with each $i \in \Sigma$ to record the number of $i$'s appearances during the search of a segment in L. Initially, each $C_i (i = 1, 2, 3, 4, 5)$ is set to 0.

In the searching, each time we meet an $L[l]$, we will increase $C_{L[l]}$ by one.

See Figure 9 for illustration.

With these counters, we change sDown(L, j, $\sigma$, x) and sUp(L, j, $\sigma$, x) to sDown(L, j, $\sigma$) and sUp(L, j, $\sigma$), respectively, to search L for all the children of $v$, but only in one scanning of L.

Let $j' = \lfloor j/\sigma \rfloor \cdot \sigma + 1$. sDown(L, j, $\sigma$) will search a segment $L[j' .. j]$ from top to bottom, and store the result in an array $E$ of length $|\Sigma|$, in which each entry $E[i]$ is the rank of $i$ (representing a character), equal to $C_i + A_s[j/\sigma]$. Remember that $C_i$ records the number of $i$'s appearances within $L[j' .. j]$.
FUNCTION sDown($L$, $j$, $\sigma$)
begin
1. $C_i := 0$ for each $i \in \Sigma$;
2. $l := \lceil j/\sigma \rceil \cdot \sigma + 1$;
3. while $l \leq j$ do {
5. $l := l + 1$;
6. }
7. for $k = 1$ to $|\Sigma|$ do {
8. $E[k] := A_k[l] + C_k$;
9. }
10. return $E$;
end

In the algorithm, $L[j' \ldots j]$ is scanned only once in the main while-loop (see lines 3–6). For each encountered entry $L[l]$ ($j' \leq l \leq j$), $C_L[l]$ will be increased by 1 to count encountered entries which are equal to $L[l]$. After the while-loop, we compute the ranks for all the characters respectively labeling the children of $v$ (see lines 7–8).

Fig. 14 helps for illustration.

FUNCTION sUp($L$, $j$, $\sigma$)
begin
1. $C_i := 0$ for each $i \in \Sigma$;
2. $l := \lfloor j/\sigma \rfloor \cdot \sigma$;
3. while $l \geq j + 1$ do {
5. $l := l - 1$;
6. }
7. for $k = 1$ to $|\Sigma|$ do {
8. $E[k] := A_k[l] - C_k$;
9. }
10. return $E$;
end

By using the above two procedures, search($z$, $\pi$) will be changed to search($\pi$) as described below.
In our experiments, we have tested altogether four different methods:

Example 2 In this example, we trace the working process to generate all the children of node $v_2 = <c, [1, 3]>$ in $T$ shown in Fig. 3.

First, we create $E$ as below:

1. $j = Y_t + \alpha \cdot 1 = 5 + 1 - 1 = 5$.
   \begin{align*}
   \end{align*}

Then, $E'$ can be computed as follows:

2. $j = Y_t + \beta \cdot 1 = 5 + 3 - 1 = 7$. $i = \lfloor j/\alpha \rfloor = \lfloor 7/4 \rfloor = 1$. $i' = \lfloor 7/\sigma \rfloor = \lfloor 5/4 \rfloor = 2$.

3. Since $j - i \cdot \sigma = 7 - 1 \times 4 = 3 > i' \cdot \sigma \cdot j = 2 \times 4 = 7 - 1$, $sUp(L, j, \sigma)$ will be invoked, where $L$ is shown in Fig. 12(a), but with $a, c, g, t, \$\$ represented by 1, 2, 3, 4, 5, respectively; $j = 7,$ and $\alpha = 4$.

4. In the execution of $sUp(L, 7, 4)$, we first set $C_1 = 0, C_2 = 0, C_3 = 0, C_4 = 0, and C_5 = 0$. Then, we scan $L[j + 1 .. \lfloor 7/\sigma \rfloor \sigma] = L[8 .. 8]$ bottom-up to find the numbers of different characters and record them in $C_i$'s. In this process, only $C_1$ is changed to 1 while all the other counters remain 0. Then, we have $E'[1] = A_c[8] \cdot C_1 = 3 - 1 = 2, E'[2] = A_c[8] = 3, E'[3] = A_c[8] = 1, E'[4] = A_c[8] = 0, and E'[5] = A_c[8] = 1.$

Since $E'[3] + 1 > E'[3] and E'[4] + 1 > E'[4]$, we will not have nodes with labels $g$ and $t$. That is, only three nodes will be created: $<\alpha, [E[1] + 1, E'[1]]>, <\alpha, [2, 2]>$, and $<\$, [1, 1]$. In addition, a node with $\$\$ must be the last node of a path. So, we simply use $<\$, [\, \, i] to represent $<\$, [i, i] for any $i$.

8 EXPERIMENTS

In our experiments, we have tested altogether four different methods:
- **BWT-based** [10] (BWT for short),
- **Amir’s method** [2] (Amir for short),
- **Cole's method** [15] (Cole for short),
- **Algorithm A discussed in this paper (A( ) for short)**

By the BWT-based method, an S-tree will be created as described in Section IV, but with \( \mu(i) \) being used to cut off branches, where \( \mu(i) \) is the number of consecutive, disjoint substrings in \( r[i .. m] \) not appearing in \( s \). By the Amir’s algorithm, a pattern \( r \) is divided into several periodic stretches separated by 2\( k \) aperiodic substrings, called breaks, as illustrated in Fig. 14. Then, for each break \( b_i \), located at a certain position \( i \), find all those substrings \( s_j \) (located at different positions \( j \)) in \( s \) such that \( b_i = s_j \), and then mark each of them. After that, discard any position that is marked less than \( k \) times. In a next step, verify every surviving position in \( s \).

![Fig. 14: Illustration for periodic stretches and breaks](image)

By the Cole’s, a suffix tree for a target is constructed. (The code for constructing suffix trees is taken from the gsuffix package: http://gsuffix.Sourceforge.net/).

All the four methods are implemented in C++, compiled by GNU make utility with optimization of level 2. In addition, all of our experiments are performed on a 64-bit Ubuntu operating system, run on a single core of a 2.40GHz Intel Xeon E5-2630 processor with 32GB RAM.

For the tests, five reference genomes shown in Table 2 are used. They are all obtained from a biological project conducted in a laboratory at University of Manitoba [28]. In addition, all the simulating reads are taken from these five genomes, with varying lengths and amounts. It is done by using the wgsim program included in the SAMtools package [38] with a default model for single reads simulation. Concretely, we take 5000 reads with length varying from 100 bps to 300 bps.

**TABLE 2: CHARACTERISTICS OF GENOMES**

<table>
<thead>
<tr>
<th>Genomes</th>
<th>Genome sizes (bp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rat (Rnor_6.0)</td>
<td>2,909,701,677</td>
</tr>
<tr>
<td>Zebra fish (GRCz10)</td>
<td>1,464,443,456</td>
</tr>
<tr>
<td>Rat chr1 (Rnor_6.0)</td>
<td>290,094,217</td>
</tr>
<tr>
<td>C. elegans (WBcel235)</td>
<td>103,022,290</td>
</tr>
<tr>
<td>C. merlae (ASM9120v1)</td>
<td>16,728,967</td>
</tr>
</tbody>
</table>

All the tests are organized in two groups: group I and group II. In group I, we compare our method with the existing methods. In group II, we test the impact of the rankAll array compression and the multiple character checking.

- **Group I**

  To store \( BWT(\tilde{s}) \), we use 2 bits to represent a character \( \in \{a, c, g, t\} \) and store 4 rankAll values (respectively in \( A_a, A_c, A_g, \) and \( A_t \)) for every 4 elements (in \( L \)) with each taking 32 bits.

  In Fig. 15(a) and (b), we report the average time of testing the Rat (Rnor_6.0) for matching 100 reads of length 100 to 300 bps. From this figure, we can see that Algorithm A( ) outperforms all the other three methods. But the Amir’s method is better than the other two
methods. The BWT-based and the Cole’s method are comparable. However, for small \( k \), the Cole’s is a little bit better than the BWT-based method while for large \( k \) their performances are reversed.

![Graph showing time vs. varying values of \( k \) for BWT, Amir’s, Cole’s, and \( A() \).]

![Graph showing time vs. varying length of reads for BWT, Amir’s, Cole’s, and \( A() \).]

Fig. 15: Test results on varying values of \( k \) and read length

To show why \( A() \) has the best running time, we show the number \( n' \) of leaf nodes in the \( M \)-trees created by \( A() \) for some tests in Table 3, which demonstrates that \( n' \) can be much smaller than \( n \). Thus, the time complexity \( O(kn') \) of \( A() \) should be a significant improvement over \( O(n \sqrt{k \log k}) \) - the time complexity of Amir’s.

TABLE 3: NUMBER OF LEAF NODES OF S-TREES

<table>
<thead>
<tr>
<th>( k )/length-of-read</th>
<th>5/50</th>
<th>10/100</th>
<th>20/150</th>
<th>30/200</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of leaf nodes</td>
<td>2K</td>
<td>0.7M</td>
<td>16.5M</td>
<td>102M</td>
</tr>
</tbody>
</table>

In this test (and also in the subsequent tests), the time for constructing \( BWT(\tilde{s}) \) is not included as it is completely independent of \( r \). Once it is created, it can be repeatedly used.

In Fig. 11(b), we show the impact of read lengths. For this test, \( k \) is set to 25. It can be seen that only the BWT-based and the Cole’s are sensitive to the length of reads. For the BWT-based, more time is required to construct S-trees for longer reads while for the Cole’s longer paths in a suffix tree will be searched as the lengths of reads increase. For the other two methods: \( A() \) and the the Amir’s, the lengths of reads only impact the time for the read pre-processing, but it is completely overshadowed by the time spent on searching genomes. For the Amir’s, the time for recognizing breaks is linear in \( |r| \) [2] while for \( A() \) the time for generating the mismatch information is bounded by \( O(|r| \log |r|) \). No significant difference between them can be measured.

In Fig. 16(a) and (b), we report the test results of searching the Zebra fish (GRCz10).
Again, similar to Fig. 15(a), the performance of Algorithm \( A( ) \) is best, and the Amir’s is still better than both the BWT-based and the Cole’s. In Table 4, we show the number \( n' \).

**TABLE 4: NUMBER OF LEAF NODES OF S-TREES**

<table>
<thead>
<tr>
<th>( k/\text{length-of-read} )</th>
<th>5/50</th>
<th>10/100</th>
<th>20/150</th>
<th>30/200</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of leaf nodes</td>
<td>0.7K</td>
<td>0.30M</td>
<td>9.2M</td>
<td>89M</td>
</tr>
</tbody>
</table>

Fig. 16(b) shares the same features as Fig. 16(b). It also shows that only the BWT-based and the Cole’s are sensitive to the length of reads.

In Fig. 17, 18, and 159, we show the tests on Rat chr1 (Rnor_6.0), C. elegans (WBcel235), and C. meria (ASM9120v1), respectively.
From these figures, the most important feature we can observe is that as the size of genomes becomes smaller, the difference between the Amir’s and Cole’s diminishes. But the BWT-based and \( A() \) remain the worst and the best, respectively. Although \( A() \) is impacted by the number of leaf nodes of an \( S \)-tree, the impact factor is small in comparison with the size of the whole \( S \)-tree, which dominates the time complexity of the BWT-based method. Also, the big difference between \( A() \) and Amir’s shows that using \( M \)-trees the cost for creating mismatch information of \( r \)’s occurrences in \( s \) can be significantly reduced.

From these figures, the most important feature we can observe is that as the size of genomes becomes smaller, the difference between the Amir’s and Cole’s diminishes. But the BWT-based and \( A() \) remain the worst and the best, respectively. Although \( A() \) is impacted by the number of leaf nodes of an \( S \)-tree, the impact factor is small in comparison with the size of the whole \( S \)-tree, which dominates the time complexity of the BWT-based method. Also, the big difference between \( A() \) and Amir’s shows that using \( M \)-trees the cost for creating mismatch information of \( r \)’s occurrences in \( s \) can be significantly reduced.

- **Group II**

In this group, we check the impact of the impact of the rankAll array compression and the multiple character checking. For this purpose, we set \( \sigma \) to different values for different tests and then each time the sizes of \( A_x \)'s (\( x \in \Sigma \)) are different. However, for each test, we use the same size of reads and the same value of \( k \).

In Fig. 20(a), (b), (c) and (d), we show the test results by checking a read containing 150 characters against the first four gnomes: Rat (Rnor_6.0), Zebra fish (GRCz10), Rat chr1 (Rnor_6.0), and C. elegans (WBcel35), respectively. For each test, we have run the two versions of our method: with multiple character checking (with MC), and without multiple character checking (without MC), with \( k \) set to be 20. From these figures, we can see that the performance can be significantly improved if the multiple character checking is used.
9 CONCLUSION AND FUTURE WORK

In this paper, a new method to do the string matching with \( k \) mismatches is proposed. Its main idea is to transform the reverse \( \overline{s} \) of target string \( s \) to \( BWT(\overline{s}) \) and use the mismatch information over a pattern string \( r \) to speed up the computation. Its time complexity is bounded by \( O(\kappa n' + n \cdot \log m) \), where \( m = |r| \), \( n = |s| \), and \( n' \) is the number of leaf nodes of a tree structure produced during the search of a \( BWT(s) \). In the case of \( m \geq 2(k+1) \), the average value of \( n' \) is bounded by \( O\left(1 + \frac{1}{|\Sigma|}\right)^{k+1} \). Our experiments show that it has a better running time than any existing on-line and index-based algorithms.

As a future work, we will use the BWT to solve another important problem, the string matching with \( k \) errors. It seems to be more challenging than the \( k \) mismatches since the Levenshtein distance is more difficult to handle than the Hamming distance.

REFERENCES


Dear Sir/Madam,

attached please find a paper entitled “BWT Arrays and Mismatching Trees: A New Way for String Matching with k Mismatches”, submitted to *Theoretical Computer Science*.

The article is a modification and extension of a paper published in *Proc. Int. Conf. on Data Engineering, IEEE, April 19-22, 2017, USA, pp. 339 – 410*.

In advance, I would like to thank you and the referees for reading the manuscript.

Yours Sincerely,

Dr. Yangjun Chen