Theoretical Computer Science 726 (2018) 5–29

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the string matching with k mismatches ☆

Yangjun Chen *, Yujia Wu

Dept. of Applied Computer Science, University of Winnipeg, 515 Portage Ave., Winnipeg, Manitoba, R3B 2E9, Canada

ARTICLE INFO

Article history:
Received 26 June 2017
Received in revised form 12 January 2018
Accepted 1 February 2018
Available online 13 February 2018
Communicated by R. Giancarlo

Keywords:
String matching
DNA sequences
Tries
Burrows–Wheeler transformation

ABSTRACT

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 the Burrows–Wheeler transformation of s, denoted as $BWT(s)$, used as an index to search r against it. During the process, the precomputed mismatch information of r will be utilized to speed up the $BWT(s)$'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 $BWT(s)$. In the case of $m \geq 2(k+1)$, the average value of n' is bounded by $O((1 + \frac{1}{|\Sigma|})^{k+1})$, where Σ 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

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 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.

As an example, consider a target $s = ccacacagaagcc$, and a pattern $r = aaaaacaaac$. 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.

☆ This work is supported by NSERC 239074-01 (242523), Canada. The article is a modification and extension of a paper published in Proc. Int. Conf. on Data Engineering, IEEE, April 19–22, 2017, USA [13].

* Corresponding author.
E-mail address: y.chen@uwinnipeg.ca (Y. Chen).

https://doi.org/10.1016/j.tcs.2018.02.001
0304-3975/

```
    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.

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

This topic has received much attention in the research community and many efficient algorithms have been proposed, such as [2,15,20,27,32,41]. Among them, [20] and [27] are two on-line algorithms (using no indexes) with the worst-case time complexities bounded by $O(kn + m \log m)$, 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 [41] are also on-line strategies, but with a slightly better time complexity $O(n\sqrt{k}\log k)$ by utilizing the *periodicity* within r. Only the algorithms discussed in [15,32] 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 $O(m + n + (c \log n)^k / k!)$, 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 [32], s is transformed to an array by using the *Burrows-Wheeler* transformation (denoted $BWT(s)$) [1] as part of an index, called FM-index [19]. In comparison with *suffix trees*, $BWT(s)$ uses much less space [19]. However, the time complexity of [32] is bounded by $O(mn' + n)$, where n' is the number of leaf nodes of a tree 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 the Burrows–Wheeler transformation (BWT-transformation for short), but with the mismatching information within r being effectively utilized.

Specifically, two techniques are introduced, which will be combined with a scanning of $BWT(s)$:

- An efficient method to calculate the mismatches between $r[i..m]$ and $r[j..m]$ $(i, j \in \{1, \ldots, m\},\ i \neq j)$, 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 $r[i + q - 1] \neq r[j + q - 1]$ and it is their pth mismatch.
- A new tree 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 $i_1, i_2, \ldots, i_l$ in s such that $s[i_q + i - 1] \neq r[i]$ $(q = 1, \ldots, l)$. If v is at the pth level of D, it also shows that it is the pth mismatch between each $s[i_q..i_q + i - 1]$ 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 $O((1 + \frac{1}{|\Sigma|})^{k+1})$, where Σ 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 review the related work. In Section 3, we briefly describe how a Burrows–Wheeler transformation can be used to speed up string matches. Section 4 is devoted to the discussion of our algorithm to find all the occurrences of r in s, but with up to k mismatches. In Section 5, the time complexity of the algorithm is analyzed. In Section 6, we discuss how our method can be further improved by using the so-called *rankALL* mechanism and the *multi-character* checking. Section 7 reports the test results. Finally, we conclude with a short summary and a brief discussion on the future work in Section 8.

In the Appendix A, we show all the notations and symbols used in the paper for reference. Also, in the Appendix B, a detailed description of a step in the algorithm in Section 4 is given.

2. 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*. In the first case, all the occurrences of a pattern string r in a target string s will be searched. In the second case, 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 [45], 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 [25], 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 [8,20] works a little bit better than the *Knuth–Morris–Pratt*'s. In addition to the next-table, a skip-table *skip* of size $|\Sigma|$ (also for r) is kept, in which each entry $skip[w]$ is a smallest integer j such that $r[m-j] = w$. For a large alphabet and small pattern, the expected number of character comparisons is about n/m, and is $O(m+n)$ in the worst case. These two methods have sparked a series of subsequent research on this problem [1,3,16,29,31,40,46,47]. 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 $O\sum_{i=1}^{l} |r_1| + n)$ time. This method has been slightly improved in different ways. In [17], Commentz-Walter combines the Boyer–Moore's technique into the Aho–Corasick's algorithm. In [51], Wu and Manber 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 [52], 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 [11], 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 [39,49], suffix arrays [37], and more recently the Burrows–Wheeler transformation [9,12,32,33]. A suffix tree is in fact a *trie* structure [24] over all the suffixes of s; and by using the Weiner's algorithm [49] it can be built in $O(n)$ time. However, in comparison with the Burrows–Wheeler transformation, a suffix tree needs much more space. Especially, for DNA sequences the Burrows–Wheeler transformation works highly efficiently due to the small alphabet Σ of DNA strings. By the Burrows–Wheeler transformation, the smaller Σ is, the less space will be occupied by the corresponding indexes. According to a survey done by Li and Homer [34] 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 Burrows–Wheeler transformation. Our experiments also confirm this distinction [12,14]. 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 Burrows–Wheeler transformation 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 [23], 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 [22,30,35,36,43]. However, almost all experiments show that they are generally inferior to the suffix tree and the FM 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* [2,28]. 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 [2,4,5,21,27,28,41,46,47]. They are all on-line algorithms. Except those discussed in [2,5,21,27,28,41], 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 [21] and [28], 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 [2,41] 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 [32] is index-based, by which s is transformed to $BWT(s)$, 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 $BWT(s)$. If m is large, it can be worse than all those on-line methods discussed in [2,21,28,41]. Another index-based method is based on a brute-force searching of suffix trees [15]. Its time complexity is bounded by $O(m + n + (c \log n)^k/k!)$, where c is a very large constant. It can also be worse than an on-line algorithm when n is large and k is larger than a certain constant.

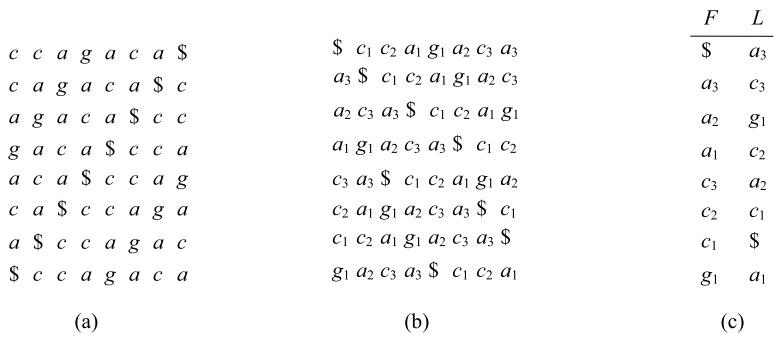

Fig. 1. Rotation of a string.

k errors When the distance function is the *Levenshtein* distance, the problem is known as the string matching with k *errors* [4]. By the Levenshtein distance, we have

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

where $d_{i,j}$ represents the distance between $r[1..i]$ and $s'[1..j]$, r_i (s'_j) the ith character in r (resp. jth character in s'), ϕ 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 [6,10,18,48]. They are all some kinds of variants of the *dynamic programming* paradigm with the worst-case time complexity bounded by O(mn). However, by the algorithm discussed in [10], the expected time can reach O(kn).

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 skillful and in general need a quadratic time [42]. 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 [38].

3. Burrows–Wheeler transformation

In this section, we give a brief description of the Burrows–Wheeler transformation to provide a discussion background.

3.1. 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 $s = ccagaca\$$. We can rotate s consecutively to create eight different strings, and put them in a matrix as illustrated in Fig. 1(a).

Now we sort the rows of the matrix alphabetically, and get another matrix, as shown in Fig. 1(b), which is called the *Burrow-Wheeler Matrix* [9] and denoted as $BWM(s)$. In particular, the last column L of $BWM(s)$, read from top to bottom, is called the Burrows–Wheeler transform and denoted as $BWT(s)$. So, for $s = ccagaca\$$, we have $BWT(s) = acgcac\$a$ (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 $c_1c_2a_1g_1a_2c_3a_3\$$.) For example, a_2 representing the second occurrence of a in s; and c_1 the first occurrence of c in s. In the same way, we can check all the other occurrences of different characters.

When ranking the elements e in both F and L in such a way that if e is the ith appearance of a certain character it will be assigned i, the same element will get the same number in these two columns. For example, in F the rank of a_3, denoted as $rk_F(a_3)$, is 1, showing that a_3 is the first occurrence of a in F. Its rank in L, $rk_L(a_3)$ 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* (also known as LF-mapping [19]), holds for all the elements. That is, for any element e in s, we always have

$$rk_F(e) = rk_L(e) \tag{1}$$

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

First, we notice that we can store F as $|\Sigma|+1$ intervals, such as $F_\$ = F[1..1]$, $F_a = F[2..4]$, $F_c = F[5..7]$, $F_g = F[8..8]$, and $F_t = \varnothing$ (empty set) for the above example (see Fig. 1(c).) We can also represent a segment within an F_x (with $x \in \Sigma$) as a

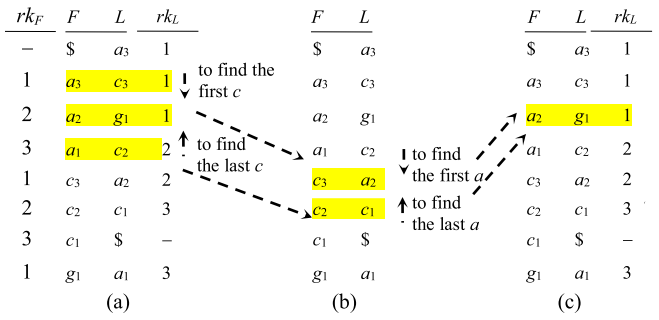

Fig. 2. Sample trace.

pair of the form $\langle x, [\alpha, \beta]\rangle$, where $\alpha \le \beta$ are two ranks of x. Thus, we have $F_a = F[2..4] = \langle a, [1, 3]\rangle$, $F_c = F[5..7] = \langle c, [1, 3]\rangle$, and $F_g = F[8..8] = \langle g, [1, 1]\rangle$.

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_a is 2 and Z_a is 4. Then, for a given $\langle x, [\alpha, \beta]\rangle$, its segment in F can be easily determined: $F[Y_x + \alpha - 1..Y_x + \beta - 1]$. In addition, we can use L_π to represent a range in L corresponding to a pair $\pi = \langle x, [\alpha, \beta]\rangle$. For example, in Fig. 1(c), $L_{\langle a,[1,3]\rangle} = L[2..4]$, $L_{\langle a,[2,3]\rangle} = L[3..4]$, $L_{\langle c,[1,2]\rangle} = L[5..6]$, and so on.

Finally, we use a procedure *search*(z, π) to search L_π to find the first and the last rank of z (denoted as α' and β', respectively, which then make up an interval $[\alpha', \beta']$) within L_π, and return $\langle z, [\alpha', \beta']\rangle$ as the result:

$$search(z, \pi) = \begin{cases} \langle z, [\alpha', \beta']\rangle, & \text{if } z \text{ appears in } L_\pi; \\ \phi, & \text{otherwise.} \end{cases} \tag{2}$$

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

Trace of searching $\bar{r}$ against s:

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

Step 2: Check $r[2] = c$. Call *search*$(c, L_{\langle a,[1,3]\rangle})$. It will search $L_{\langle a,[1,3]\rangle} = L[2..4]$ to find a range bounded by the first and last rank of c. Concretely, we will find $rk_L(c_3) = 1$ and $rk_L(c_2) = 2$. So, *search*$(c, L_{\langle a,[1,3]\rangle})$ returns $\langle c, [1, 2]\rangle$. 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*$(a, L_{\langle c,[1,2]\rangle})$. Notice that $L_{\langle c,[1,2]\rangle} = L[5..6]$. So, *search*$(a, L_{\langle c,[1,2]\rangle})$ returns $\langle a, [2, 2]\rangle$. 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: $\langle a, [1, 3]\rangle, \langle c, [1, 2]\rangle, \langle a, [2, 2]\rangle$. In general, for $\bar{r} = c_1 \cdots c_m$, its search against *BWT*(s) can always be represented as a sequence of pairs (with each made up of a character and an interval):

$$\langle x_1, [\alpha_1, \beta_1]\rangle, \ldots, \langle x_m, [\alpha_m, \beta_m]\rangle,$$

where $\langle x_1, [\alpha_1, \beta_1]\rangle = F_{x_1}$, and $\langle x_i, [\alpha_i, \beta_i]\rangle = search(x_i, L_{x_{i-1},[\alpha_{i-1},\beta_{i-1}]})$ for $1 < i \le m$. We call such a sequence as a *search sequence*. Thus, the time used for this process is bounded by $O(\sum_{i=1}^{m} \tau_i)$, where τ_i is the time for an execution of *search*$(x_i, L_{x_{i-1},[\alpha_{i-1},\beta_{i-1}]})$. However, this time complexity can be reduced to $O(m)$ by using the so-called *rankAll* method (with more space to be used [9]) and the multi-character checking to be discussed in Section 6.

From the above discussion, we can observe a most important property of the Burrows–Wheeler 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.

3.2. Construction of BWT(s)

A *BWT*(s) can be constructed in terms of a relationship to the *suffix arrays* [9,44].

As mentioned above, a string $s = a_1 a_1 ... a_n$ is always ended with \$ (i.e., $a_i \in \Sigma$ for $i = 1, \ldots, n-1$, and a_n = \$). Let $s[i] = a_i$ $(i = 1, 2, \ldots, n)$ be the ith character of s, $s[i..j] = a_i ... a_j$ a substring and $s[i..n]$ a suffix of s. Suffix array H of s is a permutation of the integers $1, ..., n$ such that $H[i]$ is the start position of the ith smallest suffix [37]. The relationship between H and $L = BWT(s)$ can be determined by the following formulas:

$$\begin{cases} L[i] = \$, & \text{if } H[i] = 1; \\ L[i] = s[H[i] - 1], & \text{otherwise.} \end{cases} \tag{3}$$

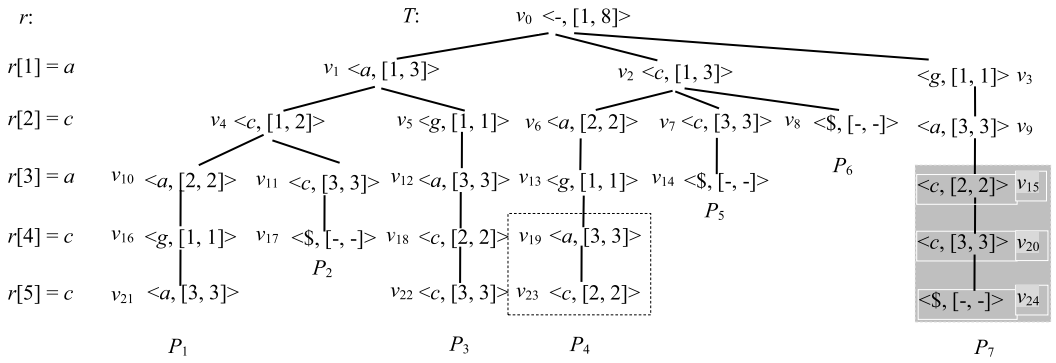

Fig. 3. Search for string matching with 2 mismatches.

Since a suffix array can be generated in $O(n)$ time (cf. for instance [53]), L can then be created in 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. [50] 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. This algorithm is further improved by Nong [54], whose approach needs only $O(|\Sigma| \log n)$-bits space. Either of the methods can be used for our purpose.

4. String matching with *k* mismatches

4.1. 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 FM 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 *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. The search of r against $BWT(\bar{s})$ (which is equivalent to the search of $\bar{r}$ against $BWT(s)$) can be represented as a tree structure called a *search tree* T (S-tree for short). In T, each node is a pair of the form $\langle x, [\alpha, \beta]\rangle$), and there is an edge from $v(=\langle x, [\alpha, \beta]\rangle)$ to $u(=\langle x', [\alpha', \beta']\rangle)$ if $search(x, L_v) = u$. Then, each path in T represents a process to find an occurrence of r in s with at most k mismatches.

As an example, consider the case where $r = acacc$, $s = acagacc$ 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.

In Fig. 3, v_0 is a virtual root, representing the whole L, and 'virtually' corresponds to the virtual starting character $r[0]$ = '–'. By exploring paths $P_1 = v_1 \to v_4 \to v_{10} \to v_{16} \to v_{21}$ and $P_3 = v_1 \to v_5 \to v_{12} \to v_{18} \to v_{22}$, we will find two occurrences of r with at most 2 mismatches: $s[1..5]$ $(= a_1c_1a_2g_1a_3)$ and $s[3..7]$ $(= a_2g_1a_3c_2c_3)$ while by any of other paths P_2, P_4, P_5, P_6 and P_7 no string matching with at most 2 mismatches can be found.

A node $\langle x, [\alpha, \beta]\rangle$ 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 $v_1 = \langle a, [1, 3]\rangle$ is a matching node since it corresponds to $r[1] = a$ while $v_{16} = \langle g, [1, 1]\rangle$ is a mismatching node since it corresponds to $r[4] = c$.

For a path P_l, we can store all its mismatching positions in an array B_l of length $k + 1$ such that $B_l[i] = j$ if $P_l[j] \neq r[j]$ and this is the ith mismatch between P_l and r, where $P_l[j]$ is the jth character appearing on P_l. If the number of mismatches, k', say, between P_l and r is less than $k + 1$, then the default value from position $k' + 1$ to $k + 1$ is ∞, i.e.,

$$B_l[k' + 1] = B_l[k' + 2] = \cdots = B_l[k + 1] = \infty.$$

We call B_l a *mismatch array*. For instance, in Fig. 3, for P_1, we have $B_1 = [4, 5, \infty]$, indicating that at position 4, 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 ∞ 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 ∞), we will store B as a B_l (and associate it with the leaf node of the corresponding P_l). 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 ∞, i.e., B will be changed to $[\infty, \infty, \infty]$.

Now we consider another path P_4. 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_{19}, 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, what is described above is the main process discussed in [32]. That is, during the execution of the algorithm proposed in [32], an S-tree will be generated. The only difference is that in [32] a simple heuristics is used, which precomputes, for each position i in r, the 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 $r[1..5] = acacc$ we have a substring: $r[1..4] = acac$, which 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 $O(n)$ by using $BWT(s)$ [32]. However, the theoretic time complexity of this method is still $O(mn' + n)$. 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_7 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.

4.2. 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 $\langle g, [1, 1]\rangle$ (v_{16} in Fig. 3) is compared to $r[4]$ while its second appearance of it (v_5) is to $r[2]$. Hence, we cannot use the result computed for v_{16} (when $\langle g, [1, 1]\rangle$ is first met) as the result for v_5.

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_1 (the mismatches recorded for P_1), instead of simply exploring P_3 again in a way done for P_1. To do so, for each pair i, $j \in \{1, \ldots, m\}$, 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 lth mismatch between them. For example, for $r = acacc$, we have $R_{23} = [1, 2, \infty, \infty, \infty]$. It is because $r_2 = r[2..5] = cacc$, $r_3 = r[3..5] = acc$, and the first two characters of them are not equal.

This design is inspired by the method discussed in [28], but with essential difference. By [28], only $R_{12}, \ldots, R_{1m}$ (giving the positions of the mismatches between the pattern r and its suffixes $r[2..m]$, $r[3..m]$, $\ldots$, $r[m..m]$) are created. (See Fig. 4(b) for illustration.) During the scanning of s, r will be matched with the substrings of s from left to right. At each step, when r is checked against $s[j..j+|r|-1]$, for instance, R_{1j} and R_{1i} for some $i < j$, as well as the mismatching information between r and $s[i..i+|r|-1]$ will be used to derive part of mismatches between r and $s[j..j+|r|-1]$ (see [28] to know how i is determined).

By ours, however, all R_{ij}'s ($i, j \in \{1, \ldots, m\}$) may be required, and clearly $O(km^2)$ time and space are needed to generate such information. For this reason, we will only precompute $R_{12}, \ldots, R_{1m}$, and derive the others whenever it is necessary.

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 $R_{12}, \ldots, R_{15}$ for r_1. In an R_{1i} ($i = 2, \ldots, 5$), 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 ∞ from position $k'+1$ onwards, i.e.,

$R_{1i}[k'+1] = R_{1i}[k'+2] = \cdots = R_{1i}[2k+1] = \infty$.

We will also use $\delta(R_{1i})$ to represent the number of all those entries in R_{1i}, which are not ∞. Trivially, R_{11} is set to be $[\infty, \ldots, \infty]$ and therefore $\delta(R_{11}) = 0$.

Using the algorithm of [28], $R_{12}, \ldots, R_{1m}$ 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_{1i} ($i = 2, \ldots, m$), since for generating an R_{1j}, up to $2k+1$ mismatches in some R_{1i} with $i < j$ are needed to get an efficient algorithm (see [28] 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_{1i} and R_{1j}, as well as r, to compute mismatching information for some new paths to avoid exploring them by using *search*( ). (A node $\langle x, [\alpha, \beta]\rangle$ is said to be the same as another node $\langle x', [\alpha', \beta']\rangle$ if $x = x'$, $\alpha = \alpha'$ and $\beta = \beta'$.) For this purpose, we design a general algorithm to create R_{ij} efficiently.

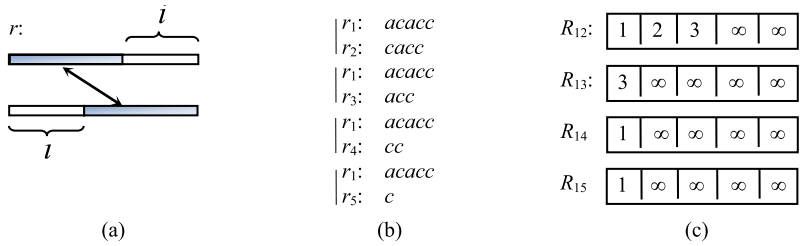

Fig. 4. Illustration for table R.

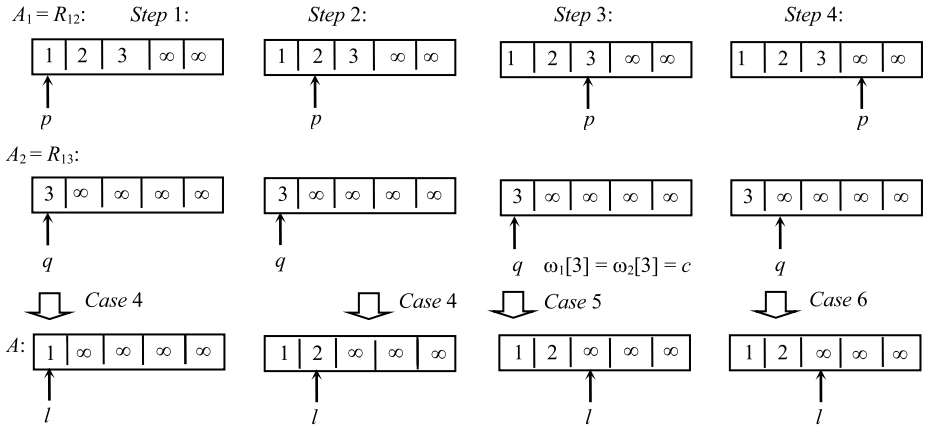

Fig. 5. Illustration for *merge*( ).

- Let ω, ω_1 and ω_2 be three strings. Let A_1 and A_2 be two arrays containing all the positions of mismatches between ω and ω_1, and ω and ω_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 exists. It is the ith mismatch between them.

The algorithm works in a way similar to the *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 ϕ, set A to ϕ and return. ($A = \phi$ means that A is empty.)
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 {if $\omega_1[p] \neq \omega_2[q]$, then $\{A[l] := q; l := l + 1;\}$ $p := p + 1; q := q + 1;\}$
6. If $p > |A_1|, q > |A_2|$, or both $A_1[p]$ and $A_2[q]$ are ∞, stop (if A_1 or A_2 has some remaining elements, which are not ∞, first append them to the rear of A, and then stop.)
7. Otherwise, go to (3).

We denote this process as *merge*(A_1, A_2, ω_1, ω_2). As an example, let us consider the case where $A_1 = R_{12} = [1, 2, 3, \infty, \infty]$ (representing the mismatches between $\omega = r[1..5] = acacc$ and $\omega_1 = r[2..4] = cacc$), $A_2 = R_{13} = [3, \infty, \infty, \infty, \infty]$ (the mismatches between ω and $\omega_2 = r[3..5] = acc$), and demonstrate the execution of $merge(A_1, A_2, \omega_1, \omega_2)$ in Fig. 5. The result is $A = [1, 2, \infty, \infty, \infty]$, showing the mismatches between these two substrings.

In step 1: $p = 1$, $q = 1$, $l = 1$. We compare $A_1[p] = A_1[1]$ and $A_2[q] = A_2[1]$. Since $A_1[1] = 1 < A_2[1] = 3$, $A[1]$ is set to be equal to $A_1[1] = 1$. $p := p + 1 = 2$, q is not changed, $l := l + 1 = 2$.

In step 2: $p = 2$, $q = 1$, $l = 2$. We compare $A_1[2]$ and $A_2[1]$. Since $A_1[2] = 2 < A_2[1] = 3$, $A[2]$ is set to be equal to $A_1[2] = 2$. $p := p + 1 = 3$, q is not changed, $l := l + 1 = 3$.

In step 3: $p = 3$, $q = 1$, $l = 3$. We compare $A_1[3]$ and $A_2[1]$, and find that $A_1[3] = A_2[1] = 3$. So, we need to compare $\omega_1[3]$ and $\omega_2[3]$. Since $\omega_1[3] = \omega_2[3] = c$, $A[3]$ is still ∞. $p := p + 1 = 4$, $q := q + 1 = 2$, l is not changed.

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 ∞. Stop.

Obviously, the running time of this process is bounded by $O(2k + 1)$.

Proposition 1. *Let A be the result of merge(A_1, A_2, ω_1, ω_2) with A_1, A_2, ω_1, ω_2 defined as above. Let k' be the number of mismatches between ω_1 and ω_2. Then, $A[i]$ must be the position of the ith mismatch between ω_1 and ω_2, or ∞, 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 lth mismatch between ω and ω_2 for some l, i.e., $\omega[j] \neq \omega_2[j]$ and $A_2[l] = j$.
ii) j corresponds to the fth mismatch between ω and ω_1 for some f, i.e., $\omega[j] \neq \omega_1[j]$ and $A_1[f] = j$.
If (i) holds, but (ii) not, step (3) in *merge*(A_1, A_2, ω_1, ω_2) will be executed. Since in this case, we have $\omega[j] \neq \omega_2[j]$ and $\omega[j] = \omega_1[j]$, step (3) is correct.

If (ii) holds, but (i) not, step (4) will be executed. Since in this case, we have $\omega[j] \neq \omega_1[j]$ and $\omega[j] = \omega_2[j]$, step (4) is also correct.

If both (i) and (ii) hold, no conclusion concerning $\omega_1[j]$ and $\omega_2[j]$ can be drawn and we need to compare them. In this case, step (5) is executed. If neither (i) nor (ii) is satisfied, we must have $\omega[j] = \omega_2[j]$ and $\omega[j] = \omega_1[j]$. So $\omega_2[j] = \omega_1[j]$, i.e., we have a matching at j. □

4.3. 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 4.1. 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_{ij} by executing *merge*($R_{1i}, R_{1j}, r[i..m-q+i], r[j..m-q+j]$), where $q = \max\{i, j\}$. Then, we will create 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_l going through v, figure out a sub-array of B_l, denoted as B_l^i, containing only those values in B_l, which are larger than or equal to i. Moreover, each value in it will be decreased by $i-1$. (For example, for $B_4 = [1, 2, 3]$, we have $B_4^1 = [1, 2, 3]$, $B_4^2 = [1, 2]$, $B_4^3 = [2]$, $B_4^4 = B_4^5 = \phi$.)
- Create the mismatch arrays for all the paths going through u by executing *merge*(B_l^i, R_{ij}, $P_l[i..m_l]$, $r[j..m]$) for each P_l, where $m_l = |P_l|$.

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

As an example, consider v_3 (in Fig. 3, labeled $\langle g, [1, 1]\rangle$ 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 generation of the corresponding subtree (i.e., part of P_7 shown in Fig. 6(a)):

1. Create R_{21}:
$R_{12} = [1, 2, 3, 4, \infty]$, $R_{11} = [\infty, \infty, \infty, \infty, \infty]$.
$R_{21} = merge(R_{12}, R_{11}, r[2..5], r[1..4]) = [1, 2, 3, 4]$.
2. Create part of mismatch information for P_7:
$B_3 = [2, \infty, \infty]$, $B_3^2 = [1, \infty, \infty]$. $P_3[2..5] = gacc$, $r[1..4]) = acac$. $merge(B_3^2, R_{21}, P_1[2..5], r[1..4]) = [1, 2, 3]$.

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.*

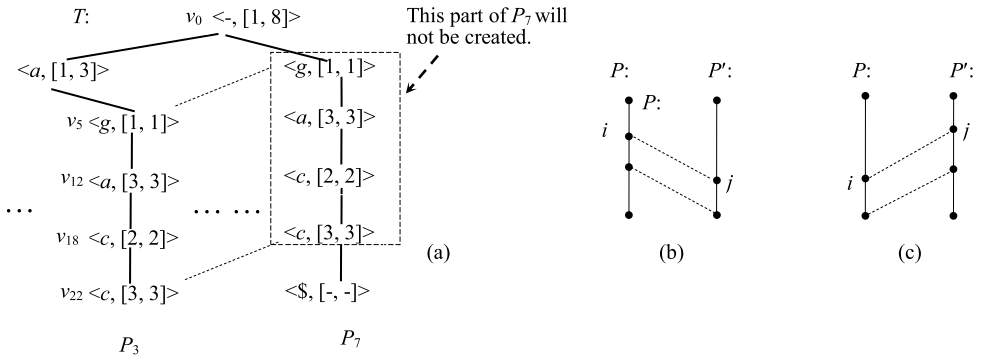

Fig. 6. Illustration for derivation of mismatch information.

4.4. Algorithm description

The main idea presented in the previous subsection can be dramatically improved. Instead of keeping a B_l for each P_l, we can organize all B_l's into a 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 the root or a mismatching node and its last node is a leaf node or has only mismatching nodes as its children.

For example, edge $v_1 \to v_4 \to v_{10}$ in T shown in Fig. 3 is a *MM-path*. Path $v_{12} \to v_{18} \to v_{22}$ 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 $\langle x, [\alpha, \beta]\rangle$ (compared to $r[i]$ for some i) in T we have a node of the form $\langle x, i\rangle$, and for each *MM*-path we have a node of the form $\langle -, 0\rangle$. There is an edge from u to u' if one of the following three conditions is satisfied:

- u is of the form $\langle x, i\rangle$ corresponding to a pair $\langle x, [\alpha, \beta]\rangle$ (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 $\langle -, 0\rangle$ 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 $\langle -, 0\rangle$ in D a *matching node*, and $\langle x, i\rangle$ 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 $\langle -, 0\rangle$ since it will be handled as a matching node. Then, each path in the M-tree corresponds to a B_l. For instance, path $u_0 \to u_1 \to u_4 \to u_{11}$ corresponds to $B_1 = [4, 5, \infty]$ if all the matching nodes on the path are ignored. For the same reason, $u_0 \to u_1 \to u_5 \to u_{12}$ corresponds to $B_2 = [3, 4, \infty]$.

In addition, we can store all the different nodes $v(= \langle x, [\alpha, \beta]\rangle)$ 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 $u = \langle x, i\rangle$ will be created in D and a pointer (associated with v) to u will be generated.
- If v is a matching node (compared to $r[i]$), a node $u = \langle -, 0\rangle$ 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 $\langle -, 0\rangle$, 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 $\langle a, [1, 3]\rangle$ (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 $u_1 = \langle -, 0\rangle$ in the M-tree D will be created. At the same time, we will insert $\langle a, [1, 3]\rangle$ into the hash table and produce a pointer associated with it to u_1 (see Fig. 7 for illustration). When $\langle c, [1, 2]\rangle$ (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 $\langle c, [1, 2]\rangle$ (in the hash table) to u_1. However, when $\langle g, [1, 1]\rangle$ (v_{16} in T shown in Fig. 3) is created, it is

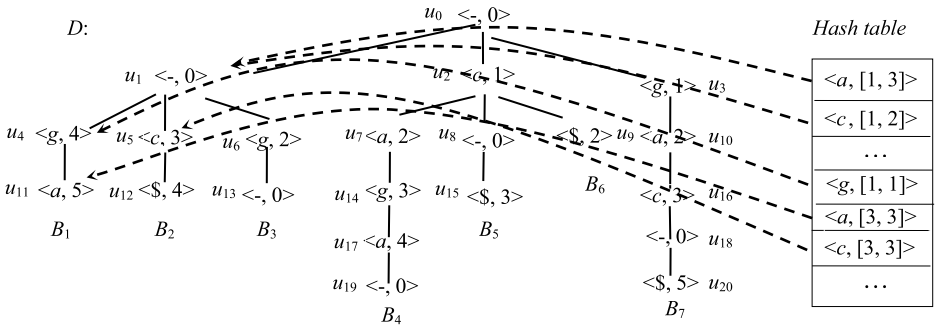

Fig. 7. A mismatch tree.

compared to $r[4] = c$ and we will create a mismatching node $u_4 = \langle g, 4\rangle$ in D since $g \neq c$. Again, we will insert $\langle g, [1, 1]\rangle$ into the hash table and produce a pointer associated with it to u_4 (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, κ, u), where

v – a node inserted into the hash table.

j – j is an integer to indicate that v is compared to $r[j]$.

κ – the number of mismatches between the path and $r[0..j]$ (recall that $r[0] =$ '–').

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 $e = (v, j, \kappa, u)$ with $v = \langle x, [\alpha, \beta]\rangle$ is popped out from S, we will check whether $x = r[j]$.

i) If $x \neq r[j]$, we will generate a node $u' = \langle x, j\rangle$ (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 $\langle -, 0\rangle$. If it is not the case or u is the root of D, generate a node $u' = \langle -, 0\rangle$. Otherwise, set u' to be u.

iii) Using *search*( ) to find all the children of v: $v_1, \ldots, v_l$. Let $v_i = \langle y_i, [\alpha_i, \beta_i]\rangle$ $(i = 1, \ldots, l)$. Push each $(v_i, j + 1, \kappa', u')$ into S with κ' being κ or $\kappa + 1$, depending on whether $y_i = r[j + 1]$. If $y_i = r[j + 1], \kappa' = \kappa$. Otherwise, $\kappa' = \kappa + 1$.

Obviously, in the case that $x = \$$, $j = |r|$, or $\kappa > k + 1$, (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 1. In this example, we run the above process on $r = acacc$ and $L = BWT(\bar{s})(\bar{s} = ccagaca)$ 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*, $v_0 = \langle -, [1, 8]\rangle$. Push $(v_0, 0, 0, \phi)$ into S, where ϕ 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 ϕ. Push $\langle v_3, 1, 1, u_0\rangle$, $\langle v_2, 1, 1, u_0\rangle$, $\langle v_1, 1, 0, u_0\rangle$ 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. $v_1 = \langle a, [1.3]\rangle$. Since $r[1] = a$, a matching node $u_1 = \langle -, 0\rangle$ will be created and set to be a child of u_0. Then, push two entries $\langle v_5, 2, 1, u_1\rangle$ and $\langle v_4, 2, 0, u_1\rangle$ into S, where v_4 is the child of v_1. See Fig. 8(c).

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

Step 5: Pop out $\langle v_{10}, 3, 0, u_1\rangle$ from S. $v_8 = \langle a, [2, 2]\rangle$. $r[3] = a$. 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 $\langle v_{16}, 4, 1, u_1\rangle$ from S. $v_{16} = \langle g, [1, 1]\rangle$. $r[4] = c \neq g$. A mismatching node $u_4 = \langle g, 4\rangle$ will be generated. Push $\langle v_{21}, 5, 2, u_4\rangle$ into S. See Fig. 8(f).

Step 7: Pop out $\langle v_{21}, 5, 2, u_1\rangle$ from S. $v_{21} = \langle a, [3, 3]\rangle$. $r[5] = c \neq a$. A mismatching node $u_{11} = \langle a, 5\rangle$ will be generated. No new entries will be pushed into S since now $j = |r|$. See Fig. 8(g).

Step 8: Pop out $\langle v_{11}, 3, 1, u_1\rangle$ from S. $v_{11} = \langle c, [3, 3]\rangle$. $r[3] = a \neq c$. A mismatching node $u_5 = \langle c, 3\rangle$ will be generated. Push $\langle v_{17}, 4, 1, u_5\rangle$ 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.

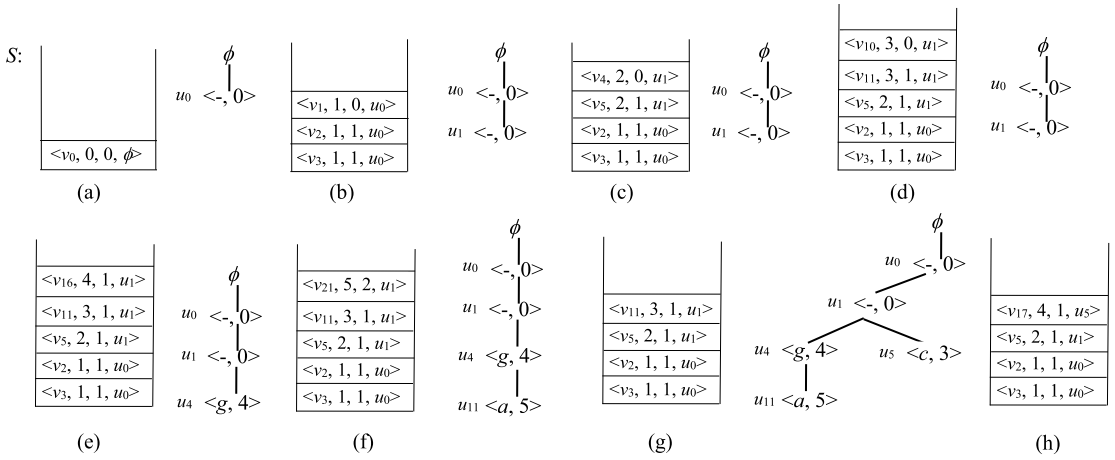

Fig. 8. Illustration for stack changes.

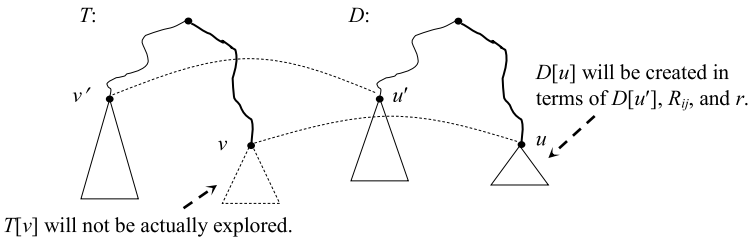

Fig. 9. Illustration for generation of subtrees in T'.

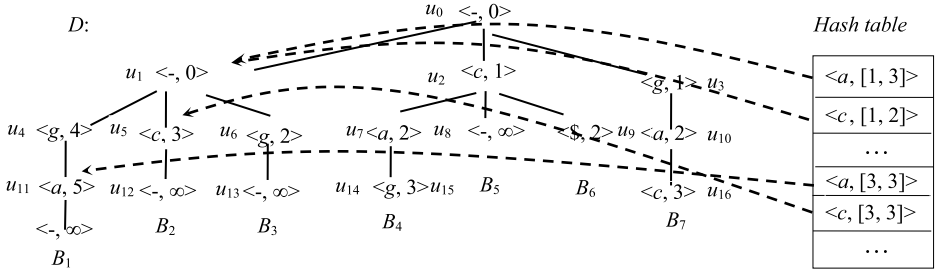

Fig. 10. A modified mismatch tree.

As with the basic process, each time a node $v = \langle x, [\alpha, \beta]\rangle$ (compared to $r[j]$) is encountered, which is the same as a previous one $v' = \langle x', [\alpha', \beta']\rangle$ (compared to $r[i]$), 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 4.2 and 4.3. In other words, $D[u]$ (to be created) corresponds to the mismatch arrays for all the paths going though v in T, which will not be actually produced. See Fig. 9 for illustration.

For this purpose, we introduce a third kind of nodes of the form $\langle -, \infty\rangle$ into D to represent symbol ∞ 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 \$ will be removed if its parent has only one child since in this case it needn't be checked.)

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, γ, h) with w being a node in $D[u']$, γ an entry in R_{ij}, and h the number of mismatching nodes on the path from the root to the node to be created in $D[u]$. Initially, put $(u', R_{ij}[1], h')$ 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 $e = (w, R_{ij}[l], h)$. Assume that $w = \langle z, f\rangle$ and $R_{ij}[l] = val$.
   - If $\langle z, f\rangle$ is equal to $\langle -, 0\rangle$, then create a copy of $\langle -, 0\rangle$ added to $D[u]$. Let $u_1, \ldots, u_g$ be the children of w. We will enqueue (append at the end) $(u_1, R_{ij}[l], h), \ldots, (u_g, R_{ij}[l], h)$ into Q in turn.

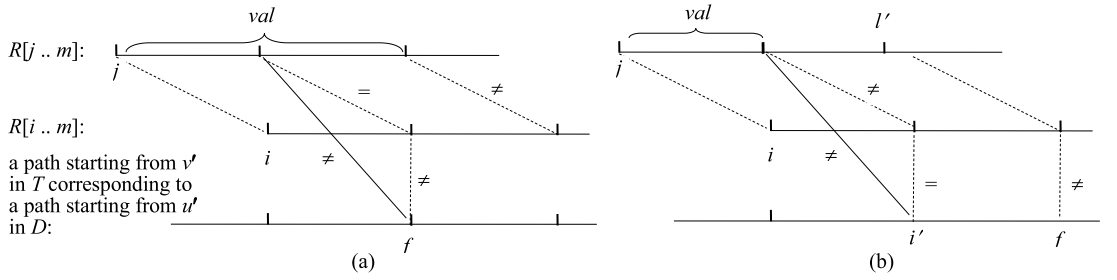

Fig. 11. Illustration for generation of nodes in D.

- If $\langle z, f\rangle$ is a mismatching node, do (2), (3), or (4).
- If $\langle z, f\rangle$ is equal to $\langle -, \infty\rangle$, do (5).

2. If $f < i + val - 1$, add $\langle z, j + f - i + 1\rangle$ to $D[u]$. If $h < k + 1$, enqueue $(u_1, R_{ij}[l], h + 14), \ldots, (u_g, R_{ij}[l], h + 1)$ into Q.
3. If $f > i + val - 1$ (and $f \neq \infty$), we will scan R_{ij} starting from $R_{ij}[l]$ until we meet the largest $l' \leq k - h + l$ such that $f > i + R_{ij}[l'] - 1$. For each $R_{ij}[q]$ $(l \leq q \leq l')$, we create a new node $\langle r[i + R_{ij}[q] - 1], j + R_{ij}[q] - 1\rangle$ added to $D[u]$. If $l' < k - h + l$, enqueue $\langle w, R_{ij}[l' + 1], h + l' - l + 1\rangle$ 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 $(u_1, R_{ij}[l + 1], h + 1), \ldots, (u_g, R_{ij}[l + 1], h + 1)$ into Q. If $z = r[j + val - 1]$, create a node $\langle -, 0\rangle$ added to $D[u]$. (If its parent is also $\langle -, 0\rangle$, it will be merged into its parent.) Also, enqueue $< u_1, R_{ij}[l + 1], h), \ldots, < u_g, R_{ij}[l + 1], h)$ into Q.
5. If $w = \langle -, \infty\rangle$, scan R_{ij} starting from $R_{ij}[l]$ until we find the largest $l' \leq k - h + l$ such that $R_{ij}[l] \neq \infty$. For each $R_{ij}[q]$ $(l \leq q \leq l')$, we create a new node $\langle r[i + R_{ij}[q] - 1], j + R_{ij}[q] - 1\rangle$ added to $D[u]$. If $l' < k - h + l$, add $\langle -, \infty\rangle$ to $D[u]$.

Roughly speaking, in the above process, (2) corresponds to step 3 in *merge*( ), (3) to step 4 in *merge*( ), (4) to step 5 in *merge*( ), and (5) to step 6 in *merge*( ). Specifically, the following correspondence between the above process and *merge*( ) can be recognized:

$R_{ij} - A_1$,

$r[i..m] - \omega_1$,

a path in $D[u'] - A_2$,

$r[j..m] - \omega_2$,

a path in $D[u'] - A$.

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

$$P[f] \neq r[f], r[f] = r[j + f - i] \Rightarrow P[f] \neq r[j + f - i] \quad \text{(see Fig. 11(a) for illustration)},$$

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 fth 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 $R_{ij}[q] = i'$ $(l \leq q \leq l')$,

$$P[i'] = r[i'], r[i'] \neq r[j + i' - i] \Rightarrow P[i'] \neq r[j + i' - i] \quad \text{(see Fig. 11(b) for illustration)}.$$

Thus, for each $R_{ij}[q]$ $(l \leq q \leq l')$, 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. (See Appendix B for a detailed description of this process.)

Denote the above process by *node-creation*$(w, h, \gamma, i, j, R_{ij})$. We have the following proposition.

Proposition 2. *Node-creation*$(w, h, \gamma, i, j, R_{ij})$ *creates nodes in* $D[u]$ *correctly.*

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

Example 2. As an example, consider Fig. 3 and Fig. 10 once again. Assume that $D[u_6]$ (shown in Fig. 10) has been created. Here, $u_6 = \langle g, 2\rangle$ is generated when we meet $\langle g, [1,1]\rangle$ (compared to $r[2] = c$). When we meet $\langle g, [1,1]\rangle$ (compared to $r[1] = a$) again, we generate the corresponding subtree (see u_3 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 $(\langle g, 2\rangle, R_{21}[1] = 1, 0)$ into Q.
Step 3: $(\langle g, 2\rangle, 1, 0) :=$ dequeue(Q). $h = 0$. $f = 2 = R_{21}[1] + i - 1 = 1 + 2 - 1 = 2$. (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 $\langle g, 1\rangle$ (u_3 in Fig. 1) will be generated. Enqueue $(u_{13}, R_{21}[2], h + 1) = (\langle -, \infty\rangle, 2, 1)$.
Step 4: $(\langle -, \infty\rangle, 2, 1) :=$ dequeue(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: $\langle r[i + R_{ij}[2] - 1],\ j + R_{ij}[2] - 1\rangle = \langle r[3], 2\rangle = \langle a, 2\rangle$ and $\langle r[i + R_{ij}[3] - 1],\ j + R_{ij}[3] - 1\rangle = \langle r[4], 3\rangle = \langle c, 3\rangle$ 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 $\langle -, \infty\rangle$ 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 4.3.

The following is the formal description of the working process.

Algorithm *strMisMatch*(L, r, k).

```
begin
1.  create root of T; push(S, (root, 0, 0, φ));
2.  while S is not empty do {
3.    (v, j, κ, 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_ij := merge(R_1i, R_1j, r[i..m − q + i], r[j..m − q + j]);
7.              enqueue(Q, (p(v′), R_ij[1], h)), where h is the number of mismatching nodes on the path from the root
                to the node to be created (for v) in D[u];
8.        while Q is not empty do {
9.            (w, γ, h′) := dequeue(Q); node-creation(w, h′, γ, i, j, R_ij); } }
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 κ ≤ k then {
16.           for each y ∈ ∑ within L_v do {
17.           w := search(y, L_v);
18.           if w ≠ φ then {
19.               if y = r[j + 1] then push(S, (w, j + 1, κ, u′));
20.               if y ≠ r[j + 1] and κ < k then {push(S, (w, j + 1, κ + 1, u′));
21. }}}}}
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 $(v,\ j,\ \kappa,\ u)$ 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.) This can be done in O(1) time by maintaining a hash table $\Im$ as illustrated in Fig. 7, by which we will use a hash function *HASH* to create a hash address $HASH(v) = j$ in $\Im$. If $\Im[j]$ is not Nil, we will compare v with each value in the corresponding *conflict* chain. In this way, we can find whether v is the same as a previous one. If not, we will append v to the end of the conflict chain (if $\Im[j]$ is Nil, we will create a conflict chain associated with $\Im[j]$ and add v to it as the first value). At the same time, 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 *search*( ) (see line 17) and pushed into S (see lines 18, and 19). Otherwise, we will first create R_{ij} by executing *merge*$(R_{1i},\ R_{1j},\ r[i..m - q + i],\ r[j..m - q + j])$, 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-transformation for the reverse $\bar{s}$ of a target string s, and r a pattern. Algorithm strMisMatch(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 *strMisMatch*(L, r, k), two data structures will be generated: a hash table and a mismatching tree D, in which some subtrees in D are derived by using the mismatching information over r. Replacing each matching node in

D with its corresponding maximum matching path and each mismatching node $\langle x, i\rangle$ with its corresponding pair $\langle x, [\alpha, \beta]\rangle$ (compared to $r[i]$), we will get an S-tree, in which each path corresponds to a *search sequence* discussed in Section 4. 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. □

5. 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'. In the worst case, n' is bounded by $O(|\Sigma|^{k+1})$ since each node in a M-tree has at most $|\Sigma|$ children and the height of the M-tree is bounded by $k+1$. In following, we estimate the average value of n'.

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

$$\binom{k+1}{i} p^{m-i} q^i = \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{m-i} \left(1 - \frac{1}{|\Sigma|}\right)^i. \tag{4}$$

The number of such paths is obviously bounded by $|\Sigma|^i$.

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

$$\sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{m-i} \left(1 - \frac{1}{|\Sigma|}\right)^i |\Sigma|^i = \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} (|\Sigma| - 1)^i \left(\frac{1}{|\Sigma|}\right)^{m-k-1}. \tag{5}$$

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

$$\sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} (|\Sigma| - 1)^i \left(\frac{1}{|\Sigma|}\right)^{m-k-1} \leq \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} (|\Sigma| - 1)^i \left(\frac{1}{|\Sigma|}\right)^{k+1}$$
$$\leq \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} = \left(1 + \frac{1}{|\Sigma|}\right)^{k+1}.$$

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

$$\sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} (|\Sigma| - 1)^i \left(\frac{1}{|\Sigma|}\right)^{m-k-1} = \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} \left(\frac{|\Sigma| - 1}{|\Sigma|^{\frac{m-k-1}{i}}}\right)^i$$
$$\leq \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{|\Sigma|}\right)^{k-i+1} \left(\frac{|\Sigma| - 1}{|\Sigma|^{\delta}}\right)^i = \left(\frac{1}{|\Sigma|} + |\Sigma|^{1-\delta} - \frac{1}{|\Sigma|^{\delta}}\right)^{k+1}.$$

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 $O((1 + \frac{1}{|\Sigma|})^{k+1})$.

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 $O(k(1 + \frac{1}{|\Sigma|})^{k+1} + n + m \log m)$ if the execution of 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 *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 $O(k(1 + \frac{1}{|\Sigma|})^{k+1} + n + m \log m)$. □

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

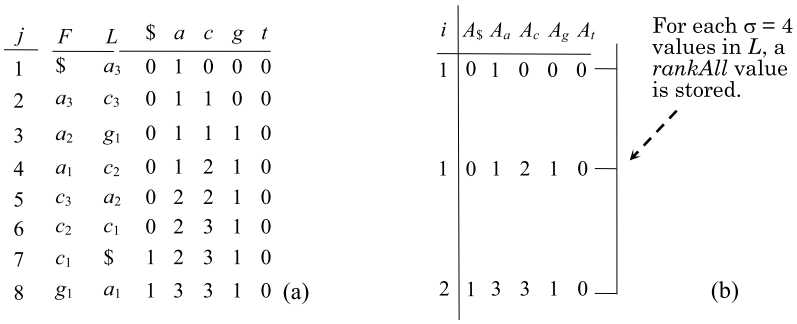

Fig. 12. LF-mapping and rank-correspondence.

6. Further improvements

The operation *search*( ) discussed in 3.1 can be greatly improved in two ways. One is to use the so-called *rankAll* mechanism, by which $|\Sigma|$ arrays are arranged each for a character $x \in \Sigma$ [9]. With help of such data structures, the cost of *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.

6.1. RankAll

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

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 α 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 $L[i](i \in \{1, \ldots, n\})$, 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_x. Obviously, doing so, more search will be required. In practice, the size σ 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_x) will be stored. That is, we will not store all the values in Fig. 12(a), but only store \$[4], a[4], c[4], g[4], t[4], and \$[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_x by using the following formulas:

$$x[j] = A_x[i] + \rho, \tag{6}$$

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

$$x[j] = A_x\left[i'\right] - \rho', \tag{7}$$

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

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

Another way to implement *search*( ) is to construct a *Wavelet tree* [56], instead of rankAll arrays, for L to support rank queries. By a rank query, we will find the number of a certain character's appearances up to a given position in L. Thus, by using a Wavelet tree, we can dynamically compute the values in each A_x $(x \in \Sigma)$, rather than storing them explicitly. The size of a Wavelet tree is bounded by $O(|L| \log |\Sigma|)$ bits. If the Wavelet tree is well balanced, $O(\log |\Sigma|)$ rank operations over a binary string need to be performed (see the discussion in Chapter 7 of [55]). Each of such operations requires only a constant time by using an auxiliary data structure, called *RRR* [57]. However, the size of this data structure is comparable to the rankAll when Σ is small (see page 15 in [57]).

6.2. Multiple character checking

From the above discussion, we can see that each time we meet a node $v = \langle x, [\alpha, \beta] \rangle$ 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

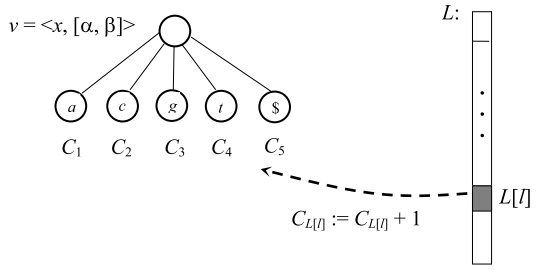

Fig. 13. Illustration for counters.

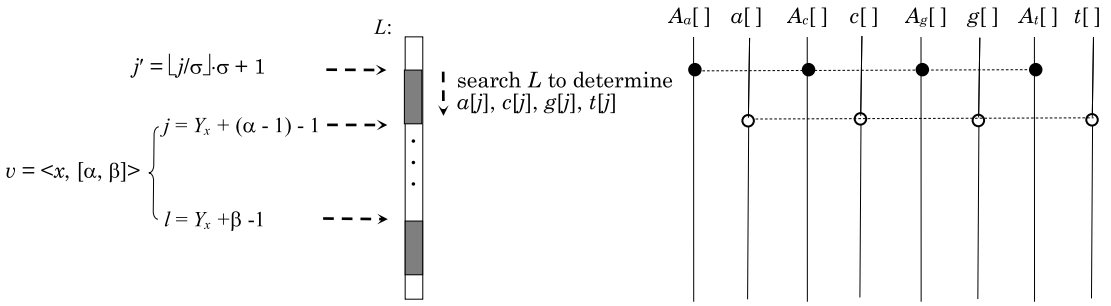

Fig. 14. Illustration for *sDown*( ).

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 Σ. 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 Fig. 13 for illustration.

With these counters, we change *sDown*(L, j, σ, x) and *sUp*(L, j, σ, x) to *sDown*(L, j, σ) and *sUp*(L, j, σ), 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, σ) 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 $C_i + A_i[\lfloor j/\sigma \rfloor]$. Remember that C_i records the number of i's appearances within $L[j'..j]$.

Function *sDown*(L, j, σ)

```
begin
1.  C_i := 0 for each i ∈ ∑;
2.  l := ⌊j/σ⌋ · σ + 1;
3.  while l ≤ j do {
4.      C_L[l] := C_L[l] + 1;
5.      l := l + 1;
6.  }
7.  for k = 1 to |∑| do {
8.      E[k] := A_k[⌊j/σ⌋] + C_k;
9.  }
10. return E;
end
```

In the algorithm *sDown*(L, j, σ), $L[j'..j]$ is scanned only once in the main **while**-loop (see lines 3–6). For each encountered entry $L[l]$ $(j' \le l \le j)$, $C_{L[l]}$ 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.

sUp(L, j, σ) is dual to *sDown*(L, j, σ), in which a segment of L will be searched bottom-up.

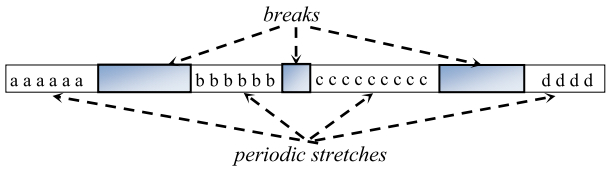

Fig. 15. Illustration for periodic stretches and breaks.

Function *sUp*(L, j, σ)

begin
1. $C_i := 0$ for each $i \in \sum$;
2. $l := \lceil j/\sigma \rceil \cdot \sigma$;
3. **while** $l \geq j+1$ **do** {
4. $\quad C_{L[l]} := C_{L[l]} + 1$;
5. $\quad l := l - 1$;
6. }
7. **for** $k = 1$ to $|\sum|$ **do** {
8. $\quad E[k] := A_k[\lceil j/\sigma \rceil] - C_k$;
9. }
10. return E;

end

By using the above two procedures, *search*(z, π) will be changed to *search*(π) as described below.

Function *search*(π)

begin
1. let $\pi = \langle x, [\alpha, \beta] \rangle$;
2. $j := Y_x + \alpha - 1$; $j := j - 1$; $i := \lfloor j/\sigma \rfloor$; $i' \leftarrow \lceil j/\sigma \rceil$;
3. **if** $j - i \cdot \sigma = 0$ **then** { **for** $k = 1$ to $|\sum|$ **do** $\{E[k] \leftarrow A_k[j];\}$
4. **if** $j - i \cdot \sigma \leq i' \cdot \sigma - j$ **then** $E := sDown(L, j, \sigma)$
5. **else** $E := sUp(L, j, \sigma)$
6. $j := Y_x + \beta - 1$; $i := \lfloor j/\sigma \rfloor$; $i' := \lceil j/\sigma \rceil$;
7. **if** $j - i \cdot \sigma \leq i' \cdot \sigma - j$ **then** $E' := sDown(L, j, \sigma)$
8. **else** $E' := sUp(L, j, \sigma)$
9. }
10. return (E, E');

end

The return value of the algorithm *search*(π) is a pair of arrays: (E, E'), from which we can construct all the five children of a node v representing the input $\pi = \langle x, [\alpha, \beta] \rangle$: $\langle a, [E[1]+1, E'[1]] \rangle$, $\langle c, [E[2]+1, E'[2]] \rangle$, $\langle g, [E[3]+1, E'[3]] \rangle$, $\langle t, [E[4]+1, E'[4]] \rangle$, and $\langle \$, [E[5]+1, E'[5]] \rangle$.

7. Experiments

In our experiments, we have tested altogether four different methods:

- *BWA* [32],
- *Amir et al.'s method* [2] (Amir for short),
- *Cole et al.'s method* [15] (Cole for short),
- *Algorithm strMisMatch discussed in this paper* (*sMM*( ) for short)

By the BWA method, an S-tree will be created as described in Section 4, 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 $2k$ aperiodic substrings, called breaks, as illustrated in Fig. 15. 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.

By the Cole's, a suffix tree for a target is constructed. (The code for constructing suffix trees is taken from the *gsuffix* package [7]: 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 2.40 GHz Intel Xeon E5-2630 processor with 32 GB RAM.

Table 2
Characteristics of genomes.

Genomes	Genome sizes (bp)	Size of BWT (Mbyte)	Time for creating BWT (s)
Rat (Rnor_6.0)	2,909,701,677	11,638.81	3127.8
Zebra fish (GRCz10)	1,464,443,456	5,857.78	1562.3
Rat chr1 (Rnor_6.0)	290,094,217	1,160.4	297.04
C. elegans (WBcel235)	103,022,290	412.1	135.2
C. merolae (ASM9120v1)	16,728,967	66.9	18.15

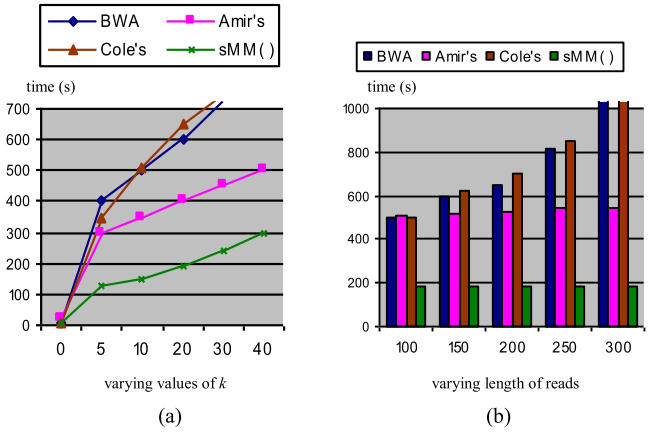

Fig. 16. Test results on varying values of k and read length (on the Rat genome).

Table 3
Number of leaf nodes of S-trees.

k/length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	2 K	0.7 M	16.5 M	102 M

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 [26]. For them, all their lengths, and the corresponding BWT arrays, as well as the time for creating these BWT arrays are listed. 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 [35] with a default model for single reads simulation. Concretely, we take 5000 reads with length varying from 100 bps to 300 bps.

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 $BWT(\bar{s})$, 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. 16(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 *strMisMatch*( ) outperforms all the other three methods. But the Amir's method is better than the other two methods. The BWA and the Cole's method are comparable. However, for small k, the Cole's is a little bit better than the BWA method while for large k their performances are reversed.

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

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

In Table 4, we show the space used by all the four tested algorithms when checking reads of 200 characters long with at most 30 mismatches.

From Table 4, we can see that the Amir's uses the least space, by which only a space for accommodating the target string and the pattern string is used. Our method uses a little more space than the BWA since beside the BWT array some memory has to be arranged to store the M-tree created during the working process. The space overhead of the Cole's is worst among all the other methods since the suffix tree for the genome has to be loaded and manipulated by it.

Table 4
Space usage.

Algorithms	BWA	Amir's	Cole's	sMM
Space usage (Mbyte)	11,638	2,909	26,656	11,751

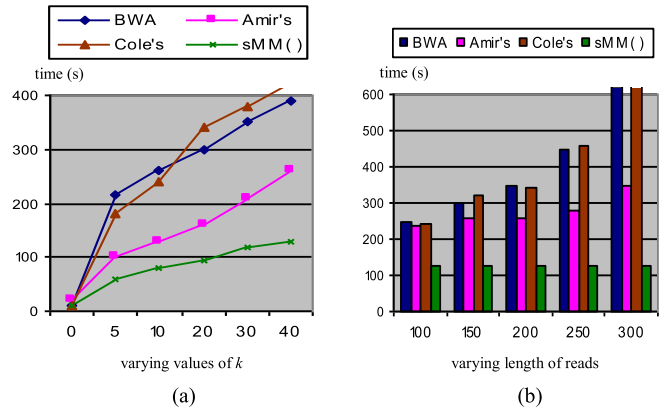

Fig. 17. Test results on varying values of k and read length (on the Zebra fish genome).

Table 5
Number of leaf nodes of S-trees.

k/length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	0.7 K	0.30 M	9.2 M	89 M

Table 6
Space usage.

Algorithms	BWA	Amir's	Cole's	sMM
Space usage (Mbyte)	5,857	1,464	21,012	5,941

Table 7
Space usage.

Algorithms	BWA	Amir's	Cole's	sMM
Space usage (Mbyte)	1,161	290	13,653	1,163

In Fig. 16(b), we show the impact of read lengths. For this test, k is set to 25. It can be seen that the BWA and the Cole's are more sensitive to the length of reads than *strMisMatch*( ) and the Amir's. For the BWA, 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: *strMisMatch*( ) and 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 *strMisMatch*( ) the time for generating the mismatch information is bounded by $O(|r| \log |r|)$. No significant difference between them can be measured since r is quite short in the tests.

In Fig. 17(a) and (b), we report the test results of searching the Zebra fish (GRCz10).

Again, similar to Fig. 16(a), the performance of Algorithm *strMisMatch*( ) is best, and the Amir's is still better than both the BWA and the Cole's.

In Table 5, we show the number n'.

In Table 6, we show the space usage when checking reads of 200 characters long with at most 30 mismatches.

Fig. 17(b) shares the same features as Fig. 16(b). It also shows that only the BWA and the Cole's are sensitive to the length of reads.

In Fig. 18, 19, and 20, we show the tests on Rat chr1 (Rnor_6.0), C. elegans (WBcel235), and C. merolae (ASM9120v1), respectively. Their space usage when checking reads of 200 characters long with at most 30 mismatches are shown respectively in Tables 7, 8 and 9.

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 BWA and *strMisMatch*( ) remain the worst and the best, respectively. Although *strMisMatch*( ) is impacted by the number of leaf nodes of an S-tree, the impact factor is small in

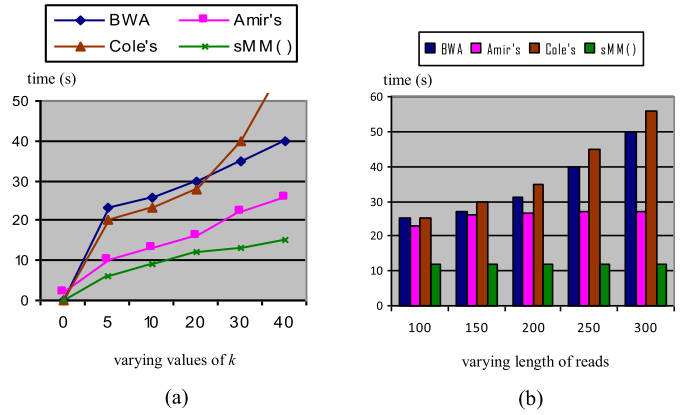

Fig. 18. Test results on varying values of k and read length (on the Rat chr1 genome).

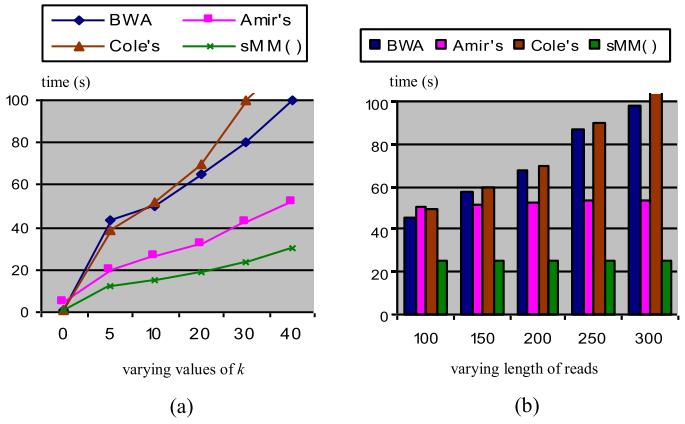

Fig. 19. Test results on varying values of k and read length (on the C. elegans genome).

Table 8
Space usage.

Algorithms	BWA	Amir's	Cole's	sMM
Space usage (Mbyte)	412.1	103	4,945	412.5

Table 9
Space usage.

Algorithms	BWA	Amir's	Cole's	sMM
Space usage (Mbyte)	66.90	16	602.1	66.92

comparison with the size of the whole S-tree, which dominates the time complexity of the BWA method. Also, the big difference between *strMisMatch*( ) 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 σ 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. 21(a), (b), (c), (d), and (e), we show the test results by checking a read containing 150 characters against all the five gnomes: Rat (Rnor_6.0), Zebra fish (GRCz10), Rat chr1 (Rnor_6.0), C. elegans (WBcel35), and C. merolae (ASM9120v1), respectively. For each test, we have run the two versions of our method: with multi-character checking (with MC), and without multi-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 multi-character checking is used.

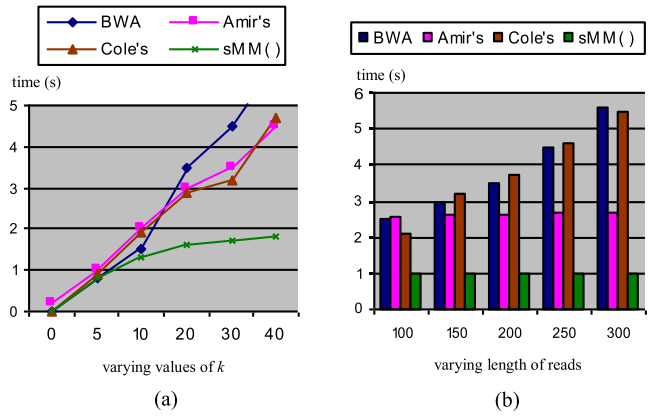

Fig. 20. Test results on varying values of k and read length (on the C. merolae genome).

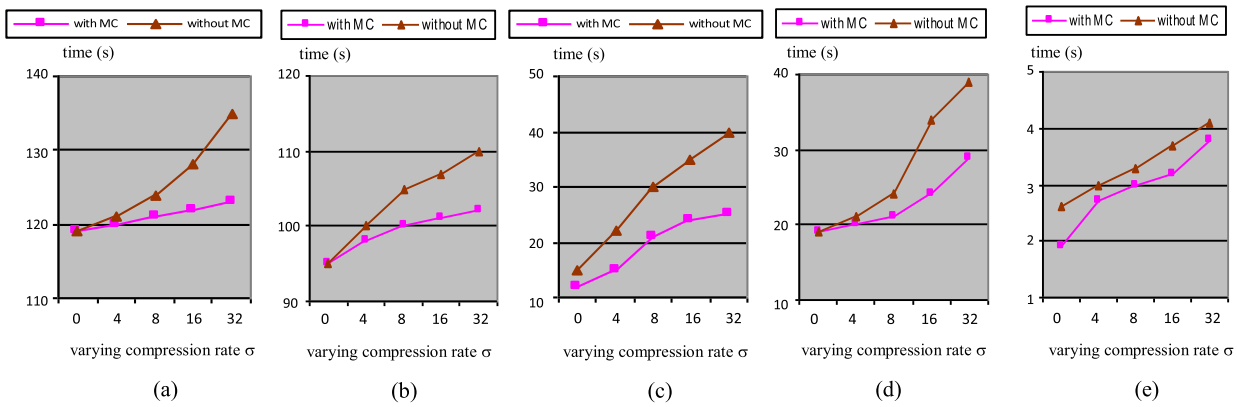

Fig. 21. Impact of rankAll array compression and multiple character checking.

Table 10
Space usage.

	Rat (Rnor_6.0)	Zebra fish (GRCz10)	Rat chr1 (Rnor_6.0)	C. elegans (WBcel235)	C. merolae (ASM9120v1)
$\sigma = 0$	–	23,796.7 Mb	4,712.2 Mb	1,673.1 Mb	260.4 Mb
$\sigma = 4$	11,655 Mb	5,863.9 Mb	1,162 Mb	412.1 Mb	66.9 Mb
$\sigma = 8$	6,546.8 Mb	3,294.9 Mb	652.5 Mb	231.75 Mb	36 Mb
$\sigma = 16$	3,637.12 Mb	1,830.5 Mb	362.6 Mb	128.7 Mb	20 Mb
$\sigma = 32$	2,182.27 Mb	1,098.3 Mb	217.1 Mb	77.2 Mb	12 Mb

In Table 10, we show the space overhead. We notice that no matter whether the multiple character checking is used, the space consumed is almost the same for both of them. It is because by the multiple character checking we need only to maintain five counters C_i $(i = 1, \ldots, 5)$ for the children of a node in an M-tree to record how many times each character is encountered during a search of a segment in L. The values of these counters can be dynamically changed for each node created in a M-tree and no extra space is required. Therefore, in Table 10, we only show the usage of space when the multi-character checking is utilized, which is almost the same as when the multi-character checking not used.

From Table 10, we can also see that the space overhead is roughly linearly decreased as σ is increased. In addition, when σ is equal to 0, the BWT array for Rat (Rnor_6.0) cannot be generated due to its huge volume.

8. 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 $\bar{s}$ of target string s to $BWT(\bar{s})$ and use the mismatch information over a pattern string r to speed up the computation. Its time complexity is bounded by $O(kn' + n + m \log m)$, 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 $O((1 + \frac{1}{|\Sigma|})^{k+1})$. 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 two important problems: the string matching with k errors and the string matching with don't-care symbols.

For 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.

For the string matching with don't-care symbols, we need to distinguish between two cases. For the case that only the pattern contains don't-care symbols, it is possible to modify the method discussed in this paper to develop an algorithm to handle the problem efficiently. However, for the case that both the pattern and target contain don't-care symbols, it is not clear whether the BWT technique can be effectively utilized.

Acknowledgement

The authors are grateful to the anonymous reviewers for their very detailed comments, which enable us to greatly improve the manuscript.

Appendix A. Notations

In this appendix, we summarize all the symbols and notations used throughout the paper (see Table 11).

Table 11
symbols and notations.

r	a pattern string
$r[i]$	i-th character in r
$r[i..j]$	substring from i-th to j-th position in r
m	$m = \|r\|$
s	a target string
n	$n = \|s\|$
Σ	an alphabet
x, y, z, a, b,c	characters in a string
$\bar{s}$	reverse of s
$L = BWT(s)$	BWT array for s
F	an array, in which the equal characters are clustered in alphabetic order
$rk_L(e)$	rank of an element in L
$rk_F(e)$	rank of an element in F
T	search tree, which is constructed when searching s to find all occurrences of r in s
P	a path in T
$P[i]$	the ith node on P
D	match tree, whose nodes are created to represent mismatching information
u, v, w	nodes in T
$\pi = \langle x, [\alpha, \beta]\rangle$	a pair representing a segment of F
L_π	a segment in L, corresponding to π
$x[]$	an array created for $x \in \Sigma$
$A_x[]$	an compact version of $x[]$
search(z, π)	a procedure to find the first and last appearance of z within a segment corresponding to π
search(π)	an improved version of *search*(z, π)
Y_a, Y_c, Y_g, Y_t	starting position of a-segment, c-segment, g-segment, t-segment in F
Z_a, Z_c, Z_g, Z_t	ending position of a-segment, c-segment, g-segment, t-segment in F
sUP( )	a procedure to search L from a certain position upwards to a higher position
sDown( )	a procedure to search L from a certain position downwards to a lower position
Q	a queue used to search a subtree of D in the breadth-first manner
R_{ij}	an array storing the mismatching information between $r[i..m]$ and $r[j..m]$

Appendix B. On the generation of $D[u]$

Continued with Section 4.4. Let $w = \langle z, f\rangle$ be a node in $D[u']$, $\gamma = R_{ij}[l]$ for some l, and h be the number of mismatching nodes on the path from the root to the node to be created in $D[u]$. Then, in the execution of *node-generation*$(w, \gamma, h, i, j, R_{ij})$, for w, a node x will be created in $D[u]$ if $f \le i + val - 1$, where $val = R_{ij}[l]$. But if $w = \langle 0, \infty\rangle$ or $f > i + val - 1$, we will create a path $x_1 \to x_2 \to \cdots \to x_a$ (for some $a \ge 1$), rather than a single node (see step 4 and 5 in *node-generation*( )). In Fig. 22(a), we illustrate this difference (in which we assume that for node w in $D[u']$ a node x in $D[u]$ while for w' a path is created).

Now we consider a right sibling w'' of w' and assume that for w'' also a path $x_1 \to x_2 \to \cdots \to x_b \to y$ $(a > b \ge 1, y \ne x_{b+1})$ will be created. Then, for efficiency, we should organize these two paths into a subtree in $D[u]$ as illustrated in Fig. 22(b).

In general, let $w_1, \ldots, w_g$ be the children of w in $D[u']$. Let $w_b = \langle z_b, f_b\rangle$ $(b = 1, \ldots, g)$. Without loss of generality, assume that $f_1 \le f_2 \le \cdots \le f_g$, and we can find an integer c such that $f_1 \le \cdots \le f_c \le i + val - 1 < f_{c+1} \le \cdots \le f_g$. Thus,

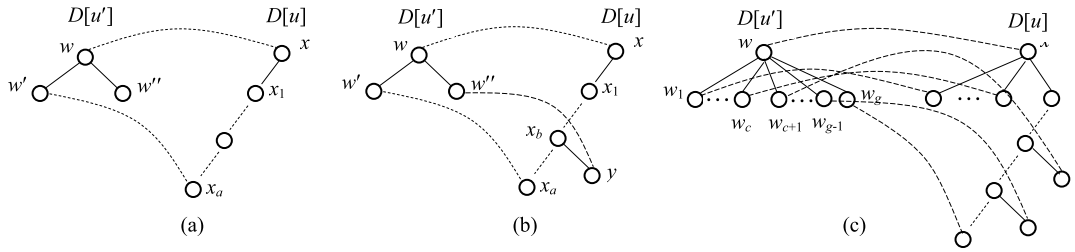

Fig. 22. Illustration for node generation.

what will be done is to create a single node in $D[u]$ for each $w_q \in \{w_1, \ldots, w_c\}$, but generate a subtree for $w_{c+1}, \ldots, w_g$ altogether, as illustrated in Fig. 22(c).

According to the above discussion, we rewrite the algorithm for the node generation in $D[u]$ as follows.

Algorithm *genD*(u').

begin
1. generate node u according u';
2. *enqueue*(Q,(u', $R_{ij}[1]$, h));
3. **while** Q is not empty **do** {
4. $(w, \gamma, h') :=$ *dequeue*(Q); let $\gamma = R_{ij}[l]$ for some l;
5. let $w_1, \ldots, w_g$ be the children of w; let $w_b = \langle z_b, f_b \rangle$ $(b = 1, \ldots, g)$;
6. assume that $f_1 \leq \ldots \leq f_c \leq i + val - 1 < f_{c+1} \leq \ldots \leq f_g$ for some c;
7. **for** each $w' \in \{w_1, \ldots, w_c\}$ **do** {
8. create a node y in $D[u]$; set y be a child of x, where x is the node created for w;
9. if $h < k + 1$, *enqueue*(w', $R_{ij}[l]$, $h + 1$) into Q;
10. }
11. create a subtree as illustrated in Fig. 22(c) for $w_{c+1}, \ldots, w_g$ with leaf nodes $y_{c+1}, \ldots, y_g$;
12. **for** each w_q with $f_q \neq \infty$ in $\{w_1, \ldots, w_c\}$ **do** {
13. *enqueue*(w_q, $R_{ij}[l' + 1]$, $h + l' - l + 1$), where l' is a largest integer $\leq k + h + l$ such that $f_g > i + R_{ij}[l'] - 1$;
14. }

end

The main difference of the above algorithm from *node-generation*( ) given in 4.4 consists in that the paths created by executing step 4 and 5 in *node-generation*( ) are separated, but in this algorithm all such paths are organized into a subtree.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 23 (1) (June 1975) 333–340.
[2] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, J. Algorithms 50 (2) (Feb. 2004) 257–275.
[3] A. Apostolico, R. Giancarlo, The Boyer–Moore–Galil string searching strategies revisited, SIAM J. Comput. 15 (1) (Feb. 1986) 98–105.
[4] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, in: N.J. Belkin, C.J. van Rijsbergen (Eds.), SIGIR 89, Proc. 12th Annual Intl. ACM Conf. on Research and Development in Information Retrieval, 1989, pp. 168–175.
[5] R.A. Baeza-Yates, G.H. Gonnet, A new approach in text searching, Commun. ACM 35 (10) (Oct. 1992) 74–82.
[6] R.A. Baeza-Yates, C.H. Perleberg, Fast and practical approximate string matching, in: A. Apostolico, M. Crocchemore, Z. Galil, U. Manber (Eds.), Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 644, Springer-Verlag, Berlin, 1992, pp. 185–192.
[7] S. Bauer, M.H. Schulz, P.N. Robinson, gsuffix, http://gsuffix.Sourceforge.net/, 2014.
[8] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (Oct. 1977) 762–772.
[9] M. Burrows, D.J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, 1994.
[10] W.L. Chang, J. Lampe, Theoretical and empirical comparisons of approximate string matching algorithms, in: A. Apostolico, M. Crocchemore, Z. Galil, U. Manber (Eds.), Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 644, Springer-Verlag, Berlin, 1992, pp. 175–184.
[11] M. Crochemore, et al., Fast practical multi-pattern matching, Inform. Process. Lett. 71 (1999) 107–113.
[12] Y. Chen, Y. Wu, On the massive string matching problem, in: Proc. ICNC–FSKD 2016, IEEE, Changsha, China, August 2016, pp. 13–15.
[13] Y. Chen, Y. Wu, Mismatching Trees and BWT Arrays: a New Way for String Matching with k-Mismatches, in: Proc. ICDE2017, April 19–22, 2017, IEEE, San Diego, USA, pp. 339–410.
[14] Y. Chen, Y. Wu, Searching BWT against pattern matching machine to find multiple string matches, in: Proc. 9th Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery, IEEE, 2017, pp. 167–176.
[15] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: STOC'04, 2004, pp. 91–100.
[16] L. Colussi, Z. Galil, R. Giancarlo, On the exact complexity of string matching, in: Proc. 31st Annual IEEE Symposium of Foundation of Computer Science, vol. 1, 1990, pp. 135–144.
[17] B. Commentz-Walter, A string matching algorithm fast on the average, in: Proc. 6th Colloquium on Automata, Languages and Programming, July 16–20, 1979, pp. 118–132.
[18] A. Ehrenfeucht, D. Haussler, A new distance metric on strings computable in linear time, Discrete Appl. Math. 20 (1988) 191–203.
[19] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 390–398.
[20] Z. Galil, On improving the worst case running time of the Boyer–Moore string searching algorithm, Commun. ACM 22 (9) (1977) 505–508.
[21] Z. Galil, R. Giancarlo, Improved string matching with k mismatches, ACM SIGACT News 17 (4) (1986) 52b–54.

[22] M.C. Harrison, Implementation of the substring test by hashing, Commun. ACM 14 (12) (1971) 777–779.
[23] R.L. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (2) (March 1987) 249–260.
[24] D.E. Knuth, The Art of Computer Programming, vol. 3, Addison–Wesley Publish Com., Massachusetts, 1975.
[25] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (June 1977) 323–350.
[26] lab website, http://home.cc.umanitoba.ca/~xiej/, 2014.
[27] G.M. Landau, U. Vishkin, Efficient string matching in the presence of errors, in: Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 126–136.
[28] G.M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theoret. Comput. Sci. 43 (1986) 239–249.
[29] T. Lecroq, A variation on the Boyer–Moore algorithm, Theoret. Comput. Sci. 92 (1) (Jan. 1992) 119–144.
[30] H. Li, et al., Mapping short DNA sequencing reads and calling variants using mapping quality scores, Genome Res. 18 (2008) 1851–1858.
[31] R. Li, et al., SOAP: short oligonucleotide alignment program, Bioinformatics 24 (2008) 713–714.
[32] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[33] H. Li, R. Durbin, Fast and accurate long-read alignment with Burrows–Wheeler transform, Bioinformatics 26 (5) (2010) 589–595.
[34] H. Li, Homer, A survey of sequence alignment algorithms for next-generation sequencing, Brief. Bioinform. 11 (5) (2010) 473–483, https://doi.org/10.1093/bib/bbq015.
[35] H. Li, wgsim: a small tool for simulating sequence reads from a reference genome, https://github.com/lh3/wgsim/, 2014.
[36] H. Lin, et al., ZOOM! Zillions of oligos mapped, Bioinformatics 24 (2008) 2431–2437.
[37] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, in: Proc. the 1st Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, PA, 1990, pp. 319–327.
[38] U. Manber, R.A. Baeza-Yates, An algorithm for string matching with a sequence of don't cares, Inform. Process. Lett. 37 (Feb. 1991) 133–136.
[39] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (April 1976) 262–272.
[40] G. Navarro, M. Raffinot, Pattern Matching in Strings, Cambridge University Press, 2002.
[41] M. Nicolas, S. Rajasekarian, On string matching with k mismatches, https://arxiv.org/pdf/1307.1406, 2013.
[42] R.Y. Pinter, Efficient string matching with don't' care patterns, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series, vol. F12, Springer-Verlag, Berlin, 1985, pp. 11–29.
[43] M. Schatz, Cloudburst: highly sensitive read mapping with mapreduce, Bioinformatics 25 (2009) 1363–1369.
[44] J. Seward, bzip2 and libbzip2, version 1.0. 5: a program and library for data compression, http://www.bzip.org, 2007.
[45] A.D. Smith, et al., Using quality scores and longer reads improves accuracy of Solexa read mapping, BMC Bioinform. 9 (2008) 128.
[46] J. Tarhio, E. Ukkonen, Boyer–Moore approach to approximate string matching, in: J.R. Gilbert, R. Karlsson (Eds.), SWAT 90: Proc. 2nd Scandinavian Workshop on Algorithm Theory, in: Lecture Notes in Computer Science, vol. 447, Springer-Verlag, Berlin, 1990, pp. 348–359.
[47] J. Tarhio, E. Ukkonen, Approximate Boyer–Moore string matching, SIAM J. Comput. 22 (2) (1990) 243–260.
[48] E. Ukkonen, Approximate string-matching with q-grams and maximal matches, Theoret. Comput. Sci. 92 (1992) 191–211.
[49] P. Weiner, Linear pattern matching algorithm, in: Proc. 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[50] W. Hon, et al., A space and time efficient algorithm for constructing compressed suffix arrays, Algorithmica 48 (2007) 23–36.
[51] S. Wu, U. Manber, A Fast Algorithm for Multi-Pattern Searching, Technical Report TR-94-17, Dept. Computer Science, Chung-Cheng University, 1994.
[52] L. Salmela, J. Tarhio, J. Kytöjoki, Multi-pattern string matching with q-grams, ACM J. Exp. Algorithmics 11 (2006).
[53] G. Nong, S. Zhang, W.H. Chan, Two efficient algorithms for linear time suffix array construction, IEEE Trans. Comput. 60 (10) (2011) 1471–1484.
[54] G. Nong, Practical linear-time O(1)-workspace suffix sorting for constant alphabets, ACM Trans. Inf. Syst 31 (3) (2013) 15.
[55] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013.
[56] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th SODA, 2003, pp. 841–850.
[57] A. Bowe, Multiary Wavelet Trees in Practice, Master Thesis, School of Computer Science and Information Technology, RMIT University, Melbourne, Australia, 2010.