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

Yangjun Chen[1], Yujia Wu[2]
Dept. Applied Computer Science
University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]wyj1128@yahoo.com

***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 to transform *s* to a BWT-array as index, denoted as *BWT*(*s*), and 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*). Extensive experiments have been conducted, which show that our method for this problem is promising.**

Key words: *String matching; DNA sequences; tries; BWT-transformation*

I. Introduction

By the string matching with k mismatches, we mean a problem to find all the occurrences of a pattern string r in a target string s with each occurrence having up to k positions different between r and s. This problem is important for DNA databases to support the biological research, where we need to locate all the appearances of a read (a short DNA sequence) in a genome (a very long NDA sequence) for disease diagnosis or some other purposes. Due to polymorphisms or mutations among individuals or even sequencing errors, the read may disagree in some positions at any of its occurrences in the genome.

As an example, consider a target s = *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.

```
    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 string matching problem.

This topic has received much attention in the research community and many efficient algorithms have been proposed, such as [2, 14, 21, 29, 34, 43]. Among them, [21] and [29] 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 *mismatch information* among substrings of r is used to speed up the working process. The methods discussed in [2] and [43] 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 [14, 34] are index-based. By the method discussed in [14], 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 [34], s is transformed to a *BWT*-array (denoted *BWT*(s)) as an index [30]. In comparison with *suffix trees*, *BWT*(s) uses much less space [19]. However, the time complexity of [34] is bounded by O($mn' + n$), where n' is the number of leaf nodes of a tree (forest) produced during the search of *BWT*(s). Again, this time requirement can also be much worse than the best on-line algorithm for large patterns. Thus, simply indexing s is not always helpful for k mismatches. The reason for this is that in both the above index-based methods neither mismatch information nor periodicity within r is employed, leading to a lot of redundancy, which shadows the benefits brought by indexes. However, to use such information efficiently and effectively in an indexing environment is very challenging since in this case s will no longer be scanned character by character and the auxiliary information extracted from r cannot be simply integrated into an index searching process.

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

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

- An efficient method to calculate the mismatches between $r[i\,..\,m]$ and $r[j\,..\,m]$ ($i, j \in \{1, \ldots, m\}$, $i \neq j$), 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 (forest) structure D to store the mismatches between r and different segments of s. In D, each node v stores an integer i, indicating that there are some positions $i_1, i_2, \ldots, i_l$ 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 $O(kn' + n)$. Our experiment shows that $n' \ll n$.

The remainder of the paper is organized as follows. In Section II, we review the related work. In Section III, we briefly describe how a BWT-transformation can be used to speed up string matches. In Section IV, we discuss our algorithm to find all the occurrences of r in s, but up to k mismatches. Section V is devoted to the test results. Finally, we conclude with a short summary and a brief discussion on the future work in Section VI.

II. Related Work

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

- *Exact matching*

The first interesting algorithm for this problem is the famous *Knuth-Morris-Pratt*'s algorithm [26], 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 $O(m + n)$. The *Boyer-Moore*'s approach [9] works a little bit better than the *Knuth-Morris-Pratt*'s. In addition to the next-table, a *skip-table* (also for r) is kept. 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. Although these two algorithms have never been used in practice [42], they sparked a series of subsequent research on this problem, and improved by different researchers in different ways, such as the algorithms discussed in [1, 31]. However, the worst-case time complexity remains unchanged. In addition, the idea of the 'shift information' has also been adopted by Aho and Corasick [1] for the *multiple-string* matching, by which s is searched for an occurrence of any one of a set of x patterns: $\{r_1, r_2, \ldots, r_x\}$. Their algorithm needs only $O(\sum_{i=1}^{x} |r_i| + n)$ time.

In situations where a fixed string s is to be searched repeatedly, it is worthwhile constructing an index over s, such as suffix trees [41, 51], suffix arrays [39], and more recently the BWT-transformation [10, 12, 13, 19, 35, 46]. A suffix tree is in fact a *trie* structure [25] over all the suffixes of s; and by using the Weiner's algorithm it can be built in $O(n)$ time [41]. However, in comparison with the BWT-transformation, a suffix tree needs much more space. Especially, for DNA sequences the BWT-transformation works highly efficiently due to the small alphabet Σ of DNA strings. By the BWT, the smaller Σ is, the less space will be occupied by the corresponding indexes. According to a survey done by Li and Homer [36] on sequence alignment algorithms for next-generation sequencing, the average space required for each character is 12 - 17 bytes for suffix trees while only 0.5 - 2 bytes for the BWT. The experiments reported in [12] also confirm this distinction. For example, the file size of chromosome 1 of human is 270 Mb. But its suffix tree is of 26 Gb in size while its BWT needs only 390 Mb – 1 Gb for different compression rates of auxiliary arrays, completely handleable on PC or laptop machines.

By the hash-table-based algorithms [22, 24], short substrings called 'seeds' will be first extracted from a pattern r and a *signature* (a bit string) for each of them will be created. The search of a target string s is similar to that of the Brute Force searching, but rather than directly comparing the pattern at successive positions in s, their respective signatures are compared. Then, stick each matching seed together to form a complete alignment. Its expected time is $O(m + n)$, but in the worst case, which is extremely unlikely, it takes $O(mn)$ time. The hash technique has also been extensively used in the DNA sequence research [23, 32, 33, 38, 45]. However, almost all experiments show that they are generally inferior to the suffix tree and the BWT index in both running time and space requirements.

- *Inexact matching*

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

***k* mismatches** When the distance function is the *Hamming* distance, the problem is known as the string matching with k *mismatches* [4]. 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, 28, 29, 43, 48, 49]. They are all on-line algorithms. Except those discussed in [2, 21, 29, 43], all the other methods have the worst-case time complexity $O(mn)$. The methods discussed in [21] and [29], however, require only $O(kn + m\log m)$ time, by which the mismatch arrays for r are precomputed and exploited to speed up the search of s. The methods discussed in [2, 43] 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 [34] is index-based, by which s is transformed to a BWT-array, used as an index. Its time complexity is bounded by $O(mn' + n)$, where n' is the number of leaf nodes of a tree produced during the search of $BWT(s)$. If m is large, it can be worse than all those on-line methods discussed in [2, 21, 29, 43]. Another index-based method is based on a brute-force searching of suffix trees [14]. 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.

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

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

where $d_{i,j}$ represents the distance between $r[1 .. i]$ and $s'[1 .. j]$, r_i (s_j') the ith character in r (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, 11, 18, 50]. They are all some kinds of variants of the *dynamic programming* paradigm [17] with the worst-case time complexity bounded by $O(mn)$. However, by the algorithm discussed in [11], 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 [44]. 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 [40].

III. BWT-Transformation

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

A. 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, we have $\$ < a < c < g < t$. As an example, consider $s = acagaca\$$. We can rotate s consecutively to create eight different strings, and put them in a matrix as illustrated in Fig. 1(a).

		rk_F	F	L	rk_L
$a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$$	$\$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4$	–	\$	a_4	1
$c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1$	$a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2$	1	a_4	c_2	1
$a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1$	$a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1$	2	a_3	g_1	1
$g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2$	$a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$$	3	a_1	\$	–
$a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1$	$a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1$	4	a_2	c_1	2
$c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3$	$c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3$	1	c_2	a_3	2
$a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2$	$c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1$	2	c_1	a_1	3
$\$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4$	$g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2$	1	g_1	a_2	4
(a)	(b)		(c)		

Fig. 1: Rotation of a string

In Fig. 1(a), for ease of explanation, the position of a character in s is represented by its subscript. (That is, we rewrite s as $a_1c_1a_2g_1a_3c_2a_4\$$.) For example, a_2 representing the second appearance of a in s; and c_1 the first appearance of c in s. In the same way, we can check all the other appearances of different characters.

Now we sort the rows of the matrix alphabetically, and get another matrix, as demonstrated in Fig. 1(b), which is called the *Burrow-Wheeler Matrix* [7, 15, 27] and denoted as $BWM(s)$. Especially, the last column L of $BWM(s)$, read from top to bottom, is called the *BWT*-transformation (or the *BWT*-array) and denoted as $BWT(s)$. So for $s = acagaca\$$, we have $BWT(s) = acg\$caaa$. The first column is referred to as F.

When ranking the elements x in both F and L in such a way that if x is the ith appearance of a certain character it will be assigned i, the same element will get the same number in the two columns. For example, in F the rank of a_4, denoted as $rk_F(a_4)$, is 1 (showing that a_4 is the first appearance of a in F). Its rank in L, $rk_L(a_4)$ is also 1. We can check all the other elements and find that this property, called the *rank correspondence*, holds for all the elements. That is, for any element e in s, we always have

$$rk_F(e) = rk_L(e) \qquad (1)$$

According 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 $s = acagaca\$$.

First, we notice that we can store F as $|\Sigma| + 1$ intervals, such as $F_\$ = F[1 .. 1]$, $F_A = F[2 .. 5]$, $F_C = F[6 .. 7]$, $F_G = F[8 .. 8]$, and $F_T = \Phi$ for the above example (see Fig. 1(c).) We can also represent a segment within an F_x with $x \in \Sigma$ as a pair π of the form $<x, [\alpha, \beta]>$, where $\alpha \le \beta$ are two ranks of x. Thus, we have $F_A = F[2 .. 5] = <a, [1, 4]>$, $F_C = F[6 .. 7] = <c, [1, 2]>$, and $F_G = F[8 .. 8] = <g, [1, 1]>$. In addition, we can use L_π to represent a range in L corresponding to a pair π. For example, in Fig. 1(c), $L_{<a, [1, 4]>} = L[2 .. 5]$, $L_{<c, [1, 2]>} = L[6 .. 7]$. $L_{<a, [2, 3]>} = L[3 .. 4]$, and so on.

We will also use a procedure $search(z, \pi)$ to search L_π to find the first and the last rank of z (denoted as α' and β', respectively) within L_π, and return $<z, [\alpha', \beta']>$ as the result:

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

Then, 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.

Step 1: Check $r[3] = a$ in the pattern string r, and then figure out $F_A = F[2 .. 5] = <a, [1, 4]>$.

Step 2: Check $r[2] = c$. Call $search(c, L_{<a, [1, 4]>})$. It will search $L_{<a, [1, 4]>} = L[2 .. 5]$ to find a range bounded by the first and last rank of c. Concretely, we will find $rk_L(c_2) = 1$ and $rk_L(c_1) = 2$. So, $search(c, L_{<a, [1, 4]>})$ returns $<c, [1, 2]>$. It is $F[6 .. 7]$.

Step 3: Check $r[3] = a$. Call $search(a, L_{<c, [1, 2]>})$. Notice that $L_{<c, [1, 2]>} = L[6 .. 7]$. So, $search(a, L_{<c, [1, 2]>})$ returns $<a, [2, 3]>$.

It is $F[3 .. 4]$. Since now we have exhausted all the characters in r and $F[3 .. 4]$ contains only two elements, two occurrences of r in s are found. They are a_1 and a_3 in s, respectively.

The above working process can be represented as a sequence of three pairs: $<a, [1, 4]>$, $<c, [1, 2]>$, $<a, [2, 3]>$. In general, for $\bar{r} = c_1 \ldots c_m$, its search against $BWT(s)$ can always be represented as a sequence:

$<x_1, [\alpha_1, \beta_1]>, \ldots, <x_m, [\alpha_m, \beta_m]>$,

where $<x_1, [\alpha_1, \beta_1]> = F_{x_1}$, and $<x_i, [\alpha_i, \beta_i]> = search(x_i, L_{<x_{i-1}, [\alpha_{i-1}, \beta_{i-1}]>})$ for $1 < i \leq m$. 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 [29], by which $|\Sigma|$ arrays each for a character $x \in \Sigma$ are arranged such that $A_x[k]$ (the kth entry in the array for x) is the number of appearances of x within $L[1 .. k]$ (i.e, the number of x-characters appearing befor $L[k + 1]$.) (See Fig. 2(a) for illustration.)

j	F	L	$	A_a	A_c	A_g	A_t
1	$	a_4	0	1	0	0	0
2	a_4	c_2	0	1	1	0	0
3	a_3	g_1	0	1	1	1	0
4	a_1	$	1	1	1	1	0
5	a_2	c_1	1	1	2	1	0
6	c_2	a_3	1	2	2	1	0
7	c_1	a_1	1	3	2	1	0
8	g_1	a_2	1	4	2	1	0

(a)

i	A_a	A_c	A_g	A_t
0	0	0	0	0
1	1	1	1	0
2	4	2	1	0

For each 4 values in L, a *rankAll* value is stored.

(b)

Fig. 2: Illustration for *rankAlls*

Now, instead of scanning a certain segment $L[i .. j]$ $(i \leq j)$ to find a subrange for a certain $x \in \Sigma$, we can simply look up the array for x to see whether $A_x[i - 1] = A_x[j]$. If it is the case, then x does not occur in $L[i .. j]$. Otherwise, $[A_x[i - 1] + 1, A_x[j]]$ should be the range to be found.

For instance, to find the subrange for g within $L[6 .. 7]$, we will first check whether $A_g[6 - 1] = A_g[7]$. Since $A_g[6 - 1] = A_g[5] = A_g[7] = 1$, we know that g does not appear in $L[6 .. 7]$. However, since $A_c[2 - 1] \neq A_c[5]$, we immediately get the subrange for c within $L[2 .. 5]$: $[A_c[2 - 1] + 1, A_c[5]] = [1, 2]$.

We notice that the column for $ needn't be stored since it will never be searched. We can also create *rankAll*s only for part of the elements to reduce the space overhead, but at cost of some more searches. See Fig. 2(b) for illustration.

B. Construction of BWT arrays

A BWT-array can be constructed in terms of a relationship to the *suffix arrays* [10, 19, 30, 46].

As mentioned above, a string $s = a_1a_1 \ldots a_n$ 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, $s[i.. j] = a_i \ldots a_j$ a substring and $s[i .. n]$ a suffix of s. Suffix array H of s is a permutation of the integers 1, ..., n such that $H[i]$ is the start position of the ith smallest suffix. The relationship between H and the BWT-array L can be determined by the following formulas:

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

Since a suffix array can be generated in $O(n)$ time [52], L can then be created in a linear time. However, most algorithms for constructing suffix arrays require at least $O(n\log n)$ bits of working space, which is prohibitively high and amounts to 12 GB for the human genome. Recently, Hon *et al* [52] proposed a space-economical algorithm that uses n bits of working space and requires only < 1 GB memory at peak time for constructing L of the human genome. We use this for our purpose.

IV. STRING MATCHING WITH K MISMATCHES

A. Basic working process

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

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

As an example, consider the case where r = *tcaca*, s = *acagaca* 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.

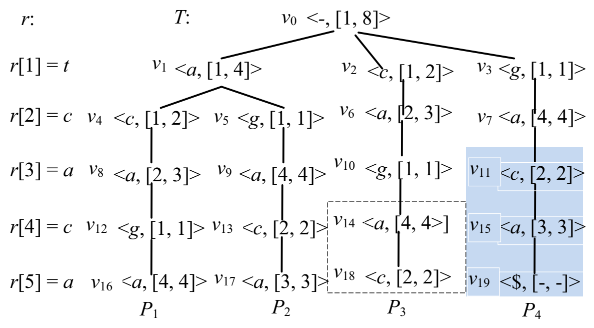


Fig. 3: Search for string matching with 2 mismatches

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 $P_1 = v_1 \rightarrow v_4 \rightarrow v_8 \rightarrow v_{12} \rightarrow v_{16}$ and $P_2 = v_1 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{13} \rightarrow v_{16}$, we will find two occurrences of r with 2 mismatches: $s[1 .. 5]$ (= $a_1c_1a_2g_1a_3$) and $s[3 .. 7]$ (= $a_2g_1a_3c_2a_4$) while by either $P_3 = v_2 \rightarrow v_6 \rightarrow v_{10} \rightarrow v_{14} \rightarrow v_{18}$ or $P_4 = v_3 \rightarrow v_7 \rightarrow v_{11} \rightarrow v_{15} \rightarrow v_{19}$ no string matching with at most 2 mismatches can be found.

A node $<x, [\alpha, \beta]>$ in such a tree is called a *matching node* if it corresponds to a same character in r. Otherwise, it is called a *mismatching node*. For example, node $v_4 = <c, [1, 2]>$ is a matching node since it corresponds to $r[2] = c$ while $v_1 = <a, [1, 4]>$ is a mismatching node since it corresponds to $r[1] = t$.

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 ∞ onwards, i.e.,

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

We call B_l a *mismatch array*. For instance, in Fig. 3, for P_1, we have $B_1 = [1, 4, \infty]$, indicating that at position 1, we have the first mismatch $P_1[1] = a \neq r[1] = t$ and at position 4 we have the second mismatch $P_1[4] = g \neq r[4] = a$. For the same reason, we have $B_2 = [1, 2, \infty]$, $B_3 = [1, 2, 3]$, and $B_4 = [1, 2, 3]$.

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 an 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_{16}, r is exhausted and the current value of B is $[1, 4, \infty]$. We will store B in B_1 (the array associated with the leaf node v_{16} of P_1) and 'backtrack' to v_1 to explore a new path. At the same time, all those values in B, which are set after v_1, will be reset to ∞, i.e., B will be changed to $[1, \infty, \infty]$.

Now we consider another path P_3. The search along P_3 will stop at v_{10} since when reaching it B becomes full ($B = [1, 2, 3]$). Therefore, the search will not be continued, and v_{14}, v_{18} will not be created.

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

In fact, it is the main process discussed in [34]. The only difference is that in [34] a simple heuristics is used, which precomputes, for each position i in r, the number $\sigma(i)$ of consecutive, disjoint substrings in $r[i .. m]$, which do not appear in s. For example, in Fig. 3, $\sigma(1) = 2$ since in $r[1 .. 5] = tcaca$ both $r[1 .. 1] = t$ and $r[2 .. 4] = cac$ do not occur in $s = acagaca$. But $\sigma(3) = 0$ since any substring in $r[1 .. 3] = aca$ 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 $k - l < \sigma(i)$, 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 $BWT(s)$ [33]. However, the theoretic time complexity of this method is still $O(mn')$. Even in practice, this heuristics is not quite helpful since $\sigma(i)$ 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 $\sigma(3) = 0$, the search along P_4 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 a next step.

B. Mismatch information

Searching S-trees in an improvement over sanning strings, but it often happens that there are repetitive traversals of similar subtrees due to the multiple appearnces of a same pair. 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 $<c, [1, 2]>$ (v_4 in Fig. 3) is compared to $r[2]$ while its second appearance (v_2) is to $r[1]$. Hence, we cannot use the result computed for v_4 (when $<c, [1, 2]>$ is first met) as the result for v_2.

However, if we have stored the mismatch information R between substrings of r, like $r[2 .. 4]$ and $r[1 .. 3]$, in some way, the mismatches along P_3 can be 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 $R_{ij}[l] = x$ ($\neq \infty$) then $r[i + x - 1] \neq r[j + x - 1]$ or one of them does not exist, and it is the lth mismatch between them.

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

For this reason, we will precompute only part of R, instead of R_{ij} for all $i, j \in \{1, \ldots, m\}$. Specifically, $R_{12}, \ldots, R_{1m}$ for r will be pre-constructed in a way as described in [29], giving the positions of the mismatches between the pattern and itself at various relative shifts. That is, each R_{1i} ($2 \leq i \leq m$) contains the positions within r of the first $2k + 1$ mismatches between the substring $r[1 .. m - i]$ and $r[i + 1 .. m]$, i.e., the overlapping portions of the two copies of pattern r for a relative shift of i. Thus, if $R_{1i}[j] = x$, then $r[x] \neq r[i + x - 1]$ or one of them does not exist, which is the jth mismatch between $r[1 .. m - i]$ and $r[i + 1 .. m]$. (See Fig. 4(a) for illustration.)

In Fig. 4(b), we show a pattern $r_1 = tcacg$ and all the possible right-to-left shifts: $r_2 = r[2 .. 5] = cacg$, $r_3 = r[3 .. 5] = acg$, and so on. In Fig. 4(c), we give $R_{12}, \ldots, R_{15}$ for r_1. In an R_{1i}, 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 ∞ onwards, i.e.,

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

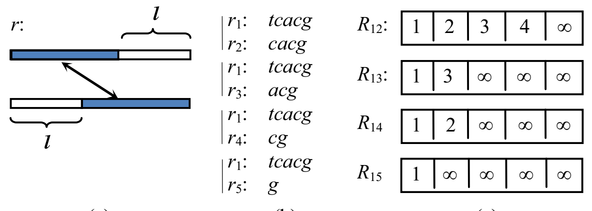


Fig. 4: Illustration for table R

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

Using the algorithm of [29], R_{12}, …, R_{1m} can be constructed in O($m\log m$) time, just before the process for the string matching gets started. In addition, we need to keep $2k + 1$, rather than $k + 1$ mismatches in each R_{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 [29] 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 mismatch information for some new paths (to avoid exploring them by using *search*( ).) (A node $<x, [\alpha, \beta]>$ is said to be the same as another node $<x', [\alpha', \beta']>$ if $x = x'$, $\alpha = \alpha'$ and $\beta = \beta'$.) For this purpose, we design a general algorithm to create R_{ij} efficiently.

- Let 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 $\omega_1[j] \neq \omega_1[j]$, 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. $p := 1; q := 1; l := 1;$
2. If $A_2[q] < A_1[p]$, then {$A[l] := A_2[q]; q := q + 1; l := l + 1;$}
3. If $A_1[p] < A_2[q]$, then {$A[l] := A_1[p]; p := p + 1; l := l + 1;$}
4. If $A_1[p] = A_2[q]$, then {if $\omega_1[p] \neq \omega_2[q]$, then {$A[l] := q; l := l + 1;$} $p := p + 1; q := q + 1;$}
5. 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.)
6. Otherwise, go to (2).
```

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

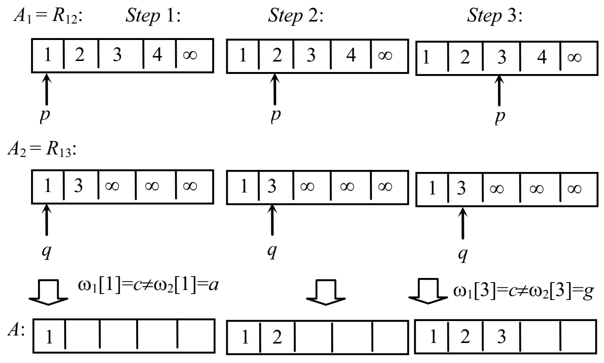


Fig. 5: Illustration for *merge*( )

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 $A_1[1] = A_2[1] = 1$, we will compare $\omega_1[1]$ and $\omega_2[1]$, and find that $\omega_1[1] = c \neq \omega_2[1] = a$. Thus, $A[1]$ is set to be 1. $p := p + 1 = 2$, $q := q + 1 = 2$, $l := l + 1 = 2$.

In step 2: $p = 2$, $q = 2$, $l = 2$. we compare $A_1[2]$ and $A_2[2]$. Since $A_1[2] = 2 < A_2[2] = 3$, $A[2]$ is set to be 2. $p := p + 1 = 3$, $q := 2$, $l := l + 1 = 3$.

In step 3: $p = 3$, $q = 2$, $l = 3$. We compare $A_1[3]$ and $A_2[2]$, and find that $A_1[3] = A_2[2] = 3$. So, we need to compare $\omega_1[3]$ and $\omega_2[3]$. Since $\omega_1[3] = c \neq \omega_2[3] = g$, $A[3]$ is set to be 3. $p := p + 1 = 4$, $q := 3$, $l := l + 1 = 4$.

In a next step, we have $p = 4$, $q = 3$, $l = 4$. We will compare $A_1[4]$ and $A_2[3]$. Since $A_1[4] = 4 < A_2[3] = \infty$, we set $A[4]$ to 4.

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

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, (2) 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 $\omega[j] = \omega_1[j]$, (2) is correct.

If (ii) holds, but (i) not, (3) will be executed. Since in this case, we have $\omega[j] \neq \omega_1[j]$ and $\omega[j] = \omega_2[j]$, (3) 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, (4) 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. □

C. *Main idea: mismatch 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 *A*. 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 created a set of mismatch arrays for all the sub-paths in $T[u]$, which start at *u* and end at a leaf node, by doing two steps explained below.

- For each path P_i going through *v*, figure out a sub-array of B_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 $B_1 = [1, 4, \infty]$, we have $B_1^1 = [1, 4, \infty]$, $B_1^2 = [3, \infty]$, $B_1^3 = [2, \infty]$, $B_1^4 = [1, \infty]$, and $B_1^5 = [\infty]$.)
- 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 $m_l = |P_i|$.

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

As an example, consider v_2 (in Fig. 3, labeled $<c, [1, 2]>$ and compared to $r[1] = t$), which is the same as v_4 (compared to $r[2] = c$). By executing $mi\text{-}creation(v_2, v_4, 1, 2)$, the following operations will be performed, to avoid repeated access of the corresponding subtree (i.e., part of P_3 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_3:

   $B_1 = [1, 4, \infty]$, $B_1^2 = [3, \infty]$. $P_1[2 .. 5] = caga$, $r[1 .. 4]) = caca$.

   $merge(B_1^2, R_{21}, P_1[2 .. 5], r[1 .. 4]) = [1, 2, 3, 4]$.

In general, we will distinguish between two cases:

(i) $i < j$. This case can be illustrated in Fig. 6(b). In this case, the mismatch 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 mismatch 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 mismatch 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. □

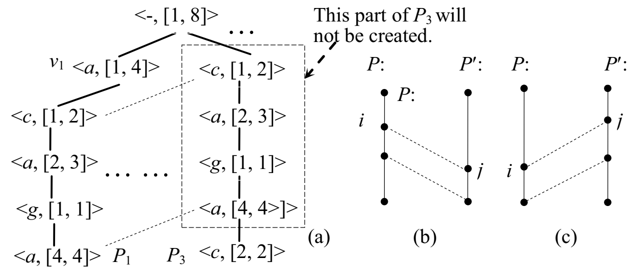


Fig. 6: Illustration for derivation of mismatch information

D. *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 maintain a general tree structure, called a *mismatch tree*, to store the mismatch information for all the created paths. First, we define two simple concepts related to *S*-trees.

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

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

For example, edge $v_4 \rightarrow v_8$ in *T* shown in Fig. 3 is a *MM-path*. Path $v_9 \rightarrow v_{13} \rightarrow v_{17}$ is another one. The node v_{16} alone is also a *MM-path* in *T*.

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

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

- *u* is of the form $<x, i>$ corresponding to a pair $<x, [\alpha, \beta]>$ (compared to $r[i]$), which is the parent of the first node of an *MM*-path (in *T*) represented by u'; or
- *u* is of the form $<-, 0>$ and u' corresponds to a mismatching node which is a child of a node on the *MM*-path represented by *u*.

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

For example, for *T* shown in Fig. 3, we have its *M*-tree shown in Fig. 7, in which u_0 is a virtual root corresponding to the virtual root of the *S*-tree shown in Fig. 3. Its value is also set to be $<-, 0>$ since it will be handled as a matching node. Then, each path in the *M*-tree corresponds to a B_l. For instance, path $u_0 \rightarrow u_1 \rightarrow u_4 \rightarrow u_8 \rightarrow u_{12}$ corresponds to $B_1 = [1, 4, \infty]$

if all the matching nodes on the path are ignored. For the same reason, $u_0 \to u_1 \to u_5 \to u_{19}$ corresponds to $B_2 = [1, 2, \infty]$.

In addition, we can store all the different nodes v (= $<x$, $[\alpha, \beta]>$) in T in a hash table with each entry associated with a pointer to a node in the corresponding M-tree D, described as follows.

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

For instance, when $<a, [1, 4]>$ (v_1 in T shown in Fig. 3) is created, it is compared to $r[1] = t$. Since $a \neq t$, we have a mismatch and then $u_1 = <a, 1>$ in the M-tree D will be generated. At the same time, we will insert $<a, [1, 4]>$ into the hash table and produce a pointer associated with it to u_1 (see Fig. 8 for illustration). However, when $<c, [1, 2]>$ (v_4 in T shown in Fig. 3) is created, it is compared to $r[2] = c$ and we have a matching. For this, a node $<\text{-}, 0>$ (u_4 in Fig. 7) will be generated, and a link from $<c, [1, 2]>$ to it will be established. But when $<a, [2, 3]>$ (v_8 in T shown in Fig. 3, compared to $r[5] = a$) is met, no node in D will be generated since it is a matching node (in T) and the parent (u_4 in Fig. 7) of the node to be created for it is also $<\text{-}, 0>$. We will simply link it to its parent u_4.

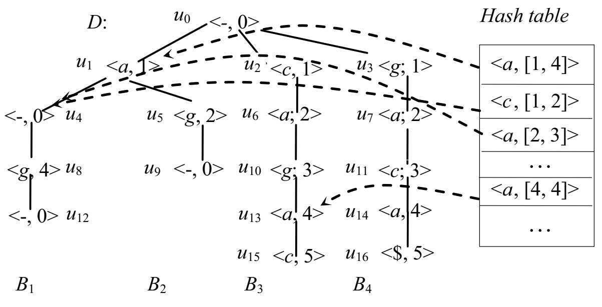


Fig. 7: A mismatch tree

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 the jth node on a path in T (counted from the root with the root as the 0th node).

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 $v = <x, [\alpha, \beta]>$ is popped out from S, we will check whether $x = r[j]$.

i) If $x = r[j]$, we will generate a node $u' = <x, j>$ and link it to u as a child.

ii) If $x \neq r[j]$, we will check whether u is a node of the form $<\text{-}, 0>$. If it is not the case, generate a node $u' = <\text{-}, 0>$. Otherwise, set u' to be u.

iii) Using *search*( ) to find all the children of v: $v_1, \ldots, v_l$. Then, push each $(v_i, j + 1, \kappa', u')$ into S with κ' being κ or $\kappa + 1$, depending on whether $y_i = r[j + 1]$, where $v_i = <y_i, [\alpha_i, \beta_i]>$.

Note that in this process it is not necessary to keep T, but insert all the nodes (of T) in the hash table as discussed above.

Example 2 In this example, we run the above process on r = *tcaca* and $L = BWT(\bar{s})$ shown in Fig. 1(c) with $k = 2$, and show its first 5 steps. The tree created is shown in Fig. 7.

Step 1: Create the *root*, $v_0 = <\text{-}, [1, 8]>$. 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 $<v_3, 1, 1, u_0>$, $<v_2, 1, 1, u_0>$, $<v_1, 1, 1, u_0>$ into S, where v_3, v_2, and v_1 are three children of v_0. See Fig. 8(b).

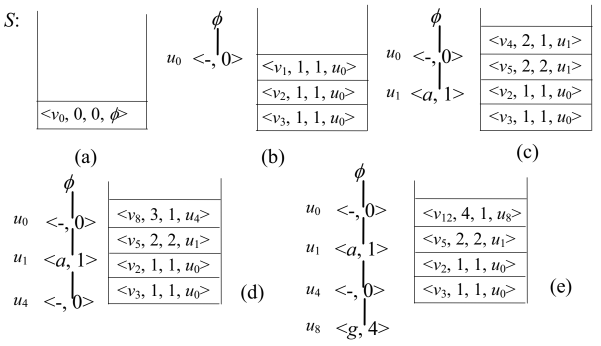


Fig. 8: Illustration for stack changes

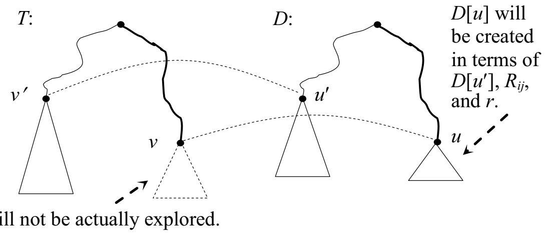
Step 3: Pop out $(v_1, 1, 1, u_0)$ from S. $v_1 = <a, [1. 4]>$. Since $r[1] = t \neq a$, a mismatching node $u_1 = <a, 1>$ will be created and set to be a child of u_0. Then, push $(v_4, 2, 1, u_1)$ into S, where v_4 is the child of v_1. See Fig. 8(c).

Step 4: Pop out $(v_4, 2, 1, u_1)$ from S. $v_4 = <c, [1, 2]>$. Since $r[2] = c$, we will check whether u_1 is a matching node. It is the case. So, a matching node $u_4 = <\text{-}, 0>$ will be created and set to be a child of u_1. Then, push $(v_8, 3, 1, u_4)$ into S, where v_8 is the child of v_4. See Fig. 8(d).

Step 5: Pop out $(v_8, 3, 1, u_4)$ from S. $v_8 = <a, [2, 3]>$. $r[3] = a$. However, no new node is created since u_4 is a matching node. Push $(v_{12}, 4, 1, u_4)$ into S, where v_{12} is the child of v_8. See Fig. 8(e). □

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

As with the basic process, each time a node $v = <x, [\alpha, \beta]>$ (compared to $r[j]$) is encountered, which is the same as a previous one $v' = <x', [\alpha', \beta']>$ (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 $p(v')$ (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 B. 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.



T:
D:
D[u] will be created in terms of D[u'], R_ij, and r.
v'
u'
v
u
T[v] will not be actually explored.

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

To this end, a queue data structure Q is used to do a breadth-first search of $D[u']$, and at the same time generate $D[u]$. In Q, each entry e is a pair (w, γ) with w being a node in $D[u']$, and γ an entry in R_{ij}. Initially, put $(u', R_{ij}[1])$ into Q, where $u' = <x, i>$. 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 $e = (w, R_{ij}[l])$. Assume that $w = <z, f>$ and $R_{ij}[l] = val$. If $<z, f> = <\text{-}, 0>$, then create a copy of w added to $D[u]$. If w is not a leaf node, let $w_1, \ldots, w_h$ be the children of w and enqueue $(w_1, R_{ij}[l]), \ldots, (w_h, R_{ij}[l])$ into Q (append at the end) in turn. If $<z, f> \neq <\text{-}, 0>$, do (2), (3), or (4).
2. If $f < i + val - 1$, add $<z, f - i + j>$ to $D[u]$. If w is not a leaf node, enqueue $(w_1, R_{ij}[l]), \ldots, (w_h, R_{ij}[l])$ into Q.
3. 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 mismatching and add a node $<z, j + val - 1>$ to $D[u]$. If $z = r[j + val - 1]$, create a node $<\text{-}, 0>$ added to $D[u]$. (If its parent is $<\text{-}, 0>$, it should be merged into its parent.)

   If w is not a leaf node, enqueue $<w_1, R_{ij}[l + 1]), \ldots, < w_h, R_{ij}[l + 1])$ into Q.
4. If $f > i + val - 1$, we will scan R_{ij} starting from $R_{ij}[l]$ until we meet $R_{ij}[l']$ such that $f \leq i + R_{ij}[l'] - 1$. For each $R_{ij}[g]$ ($l \leq g < l'$), we create a new node $<r[j + R_{ij}[g] - 1], j + R_{ij}[g] - 1>$ added to $D[u]$. Enqueue $<w, R_{ij}[l']>$ into Q.

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 respetive parents.

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

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

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

Again, if $i > j$, $D[u]$ needs to be extended, which can be done in a way similar to the extension of mismatch arrays as discussed in Subsection *C*.

As an example, consider Fig. 3 and Fig. 7 once again. When we meet $<g, [1, 1]>$ (v_5 in T, compared to $r[2]$) for a second time, we will not generate $T[v_5]$ in Fig. 3, but $D[u_5]$ in Fig. 7. Comparing T and D, we can clearly see the efficiency of this improvement. In D, an *MM*-path in T is collapsed into a single node of the form $<\text{-}, 0>$.

The following is the formal description of the working process.

```
ALGORITHM A(L, r, k)
begin
 1. create root of T; push(S, (root, 0, 0, $\phi$));
 2. while S is not empty do {
 3.   $(v, j, \kappa, u)$ := pop(S); let $v = <x, \alpha, \beta>$;
 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.     enqueuer(Q, (p(v'), $R_{ij}[1]$));
 8.     while Q is not empty do {
 9.       $(w, \gamma)$ := dequeuer(Q); node-creation$(w, \gamma, i, j, R_{ij})$;}}
10.   else {
11.     if $x \neq r[j]$ 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 $\kappa \leq k$ then {
16.       for each $y \in \Sigma$ within $L_v$ do {
17.         $w := search(y, L_v)$;
18.         if $w \neq \phi$ then {
19.           if $y = r[j + 1]$ then push(S, $(w, j + 1, \kappa, u')$);
20.           if $y \neq r[j + 1]$ and $\kappa < k$ then {push(S, $(w, j + 1, \kappa + 1, u')$);
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, κ, u) is popped out from S (see line 4), it will be checked whether v is the same as a previous one v' (compared to $r[i]$). (See line 4.) If it is not the case, a node u' for v will be created in D (see lines 11 – 14). Then, all the children of v will be found by using the procedure *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 operatons (see lines 8 – 9.)

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

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

Proof. In the execution of $\boldsymbol{A}(L, r, k)$, 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 the corresponding maximum matching path and each mismatching node <x, i> with the corresponding pair <x, [α, β]> (compared to $r[i]$), we will get an S-tree, in which each path corresponds to a *search sequence* discussed in Section III. Thus, in D each root-to-leaf path represents an occurrence of r in s having up to k positions different between r and s. □

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). So, the total running time is bounded by O($kn' + n + m$log$m$).

V. EXPERIMENTS

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

- *BWT-based* [34] (BWT for short),
- *Amir's method* [2] (Amir for short),
- *Cole's method* [14] (Cole for short),
- *Algorithm **A** discussed in this paper* (A( ) for short)

By the BWT-based method, an S-tree will be created as described in Section IV, but with σ(i) being used to cut off branches, where σ(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. 10. Then, for each break b_i, located at a certain position i, find all those substrings s_j (located at different positions j) in s such that $b_i = s_j$, and then mark each of them. After that, discard any position that is marked less than k times. In a next step, verify every surviving position in s.

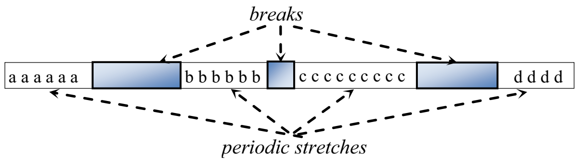


Fig. 10: Illustration for periodic stretches and breaks

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

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

For the test, five reference genomes shown in Table 1 are used. They are all obtained from a biological project conducted in a laboratory at University of Manitoba [27]. 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 [37] with a default model for single reads simulation. Concretely, we take 5000 reads with length varying from 100 bps to 300 bps.

Table 1: Characteristics of genomes

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

To store $BWT(\bar{s})$, we use 2 bits to represent a character ∈ {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. 11(a) and (b), we report the average time of testing the Rat (Rnor_6.0) for matching 100 reads of length 100 to 300 bps. From this figure, we can see that Algorithm A( ) outperforms all the other three methods. But the Amir's method is better than the other two methods. The BWT-based and the Cole's method are comparable. However, for small k, the Cole's is a little bit better than the BWT-based method while for large k their performaces are reversed.

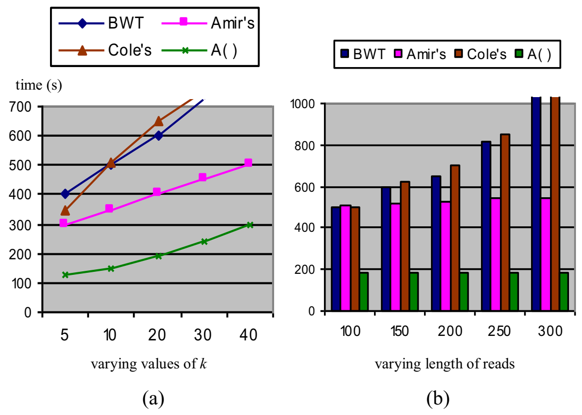


Fig. 11: Test results on varying values of k and read length

To show why A( ) has the best running time, we show the number n' of leaf nodes in the M-trees created by A( ) for some tests in Table 2, which demonstrates that n' can be much smaller than n. Thus, the time complexity O(kn') of A( ) should be a significant improvement over O($n\sqrt{k}$ log$k$) - the time complexity of Amir's.

Table 2: Number of leaf nodes of S-trees

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

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

In Fig. 12(a) and (b), we report the test results of searching the Zebra fish (GRCz10).

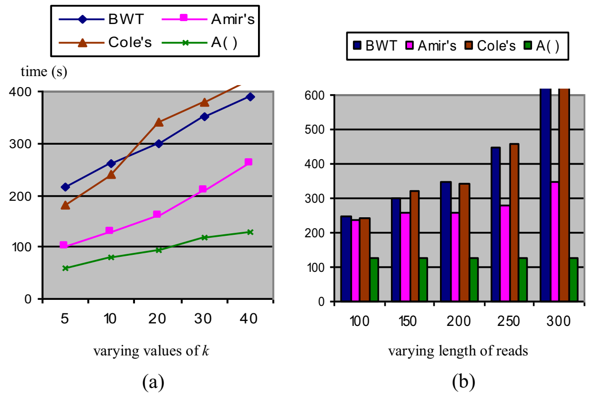


Fig. 12: Test results on varying values of *k* and read length

Again, similar to Fig. 11(a), the performance of Algorithm *A*( ) is best, and the Amir's is still better than both the BWT-based and the Cole's.

In Table 3, we show the number n'.

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	0.7K	0.30M	9.2M	89M

Fig. 12(b) shares the same features as Fig. 11(b). It also shows that only the BWT-based and the Cole's are sensitive to the length of reads.

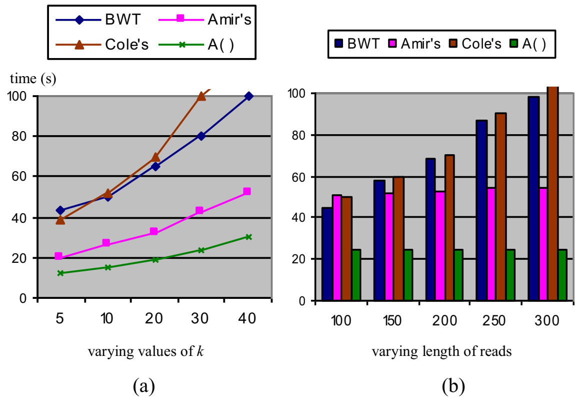


Fig. 13: Test results on varying values of *k* and read length

In Fig. 13, 14, and 15, we show the tests on Rat chr1 (Rnor_6.0), C. elegans (WBcel235), and C. merlae (ASM9120v1), respectively.

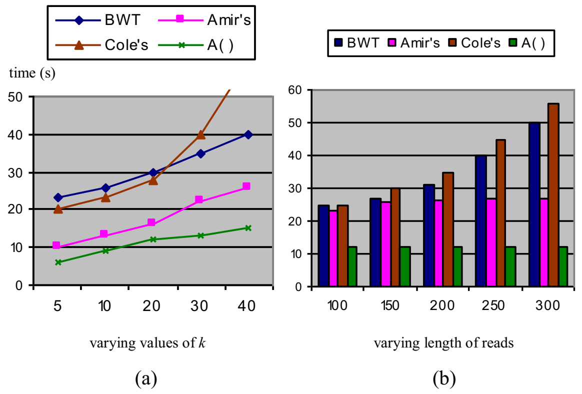


Fig. 14: Test results on varying values of *k* and read length

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

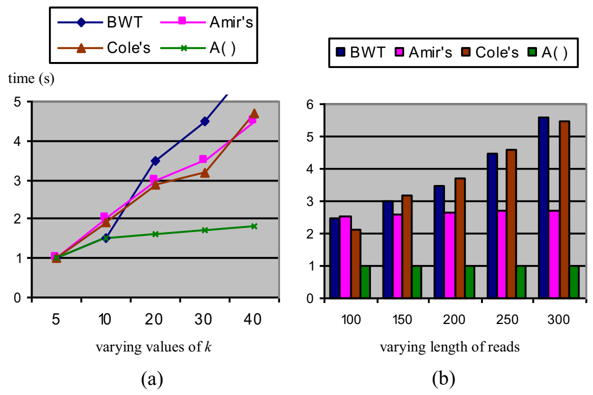


Fig. 15: Test results on varying values of *k* and read length

VI. 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*). Our experiments show that it ihas a better running time than any existing on-line and index-based algorithms.

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

REFERENCES

[1] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Communication of the ACM*, Vol. 23, No. 1, pp. 333 -340, June 1975.

[2] A. Amir, M. Lewenstein and E. Porat, Faster algorithms for string matching with *k* mismatches, *Journal of Algorithms*, Vol. 50, No. 2, Feb.2004, pp. 257-275.

[3] A. Apostolico and R. Giancarlo, The Boyer-Moore-Galil string searching strategies revisited, *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 98 – 105, Feb. 1986.

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

[5] R.A. Baeza-Yates and G.H. Gonnet, A new approach in text searching, *Communication of the ACM*, Vol. 35, No. 10, pp. 74 – 82, Oct. 1992.

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

[7] S. Bauer, M.H. Schulz, Peter N. Robinson, gsuffix: http:://gsuffix. Sourceforge.net/, 2014.

[8] A.M.Bolger, M. Lohse and B. Usadel, Trimmomatic: Bolger: A flexible trimmer for Illumina Sequence Data. Bioinformatics, btu170, 2014.

[9] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Communication of the ACM*, Vol. 20, No. 10, pp. 762 -772, Oct. 1977.

[10] M. Burrows and D.J. Wheeler, A block-sorting lossless data compression algorithm, 1994.

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

[12] Y. Chen, Y. Wu and J. Xie, An Efficient Algorithm for Read Matching in DNA Databases, in *Proc. Int. Conf. DBKDA*'2016, Lisbon, Portugal, June 26 – 30, 2016, pp. 23 – 34.

[13] Y. Chen and Y. Wu, On the Massive String Matching Problem, in *Proc. ICNC-FSKD 2016*, IEEE, Changsha, China, August 13 – 15, 2016.

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

[15] L. Colussi, Z. Galil, and R. Giancarlo, On the exact complexity of string matching, *Proc. 31st Annual IEEE Symposium of Foundation of Computer Science*, Vol. 1, pp. 135 – 144, 1990.

[16] F. Cunningham, et al., Nucleic Acids Research 2015, 43, Database issue:D662-D669.

[17] S.R. Eddy, What is dynamic programming? *Nature Biotechnology* 22, 909 - 910, (2004) doi:10.1038/nbt0704-909.

[18] A. Ehrenfeucht and D. Haussler, A new distance metric on strings computable in linear time, *Discrete Applied Mathematics*, Vol. 20, pp. 191 – 203.

[19] P. Ferragina and G. Manzini, Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pp. 390 - 398. IEEE, 2000.

[20] Z. Galil, On improving the worst case running time of the Boyer-Moore string searching algorithm, *Communication of the ACM*, Vol. 22, No. 9, pp. 505 -508, 1977.

[21] Z. Galil and R. Giancarlo, Improved string matching with *k* mismatches, *ACM SIGACT News*, Vol. 17, Issue 4, Spring 1986, pp. 52b- 54.

[22] M.C. Harrison, Implementation of the substring test by hashing, *Communication of the ACM*, Vol. 14, No. 12, pp. 777- 779, 1971.

[23] H. Jiang, and W.H. Wong, (2008) SeqMap: mapping massive amount of oligonucleotides to the genome, *Bioinformatics*, **24**, 2395–2396.

[24] R.L. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development, Vol. 31, No. 2, pp. 249 – 260, March 1987.

[25] D.E. Knuth, *The Art of Computer Programming*, *Vol. 3*, Massachusetts, Addison-Wesley Publish Com., 1975.

[26] D.E. Knuth, J.H. Morris, and V.R. Pratt, Fast pattern matching in strings, *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323 – 350, June 1977.

[27] lab website: http://home.cc.umanitoba.ca/~xiej/, 2014.

[28] G.M. Landau and U. Vishkin, Efficient string matching in the presence of errors, *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, pp. 126 – 136, 1985.

[29] G.M. Landau and U. Vishkin, Efficient string matching with *k* mismatches, *Theoretical Computer Science*, Vol. 43, pp. 239 – 249, 1986.

[30] B. Langmead, Introduction to the Burrows-Wheeler Trans- form, *www.youtube.com /watch?v=4n7N Pk5lwbI*, Sept., 2014.

[31] T. Lecroq, A variation on the Boyer-Moore algorithm, *Theoretical Computer Science*, Vol. 92, No. 1, pp. 119 – 144, Jan. 1992.

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

[33] R. Li, et al., (2008) SOAP: short oligonucleotide alignment program,*Bioinformatics*, **24**, 713–714.

[34] H. Li and R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler Transform, *Bioinformatics*, Vol. 25 no. 14 2009, pp. 1754–1760.

[35] H. Li and R. Durbin, Fast and accurate long-read alignment with Burrows–Wheeler Transform, *Bioinformatics*, Vol. 26 no. 5 2010, pp. 589–595.

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

[37] H. Li, wgsim: a small tool for simulating sequence reads from a reference genome, https://github.com/lh3/wgsim/, 2014.

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

[39] U. Manber and E.W. Myers, Suffix arrays: a new method for on-line string searches, *Proc. the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319 – 327, SIAM, Philadelphia, PA, 1990.

[40] U. Manber and R.A. Baeza-Yates, An algorithm for string matching with a sequence of don't cares, *Information Processing Letters*, Vol. 37, pp. 133 – 136, Feb. 1991.

[41] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM, Vol. 23, No. 2, pp. 262 – 272, April 1976.

[42] G. Navarro and M. Raffinot, *Pattern Matching in Strings*, Cambridge University Press, 2002.

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

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

[45] M. Schatz, (2009) Cloudburst: highly sensitive read mapping with mapreduce, *Bioinformatics*, **25**, 1363–1369.

[46] J. Seward. bzip2 and libbzip2, version 1.0. 5: A program and library for data compression. URL http://www. bzip. org, 2007.

[47] A.D. Smith, et al, (2008) Using quality scores and longer reads improves accuracy of Solexa read mapping, *BMC Bioinformatics*, **9**, 128.

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

[49] J. Tarhio and E. Ukkonen, Approximate Boyer-Moore String Matching, *SIAM Journal on Computing*, Vol. 22, No. 2, pp. 243 -260.

[50] E. Ukkonen, Approximate string-matching with *q*-grams and maximal matches, *Theoretical Computer Science*, Vol. 92, pp. 191 – 211.

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

[52] W. Hon et al., A space and time efficient algorithm for constructing compressed suffix arrays, *Alrothmica*, **48**, 23 – 36, 2007.