

# An Efficient Algorithm for Read Matching in DNA Databases

Yangjun Chen, Yujia Wu

Dept. Applied Computer Science

University of Winnipeg, Canada

email: y.chen@uwinnipeg.ca, wyj1128@yahoo.com

Jiuyong Xie

Dept. Physiology & Pathophysiology, College of Medicine

University of Manitoba, Canada

email: xiej@umanitoba.ca

**Abstract**—In this paper, we discuss an efficient and effective index mechanism to support the matching of massive reads (short DNA strings) in DNA databases. It is very important to the next generation sequencing in the biological research. The main idea behind it is to construct a trie structure over all the reads, and search the trie against a BWT-array  $L$  created for a genome sequence  $s$  to locate all the occurrences of every read in  $s$  once for all. In addition, we change a single-character checking against  $L$  to a multiple-character checking, by which multiple searches of  $L$  are reduced to a single scanning of  $L$ . In this way, high efficiency can be achieved. Experiments have been conducted, which show that our method for this problem is promising.

**Keywords**—string matching; DNA sequences; tries; BWT-transformation

## I. INTRODUCTION

The recent development of next-generation sequencing has changed the way we carry out the molecular biology and genomic studies. It has allowed us to sequence a DNA (Deoxyribonucleic acid) sequence at a significantly increased base coverage, as well as at a much faster rate. This facilitates building an excellent platform for the whole genome sequencing, and for a variety of sequencing-based analysis, including gene expressions, mapping DNA-protein interactions, whole-transcriptome sequencing, and RNA (Ribonucleic acid) splicing profiles. For example, the RNA-Seq protocol, in which processed mRNA is converted to cDNA and sequenced, is enabling the identification of previously unknown genes and alternative splice variants. The whole-genome sequencing of tumour cells can uncover previously unidentified cancer-initiating mutations.

The core and the first step to take advantage of the new sequencing technology is termed as *read aligning*, where a read is a short nucleotide sequence of 30 - 1000 base pairs (bp) generated by a high throughput sequencing machine made by Illumina, Roche, ABI/Life Technologies, which is in fact a sequence fragment fetched from a longer DNA molecule present in a sample that is fed into the machine. Most of the next-generation sequencing projects begin with a *reference sequence* which is a previously well studied, known *genome*. The process of a read aligning is to find the meaning of reads, or in other words, to determine their positions within a reference sequence, which will then be used for an effective statistical analysis.

Compared to the traditional pattern matching problems, the new challenge from the read aligning is its enormous volume, usually millions to billions of reads need to be aligned within

a same reference sequence. For example, to sequence a human molecule sample with 15X coverage, one may need to align 1.5 billion reads of length about 100 characters (bps).

In general, three kinds of alignment algorithms are practically applied: *hash-based*, *string-matching-based*, as well as *inexact matching* (including *edit-distance* computation and *k-mismatching*). By the hash-based methods, short subsequences called *seeds* are extracted from a pattern sequence and their hash values are computed, which are used to search against a reference genome sequence. By the string-matching-based methods, different efficient algorithms are utilized, such as *Knuth-Morris-Pratt* [22], *Boyer-Moore* [9], and *Apostolico-Giancarlo* [3], as well as the algorithms based on different *indexes* like *suffix trees* [37][45], *suffix arrays* [35], and *BWT-transformation* (*Burrows-Wheeler Transform*) [10, 16, 40]. By the *edit-distance* computation, a *score* matrix to represent the relevance between characters is defined and an alignment with the highest total score is searched, for which the *dynamical programming* paradigm is typically employed. However, a recent research shows that the BWT can also be used as an index structure for the *k-mismatching* problem [30].

All the methods mentioned above are *single-pattern* oriented, by which a single string pattern is checked against an entire database to find all the alignments in all the sequences stored in the database. In the current research of the molecular biology, however, we need to check a bunch of string patterns each time and the size of all string patterns can be even much larger than the database itself. This requires us considering all the string patterns as a whole, rather than separately check them one by one. By the *Aho-Corasick* algorithm [1], the multiple patterns are handled. However, it cannot be utilized in an indexing environment since it has to search a target sequence linearly while by using indexes to expedite a search this is not expected.

In this paper, we address this issue and present a holistic string matching algorithm to handle million-billion reads. Our experiment shows that it can be more than 40% faster than single-pattern oriented methods when multi-million reads are checked. The main idea behind our method is:

1. Construct a trie  $T$  over all the pattern sequences, and check  $T$  against a BWT-array created as an index for a target (reference) sequence. This enables us to avoid repeated search of the same part of different reads.
2. Change a single-character checking to a multiple-character checking. (That is, each time a set of characters respectively from more than one read will be checked

against a BWT-array in one scan, instead of checking them separately one by one in multiple scans.)

In this way, high efficiency has been achieved.

The remainder of the paper is organized as follows. In Section II, we review the related work. In Section III, we briefly describe a string matching algorithm based on the BWT-transformation. In Section IV, we discuss our basic algorithm in great detail. In Section V, we improve the basic method by using multiple-character checkings. Section VI is devoted to the test results. Finally, a short conclusion is set forth in Section VII.

## II. RELATED WORK

The matching of DNA sequences is just a special case of the general string matching problem, which has always been one of the main focuses in the computer science. All the methods developed up to now can be roughly divided into two categories: exact matching and inexact matching. By the former, all the occurrences of a pattern string  $p$  in a target string  $s$  will be searched. By the latter, a best alignment between  $p$  and  $s$  (i.e., a correspondence with the highest score) is searched in terms of a given score matrix  $M$ , which is established to indicate the relevance between characters (more exactly, the meanings represented by them).

### A. Exact Matching

**Scanning-based** By this kind of algorithms, both pattern  $p$  and  $s$  are scanned from left to right, but often with an auxiliary data structure used to speed up the search, which is typically constructed by a pre-processor. The first of them is the famous *Knuth-Morris-Pratt* algorithm [22], which employs an auxiliary *next-table* (for  $p$ ) 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  $p$  fails to match the current character in  $s$ . Its time complexity is bounded by  $O(m+n)$ , where  $m=|p|$  and  $n=|s|$ . The *Boyer-Moore* approach [9] works a little bit better than the *Knuth-Morris-Pratt*. In addition to the *next-table*, a *skip-table* (also for  $p$ ) 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, they sparked a series of research on this problem, and improved by different researchers in different ways, such as the algorithms discussed in [1][27]. However, the worst-case time complexity remains unchanged. 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  $k$  patterns:  $\{p_1, p_2, \dots, p_k\}$ . Their algorithm needs only  $O(\sum_{i=1}^k m_i + n)$  time, where  $m_i=|p_i|$  ( $i=1, \dots, k$ ). However, this algorithm cannot be adapted to an index environment due its working fashion totally unsuitable for indexes.

**Index-based** In situations where a fixed string  $s$  is to be searched repeatedly, it is worthwhile constructing an index over  $s$  [46], such as suffix trees [37][45], suffix arrays [35], and more recently the *BWT-transformation* [10][16][30][31][40]. A suffix tree is in fact a *trie* structure [21] over all the suffixes of  $s$ ; and by using the Weiner’s

algorithm it can be built in  $O(n)$  time [37]. However, in comparison with suffix trees, the BWT-transformation is more suitable for DNA sequences due to its small alphabet  $\Sigma$  since the smaller  $\Sigma$  is, the smaller space will be occupied by the corresponding BWT index. According to a survey done by Li and Homer [30] 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 byte for the BWT. Our experiments 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 handlable on PC or laptop machines. The huge size of a suffix tree may greatly affect the computation time. For example, for the Zebra fish and Rat genomes (sizes 1,464,443,456 pb, and 2,909,701,677 pb, respectively), we cannot finish the construction of their suffix trees within two days in a computer with 32GB RAM.

**Hash-based** Intrinsically, all hash-table-based algorithms [18, 20] extract short subsequences called ‘seeds’ from a pattern sequence  $p$  and create a *signature* (a bit string) for each of them. The search of a target sequence  $s$  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 [19, 28, 29, 34, 39], and all experiments shows that they are generally inferior to the suffix tree and the BWT index in both running time and space requirements.

### B. Inexact Matching

The inexact matching ranges from the score-based to the  $k$ -mismatching, as well as the  $k$ -error. By the score-based method, a score matrix  $M$  of size  $|\Sigma| \times |\Sigma|$  is used to indicate the relevance between characters. The algorithm designed is to find the best alignment (or say, the alignment with the highest scores) between two given strings, which can be DNA sequences, protein sequences, or XML documents; and the *dynamic programming* paradigm is often utilized to solve the problem [14]. By the  $k$ -mismatching, we will find all those subsequences  $q$  of  $s$  such that  $d(p, q) \leq k$ , where  $d(\ )$  is a distance function. When it is the *Hemming* distance, the problem is known as sequence matching with  $k$  mismatches [4]. When it is the *Levenshtein* distance, the problem is known as sequence matching with  $k$  errors [6]. There is a bunch of algorithms proposed for this problem, such as [4, 5, 24, 25, 42, 43] for the  $k$ -mismatch; and [6, 11, 15, 44] for the  $k$ -error. All the methods for the  $k$ -mismatch needs quadratic time  $O(mn)$  in the worst case. However, the algorithm discussed in [2] has the best expected time complexity  $O(n \cdot \sqrt{k} \cdot \log m)$ . Especially, for small  $k$  and large  $\Sigma$ , the search requires sublinear time on average. In addition, the BWT can also be used as an index structure for this problem [30]. For the  $k$ -error, the worst case time complexity is the same as the  $k$ -mismatching. But the expected time can reach  $O(kn)$  by an algorithm discussed in [11]. As a different kind of inexact matching, the string matching with *Don’t-Cares* (or *wild-cards*) has also been an

active research topic for decades, by which we may have wild-cards in  $p$ , 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*. All the methods proposed to solve this problem also needs quadratic time [38]. But 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 [36].

### III. BWT-TRANSFORMATION

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

#### A. BWT and String Compression

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 as shown in Figure 1(a).

	$F$	$L$	
$a c a g a c a \$$	$a_1$	$\$$	$\$ a c a g a c a$
$c a g a c a \$ a$	$c_1$	$a_1$	$a \$ a c a g a c$
$a g a c a \$ a c$	$a_2$	$c_1$	$a c a \$ a c a g$
$g a c a \$ a c a$	$g_1$	$a_2$	$a c a g a c a \$$
$a c a \$ a c a g$	$a_3$	$g_1$	$a g a c a \$ a c$
$c a \$ a c a g a$	$c_2$	$a_3$	$c a \$ a c a g a$
$a \$ a c a g a c$	$a_4$	$c_2$	$c a g a c a \$ a$
$\$ a c a g a c a$	$\$$	$a_4$	$g a c a \$ a c a$
(a)	(b)		(c)

Figure 1. Rotation of a string

By writing all these strings stacked vertically, we generate an  $n \times n$  matrix, where  $n = |s|$  (see Figure 1(a).) Here, special attention should be paid to the first column, denoted as  $F$ , and the last column, denoted as  $L$ . For them, the following equation, called the *LF mapping*, can be immediately observed:

$$F[i] = L[i]’s \text{ successor}, \tag{1}$$

where  $F[i]$  ( $L[i]$ ) is the  $i^{th}$  element of  $F$  (resp.  $L$ ).

From this property, another property, the so-called *rank correspondence* can be derived, by which we mean that for each character, its  $i$ th appearance in  $F$  corresponds to its  $i$ th appearance in  $L$ , as demonstrated in Figure 1(b), in which 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 2nd appearance of  $a$  in  $s$ ) is in the second place among all the  $a$ -characters in both  $F$  and  $L$  while  $c_1$  the first appearance in both  $F$  and  $L$  among all the  $c$ -characters. In the same way, we can check all the other appearances of different characters.

Now we sort the rows of the matrix alphabetically. We will get another matrix, called the *Burrow-Wheeler Matrix* [7] [12][23] and denoted as  $BWM(s)$ , as demonstrated in Figure

1(c). Especially, the last column 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  $BWT(s) = acg\$caaa$ .

By the  $BWM$  matrix, the  $LF$ -mapping is obviously not changed. Surprisingly, the rank correspondence also remains. Even though the ranks of different appearances of a certain character (in  $F$  or in  $L$ ) may be different from before, their rank correspondences are not changed as shown in Figure 2(b), in which  $a_2$  now appears in both  $F$  and  $L$  as the fourth element among all the  $a$ -characters, and  $c_1$  the second element among all the  $c$ -characters.

$rk_F$	$F$	$L$	$rk_L$		
–	$\$$	$a_4$	1	By ranking the elements in $F$ , each element in $L$ is also ranked with the same number.	$F_\$ = \langle \$; 1, 1 \rangle$
1	$a_4$	$c_2$	1		$F_a = \langle a; 2, 5 \rangle$
2	$a_3$	$g_1$	1		$F_c = \langle c; 6, 7 \rangle$
3	$a_1$	$\$$	–		$F_g = \langle g; 8, 8 \rangle$
4	$a_2$	$c_1$	2		
1	$c_2$	$a_3$	2		
2	$c_1$	$a_1$	3		
1	$g_1$	$a_2$	4	(a)	(b)

Figure 2.  $LF$ -mapping and rank-correspondence

The first purpose of  $BWT(s)$  is for the string compression since same characters with similar *right-contexts* in  $s$  tend to be clustered together in  $BWT(s)$ , as shown by the following example [10][16][40]:

$BWT(\text{tomorrow and tomorrow and tomorrow})$   
 =  $wwwd nnooaattmmrrrrrrroo \$ooo$

Such a transformed string can be effectively compressed and then decompressed. Due to the  $LF$ -mapping and the rank correspondence, it can also be easily restored to the original string.

The second purpose is for the string search, which will be discussed in the next subsection in great detail. We need this part of knowledge to develop our method.

#### B. String Search Using BWT

For the purpose of the string search, the character clustering in  $F$  has to be used. Especially, for any DNA sequence, the whole  $F$  can be divided into five or less segments:  $\$$ -segment,  $A$ -segment,  $C$ -segment,  $G$ -segment, and  $T$ -segment, denoted as  $F_\$, F_A, F_C, F_G, F_T$ , respectively. In addition, for each segment in  $F$ , we will rank all its elements from top to bottom, as illustrated in Figure 2(a).  $\$$  is not ranked since it appears only once.

From Figure 2(a), we can see that the rank of  $a_4$ , denoted as  $rk_F(a_4)$ , is 1 since it is the first element in  $F_A$ . For the same reason, we have  $rk_F(a_3) = 2$ ,  $rk_F(a_1) = 3$ ,  $rk_F(a_2) = 4$ ,  $rk_F(c_2) = 1$ ,  $rk_F(c_1) = 2$ , and  $rk_F(g_1) = 1$ .

It can also be seen that each segment in  $F$  can be effectively represented as a triplet of the form:  $\langle \alpha; x_\alpha, y_\alpha \rangle$ , where  $\alpha \in \Sigma \cup \{\$\}$ , and  $x_\alpha, y_\alpha$  are the positions of the first and last appearance of  $\alpha$  in  $F$ , respectively. So the whole  $F$

can be effectively compacted and represented as a set of  $|\Sigma| + 1$  triplets, as illustrated in Figure 2(b).

Now, we consider  $\alpha_j$  (the  $j$ th appearance of  $\alpha$  in  $s$ ). Assume that  $rk_F(\alpha_j) = i$ . Then, the position where  $\alpha_j$  appears in  $F$  can be easily determined:

$$F[x_\alpha + i - 1] = \alpha_j. \quad (2)$$

Besides, if we rank all the elements in  $L$  top-down in such a way that an  $\alpha_j$  is assigned  $i$  if it is the  $i$ th appearance among all the appearances of  $\alpha$  in  $L$ . Then, we will have

$$rk_F(\alpha_j) = rk_L(\alpha_j), \quad (3)$$

where  $rk_L(\alpha_j)$  is the rank assigned to  $\alpha_j$  in  $L$ .

This equation is due to the rank correspondence between  $F$  and  $L$ . (See [10][16][40] for a detailed discussion. Also see Figure 2(a) for ease of understanding.)

With the ranks established, a string matching can be very efficiently conducted by using the formulas (2) and (3). To see this, let's consider a pattern string  $p = aca$  and try to find all its occurrences in  $s = acagaca\$$ .

We work on the characters in  $p$  in the reverse order.

First, we check  $p[3] = a$  in the pattern string  $p$ , and then figure out a segment in  $L$ , denoted as  $L'$ , corresponding to  $F_a = \langle a; 2, 5 \rangle$ . So  $L' = L[2 .. 5]$ , as illustrated in Figure 3(a), where we still use the non-compact  $F$  for explanation. In the second step, we check  $p[2] = c$ , and then search within  $L'$  to find the first and last  $c$  in  $L'$ . We will find  $rk_L(c_2) = 1$  and  $rk_L(c_1) = 2$ . By using (3), we will get  $rk_F(c_2) = 1$  and  $rk_F(c_1) = 2$ . Then, by using (2), we will figure out a sub-segment  $F'$  in  $F$ :  $F[x_c + 1 - 1 .. x_c + 2 - 1] = F[6 + 1 - 1 .. 6 + 2 - 1] = F[6 .. 7]$ . (Note that  $x_c = 6$ . See Figure 2(b) and Figure 3(b).) In the third step, we check  $p[1] = a$ , and find  $L'' = L[6 .. 7]$  corresponding to  $F' = F[6 .. 7]$ . Repeating the above operation, we will find  $rk_L(a_3) = 2$  and  $rk_L(a_1) = 3$ . See Figure 3(c). Since now we have exhausted all the characters in  $p$  and  $F[x_a + 2 - 1, x_a + 3 - 1] = F[3, 4]$  contains only two elements, two occurrences of  $p$  in  $s$  are found. They are  $a_1$  and  $a_3$  in  $s$ , respectively.

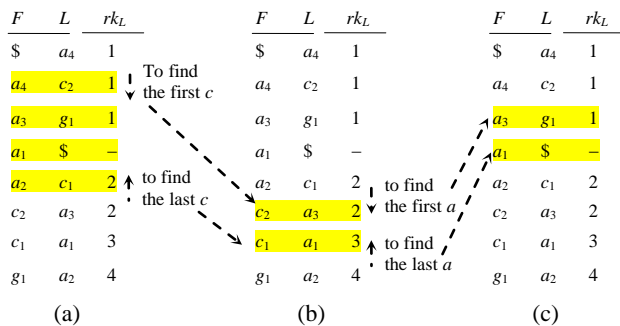


Figure 3. Sample trace

### C. RankAll

The dominant cost of the above process is the searching of  $L$  in each step. However, this can be dramatically reduced by arranging  $|\Sigma|$  arrays each for a character  $\alpha \in \Sigma$  such that  $\alpha[i]$  (the  $i$ th entry in the array for  $\alpha$ ) is the number of appearances of  $\alpha$  within  $L[1 .. i]$ . See Figure 4(a) for illustration.

Now, instead of scanning a certain segment  $L[x .. y]$  ( $x \leq y$ ) to find a subrange for a certain  $\alpha \in \Sigma$ , we can simply look up the array for  $\alpha$  to see whether  $\alpha[x - 1] = \alpha[y]$ . If it is the case, then  $\alpha$  does not occur in  $L[x .. y]$ . Otherwise,  $[\alpha[x - 1] + 1, \alpha[y]]$  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  $[c[2 - 1] + 1, c[5]] = [1, 2]$ .

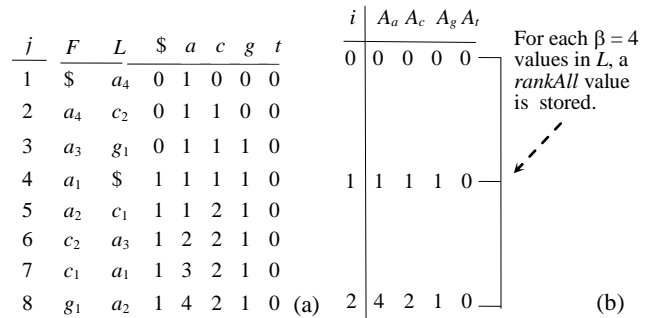


Figure 4. LF-mapping and rank-correspondence

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  $\alpha[i]$  with a compact array  $A_\alpha$  for each  $\alpha \in \Sigma$ , in which, rather than for each  $L[i]$  ( $i \in \{1, \dots, n\}$ ), 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  $A_\alpha$ . Obviously, doing so, more search will be required. In practice, the size  $\beta$  of a bucket (referred to as a *compact factor*) can be set to different values. For example, we can set  $\beta = 4$ , indicating that for each four contiguous elements in  $L$  a group of  $|\Sigma|$  integers (each in an  $A_\alpha$ ) will be stored. That is, we will not store all the values in Figure 4(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 Figure 4(b). However, each  $\alpha[j]$  for  $\alpha \in \Sigma$  can be easily derived from  $A_\alpha$  by using the following formulas:

$$\alpha[j] = A_\alpha[i] + \rho, \quad (4)$$

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

$$\alpha[j] = A_\alpha[i'] - \rho', \quad (5)$$

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

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

Finally, we notice that the column for  $\$$  can always be divided into two parts. In the first part, each entry is 0 while in the second part each entry is 1 (see Figure 4(a)). So we can

simply keep a number to indicate where it is divided, instead of storing the whole column.

#### D. Construction of BWT arrays

For self-explanation, we describe how a BWT array is constructed [10][16][26][40] in this subsection.

As mentioned above, a string  $s = a_0a_1 \dots a_{n-1}$  is always ended with \$ (i.e.,  $a_i \in \Sigma$  for  $i = 0, \dots, n-2$ , and  $a_{n-1} = \$$ ). Let  $s[i] = a_i$  ( $i = 0, 1, \dots, n-1$ ) be the  $i$ th character of  $s$ ,  $s[i..j] = a_i \dots a_j$  a substring and  $s[i..n-1]$  a suffix of  $s$ . Suffix array  $H$  of  $s$  is a permutation of the integers  $0, \dots, n-1$  such that  $H[i]$  is the start position of the  $i$ th smallest suffix. The relationship between  $H$  and the BWT array  $L$  can be determined by the following formulas:

$$\begin{cases} L[i] = \$, & \text{if } H[i] = 0; \\ L[i] = s[H[i] - 1], & \text{otherwise.} \end{cases} \quad (6)$$

Once  $L$  is determined,  $F$  can also be created immediately by using formula (1).

#### IV. MAIN ALGORITHM

In this section, we present our algorithm to search a bunch of reads against a genome  $s$ . Its main idea is to organize all the reads into a trie  $T$  and search  $T$  against  $L$  to avoid any possible redundancy. First, we present the concept of tries in Subsection A. Then, in Subsection B, we discuss our basic algorithm for the task. We improve this algorithm in Section V.

##### A. Tries over Reads

Let  $\mathbf{D} = \{s_1, \dots, s_n\}$  be a DNA database, where each  $s_i$  ( $i = 1, \dots, n$ ) is a genome, a very long string  $\in \Sigma^*$  ( $\Sigma = \{A, T, C, G\}$ ). Let  $\mathbf{R} = \{r_1, \dots, r_m\}$  be a set of reads with each  $r_j$  being a short string  $\in \Sigma^*$ . The problem is to find, for every  $r_j$ 's ( $j = 1, \dots, m$ ), all their occurrences in an  $s_i$  ( $i = 1, \dots, n$ ) in  $\mathbf{D}$ .

A simple way to do this is to check each  $r_j$  against  $s_i$  one by one, for which different string searching methods can be used, such as suffix trees [37][45], BW-transformation [10], and so on. Each of them needs only a linear time (in the size of  $s_i$ ) to find all occurrences of  $r_j$  in  $s_i$ . However, in the case of very large  $m$ , which is typical in the new genomic research, one-by-one search of reads against an  $s_i$  is no more acceptable in practice and some efforts should be spent on reducing the running time caused by huge  $m$ .

Our general idea is to organize all  $r_j$ 's into a trie structure  $T$  and search  $T$  against  $s_i$  with the BW-transformation being used to check the string matching. For this purpose, we will first attach \$ to the end of each  $s_i$  ( $i = 1, \dots, n$ ) and construct  $BWT(s_i)$ . Then, attach \$ to the end of each  $r_j$  ( $j = 1, \dots, m$ ) to construct  $T = \text{trie}(\mathbf{R})$  over  $\mathbf{R}$  as below.

If  $|\mathbf{R}| = 0$ ,  $\text{trie}(\mathbf{R})$  is, of course, empty. For  $|\mathbf{R}| = 1$ ,  $\text{trie}(\mathbf{R})$  is a single node. If  $|\mathbf{R}| > 1$ ,  $\mathbf{R}$  is split into  $|\Sigma| = 5$  (possibly empty) subsets  $\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_5$  so that each  $\mathbf{R}_i$  ( $i \in \{1, \dots, 5\}$ ) contains all those sequences with the same first character  $\alpha_i \in \{A, T, C, G\} \cup \{\$\}$ . The tries:  $\text{trie}(\mathbf{R}_1), \text{trie}(\mathbf{R}_2), \dots, \text{trie}(\mathbf{R}_5)$  are constructed in the same way except that at the  $k$ th step, the

splitting of sets is based on the  $k$ th characters in the sequences. They are then connected from their respective roots to a single node to create  $\text{trie}(\mathbf{R})$ .

**Example 1** As an example, consider a set of four reads:

$r_1$ : ACAGA  
 $r_2$ : AG  
 $r_3$ : ACAGC  
 $r_4$ : CA

For these reads, a trie can be constructed as shown in Figure 5(a). In this trie,  $v_0$  is a virtual root, labeled with an empty character  $\varepsilon$  while any other node  $v$  is labeled with a real character, denoted as  $l(v)$ . Therefore, all the characters on a path from the root to a leaf spell a read. For instance, the path from  $v_0$  to  $v_8$  corresponds to the third read  $r_3 = \text{ACAGC}\$$ . Note that each leaf node  $v$  is labelled with \$ and associated with a read identifier, denoted as  $\gamma(v)$ .

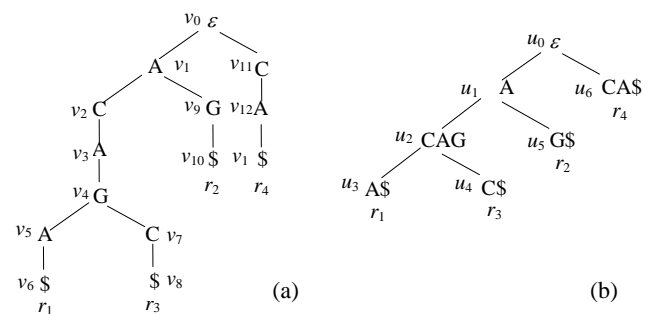


Figure 5. A trie and its compact version

The size of a trie can be significantly reduced by replacing each branchless path segment with a single edge. By a branchless path we mean a path  $P$  such that each node on  $P$ , except the starting and ending nodes, has only one incoming and one outgoing edge. For example, the trie shown in Figure 5(a) can be compacted to a reduced one as shown in Figure 5(b).

##### B. Integrating BWT Search with Trie Search

It is easy to see that exploring a path in a trie  $T$  over a set of reads  $\mathbf{R}$  corresponds to scanning a read  $r \in \mathbf{R}$ . If we explore, at the same time, the  $L$  array established over a reversed genome sequence  $\bar{s}$ , we will find all the occurrences of  $r$  (without \$ involved) in  $s$ . This idea leads to the following algorithm, which is in essence a depth-first search of  $T$  by using a stack  $S$  to control the process. However, each entry in  $S$  is a triplet  $\langle v, a, b \rangle$  with  $v$  being a node in  $T$  and  $a \leq b$ , used to indicate a subsegment in  $F_{l(v)}[a..b]$ . For example, when searching the trie shown in Figure 5(a) against the  $L$  array shown in Figure 2(a), we may have an entry like  $\langle v_1, 1, 4 \rangle$  in  $S$  to represent a subsegment  $F_A[1..4]$  (the first to the fourth entry in  $F_A$ ) since  $l(v_1) = 'A'$ . In addition, for technical convenience, we use  $F_\varepsilon$  to represent the whole  $F$ . Then,  $F_\varepsilon[a..b]$  represents the segment from the  $a$ th to the  $b$ th entry in  $F$ .

In the algorithm, we first push  $\langle \text{root}(T), 1, |s| \rangle$  into stack  $S$  (lines 1 – 2). Then, we go into the main **while-loop** (lines 3 – 16), in which we will first pop out the top element from  $S$ , stored as a triplet  $\langle v, a, b \rangle$  (line 4). Then, for each child  $v_i$  of  $v$ ,

we will check whether it is a leaf node. If it is the case, a quadruple  $\langle \gamma(v_i), l(v), a, b \rangle$  will be added to the result  $\mathcal{R}$  (see line 7), which records all the occurrences of a read represented by  $\gamma(v_i)$  in  $s$ . (In practice, we store compressed suffix arrays [35, 40] and use formulas (1) and (6) to calculate positions of reads in  $s$ .) Otherwise, we will determine a segment in  $L$  by calculating  $x'$  and  $y'$  (see lines 8 – 9). Then, we will use  $sDown(L, x' - 1, \beta, \alpha)$  or  $sUp(L, x' - 1, \beta, \alpha)$  to find  $\alpha[x' - 1]$  as discussed in the previous section. (See line 10.) Next, we will find  $\alpha[y']$  in a similar way. (See line 11.) If  $\alpha[y'] > \alpha[x' - 1]$ , there are some occurrences of  $\alpha$  in  $L[x' .. y']$  and we will push  $\langle v_i, \alpha[x' - 1] + 1, \alpha[y'] \rangle$  into  $S$ , where  $\alpha[x' - 1] + 1$  and  $\alpha[y']$  are the first and last rank of  $\alpha$ 's appearances within  $L[x' .. y']$ , respectively. (See lines 12 – 13.) If  $\alpha[y'] = \alpha[x' - 1]$ ,  $\alpha$  does not occur in  $L[x' .. y']$  at all and nothing will be done in this case. The following example helps for illustration.

---

**ALGORITHM** *readSearch*( $T, LF, \beta$ )
 

---

**begin**

```

1.  $v \leftarrow root(T)$ ;  $\mathcal{R} \leftarrow \Phi$ ;
2.  $push(S, \langle v, 1, |s| \rangle)$ ;
3. while  $S$  is not empty do {
4.    $\langle v, a, b \rangle \leftarrow pop(S)$ ;
5.   let  $v_1, \dots, v_k$  be the children of  $v$ ;
6.   for  $i = k$  downto 1 do {
7.     if  $v_i$  is a leaf then  $\mathcal{R} \leftarrow \mathcal{R} \cup \{ \langle \gamma(v_i), l(v), a, b \rangle \}$ ;
8.     else { assume that  $F_{l(v)} = \langle l(v); x, y \rangle$ ;
9.        $x' \leftarrow x + a - 1$ ;  $y' \leftarrow x + b - 1$ ;  $\alpha \leftarrow l(v_i)$ ;
10.      find  $\alpha[x' - 1]$  by  $sDown(L, x' - 1, \beta, \alpha)$  or  $sUp(L, x' - 1, \beta, \alpha)$ ;
11.      find  $\alpha[y']$  by  $sDown(L, y', \beta, \alpha)$  or  $sUp(L, y', \beta, \alpha)$ ;
12.      if  $\alpha[y'] > \alpha[x' - 1]$  then
13.         $push(S, \langle v_i, \alpha[x' - 1] + 1, \alpha[y'] \rangle)$ ;
14.      }
15.   }
16. }
end
    
```

---

 Figure 6. Algorithm *readSearch*()

**Example 2** Consider all the reads given in Example 1 again. The trie  $T$  over these reads are shown in Figure 5(a). In order to find all the occurrences of these reads in  $s = ACAGACA\$,$  we will run *readSearch*( ) on  $T$  and the  $LF$  of  $\bar{s}$  shown in Figure 7(b). (Note that  $s = \bar{s}$  for this special string, but the ordering of the subscripts of characters is reversed. In Figure 7(a), we also show the corresponding  $BWM$  matrix for ease of understanding.)

In the execution of *readSearch*( ), the following steps will be carried out.

Step 1: push  $\langle v_0, 1, 8 \rangle$  into  $S$ , as illustrated in Figure 7(c).

Step 2: pop out the top element  $\langle v_0, 1, 8 \rangle$  from  $S$ . Figure out the two children of  $v_0$ :  $v_1$  and  $v_{11}$ . First, for  $v_{11}$ , we will use  $A_c$  to find the first and last appearances of  $l(v_{11}) = 'C'$  in  $L[1 .. 8]$  and their respective ranks: 1 and 2. Assume that  $\beta = 4$  (i.e., for each 4 consecutive entries in  $L$  a *rankAll* value is stored.) Further assume that for each  $A_\alpha$  ( $\alpha \in \{a, c, g, t\}$ )  $A_\alpha[0] = 0$ . The ranks are calculated as follows.


	$j$	$F$	$L$	
$\$ A_4 C_2 A_3 G_1 A_2 C_1 A_1$	1	$\$$	$A_4$	$S:$ 
$A_1 \$ A_4 C_2 A_3 G_1 A_2 C_1$	2	$A_4$	$C_2$	
$A_2 C_1 A_1 \$ A_4 C_2 A_3 G_1$	3	$A_3$	$G_1$	
$A_4 C_2 A_3 G_1 A_2 C_1 A_1 \$$	4	$A_1$	$\$$	
$A_3 G_1 A_2 C_1 A_1 \$ A_4 C_2$	5	$A_2$	$C_1$	
$C_1 A_1 \$ A_4 C_2 A_3 G_1 A_2$	6	$C_2$	$A_3$	
$C_2 A_3 G_1 A_2 C_1 A_1 \$ A_4$	7	$C_1$	$A_1$	
$G_1 A_2 C_1 A_1 \$ A_4 C_2 A_3$	8	$G_1$	$A_2$	

Figure 7. Illustration for Step 1

- To find the rank of the first appearance of 'C' in  $L[1 .. 8]$ , we will first calculate  $C[0]$  by using formula (4) or (5) (i.e., by calling  $sDown(L, 0, 4, C)$  or  $sUp(L, 0, 4, C)$ ). Recall that whether (4) or (5) is used depends on whether  $j - i\beta \leq i'\beta - j$ , where  $i = \lfloor j/\beta \rfloor$  and  $i' = \lceil j/\beta \rceil$ . For  $C[0]$ ,  $j = 0$ . Then,  $i = i' = 0$  and (4) will be used:

$$C[0] = A_c[\lfloor 0/4 \rfloor] + \rho.$$

Since  $A_c[\lfloor 0/4 \rfloor] = A_c[0] = 0$  and the search of  $L[i\beta .. j] = L[0 .. 0]$  finds  $\rho = 0$ ,  $C[0]$  is equal to 0.

- To find the rank of the last appearance of 'C' in  $L[1 .. 8]$ , we will calculate  $C[8]$  by using (4) for the same reason as above. For  $C[8]$ , we have  $j = 8$  and  $i = 2$ . So we have

$$C[8] = A_c[\lfloor 8/4 \rfloor] + \rho.$$

Since  $A_c[\lfloor 8/4 \rfloor] = A_c[2] = 2$ , and the search of  $L[i\beta .. j] = L[8 .. 8]$  finds  $\rho = 0$ , we have  $C[8] = 2$ .

So the ranks of the first and the last appearances of 'C' are  $C[0] + 1 = 1$ , and  $C[8] = 2$ , respectively. Push  $\langle v_{11}, 1, 2 \rangle$  into  $S$ .

Next, for  $v_1$ , we will do the same work to find the first and last appearances of  $l(v_1) = 'A'$  and their respective ranks: 1 and 4; and push  $\langle v_1, 1, 4 \rangle$  into  $S$ . Now  $S$  contains two entries as shown in Figure 8(a) after step 2.

Step 3: pop out the top element  $\langle v_1, 1, 4 \rangle$  from  $S$ .  $v_1$  has two children  $v_2$  and  $v_9$ . Again, for  $v_9$  with  $l(v_9) = 'G'$ , we will use  $A_g$  to find the first and last appearances of  $G$  in  $L[2 .. 5]$  (corresponding to  $F_A[1 .. 4]$ ) and their respective ranks: 1 and 1. In the following, we show the whole working process.

- To find the rank of the first appearance of 'G' in  $L[2 .. 5]$ , we will first calculate  $G[1]$ . We have  $j = 1$ ,  $i = \lfloor j/\beta \rfloor = \lfloor 1/4 \rfloor = 0$  and  $i' = \lceil 1/4 \rceil = 1$ . Since  $j - i\beta = 0 < i'\beta - j = 3$ , formula (4) will be used:

$$G[1] = A_g[\lfloor 1/4 \rfloor] + \rho.$$

Since  $A_g[\lfloor 0/4 \rfloor] = A_g[0] = 0$  and search of  $L[i\beta .. j] = L[0 .. 0]$  finds  $\rho = 0$ ,  $G[1]$  is equal to 0.

- To find the rank of the last appearance of 'G' in  $L[2 .. 5]$ , we will calculate  $G[5]$  by using (4) based on an analysis similar to above. For  $G[5]$ , we have  $j = 5$  and  $i = \lfloor j/\beta \rfloor = 1$ . So we have

$$G[5] = A_g[\lfloor 5/4 \rfloor] + \rho.$$

Since  $A_g[\lfloor 5/4 \rfloor] = A_g[1] = 1$ , and search of  $L[i \cdot \beta .. j] = L[4 .. 5]$  finds  $\rho = 0$ , we have  $G[5] = 1$ .

We will push  $\langle v_9, G[1] + 1, G[5] \rangle = \langle v_9, 1, 1 \rangle$  into  $S$ .

For  $v_2$  with  $l(v_2) = 'C'$ , we will find the first and last appearances of  $C$  in  $L[2 .. 5]$  and their ranks: 1 and 2. Then, push  $\langle v_2, 1, 2 \rangle$  in to  $S$ . After this step,  $S$  will be changed as shown in Figure 8(b).

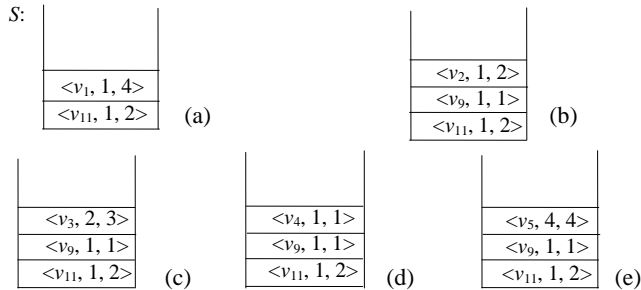


Figure 8. Illustration for stack changes

In the subsequent steps 4, 5, and 6,  $S$  will be consecutively changed as shown in Figure 8(c), (d), and (e), respectively.

In step 7, when we pop the top element  $\langle v_5, 4, 4 \rangle$ , we meet a node with a single child  $v_6$  labeled with  $\$$ . In this case, we will store  $\langle \gamma(v_6), l(v_5), 4, 4 \rangle = \langle r_1, A, 4, 4 \rangle$  in  $\mathcal{R}$  as part of the result (see line 7 in *searchRead*(.)). From this we can find that  $rk_L(A_3) = 4$  (note that the same element in both  $F$  and  $L$  has the same rank), which shows that in  $\bar{s}$  the substring of length  $|r_1|$  starting from  $A_3$  is an occurrence of  $r_1$ .  $\square$

### C. Time Complexity and Correctness Proof

In this subsection, we analyze the time complexity of *readSearch*( $T, LF, \beta$ ) and prove its correctness.

#### C.1 Time complexity

In the main **while**-loop, each node  $v$  in  $T$  is accessed only once. If the rankAll arrays are fully stored, only a constant time is needed to determine the range for  $l(v)$ . So the time complexity of the algorithm is bounded by  $O(|T|)$ . If only the compact arrays (for the rankAll information) are stored, the running time is increased to  $O(|T| \cdot \beta)$ , where  $\beta$  is the corresponding compact factor. It is because in this case, for each encountered node in  $T$ ,  $O(\frac{1}{2} \cdot \beta)$  entries in  $L$  may be checked in the worst case.

#### C.2 Correctness

**Proposition 1** Let  $T$  be a trie constructed over a collections of reads:  $r_1, \dots, r_m$ , and  $LF$  a BWT-mapping established for a reversed genome  $\bar{s}$ . Let  $\beta$  be the compact factor for the *allRank* arrays, and  $\mathcal{R}$  the result of *readSearch*( $T, LF, \beta$ ). Then, for each  $r_j$ , if it occurs in  $s$ , there is a quadruple  $\{\langle \gamma(v_i), l(v), a, b \rangle\} \in \mathcal{R}$  such that  $\gamma(v_i) = r_j$ ,  $l(v)$  is equal to the last character of  $r_j$ , and  $F_{l(v)[a]}, F_{l(v)[a+1]}, \dots, F_{l(v)[b]}$  show all the occurrences of  $r_j$  in  $s$ .

*Proof.* We prove the proposition by induction on the height  $h$  of  $T$ .

Basic step. When  $h = 1$ . The proposition trivially holds.

Induction hypothesis. Suppose that when the height of  $T$  is  $h$ , the proposition holds. We consider the case that the height of  $T$  is  $h + 1$ . Let  $v_0$  be the root with  $l(v_0) = \varepsilon$ . Let  $v_1, \dots, v_k$  be the children of  $v_0$ . Then,  $height(T[v_i]) \leq h$  ( $i = 1, \dots, k$ ), where  $T[v_i]$  stands for the subtree rooted at  $v_i$  and  $height(T[v_i])$  for the height of  $T[v_i]$ . Let  $l(v_i) = \alpha$  and  $F_\alpha = \langle \alpha; a, b \rangle$ . Let  $v_{i1}, \dots, v_{i\ell}$  be the children of  $v_i$ . Assume that  $x$  and  $y$  be the ranks of the first and last appearances of  $\alpha$  in  $L$ . According to the induction hypothesis, searching  $T[v_{ij}]$  against  $L[a' .. b']$ , where  $a' = a + x - 1$  and  $b' = a + y - 1$ , the algorithm will find all the locations of all those reads with  $l(v_i)$  as the first character. This completes the proof.  $\square$

## V. IMPROVEMENTS

The algorithm discussed in the previous section can be greatly improved by rearranging the search of a segment of  $L$  when we visit a node  $v$  in  $T$ . Such a search has to be done once for each of its children by calling *sDown*( ) or *sUp*( ) (see lines 10 - 11 in *readSearch*(.)). Instead of searching the segment for each child separately, we can manage to search the segment only once for all the children of  $v$ . To this end, we will use integers to represent characters in  $\Sigma$ . For example, we can use 1, 2, 3, 4, 5 to represent A, C, G, T, \$ in a DNA sequence. In addition, two kinds of simple data structures will be employed:

- $B_v$ : a Boolean array of size  $|\Sigma| \cup \{\$\}$  associated with node  $v$  in  $T$ , in which, for each  $i \in \Sigma$ ,  $B_v[i] = 1$  if there exists a child node  $u$  of  $v$  such that  $l(u) = i$ ; otherwise,  $B_v[i] = 0$ .
- $c_i$ : a counter associated with  $i \in \Sigma$  to record the number of  $i$ 's appearances during a search of some segment in  $L$ .

See Figure 9 for illustration.

With these data structures, we change *sDown*( $L, j, \beta, \alpha$ ) and *sUp*( $L, j, \beta, \alpha$ ) to *sDown*( $L, j, \beta, v$ ) and *sUp*( $L, j, \beta, v$ ), respectively, to search  $L$  for all the children of  $v$ , but only in one scanning of  $L$ .

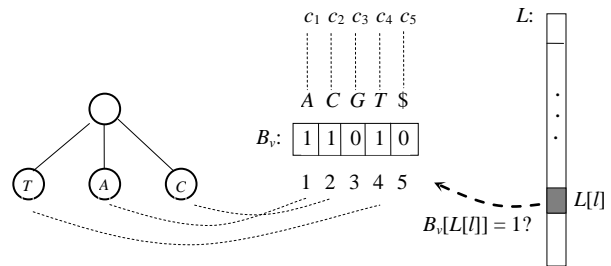


Figure 9. Illustration for extra data structures

In *sDown*( $L, j, \beta, v$ ), we will search a segment  $L[\lfloor j/\beta \rfloor \cdot \beta + 1 .. j]$  from top to bottom, and store the result in an array  $D$  of length  $|\Sigma|$ , in which each entry  $D[i]$  is the rank of  $i$  (representing a character), equal to  $c_i + A_i[\lfloor j/\beta \rfloor]$ , where  $c_i$  is the number of  $i$ 's appearances within  $L[\lfloor j/\beta \rfloor \cdot \beta + 1 .. j]$ .

In the algorithm,  $L[j' .. j]$  is scanned only once in the main **while**-loop (see lines 3 – 6), where  $j' = \lfloor j/\beta \rfloor \cdot \beta + 1$  (see line 2.) For each encountered entry  $L[l]$  ( $j' \leq l \leq j$ ), we will check whether  $B_v[L[l]] = 1$  (see line 4.) If it is the case,  $c_{L[l]}$  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).

---

**FUNCTION**  $sDown(L, j, \beta, v)$

---

**begin**

1.  $c_i \leftarrow 0$  for each  $i \in \Sigma$ ;
2.  $l \leftarrow \lfloor j/\beta \rfloor \cdot \beta + 1$ ;
3. **while**  $l \leq j$  **do** {
4.   **if**  $B_v[L[l]] = 1$  **then**  $c_{L[l]} \leftarrow c_{L[l]} + 1$ ;
5.    $l \leftarrow l + 1$ ;
6. }
7. **for**  $k = 1$  to  $|\Sigma|$  **do** {
8.   **if**  $B_v[k] = 1$  **then**  $D[k] \leftarrow A_k[\lfloor j/\beta \rfloor] + c_k$ ;
9. }
10. **return**  $D$ ;

**end**

---

Figure 10. Algorithm  $sDown()$ 

$sUp(L, j, \beta, v)$  is dual to  $sDown(L, j, \beta, v)$ , in which a segment of  $L$  will be search bottom-up.

---

**FUNCTION**  $sUp(L, j, \beta, v)$

---

**begin**

1.  $c_i \leftarrow 0$  for each  $i \in \Sigma$ ;
2.  $l \leftarrow \lceil j/\beta \rceil \cdot \beta$ ;
3. **while**  $l \geq j + 1$  **do** {
4.   **if**  $B_v[L[l]] = 1$  **then**  $c_{L[l]} \leftarrow c_{L[l]} + 1$ ;
5.    $l \leftarrow l - 1$ ;
6. }
7. **for**  $k = 1$  to  $|\Sigma|$  **do** {
8.   **if**  $B_v[k] = 1$  **then**  $D[k] \leftarrow A_k[\lceil j/\beta \rceil] - c_k$ ;
9. }
10. **return**  $D$ ;

**end**

---

Figure 11. Algorithm  $sUp()$ 

See the following example for illustration.

**Example 3** In this example, we trace the working process to generate ranges (by scanning  $L[2 .. 5]$ ) for the two children  $v_2$  and  $v_9$  of  $v_1$ . For this purpose, we will calculate  $C[1]$ ,  $C[5]$  for  $l(v_2) = 'C'$ , and  $G[1]$ ,  $G[5]$  for  $l(v_9) = 'G'$ . First, we notice that  $B_{v_1} = [0, 1, 1, 0, 0]$  and all the counters  $c_1, c_2, c_3, c_4, c_5$  are set to 0.

By running  $sDown(L, 1, 4, v_1)$  to get  $C[1]$  and  $G[1]$ , part of  $L$  will be scanned once, during which only one entry  $L[1] = 'A'$  (represented by 1) is accessed. Since  $B_{v_1}[L[1]] = B_{v_1}[1] = 0$ ,  $c_1$  remains unchanged. Especially, both  $c_2$  (for 'C') and  $c_3$  (for 'G') remain 0. Then,  $C[1] = A_c[\lfloor 1/4 \rfloor] + c_2 = 0$  and  $G[1] = A_g[\lfloor 1/4 \rfloor] + c_3 = 0$ .

By running  $sDown(L, 5, 4, v_1)$  to get  $C[5]$  and  $G[5]$ , another part of  $L$  will be scanned, also only once, during which merely one entry  $L[5] = 'C'$  (represented by 2) is

accessed. Since  $B_{v_1}[L[5]] = B_{v_1}[2] = 1$ ,  $c_2$  will be changed to 1. But  $c_3$  (for 'G') remain 0. Then, we have  $C[5] = A_c[\lfloor 5/4 \rfloor] + c_2 = 2$  and  $G[5] = A_g[\lfloor 5/4 \rfloor] + c_3 = 1$ .

Thus, the range for  $l(v_2) = 'C'$  is  $[C[1] + 1, C[5] = [1, 2]$ , and the range for  $l(v_9) = 'G'$  is  $[G[1] + 1, G[5] = [1, 1]$ .  $\square$

By using the above two procedures, our improved algorithm can be described as follows.

---

**ALGORITHM**  $rS(T, LF, \beta)$

---

**begin**

1.  $v \leftarrow root(T)$ ;
2.  $push(S, \langle v, 1, |\Sigma| \rangle)$ ;
3. **while**  $S$  is not empty **do** {
4.    $\langle v, a, b \rangle \leftarrow pop(S)$ ;
5.   let  $v_1, \dots, v_k$  be all those children of  $v$ , which are labeled with  $S$ ;
6.   let  $u_1, \dots, u_j$  be all the rest children of  $v$ ;
7.   **for** each  $j \in \{1, \dots, k\}$  **do** {  $\mathcal{R} \leftarrow \mathcal{R} \cup \{ \langle \gamma(v_j), l(v), a, b \rangle \}$ ;
8.   assume that  $F_{l(v)} = \langle l(v), x, y \rangle$ ;
9.    $x' \leftarrow x + a - 1$ ;  $y' \leftarrow x + b - 1$ ;
10.   call  $sDown(L, x' - 1, \beta, v)$  or  $sUp(L, x' - 1, \beta, v)$  to find the ranks of the first appearances of all the labels of the children of  $v$ :  $r(u_1), \dots, r(u_j)$ ;
11.   call  $sDown(L, y', \beta, v)$  or  $sUp(L, y', \beta, v)$  to find the ranks of the last appearances of all the labels of the children of  $v$ :  $r'(u_1), \dots, r'(u_j)$ ;
12.   **for**  $l = j$  **downto** 1 **do** {  $push(S, \langle u_l, r(u_l), r'(u_l) \rangle)$ ;
13. }

**end**

---

Figure 12. Algorithm  $rR()$ 

The main difference of the above algorithm from  $readSearch()$  consists in the different ways to search  $L[a .. b]$ . Here, to find the ranks of the first appearances of all the labels of the children of  $v$ ,  $sDown()$  or  $sUp()$  is called to scan part of  $L$  only once (while in  $readSearch()$  this has to be done once for each different child.) See line 10. Similarly, to find the ranks of the last appearances of these labels, another part of  $L$  is also scanned only once. See line 11. All the other operations are almost the same as in  $readSearch()$ .

## VI. EXPERIMENTS

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

- *Burrows Wheeler Transformation* (BWT for short),
- *Suffix tree based* (Suffix for short),
- *Hash table based* (Hash for short),
- *Trie-BWT* (tBWT for short, discussed in this paper),
- *Improved Trie-BWT* (itBWT for short, discussed in this paper).

Among them, the codes for the suffix tree based and hash based methods are taken from the *gsuffix* package [7] while all the other three algorithms are implemented by ourselves. All of them are able to find all occurrences of every read in a genome. The codes are written 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.

The test results are categorized in two groups: one is on a set of synthetic data and another is on a set of real data. For both of them, five reference genomes are used:

TABLE I. CHARACTERISTICS OF GENOMES

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

A. Tests on Synthetic Data Sets

All the synthetic data are created by simulating reads from the five genomes shown in Table I, with varying lengths and amounts. It is done by using the *wgsim* program included in the *SAMtools* package [33] with default model for single reads simulation.

Over such data, the impact of five factors on the searching time are tested: number  $n$  of reads, length  $l$  of reads, size  $s$  of genomes, compact factors  $f_1$  of *rankAlls* (see Subsection C in III) and compression factors  $f_2$  of suffix arrays [35][40], which are used to find locations of matching reads (in a reference genome) in terms of formula (6) (see Subsection D in III).

A.1 Tests with varying amount of reads

In this experiment, we vary the amount  $n$  of reads with  $n = 5, 10, 15, \dots, 50$  millions while the reads are 50 bps or 100 bps in length extracted randomly from *Rat chr1* and *C. merolae* genomes. For this test, the compact factors  $f_1$  of *rankAlls* are set to be 32, 64, 128, 256, and the compression factors  $f_2$  of suffix arrays are set to 8, 16, 32, 64, respectively. These two factors are increasingly set up as the amount of reads gets increased.

In Figures 13(a) and (b), we report the test results of searching the *Rat chr1* for matching reads of 50 and 100 bps, respectively. From these two figures, it can be clearly seen that the hash based method has the worst performance while ours works best. For short reads (of length 50 bps) the suffix-based is better than the BWT, but for long reads (of length 100 bps) they are comparable. The poor performance of the hash-based is due to its inefficient brute-force searching of genomes while for both the BWT and the suffix-based it is due to the huge amount of reads and each time only one read is checked. In the opposite, for both our methods tBWT and itBWT, the use of tries enables us to avoid repeated checkings for similar reads.

In these two figures, the time for constructing tries over reads is not included. It is because in the biological research a trie can be used repeatedly against different genomes, as well as often updated genomes. However, even with the time for constructing tries involved, our methods are still superior since the tries can be established very fast as demonstrated in

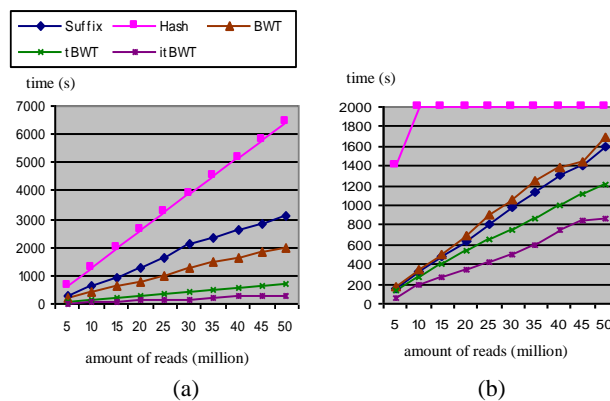


Figure 13. Test results on varying amount of reads

Table II, in which we show the times for constructing tries over different amounts of reads.

TABLE II. TIME FOR TRIE CONSTRUCTION OVER READS OF LENGTH 100 BPS

No. of reads	30M	35M	40M	45M	50M
Time for Trie Con.	51s	63s	82s	95s	110s

The difference between tBWT and itBWT is due to the different number of BWT array accesses as shown in Table III. By an access of a BWT array, we will scan a segment in the array to find the first and last appearance of a certain character from a read (by tBWT) or a set of characters from more than one read (by itBWT).

TABLE III. NO. OF BWT ARRAY ACCESSES

No. of reads	30M	35M	40M	45M	50M
tBWT	47856K	55531K	63120K	70631K	78062K
itBWT	19105K	22177K	25261K	28227K	31204K

Figures 14(a) and (b) show respectively the results for reads of length 50 bps and 100 bps over the *C. merolae* genome. Again, our methods outperform the other three methods.

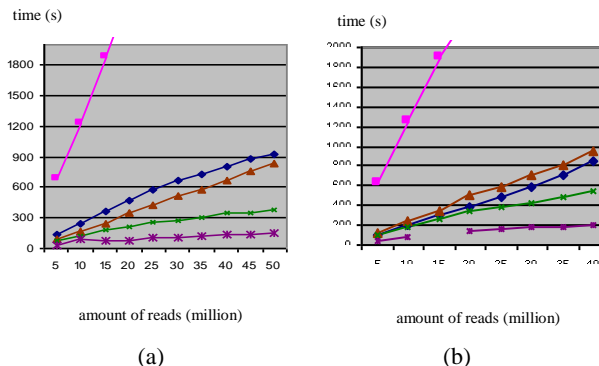


Figure 14. Test results on varying amount of reads

### A.2 Tests with varying length of reads

In this experiment, we test the impact of the read length on performance. For this, we fix all the other four factors but vary length  $l$  of simulated reads with  $l = 35, 50, 75, 100, 125, \dots, 200$ . The results in Figure 15(a) shows the difference among five methods, in which each tested set has 20 million reads simulated from the Rat chr1 genome with  $f_1 = 128$  and  $f_2 = 16$ . In Figure 15(b), the results show the the case that each set has 50 million reads. Figures 16(a) and (b) show the results of the same data settings but on *C. merlae* genome.

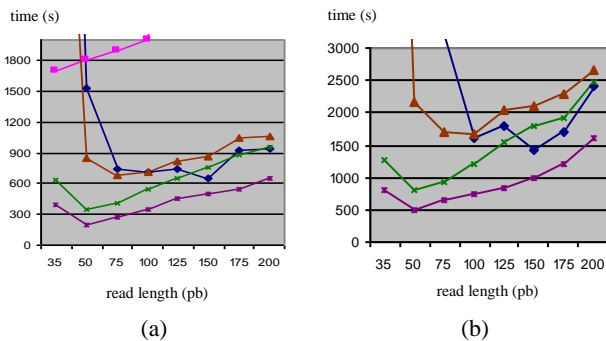


Figure 15. Test results on varying length of reads

Again, in this test, the hash based performs worst while the suffix tree and the BWT method are comparable. Both our algorithms uniformly outperform the others when searching on short reads (shorter than 100 bps). It is because shorter reads tend to have multiple occurrences in genomes, which makes the trie used in tBWT and itBWT more beneficial. However, for long reads, the suffix tree beats the BWT since on one hand long reads have fewer repeats in a genome, and on the other hand higher possibility that variations occurred in long reads may result in earlier termination of a searching process. In practice, short reads are more often than long reads.

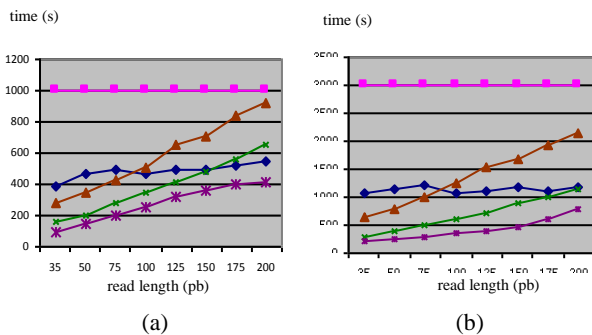


Figure 16. Test results on varying length of reads

### A.3 Tests with varying sizes of genome

To examine the impacts of varying sizes of genomes, we have made four tests with each testing a certain set of reads against different genomes shown in Table 1. To be consistent with foregoing experiments, factors except sizes of genomes

remain the same for each test with  $f_1 = 128$  and  $f_2 = 16$ . In Figure 17(a) and (b), we show the searching time on each genome for 20 million and 50 million reads of 50 bps, respectively. Figures 18(a) and (b) demonstrate the results of 20 million and 50 million reads but with each read being of 100 bps.

These figures show that, in general, as the size of a genome increases the time of read aligning for all the tested algorithms become longer. We also notice that the larger the size of a genome, the bigger the gaps between our methods and the other algorithms. The hash-based is always much slower than the others. For the suffix tree, we only show the matching time for the first three genomes. It is because the testing computer cannot meet its huge memory requirement for indexing the Zebra fish and Rat genomes (which is the main reason why people use the BWT, instead of the suffix tree, in practice.) Details for the 50 bp reads in Figure 17 and Figure 18 show that the tBWT and the itBWT are at least 30% faster than the BWT and the suffix tree, which happened on the *C. elegans* genome. For the Rat genome, our algorithms are even more than six times faster than the others.

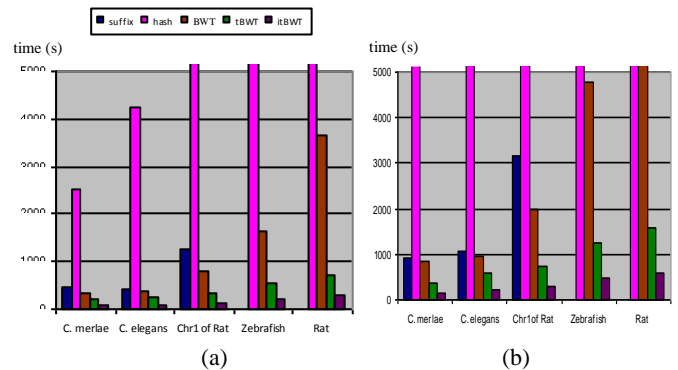


Figure 17. Test results on varying sizes of genomes

Now let us have a look at Figures 18(a) and (b). Although our methods do not perform as good as for the 50 bp reads due to the increment of length of reads, they still gain at least 22% improvement on speed and nearly 50% acceleration in the best case, compared with the BWT.

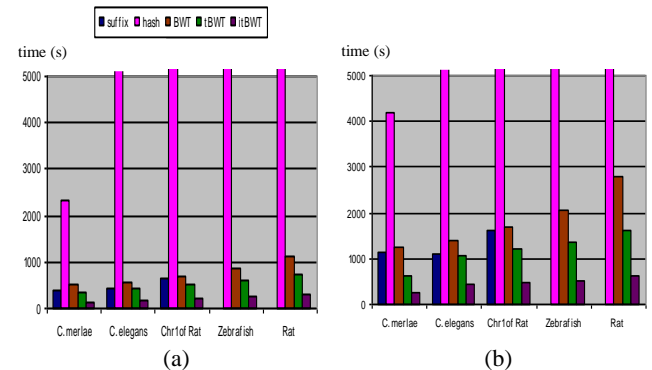


Figure 18. Test results on varying sizes of genomes

### A.4 Tests with varying compact and compression factors

In the experiments, we focus only on the BWT method, since there are no compressions in both the suffix tree and the hash-based method. The following test results are all for 20 million reads with 100 bps in length. We first show the impact of  $f_1$  on performance with  $f_2 = 16, 64$  in Figures 19(a) and (b), respectively. Then we show the effect when  $f_2$  is set to 64, 256 in Figures 20(a) and (b).

From these figures, we can see that the performance of all three methods degrade as  $f_1$  and  $f_2$  increase. Another noticeable point is that both the itBWT and the tBWT are not so sensitive to the high compression rate. Although doubling  $f_1$  or  $f_2$  will slow down their speed, they become faster compared to the BWT. For example, in Figure 19, the time used by the BWT grows 80% by increasing  $f_1$  from 8 to 64, whereas the growth of time used by the tBWT is only 50%. In addition, the factor  $f_1$  has smaller impact on the itBWT than the BWT and the tBWT, since the extra data structure used in the itBWT effectively reduced the processing time of the trie nodes by half or more.

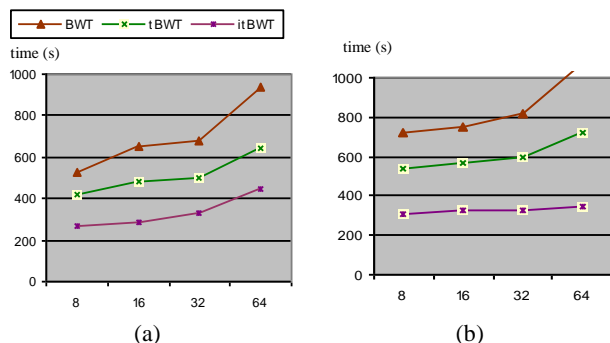


Figure 19. Test results on varying compact and compression factors

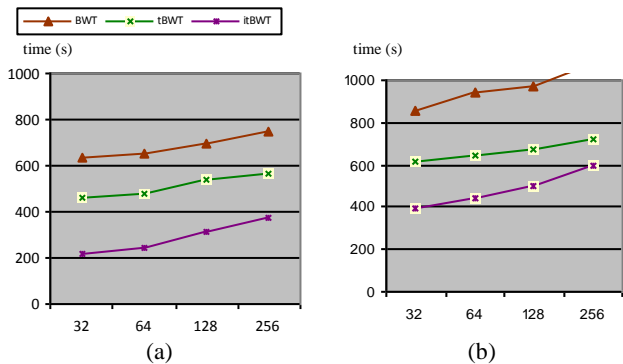


Figure 20. Test results on varying compact and compression factors

### B. Tests on Real Data Sets

For the performance assessment on real data, we obtain RNA-sequence data from the project conducted in an RNA laboratory at University of Manitoba [23]. This project includes over 500 million single reads produced by Illumina from a rat sample. Length of these reads are between 36 bps and 100 bps after trimming using Trimmomatic [8]. The reads

in the project are divided into 9 samples with different amount ranging between 20 million and 75 million. Two tests have been conducted. In the first test, we mapped the 9 samples back to rat genome of ENSEMBL release 79 [13]. We were not able to test the suffix tree due to its huge index size. The hash-based method was ignored as well since its running time was too high in comparison with the BWT. In order to balance between searching speed and memory usage of the BWT index, we set  $f_1 = 128, f_2 = 16$  and repeated the experiment 20 times. Figure 17(a) shows the average time consumed for each algorithm on the 9 samples.

Since the source of RNA-sequence data is the transcripts, the expressed part of the genome, we did a second test, in which we mapped the 9 samples again directly to the Rat *transcriptome*. This is the assembly of all transcripts in the Rat genome. This time more reads, which failed to be aligned in the first test, are able to be exactly matched. This result is showed in Figure 21(b).

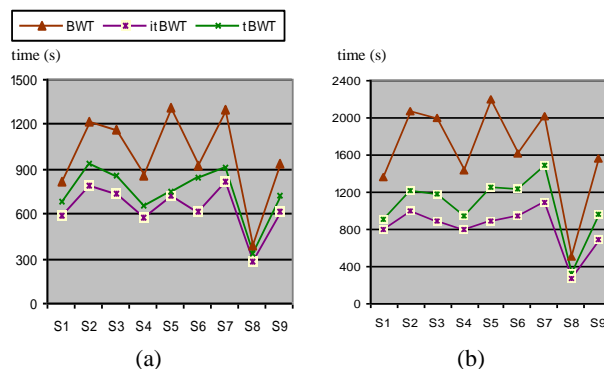


Figure 21. Test results on real data

From Figures 21(a) and (b), we can see that the test results for real data set are consistent with the simulated data. Our algorithms are faster than the BWT on all 9 samples. Counting the whole data set together, itBWT is more than 40% faster compared with the BWT. Although the performance would be dropped by taking tries' construction time into consideration, we are still able to save 35% time using itBWT.

### VII. CONCLUSION AND FUTURE WORK

In this paper, a new method to search a large volume of pattern strings against a single long target string is proposed, aiming at efficient next-generation sequencing in DNA databases. The main idea is to combine the search of tries constructed over the patterns and the search of the BWT indexes over the target. Especially, the so-called multiple-character checking has been introduced, which reduces the multiple scanning of a BWT array to a single search of it. Extensive experiments have been conducted, which show that our method improves the running time of the traditional methods by an order of magnitude or more.

As a future work, we will extend the discussed method to handle inexact string matches, such as the string matching with  $k$ -mismatches and  $k$ -errors, as well as patterns containing

'don't-cares'. It is very challenging to integrate the existing techniques for treating mismatches into the BWT-transformation.

## 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. 12<sup>th</sup> 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. Crochemore, 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, P.N. Robinson, *gsuffix*: <http://gsuffix.sourceforge.net/>, retrieved: April 2016.
- [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," <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.6177>, retrieved: 2016.
- [11] W.L. Chang and J. Lampe, "Theoretical and empirical comparisons of approximate string matching algorithms," in A. Apostolico, M. Crochemore, Z. Galil, and U. Manber (eds.) *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, Vol. 644, pp. 175-184, Springer-Verlag, Berlin, 1994.
- [12] L. Colussi, Z. Galil, and R. Giancarlo, "On the exact complexity of string matching," *Proc. 31<sup>st</sup> Annual IEEE Symposium of Foundation of Computer Science*, Vol. 1, pp. 135-144, 1990.
- [13] F. Cunningham, et al., "Nucleic Acids Research," 2015, 43, Database issue: D662-D669.
- [14] S.R. Eddy, "What is dynamic programming?" *Nature Biotechnology* 22, 909-910, (2004) doi:10.1038/nbt0704-909.
- [15] A. Ehrenfeucht and D. Haussler, "A new distance metric on strings computable in linear time," *Discrete Applied Mathematics*, Vol. 20, pp. 191-203, 1988.
- [16] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41<sup>st</sup> Annual Symposium on Foundations of Computer Science*, pp. 390-398, IEEE, 2000.
- [17] 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.
- [18] M.C. Harrison, "Implementation of the substring test by hashing," *Communication of the ACM*, Vol. 14, No. 12, pp. 777-779, 1971.
- [19] H. Jiang, and W.H. Wong, "SeqMap: mapping massive amount of oligonucleotides to the genome," *Bioinformatics*, 24, 2395-2396, 2008.
- [20] 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.
- [21] D.E. Knuth, *The Art of Computer Programming*, Vol. 3, Massachusetts, Addison-Wesley Publish Com., 1975.
- [22] 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.
- [23] lab website: <http://home.cc.umanitoba.ca/~xie/j/>, retrieved: April 2016.
- [24] G.M. Landau and U. Vishkin, "Efficient string matching in the presence of errors," *Proc. 26<sup>th</sup> Annual IEEE Symposium on Foundations of Computer Science*, pp. 126-136, 1985.
- [25] G.M. Landau and U. Vishkin, "Efficient string matching with  $k$  mismatches," *Theoretical Computer Science*, Vol. 43, pp. 239-249, 1986.
- [26] B. Langmead, "Introduction to the Burrows-Wheeler Transform," [www.youtube.com/watch?v=4n7NPK5lwbI](http://www.youtube.com/watch?v=4n7NPK5lwbI), retrieved: April 2016.
- [27] T. Lecroq, "A variation on the Boyer-Moore algorithm," *Theoretical Computer Science*, Vol. 92, No. 1, pp. 119-144, Jan. 1992.
- [28] H. Li, et al., "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Res.*, 18, 1851-1858, 2008.
- [29] R. Li, et al., "SOAP: short oligonucleotide alignment program," *Bioinformatics*, 24, 713-714, 2008.
- [30] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler Transform," *Bioinformatics*, Vol. 25 no. 14 2009, pp. 1754-1760.
- [31] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler Transform," *Bioinformatics*, Vol. 26 no. 5 2010, pp. 589-595.
- [32] 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.
- [33] H. Li, "wgsim: a small tool for simulating sequence reads from a reference genome," <https://github.com/lh3/wgsim/>, 2014.
- [34] H. Lin, et al., "ZOOM! Zillions of oligos mapped," *Bioinformatics*, 24, 2431-2437, 2008.
- [35] U. Manber and E.W. Myers, "Suffix arrays: a new method for on-line string searches," *Proc. the 1<sup>st</sup> Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319-327, SIAM, Philadelphia, PA, 1990.
- [36] 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.
- [37] E.M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, Vol. 23, No. 2, pp. 262-272, April 1976.
- [38] 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.
- [39] M. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," *Bioinformatics*, 25, 1363-1369, 2009.
- [40] J. Seward, "bzip2 and libbzip2, version 1.0.5: A program and library for data compression," URL <http://www.bzip.org>, 2007.
- [41] A.D. Smith, et al, "Using quality scores and longer reads improves accuracy of Solexa read mapping," *BMC Bioinformatics*, 9, 128, 2008.
- [42] J. Tarhio and E. Ukkonen, "Boyer-Moore approach to approximate string matching," in J.R. Gilbert and R. Karlsson (eds.) *SWAT 90, Proc. 2<sup>nd</sup> Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science*, Vol. 447, pp. 348-359, Springer-Verlag, Berlin, 1990.
- [43] J. Tarhio and E. Ukkonen, "Approximate Boyer-Moore String Matching," *SIAM Journal on Computing*, Vol. 22, No. 2, pp. 243-260, 1993.
- [44] E. Ukkonen, "Approximate string-matching with  $q$ -grams and maximal matches," *Theoretical Computer Science*, Vol. 92, pp. 191-211, 1992.
- [45] P. Weiner, "Linear pattern matching algorithm," *Proc. 14<sup>th</sup> IEEE Symposium on Switching and Automata Theory*, pp. 1-11, 1973.
- [46] Y. Chen, D. Che and K. Aberer, "On the Efficient Evaluation of Relaxed Queries in Biological Databases," in *Proc. 11th Int. Conf. on Information and Knowledge Management, Virginia, U.S.A.: ACM*, Nov. 2002, pp. 227-236.