Searching BWT against Pattern Matching Machine to Find Multiple String Matches

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

***Abstract*—In this paper, we discuss an indexing method for solving the multiple string pattern matching problem, by which we are given a set of short strings $R = \{r_1, \ldots, r_l\}$ and required to locate all substrings of a target string s such that each of them matches an r_j in R. The main idea is to construct a *pattern matching machine* A and transform the reverse $\bar{s}$ of s to a BWT-array as an index, denoted as $BWT(\bar{s})$, and search A against it. During the process, the failure function of A is used to decrease the subranges of $BWT(\bar{s})$ to be searched at each step. In addition, we change a single-character checking against $BWT(\bar{s})$ to a multiple-character checking, by which multiple searches of $BWT(\bar{s})$ are reduced to a single scanning. In this way, high efficiency can be achieved. Extensive experiments have been conducted, which shows that our method works better than almost all the existing methods for this problem.**

***Keywords*—string matching; DNA sequences; multiple pattern machine; automaton; BWT-transformation**

I. Introduction

By the multiple string pattern matching problem, we will be given a set $R = \{r_1, \ldots, r_l\}$, where each r_i ($1 \leq i \leq l$) is a (short) string (or say, a finite sequence of symbols) called a pattern, and a target (long) string s. We are required to locate and identify all substrings of s which are patterns in R. This problem becomes very important as the next-generation sequencing technique [4] comes into use, which needs to align a huge number of reads (short DNA sequences) against a very long sequence, known *genome*, which is previously well studied and often billions of characters long, for earlier diagnosis of cancers, or some other purposes. Normally, the number of reads is multiple millions and the length of a read is about 100 characters (pbs).

This problem was studied as early as mid-1970's. In [1], Aho and Corasick proposed an efficient algorithm for solving this problem, by which a pattern matching machine (*PMM* for short) or an automaton A is constructed over R and then searched against s by successively reading the characters in s, making state transition and occasionally reporting output. The running time of this process is bounded by $O(\sum_{i=1}^{l} |r_i| + |s|)$. This algorithm has been extensively used in practise, such as bioinformatics [7, 8], multiple key-word searching [9] and two-dimensional pattern searching [2]; and also improved by different researchers, such as those discussed in [6, 16, 17, 18]. However, the worst-case time complexity remains unchanged.

On the other hand, different indexes have been developed for a single string pattern searching in the past several decades, such as *suffix trees* [14, 15], *suffix arrays* [13], *hashing* [10], and *BWT-arrays* [5, 11]. However, no effort has been made to build indexes over s to expedite the multiple string pattern matching defined above.

In this paper, we address this issue. We will show that an index over s, the so-called *BWT* array, can be established quickly, and can also be used to speed up scanning of s when we search A in some way to bring down the running time to $O(\sum_{i=1}^{l} |r_i|)$. (This time complexity does not include the time for taking an index into main memory from hard disk. However, in practice, this part of time can be completely ignored. For example, for reading the BWT array of a genome of 1,464,443,456 bytes, only 3 milliseconds are used.)

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

1) *Subrange reduction*. During a search of A against the BWT array L for $\bar{s}$ (the reverse of s), a series of subranges within L will be checked. By using the failure function of A, we design a mechanism to reduce each of them effectively.
2) Change a single-character checking to a multiple-character checking. (That is, each time a set of characters respectively from more than one pattern string will be checked against L in one scan, instead of checking them separately one by one in multiple scans.)

Our extensive experiments show that the new method can improve the running time of the existing methods by 40%.

The remainder of the paper is organized as follows. In Section II, we briefly describe *PMM* and *BWT*, based on which our method is established. In Section III, we discuss our algorithm to find all occurrences of a set of string patterns in a target string. In Section IV, we discuss two possible improvements. 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. Basic Techniques: *PMM* and *BWT*

In this section, we briefly describe two basic techniques used in our method. They are the pattern matching machine and the *BWT* transformation.

978-1-5386-2209-4/17 $31.00 © 2017 IEEE
DOI 10.1109/CyberC.2017.26

A. *Pattern Matching Machine*

Similar to the Aho-Corasik's algorithm, we will first construct a pattern matching machine (*PMM*) $\boldsymbol{A}$ over $\boldsymbol{R} = \{r_1, \ldots, r_l\}$. Different from it, however, we will not search $\boldsymbol{A}$ against s, but against $BWT(\bar{s})$.

Intuitively, the pattern matching machine $\boldsymbol{A}$ over $\boldsymbol{R} = \{r_1, \ldots, r_l\}$ can be considered as a directed graph composed of two parts: a *trie T*, denoted as *trie*($\boldsymbol{R}$), and a *failure function* $f(v)$ $(v, f(v) \in \boldsymbol{A})$.

First, for each r_j $(j = 1, \ldots, m)$ we will attach \$ to its end and construct *trie*($\boldsymbol{R}$) as below.

If $|\boldsymbol{R}| = 0$, *trie*($\boldsymbol{R}$) is, of course, empty. For $|\boldsymbol{R}| = 1$, *trie*($\boldsymbol{R}$) is a single node. If $|\boldsymbol{R}| > 1$, $\boldsymbol{R}$ is split into $|\Sigma| = k$ (possibly empty) subsets $\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_k$ so that each $\boldsymbol{R}_i$ $(i \in \{1, \ldots, k\})$ contains all those sequences with the same first character $x_i \in \boldsymbol{R} \cup \{\$\}$. The tries: *trie*($\boldsymbol{R}_1$), *trie*($\boldsymbol{R}_2$), …, *trie*($\boldsymbol{R}_k$) are constructed in the same way except that at the *i*th step, the splitting of sets is based on the *i*th characters in the sequences. They are then connected from their respective roots to a single node to create *trie*($\boldsymbol{R}$).

Example 1 As an example, consider a set of four pattern strings:

r_1: *acaga*
r_2: *ag*
r_3: *acagc*
r_4: *ca*

For these pattern strings, a trie can be constructed as shown in Fig. 1(a). In this trie, v_0 is a virtual root, representing an *empty* string while any other node v stands for a string equals the concatenation of all characters labelling the nodes on the path from v_0 to v, denoted as $P(v)$. Especially, if v is a leaf, $P(v)$ must be a string in $\boldsymbol{R}$. For instance, the path from v_0 to v_8 corresponds to the third pattern r_3 = *acagc*\$. In general, however, we will associate some nodes v with an *output* such that each $r \in output(v)$ is a string in $\boldsymbol{R}$ and also a suffix of $P(v)$. For example, for v_3 shown in Fig. 1(a), we have $output(v_3) = \{r_4\}$. It is because r_4 = *'ca'* $\in \boldsymbol{R}$ is a suffix of $P(v_3)$ = *'aca'*.

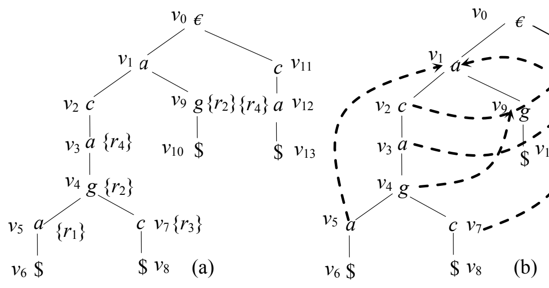


Figure 1. A trie and a pattern matching machine

In addition, for a node v, we will use $l(v)$ to represent its character.

By the *failure function* $f(v)$ (defined over $v \in T \backslash \{v_0\}$), we will give the node to be entered at a mismatch of $P(v)$. Specifically, $f(v)$ is the node labeled by the longest proper suffix w of $P(v)$ such that w is a prefix of some pattern [1], as illustrated by the dashed arrows in Fig. 1(b). For example, $f(v_3) = v_{12}$ is represented by the dashed arrow from v_3 to v_{12}. We have this since *'ca'* is a suffix of $P(v_3)$, which is a prefix of r_4, represented by $P(v_{12})$.

Formally, we have $\boldsymbol{A} = T \cup \{f(v) \mid v \in T \backslash \{v_0\}\}$. We will also simply use $f(v)$ to represent a link from v to $f(v)$.

B. *BWT and String Searching*

Now, we describe the *BWT* transformation. We will use s to denote a string that we would like to transform. Again, assume that s terminates with \$, 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 = *ccagaca*\$.

First, we can rotate s consecutively to create eight different strings, and put them in a matrix as illustrated in Fig. 2(a).

		F	L
$c\ c\ a\ g\ a\ c\ a\ \$$	$\$\ c_1\ c_2\ a_1\ g_1\ a_2\ c_3\ a_3$	\$	a_3
$c\ a\ g\ a\ c\ a\ \$\ c$	$a_3\ \$\ c_1\ c_2\ a_1\ g_1\ a_2\ c_3$	a_3	c_3
$a\ g\ a\ c\ a\ \$\ c\ c$	$a_2\ c_3\ a_3\ \$\ c_1\ c_2\ a_1\ g_1$	a_2	g_1
$g\ a\ c\ a\ \$\ c\ c\ a$	$a_1\ g_1\ a_2\ c_3\ a_3\ \$\ c_1\ c_2$	a_1	c_2
$a\ c\ a\ \$\ c\ c\ a\ g$	$c_3\ a_3\ \$\ c_1\ c_2\ a_1\ g_1\ a_2$	c_3	a_2
$c\ a\ \$\ c\ c\ a\ g\ a$	$c_2\ a_1\ g_1\ a_2\ c_3\ a_3\ \$\ c_1$	c_2	c_1
$a\ \$\ c\ c\ a\ g\ a\ c$	$c_1\ c_2\ a_1\ g_1\ a_2\ c_3\ a_3\ \$$	c_1	\$
$\$\ c\ c\ a\ g\ a\ c\ a$	$g_1\ a_2\ c_3\ a_3\ \$\ c_1\ c_2\ a_1$	g_1	a_1
(a)	(b)	(c)	

Figure 2. Rotation of a string

Next, we sort the rows of the matrix alphabetically, and get another matrix, as demonstrated in Fig. 2(b), which is called the *Burrow-Wheeler Matrix* [5, 11] 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 = *ccagaca*\$, we have *BWT*(s) = *acgcac*\$*a* (see Fig. 2(c)). The first column is referred to as F.

Special attention should be paid to Fig. 2(b) and 2(c). In both of them, for ease of explanation, the position of a character in s is represented by its subscript. (That is, we rewrite s as $c_1c_2a_1g_1a_2c_3a_3\$$.) For example, a_2 represents 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.

In addition, when ranking the elements x in both F and L in such a way that if x is the *i*th appearance of a certain character it will be assigned i, the same element will get the same number in the two columns. For instance, in F the rank of a_3, denoted as $rk_F(a_3)$, is 1 (showing that a_3 is the first appearance of a in F). Its rank in L, $rk_L(a_3)$ 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 = *ccagaca*\$.

First, we notice that we can store F as $|\Sigma| + 1$ intervals, such as $F_\$ = F[1 .. 1]$, $F_a = F[2 .. 4]$, $F_c = F[5 .. 7]$, $F_g = F[8 .. 8]$, and $F_t = \Phi$ for the above example (see Fig. 2(c).) We can also represent a segment within an F_x (with $x \in \Sigma$) as a pair of the form $<x, [\alpha, \beta]>$, where $\alpha \le \beta$ are two ranks of x. Thus, we have $F_a = F[2 .. 4] = <a, [1, 3]>$, $F_c = F[5 .. 7] = <c, [1, 3]>$, and $F_g = F[8 .. 8] = <g, [1, 1]>$.

We will also use Y_x and Z_x to represent the positions of the first and the last element of F_x in F, respectively. For example, Y_a is 2 and Z_a is 4. Also, we can use L_π to represent a range in L corresponding to a pair $\pi = <x, [\alpha, \beta]>$. For example, in Fig. 2(c), $L_{<a, [1, 3]>} = L[2 .. 4]$, $L_{<c, [1, 2]>} = L[5 .. 6]$. $L_{<a, [2, 3]>} = L[3 .. 4]$, and so on.

Finally, we 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:

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

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

Step 1 (*checking the last character in r*): Check $r[3] = a$ in the pattern string r, and then figure out $F_a = F[2 .. 4] = <a, [1, 3]>$.

Step 2 (*checking the second character from last*): Check $r[2] = c$. Call $search(c, L_{<a, [1, 3]>})$. It will search $L_{<a, [1, 3]>} = L[2 .. 4]$ to find a range bounded by the first and last rank of c. Concretely, we will find $rk_L(c_3) = 1$ and $rk_L(c_2) = 2$. So, $search(c, L_{<a, [1, 3]>})$ returns $<c, [1, 2]>$. It is $F[5 .. 6]$.

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

The above working process can be represented as a sequence of three pairs:

$<a, [1, 3]>, <c, [1, 2]>, <a, [2, 2]>$.

In general, for $\bar{r} = c_1 \ldots c_m$, its search against $BWT(s)$ can always be represented as a sequence of pairs (with each representing a segment in F_x for some $x \in \Sigma$):

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

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 $1 < i \le m$. We call such a sequence as a *search sequence*. Thus, the time used for this process is bounded by $O(\sum_{i=1}^{m} \tau_i)$, where τ_i is the time for an execution of $search(x_i, L_{<x_{i-1}, [\alpha_{i-1}, \beta_{i-1}]>})$. However, this time complexity can be reduced to $O(m)$ by using the so-called *rankAll* method (with more space to be used [11]) and the multi-character checking to be discussed in Section IV.

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

Finally, we point out that $BWT(s)$ (or $BWT(\bar{s})$) can be constructed in $O(|s|)$ time by using its relationship to the suffix array of s [5].

III. MAIN ALGORITHM

In this section, we present our main algorithm. First, we show a breadth-first search of $trie(\boldsymbol{R})$ against $BWT(\bar{s})$ in Subsection *A*. Then, in Subsection *B*, we discuss how the failure functions in an *PMM* can be used to speed up the working process. Subsection *C* is devoted to the correctness proof and the time complexity analysis.

A. Searching Tries over Pattern Strings

It is easy to see that exploring a path in a trie T over $\boldsymbol{R}$ corresponds to scanning a pattern $r \in \boldsymbol{R}$. If we explore, at the same time, the L array (= $BWT(\bar{s})$) established over a *reversed* target sequence $\bar{s}$, we will find all the occurrences of r (without \$ involved) in s. Obviously, by a depth-first search of T, this can be done very efficiently. However, to use the failure function to reduce the subrange (within L) to be checked at each step, we need to explore T in the breadth-first manner. For this purpose, we use a queue Q to control the process, in which each entry is a triplet $<v, a, b>$ with v being a node in T and $a \le b$, used to indicate a subsegment within $F_{l(v)}$. For example, when searching the trie shown in Fig. 1(a) against the L array shown in Fig. 2(c), we may have an entry like $<v_1, 1, 3>$ in Q to represent a subsegment $F_a[1 .. 3]$ (the first to the third entry in F_a) since $l(v_1) = 'a'$. In addition, for technical convenience, we use F_ϵ to represent the whole F. Then, $F_\epsilon[a .. b]$ represents the segment from the ath to the bth entry in F.

ALGORITHM *trieSearch*(T, LF)

```
begin
1.  $v \leftarrow root(T)$; $\Re \leftarrow \Phi$;
2.  $enqueue(Q, <v, 1, |s|>)$;
3.  while $Q$ is not empty do {
4.    $<v, a, b> \leftarrow dequeue(S)$;
5.    if $output(v) \ne \Phi$ then $\Re \leftarrow \Re \cup \{<output(v), l(v), a, b>\}$;
6.    let $v_1, \ldots, v_k$ be the children of $v$;
7.    for $i = 1$ to $k$ do {
8.      {denote $F_{l(v)}[a .. b]$ by $\pi$; let $x = l(v_i)$;
9.       if $search(x, \pi) \ne \phi$ then
10.       {let $search(x, \pi) = <x, [\alpha, \beta]>$; $enqueue(Q, <v_i, \alpha, \beta>)$; }
11.   }
12. }
end
```

In the algorithm, we first enqueue <$root(T)$, 1, $|s|$> into queue Q (append at the end of Q) (lines 1 – 2). Then, we go into the main **while-loop** (lines 3 – 12), in which we will first dequeue the first element from Q (taken out from the front of Q), stored as a triplet <v, a, b> (line 4). Then, we will check whether $output(v)$ is empty. If it is not the case, a quadruple <$output(v)$, $l(v)$, a, b> will be added to the result $\Re$ (see line 5), which records all the occurrences of all those pattern strings represented by $output(v)$ in s. (In practice, we store compressed suffix arrays [13, 15] and use their relationship with BWT to calculate positions of pattern strings in s.) For each child v_i of v, we will determine a segment in L by executing $search(x, \pi)$, where $x = l(v_i)$ and π = <$l(v)$, [a .. b]> (= $F_{l(v)}[a .. b]$). Let $search(x, \pi)$ = <x, [α, β]>. We will then enqueue each <v_i, α, β> into Q. (see line 10.)

Example 2 Consider all pattern strings given in Example 1 again. The trie T over these short strings are shown in Fig. 1(a). In order to find all the occurrences of them in s = $ccagaca\$$, we will run *trieSearch*( ) on T and the LF of $\bar{s}$ shown in Fig. 2(c). (By LF, we mean the L and F arrays together.)

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

Step 1: Enqueue <v_0, 1, 8> into Q, as illustrated in Fig. 3(a).

Step 2: Dequeue the first element <v_0, 1, 8> from Q. Figure out the two children of v_0: v_1 and v_{11}. First, for v_1, we have $l(v_1)$ = a. By executing $search(a, F_\epsilon[1 .. 8])$, we get <$a$, [1, 3]> and then enqueue < v_1, 1, 3> into Q. For v_{11}, we have $l(v_{11}) = c$ and get <c, [1, 3]> by executing $search(c, F_\epsilon[1 .. 8])$. So, <$v_{11}$, 1, 3> will also be enqueued into Q. See Fig. 3(b) for illustration.

Step 3: Dequeue the first element <v_1, 1, 3> from Q. v_1 also has two children: v_2 and v_9. For v_2, we have $l(v_2) = c$. By executing $search(c, F_a[1 .. 3])$, we get <c, [1, 2]>. For v_9, we have $l(v_9) = g$ and get <g, [1, 1]> by executing $search(g, F_a[1 .. 3])$. Similarly, we will consecutively enqueue <v_2, 1, 2> and <v_9, 1, 1> into Q. See Fig. 3(c).

The remaining steps 4, 5, 6, 7, 8, 9 will be done in the same way as above and Q will be accordingly changed as shown in Fig. 3(d), (e), (f), (g), (h), and (i), respectively. Here, special attention should be paid to Step 5 when <v_9, 1, 1> is dequeued from Q. Since $output(v_9) = \{r_2\}$, we will store <$\{r_2\}$, g, 1, 1> in $\Re$ as part of the result (see line 5 in *trieSearch*( )), which shows that r_2 appears at g_1-possition in s. □

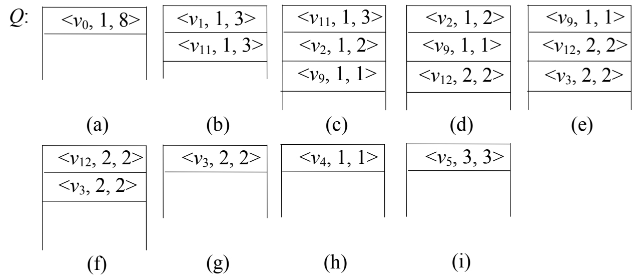


Figure 3. Illustration for Step 1 - 9

B. Searching PMMs over Pattern Strings

In the algorithm discussed in the previous subsection, the failure function is totally ignored. Indeed, due to the difference between the scanning of s and the searching of $BWT(\bar{s})$, the failure function cannot be used in a way as the Aho-Corasik's algorithm [1]. It is because by searching $BWT(\bar{s})$, what we will produce is a sequence of pairs, and two such sequences corresponding to a same sequence of characters (respectively along two paths in T) may have different sequences of intervals. For instance, along the path $v_2 \rightarrow v_3$ shown in Fig. 4, we will create a sequence of pairs: <c, [1, 2]>, <a, [2, 2]> while along the path $v_{11} \rightarrow v_{12}$ the sequence generated is <c, [1, 3]>, <a, [2, 2]>. Although they have the same sequence of characters: ca, their sequences of intervals are different: one is [1, 2][2, 2] and the other is [1, 3][2, 2].

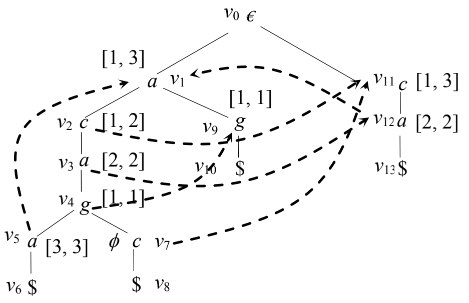


Figure 4. Illustration for intervals in a PMM

In addition, since a pair sequence cannot be created in a reverse order (by searching a *PMM* bottom-up), it is completely impossible for us to use the skip-table utilized in the Boyler-Moore's algorithm [19] (by which substrings of s need to be scanned backwards), or the DAWG structure (directed acyclic word graph) in the Crochemore's algorithm [18] (by which a DAWG needs to be searched bottom-up.) However, the failure function can be really employed to reduce the size of subranges of L to be searched during an execution of *search*( ).

To this end, we will associate each node v in T with the corresponding interval [α_v, β_v], referred to as $I(v)$, which is found for $l(v)$ by running *search*( ). That is, along an edge $w \rightarrow v$ in T, we will have $search(l(v)$, <$l(w)$, [α_w, β_w]>) = <$l(v)$, [α_v, β_v]>. (See Fig. 4 for illustration.) With the help of such intervals, the failure function can be utilized as follows.

Lemma 1 Let u, v be two nodes in $\boldsymbol{A}$ such that $f(v) = u$. Then, $I(v) \subseteq I(u)$.

Proof. According to the definition of $f(v) = u$, $P(u)$ is a suffix of $P(v)$. Assume that $P(v) = x_1 \ldots x_i x_{i+1} \ldots x_{i+j}$ and $P(u) = x_{i+1} \ldots x_{i+j}$ with $i, j \geq 0$. Then, by the execution of $trieSearch(T, LF)$, along $P(v)$ and $P(u)$, two sequences of pairs will be generated:

$\pi_1, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_{i+j}$; and

$\pi_1', \ldots, \pi_j'$,

where π_1 = <x_1, F_{x_1}>, π_1' = <$x_{i+1}, F_{x_{i+1}}$>, $\pi_{l+1} = search(x_l, \pi_l)$ (l = 1, …, $i + j$ – 1), and $\pi_{k+1}' = search(x_k, \pi_k')$ ($k = i + 1, \ldots, i + j$ -1). Let I_l be the interval in pair π_l (l = 1, …, $i + j$). Let I_k'

be the interval in pair π_k' ($k = i + 1, \ldots, i + j$). We must have $I_k \subseteq I_k'$ ($k = i + 1, \ldots, i + j$). Thus, $I(v) = I_{i+j} \subseteq I_j' = I(u)$. □

This lemma enables us to design an efficient procedure to replace *search*( ) for creating $I(v)$'s as follows.

Let $w \rightarrow v$ be an edge in T. Assume that $f(v) = u$. Let $l(u) = x$. Since we explore T in the breadth-first manner, u must be visited before v. So its interval $I(u) = [\alpha_u, \beta_u]$ must have been created when we meet v. Then, in terms of α_u, we can find an integer j such that $\alpha_u = rk_L(x_j)$ (recall that x_j represents the jth appearance of x in s). Next, in terms of j, we can obtain another integer i such that $x_j = L[i]$. Finally, using i, α_v can be immediately determined:

1. If $Y_{l(w)} + \alpha_w - 1 \leq i$, we will simply set α_v equal to α_u. (Recall that $Y_{l(w)}$ represents the position of the first entry of $F_{l(w)}$ in F.) It is because when searching a subrange in L, which corresponds to $I(w)$, we will definitely meet $L[Y_{l(w)} + \alpha_u - 1]$ as the first character equal to x according to Lemma 1.
2. Otherwise ($Y_{l(w)} + \alpha_w > i$), we have to search L starting from $L[Y_{l(w)} + \alpha_w - 1]$ downwards to find the first appearance of x and use it as α_v.

Similarly, Let i', j' be two integers such that $\beta_u = rk_L(x_{j'})$ and $x_{j'} = L[i']$.

3. If $Y_{l(w)} + \beta_w - 1 \geq i'$, we will simply set β_v equal to β_u.
4. Otherwise ($Y_{l(u)} + \beta_u < i'$), we have to search L starting from $L[Y_{l(w)} + \beta_w - 1]$ upwards to find the last appearance of x and use it as β_v.

As an example, consider the search of A shown in Fig. 4 against the LF of $\bar{s}$ shown in Fig. 2(c), where $s = ccagaca\$$. By the breadth-first search of T, v_1 and v_{11} will be visited before v_2. $f(v_2) = v_{11}$. As shown in Fig. 4(a), we have $I(v_1) = [1, 3]$ and $I(v_{11}) = [1, 3]$, and $v_1 \rightarrow v_2$ is an edge in T. According to the above discussion, $I(v_2) = [\alpha_{v_2}, \beta_{v_2}]$ will be determined as follows.

- To find α_{v_2}, we will first compare $Y_a + \alpha_{v_1} - 1$ and i, where $rk_L(L[i]) = \alpha_{v_{11}} = 1$. Since $Y_a + \alpha_{v_1} - 1 = 2 + 1 - 1 = 2$, which is equal to $i = 2$, α_{v_2} should be set equal to $\alpha_{v_{11}}$ and no search will be conducted to find this value. (Here, we have $i = 2$ since $l(v_{11}) = 'c'$ and $L[2] = 'c_3'$ with rank equal to $\alpha_{v_{11}} = 1$. See Fig. 4.)
- To determine β_{v_2}, we will compare $Y_a + \beta_{v_1} - 1$ and i', where $rk_L(L[i']) = \beta_{v_{11}} = 3$. Since $Y_a + \beta_{v_1} - 1 = 2 + 3 - 1 = 4 < i' = 6$, we need to search L starting from $L[Y_a + \beta_{v_1} - 1] = L[4]$ upwards to find the last appearance of c within a range (in L) corresponding to $<a, [\alpha_{v_1}, \beta_{v_1}]>$. It is $L[3] = c_2$ (with $rk_L(c_2) = 2$.) So, we get $\beta_{v_2} = 2$.

Now, we further consider the evaluation of $I(v_3)$. Assume that $I(v_{12}) = [2, 2]$ has already been established. By doing a checking similar to the above, we will immediately get $I(v_{12}) = [2, 2]$, no searching of L at all. In this way, a lot of time can be saved.

We will refer to the above process as *searchI*(v, w, $f(v)$, LF) to indicate its difference from *search*( ). Its output is an interval to be associated with v.

According to the above discussion, we give the following algorithm, which works almost in the same way as *trieSearch*( ). The only difference is in the use of *searchI*( ). That is, we will still explore T *breadth-first*. However, each time we encounter a node v, we will call *searchI*(v, w, $f(v)$, LF) (instead of *search*( )) to determine the interval for v, where w represents the parent of v.

ALGORITHM *pmmSearch*(T, LF)

```
begin
1.  $v \leftarrow root(T)$; $\Re \leftarrow \Phi$;
2.  $enqueue(Q, v)$;
3.  while $Q$ is not empty do {
4.    $v \leftarrow dequeue(S)$;
5.    if $output(v) \neq \Phi$ then $\Re \leftarrow \Re \cup \{<output(v), l(v), I(v)>\}$;
6.    let $v_1, \ldots, v_k$ be the children of $v$;
7.    for $i = 1$ to $k$ do {
8.      $I \leftarrow searchI(v_i, v, f(v_i), LF)$; associate $I$ with $v_i$;
9.      if $I \neq \phi$ then $enqueue(Q, v_i)$; }
10. }
end
```

C. Correctness and Time Complexity

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

First, we have the following lemma.

Lemma 2 Let u, v be two nodes in A such that $f(v) = u$. Let w be the parent of v in T. The interval returned by *searchI*(v, w, $f(v)$, LF) is correct.

Proof. The lemma can be directly derived from Lemma 1. □

Proposition 1 Let A be a trie constructed over a collections of pattern strings: $r_1, \ldots, r_m$, and LF a *BWT*-mapping established for a reversed genome $\bar{s}$. Let $\Re$ be the result of *pmmSearch*(T, LF). Then, for each r_j, if it occurs in s, there is a quadruple $<output(v), l(v), [\alpha, \beta]> \in \Re$ such that $r_j \in output(v)$, $l(v)$ is equal to the last character of r_j, and $F_{l(v)}[\alpha]$, $F_{l(v)}[\alpha + 1]$, …, $F_{l(v)}[\beta]$ show all the occurrences of r_j in s.

Proof. We prove the proposition by induction on the height h of A, which is defined to be the number of edges on the longest downward path from the root to a leaf node.

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

Induction hypothesis. Suppose that when the height of A is h, the proposition holds. We consider the case that the height of A is $h + 1$. Let A' be a PMM obtained by removing all the leaf nodes in A. Then, the height of A' is at most h. According to the induction hypothesis, the interval generated by applying *pmmSearch*( ) to A' must be correct. Now, we consider a leaf node v in A'. Let $v_1, \ldots, v_k$ be the children of v in A. Then, in terms of Lemma 2, $I(v_i)$ produced by executing *searchI*(v_i, v, $f(v_i)$, LF) for $i = 1, \ldots, k$ must also be correct. Considering that all the nodes in ***A*** are visited in the breadth-first manner, the claim in the proposition is correct. □

Concerning the time complexity, we check the main **while**-loop, in which each node v in T is accessed only once. So the running time of *trieSearch*(T, *LF*) is bounded by O($\sum_{v \in T} \delta_v$), where δ_v represents the cost for an execution of *searchI*( ) to find $I(v)$. In the next section, our focus will be on how to further reduce this cost.

IV. IMPROVEMENTS

The algorithm discussed in the previous section can be further improved in two ways. One is to use the so-called *rankAll* mechanism [5, 11]. The other is to rearrange the search of a segment of L when we visit a node v in T to do the so-called multi-character checking to effectively decrease the searching cost of L.

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

A. *rankAll*

In this subsection, we first show the *rankAll* mechanism. Then, how it can be integrated into our general method will be described.

As mentioned above, the dominant cost of the whole process is the searching of L at each step. As shown in the previous section, by using the failure function, this problem can be mitigated to some extent.

A quite different way for this purpose is to arrange $|\Sigma|$ arrays, each for a character x in Σ, denoted as $x[\ ]$, in which $x[i]$ (the ith entry in the array for x) is the number of appearances of x within $L[1 .. i]$. For example, for the L array shown in Fig. 2(c), we will have five arrays: $\$[\ ]$, $a[\ ]$, $c[\ ]$, $g[\ ]$, and $t[\ ]$, as illustrated in Fig. 5(a). Especially, we have $[1] = 1$ while $a[5] = 2$. It is because in $L[1 .. 1]$ 'a' appears only once while in $L[1 .. 5]$ 'a' appears two times. In the same way, we can check all the other entries in these arrays.

We also notice that it is not necessary to store the column for '\$' since it will never be actually checked.

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

Thus, with the help of such data structures, the time of *search*( ) can be reduced to O(1).

The problem of this method is its high space requirement. For this reason, we will replace each $x[\ ]$ with a compact array A_x for $x \in \Sigma$, in which, rather than for each $L[i]$ ($i \in \{1, \ldots, n\}$), only for some entries in L the number of their appearances will be stored. For example, we can divide L into a set of buckets of the same size and only for each bucket a value will be stored in A_x. Obviously, doing so, more searching will be required to find missing values. In practice, the size ω of a bucket (referred to as a *compact factor*) can be set to different values. For instance, we can set $\omega = 4$, indicating that for each four contiguous elements in L a group of $|\Sigma|$ integers (each in an A_x) will be stored. That is, we will not store all the values in Fig. 5(a), but only store $a[4]$, $c[4]$, $g[4]$, $t[4]$, and $a[8]$, $c[8]$, $g[8]$, $t[8]$ in the corresponding compact arrays, as shown in Fig. 5(b).

j	F	L	\$	a	c	g	t
1	\$	a_3	0	1	0	0	0
2	a_3	c_3	0	1	1	0	0
3	a_2	g_1	0	1	1	1	0
4	a_1	c_2	0	1	2	1	0
5	c_3	a_2	0	2	2	1	0
6	c_2	c_1	0	2	3	1	0
7	c_1	\$	1	2	3	1	0
8	g_1	a_1	1	3	3	1	0

(a)

i	A_a	A_c	A_g	A_t
0	0	0	0	0
1	1	2	1	0
2	3	3	1	0

For each $\omega = 4$ values in L, a *rankAll* value is stored.

(b)

Figure 5. *LF*-mapping and rank-correspondence

Obviously, each $x[j]$ for $x \in \Sigma$ can be easily derived from A_x by using one of the following formulas:

$$x[j] = A_x[i] + \rho, \qquad (4)$$

where $i = \lfloor j/\omega \rfloor$ and ρ is the number of x's appearances within $L[i \cdot \omega + 1 .. j]$ which have to be searched, or

$$x[j] = A_x[i'] - \rho', \qquad (5)$$

where $i' = \lceil j/\omega \rceil$ and ρ' is the number of x's appearances within $L[j + 1 .. i' \cdot \omega]$. Also, ρ' has to be obtained by searching part of L.

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

More importantly, this method can be easily combined with the use of failure functions (as discussed in 4.3) to form a powerful strategy. To this end, step (2) and (4) in *searchI*(v, w, $f(v)$, *LF*) need to be slightly modified.

1. If $Y_{l(w)} + \alpha_w - 1 \le i$, we will simply set α_v equal to α_u, where $w \to v$ is an edge in T, $f(v) = u$, $l(u) = x$, $\alpha_u = rk_L(x_j)$, and $x_j = L[i]$.
2. Otherwise ($Y_{l(w)} + \alpha_w > i$), do the following operations to determine α_v: $l \leftarrow Y_{l(w)} + \alpha_w - 2$, $\rho \leftarrow$ *sDown*(L, l, ω, x) (or $\rho' \leftarrow$ *sUp*(L, l, ω, x)), and $\alpha_v \leftarrow A_x[\lfloor l/\omega \rfloor] + \rho + 1$ (or $\alpha_v \leftarrow A_x[\lceil l/\omega \rceil] - \rho' + 1$).
3. If $Y_{l(w)} + \beta_w - 1 \ge i'$, we will simply set β_v equal to β_u, where $\beta_u = rk_L(x_{j'})$ and $x_{j'} = L[i']$.
4. Otherwise ($Y_{l(u)} + \beta_u < i'$), do the following operations to determine β_v: $l \leftarrow Y_{l(w)} + \beta_w - 1$, $\rho \leftarrow$ *sDown*(L, l, ω, x) (or $\rho' \leftarrow$ *sUp*(L, l, ω, x)), and $\beta_v \leftarrow A_x[\lfloor l/\omega \rfloor] + \rho$ (or $\beta_v \leftarrow A_x[\lceil l/\omega \rceil] - \rho'$).

In the above process, (1) and (3) are exactly the same as in *searchI*( ). But in (2) and (4) a simple search of L is replaced with a function call *sUp*( ) (or *sDown*( )), by which the number of checked entries will be dramatically decreased by using the *rankAll* arrays.

B. Multiple Character Checking

In *sUp*( ) or in *sDown*( ), we search L once for each child of a certain node v. But we can manage to search the corresponding segment of L only once for all the children of v. This arrangement can be very useful for applications with large alphabets, such as protein sequences, whose alphabet contains as many as 20 characters. Thus, in many cases, 20 times of searching of L can be reduced to a single searching of L. To this end, we will use integers to represent characters in Σ. For example, we can use 1, 2, 3, 4 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 $|\Sigma|$ 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 Fig. 6 for illustration.

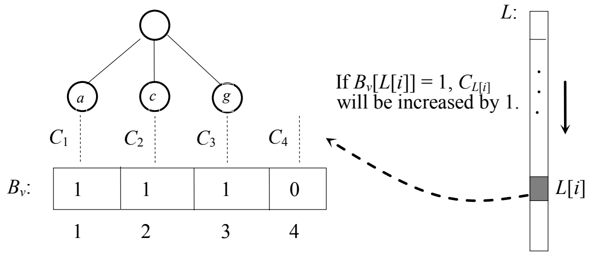


Figure 6. Illustration for extra data structures

With these two data structures, we change *sDown*(L j ,ω ,x) and *sUp*(L, j, ω, x) to *sDown*(L, j, ω, v) and *sUp*(L, j, ω, v), respectively, to search part of L for all the children of v, but only in one scanning of it.

In *sDown*(L, j, ω, v), we will search a segment $L[\lfloor j/\omega \rfloor \cdot \omega + 1 .. j]$ 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 $C_i + A_i[\lfloor j/\omega \rfloor]$, where C_i is the number of i's appearances within $L[\lfloor j/\omega \rfloor \cdot \omega + 1 .. j]$.

```
FUNCTION sDown(L, j, ω, v)
begin
1.  $c_i \leftarrow 0$ for each $i \in \Sigma$;
2.  $l \leftarrow \lfloor j/\omega \rfloor \cdot \omega + 1$;
3.  while $l \le j$ 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 $D[k] \leftarrow A_k[\lfloor j/\omega \rfloor] + C_k$;
9.  }
10. return $D$;
end
```

In the algorithm, we search $L[j' .. j]$ only in one scanning in the main **while**-loop (see lines 3 – 6), where $j' = \lfloor j/\omega \rfloor \cdot \omega + 1$ (see line 2.) For each encountered entry $L[l]$ ($j' \le l \le j$), 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).

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

```
FUNCTION sUp(L, j, ω, v)
begin
1.  $c_i \leftarrow 0$ for each $i \in \Sigma$;
2.  $l \leftarrow \lceil j/\omega \rceil \cdot \omega$;
3.  while $l \ge j + 1$ 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 $D[k] \leftarrow A_k[\lceil j/\omega \rceil] - C_k$;
9.  }
10. return $D$;
end
```

The following example helps for illustration.

Example 3 In this example, we trace the working process to generate ranges (by scanning $L[2 .. 5]$ shown in Fig. 2(c)) for the two children v_2 and v_9 of v_1 shown in Fig. 5. 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 $B_{v_1} = [0, 1, 1, 0]$ and all the counters C_1, C_2, C_3, C_4 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 $g[1] = A_g[\lfloor 1/4 \rfloor] + C_3 = 0$. This shows that in a single scanning of L, both $c[1]$ and $g[1]$ are found.

Next, to get $c[5]$ and $g[5]$, we will run *sDown*(L, 5, 4, v_1) to scan another part of L, also only once. In this process, $L[5] = 'a'$ (represented by 1) is accessed. Since $B_{v_1}[L[5]] = B_{v_1}[1] = 0$, C_2 is still 0. In addition, since C_3 (for $'g'$) is also 0, we have $c[5] = A_c[\lfloor 5/4 \rfloor] + C_2 = 2 + 0 = 2$ and $g[5] = A_g[\lfloor 5/4 \rfloor] + C_3 = 1 + 0 = 1$.

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 $[g[1] + 1, g[5]] = [1, 1]$. □

Again, we need to integrate this mechanism with the use of failure functions.

Let $v_1, \ldots, v_k$ be the children of v. Let $f(v_j) = u_j$ ($j = 1, \ldots, k$). Let i_j be the position in L corresponding to α_{u_j} ($j = 1, \ldots, k$). If $Y_{l(v)} + \alpha_v - 1 \le min\{i_1, \ldots, i_k\}$, do $\alpha_{v_j} \leftarrow \alpha_{u_j}$ for each $j \in \{1, \ldots, k\}$. Otherwise, divide $\{v_1, \ldots, v_k\}$ into two groups: G_1 and G_2 such that for any $v_j \in G_1$, we have $Y_{l(v)} + \alpha_v - 1 \le i_j$, and for any $v_{j'} \in G_2$ $Y_{l(v)} + \alpha_v - 1 > i_{j'}$. Obviously, for each $v_j \in G_1$, α_{v_j} can be directly determined as above. Thus, by setting

$B_v[L[v_j]] = 0$ for each $v_j \in G_1$, and $B_v[L[v_{j'}]] = 1$ for each $i_{j'} \in G_2$, we can then use *sDown*( ) or *sUP*( ) to determine $\alpha_{v_{j'}}$ for each $v_{j'} \in G_2$.

In a similar way, we can determine β_{v_j} for $j = 1, \ldots, k$.

According to the above discussion, our final algorithm can be described as follows.

ALGORITHM *pmmS*(*T*, *LF*, ω)

```
begin
1.  $v \leftarrow root(T)$; $\Re \leftarrow \Phi$;
2.  $enqueue(Q, v)$;
3.  while $Q$ is not empty do {
4.    $v \leftarrow dequeue(S)$;
5.    if $output(v)$ is not empty then $\Re \leftarrow \Re \cup <output(v), l(v), I(v)>$;
6.    let $v_1, \ldots, v_k$ be the children of $v$;
7.    let $i_1, \ldots, i_k$ be positions in $L$ corresponding to $\alpha_{u_1}, \ldots, \alpha_{u_k}$;
8.    divide $\{v_1, \ldots, v_k\}$ into $G_1$ and $G_2$ such that for any $v_j \in G_1$, $Y_{l(v)}$
      $+ \alpha_v - 1 \le i_j$, and for $v_{j'} \in G_2$, $Y_{l(v)} + \alpha_v - 1 > i_{j'}$;
9.    if $G_2 \neq \Phi$ then determine $\alpha_{v_1}, \ldots, \alpha_{v_k}$ as described above;
10.   else { $\alpha_{v_j} \leftarrow \alpha_{u_j}$ for $j = 1, \ldots k$;}
11.   let $l_1, \ldots, l_k$ be positions in $L$ corresponding to $\beta_{u_1}, \ldots, \beta_{u_k}$;
12.   divide $\{v_1, \ldots, v_k\}$ into $H_1$ and $H_2$ such that for any $v_j \in H_1$, $Y_{l(w)}$
      $+ \beta_w - 1 \ge l_j$, and for $v_{j'} \in H_2$, $Y_{l(w)} + \beta_w - 1 < l_{j'}$;
13.   if $H_2 \neq \Phi$ then determine $\beta_{v_1}, \ldots, \beta_{v_k}$ as described above;
14.   else { $\beta_{v_j} \leftarrow \beta_{u_j}$ for $j = 1, \ldots k$;}
15.   for $j = 1$ to $k$ do if $[\alpha_{v_j}, \beta_{v_j}] \neq \phi$ then $enqueue(Q, v_i)$;
18.}
end
```

The main difference of this algorithm from *pmmSearch*( ) is in their different ways to search *L*[*a* .. *b*]. Here, to find the ranks of the first appearances of all the labels of *v*'s children, *sDown*( ) or *sUp*( ) is called to scan part of *L* only once (while by *pmmSearch*( ) this has to be done multiple times each for a different child.) See line 9 and 10. Similarly, to find the ranks of the last appearances of these labels, another part of *L* will be scanned, also only once. See line 13 and 14. Besides these, all the other operations are almost the same as in *pmmSearch*( ).

V. EXPERIMENTS

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

- *Burrows Wheeler Transformation* [11] (*BWT* for short),
- *Suffix tree based* [14] (*Suffix* for short),
- *Hash table based* [10] (*Hash* for short),
- *Commentz-Walter's Algorithm* [6] (*CW* for short),
- *Crochemore's Algorithm* [18] (*Cr* for short), and
- *pmmS* (*pS* 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 [3] while all the other four 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, which are taken from an RNA laboratory at University of Manitoba (http://home.cc.umanitoba.ca/~xiej/):

TABLE I. CHARACTERISTICS OF GENOMES

Genomes	Genome sizes (bp)
Rat chr1 (Rnor_6.0)	290,094,217
C. merolae (ASM9120v1)	16,728,967
C. elegans (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 [12] 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 *rankAll*s (see Section IV) and compression factors f_2 of suffix arrays [13], which are used to find locations of matching reads (in a reference genome) in terms of their relationship with *BWT* arrays.

A.1 Tests with varying amount of reads

In this experiment, we vary the amount *n* of reads with *n* = 5, 10, 15, …, 50 millions while the reads are 50 bps or 100 bps in length extracted randomly from *Rat chr1* and *C. merlae* genomes. For this test, the compact factors f_1 of *rankAll*s 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 Fig. 7(a) and (b), we report the test results of searching the Rat chr1 for matching reads of 100 and 50 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 long reads (of length 100 bps) the suffix-based is worse than the BWT, but for short reads (of length 50 bps) they are comparable. Both the the Crochemore's and the Commentz-Walter's are worse than the BWT. But the Crochemore's is better than the Commentz-Walter's. 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 our method, the combination of PMMs and BWT arrays enables us to avoid repeated checking for similar reads. In these two figures, the time for constructing PMMs over reads is included. To see the impact of the construction of PMMs, we show the times for constructing them over different amounts of reads (of length 100 pbs), demonstrated in Table II.

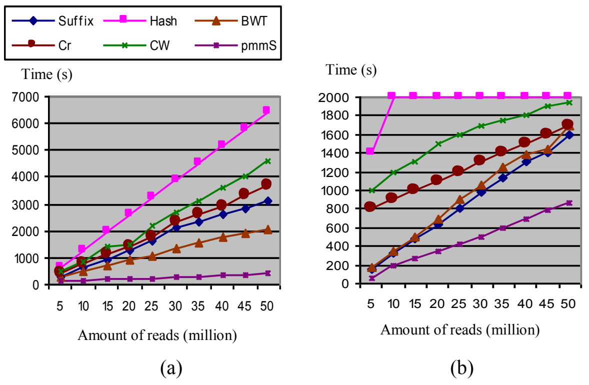


Figure 7. Test results on varying amount of reads

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

No. of reads	30M	35M	40M	45M	50M
Time for PMM Con.	91s	123s	152s	195s	210s

The difference between the BWT and ours 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 BWT) or a set of characters from more than one read (by ours).

TABLE III. NO. OF BWT ARRAY ACCESSES

No. of reads	30M	35M	40M	45M	50M
BWT	67954K	75632K	83321K	90732K	98165K
pmmS	19105K	22177K	25261K	28227K	31204K

Fig. 8(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.

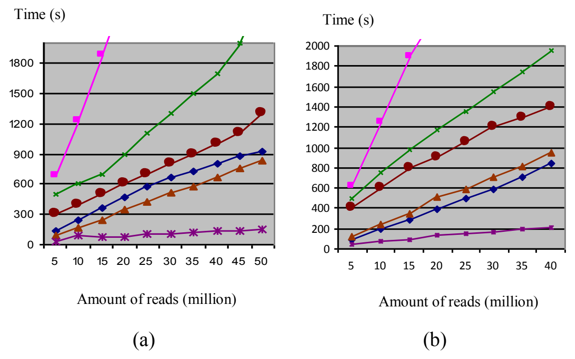


Figure 8. Test results on varying amount of reads

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, …, 200. The results in Fig. 9(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 Fig. 9(b), the results show the the case that each set has 50 million reads. Fig. 10(a) and (b) show the results of the same data settings but on C. merlae genome.

Again, in this test, the hash based performs worst while the suffix tree and the BWT method are comparable, and both the Commentz-Walter's and Wu-Manber's are worse than them. Our algorithm uniformly outperforms 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 ours 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.

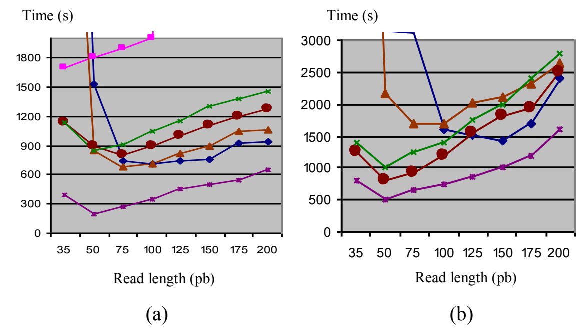


Figure 9. Test results on varying length of reads

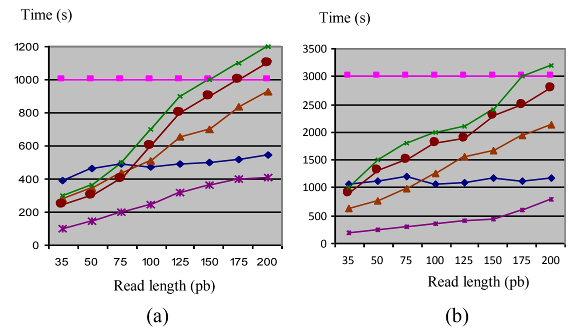


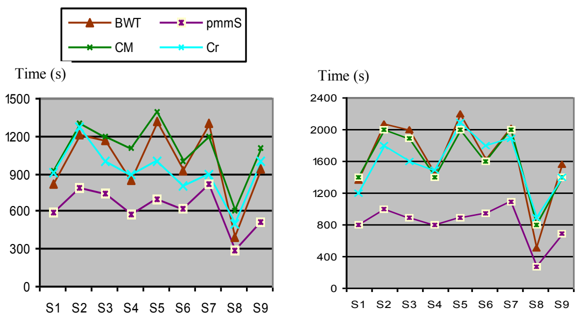
Figure 10. Test results on varying length of reads

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 (lab website: http://home.cc.umanitoba.ca/~xiej/, retrieved: 2014). This project includes over 500 million single reads produced by Illumina from a rat sample. Length of these reads is between 36 bps and 100 bps after trimming using Trimmomatic [4].

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 [6]. 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. Fig. 11(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 Fig. 11(b).



(a) (b)

Figure 11. Test results on real data

From Fig. 11(a) and (b), we can see that the test results for real data set are consistent with the simulated data. Our algorithm is faster than the BWT, the Crochemore's and the Commentz-Walter's on all 9 samples. Counting all the data sets together, ours is more than 45% faster compared with these methods. Although the performance would be dropped by taking PMMs' construction time into consideration, we are still able to save 40% time using our method.

VI. CONCLUSION AND FUTURE WORK

In this paper, an efficient algorithm for solving the set matching problem has been discussed, by which we are required to locate and identify all substrings of a long string s which match some short strings from a set $R = \{r_1, \ldots, r_m\}$. The main idea is to construct a *pattern matching machine A* and transform the reverse $\overline{s}$ of s to a BWT-array as index, $BWT(\overline{s})$, and search A against it. During the process, the failure function of A is used to reduce the subranges of $BWT(\overline{s})$ at each step. In addition, we change a single-character checking against $BWT(\overline{s})$ to a multiple-character checking, by which multiple searches of $BWT(\overline{s})$ are reduced to a single scanning of it. In this way, high efficiency can be achieved. Extensive experiments have been conducted, which shows that our method works better than the existing method for this problem.

As a future work, we will use the BWT to solve another important problem, 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.

VII. 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] R.A. Baeza-Yates and M. Régnier, Fast algorithms for two-dimensional and multiple pattern matching, in *Proc. SWAT '90 the second Scandinavian workshop on Algorithm theory*, Springer-Verlag, Bergen, Sweden, pp. 332-347.

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

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

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

[6] B. Commentz-Walter, A String Matching Algorithm Fast on the Average, in *Proc. 6th Colloquium on Automata, Languages and Programming*, July 16-20, 1979, pp. 118-132.

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

[8] Y. S. Dandass, S. C. Burgess, M. Lawrence, and S. M. Bridges, Accelerating String Set Matching in FPGA Hardware for Bioinformatics Research, *BMC Bioinformatics*2008, **9**:197.

[9] J. Y. Kim and J. S. Yaylor, Fast Multiple Keyword Searching, in *Proc. Third Annual Symposium on Combinatorial Pattern Matching*, Springer-Verlag, April 29 - May 01, 1992, pp. 41-51.

[10] 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.

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

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

[13] 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.

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

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

[16] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," *Technical Report TR-94-17*, Dept. Computer Science, Chung-Cheng University, 1994.

[17] L. Salmela, J. Tarhio and J. Kyt¨ojoki: Multi-pattern string matching with q-grams, *ACM Journal of Experimental Algorithmics*, Volume 11, 2006.

[18] M. Crochemore, at al., Fast practical multi-pattern matching, *Information Processing Letters*, 71 (1999) 107–113.

[19] 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.