On the Massive String Matching Problem

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

***Abstract*—In this paper, we discuss an efficient and effective index mechanism to support the matching of massive pattern strings in against a very long target string. It is very important to the next generation sequencing in the biological research. The main idea behind it is to construct an automaton over all the pattern strings, and search the automaton against a BWT-array *L* created for a target string *s* to locate all the occurrences of every pattern in *s* once for all. Experiments have been conducted, which show that our method for this problem is better than the existing approaches.**

Keywords—string matching; DNA sequences; automata; 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 requires us considering all the string patterns as a whole, rather than separately check them one by one. Two kinds of string matching need to be handled: *exact matching* and *inexact matching*. By the exact matching, we will find all the occurrences of a pattern string r in a target string s, but by the inexact matching we allow each occurrence having up to k positions different between r and s. The inexact matching is important due to the polymorphisms or mutations among individuals or even sequencing errors. In these cases, the pattern may disagree in some positions at an occurrence of r in the target s.

The string matching is always an interesting and important research topic in the computer science and computer engineering. In the past several decades, a bunch of efficient strategies have been proposed to find all the occurrences of a pattern in a target very fast, such as those discussed in [2, 3, 7, 8, 9, 11]. Roughly speaking, all these methods can be classified as illustrated in Fig. 1.

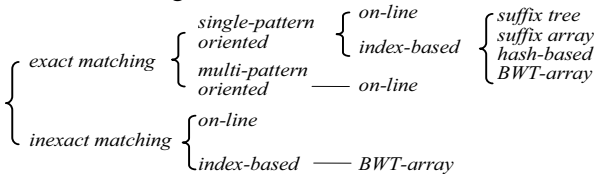

Figure 1. Classification of methods for string matching

From Fig. 1, we can see that for the exact matching problem we distinguish between two kinds of strategies: the single-pattern oriented strategies and the multi-pattern oriented ones. By the former, each time only one pattern string will be mapped to the target string, and for this we have both the on-line methods such as *Knuth-Morris-Pratt* [8], *Boyer-Moore* [11], and *Apostolico-Giancarlo* [3], and the off-line (index-based) methods like *suffix* trees [4], *suffix* arrays [14], and *BWT*-transformation (*Burrows-Wheeler* Transformation) [6]. However, for the latter, only one method can be found in the literature, which is the algorithm proposed [1]. By this method an automaton is established over all the patterns and searched against a target in one scanning.

For the inexact matching problem, we also have both the on-line strategies, such as the methods discussed in [2, 7, 9], and the index-based method such as the method proposed in [12]. The methods of [7, 9] have the worst-case time complexities bounded by $O(kn + m\log m)$, where $n = |s|$ and $m = |r|$. By these two methods, the *mismatch information* among substrings of r is used to speed up the working process. The method discussed in [2] is with a slightly better time complexity $O(n\sqrt{k}\log k)$. By this method, the *periodicity* within r is utilized. In [12], and target string s is transformed to a BWT-array (denoted as $BWT(s)$) as an index [6]. In comparison with *suffix trees* [4], $BWT(s)$ uses much less space [12]. However, its time complexity is bounded by $O(mn' + n)$, where n' is the number of leaf nodes of a tree produced during the search of $BWT(s)$. This time requirement can be much worse than the best on-line algorithm for large patterns. The reason for this is that by this method neither mismatch information nor periodicity within r is employed.

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

1. Constructing an automaton A over all the pattern strings, and check A against a BWT-array created as an index for a target string. This enables us to avoid repeated search of the same part of different patterns.
2. Changing a single-character checking to a multiple-character checking. (That is, each time a set of characters respectively from more than one pattern 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 briefly describe a string matching algorithm

978-1-5090-4093-3/16/$31.00 ©2016 IEEE

based on the BWT-transformation. Then, in Section III, we discuss our basic algorithm in great detail. In Section IV, we improve the basic method by using multiple-character checks. Section V is devoted to the test results. Finally, a short conclusion is set forth in Section VI.

II. BWT-TRANSFORMATION

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

A. BWT Arrays

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 Fig. 2(a).

(a)

a	c	a	g	a	c	a	\$
c	a	g	a	c	a	\$	a
a	g	a	c	a	\$	a	c
g	a	c	a	\$	a	c	a
a	c	a	\$	a	c	a	g
c	a	\$	a	c	a	g	a
a	\$	a	c	a	g	a	c
\$	a	c	a	g	a	c	a

(b)

F	L
a_1	\$
c_1	a_1
a_2	c_1
g_1	a_2
a_3	g_1
c_2	a_3
a_4	c_2
\$	a_4

(c)

\$	a	c	a	g	a	c	a
a	\$	a	c	a	g	a	c
a	c	a	\$	a	c	a	g
a	c	a	g	a	c	a	\$
a	g	a	c	a	\$	a	c
c	a	\$	a	c	a	g	a
c	a	g	a	c	a	\$	a
g	a	c	a	\$	a	c	a

Figure 2. Rotation of a string

By writing all these strings stacked vertically, we generate an $n \times n$ matrix, where $n = |s|$ (see Fig. 2(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:

$$F[i] = L[i]\text{'s successor}, \quad (1)$$

where $F[i]$ ($L[i]$) is the ith 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 element, its ith appearance (among all those elements with the same character) in F corresponds to its ith appearance in L, as demonstrated in Fig. 2(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 second 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* [6] and denoted as $BWM(s)$, as demonstrated in Fig. 2(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 Fig. 3(a), 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.

(a)

rk_F	F	L	rk_L
–	\$	a_4	1
1	a_4	c_2	1
2	a_3	g_1	1
3	a_1	\$	–
4	a_2	c_1	2
1	c_2	a_3	2
2	c_1	a_1	3
1	g_1	a_2	4

By ranking the elements in F, each element in L is also ranked with the same number.

(b)

$F_\$ = <\$, [1, 1]>$

$F_a = <a, [2, 5]>$

$F_c = <c, [6, 7]>$

$F_g = <g, [8, 8]>$

Figure 3. *LF*-mapping and tank-correspondence

Due to the *LF*-mapping and the rank correspondence, the *BWT*-transformation can be used to do efficient string matching, 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 shown in Fig. 3(a). \$ is not ranked since it appears only once.

From Fig. 3(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 for a certain $\alpha \in \Sigma \cup \{\$\}$ in F can be effectively represented as a pair of the form: $<\alpha, [x_\alpha, y_\alpha]>$, where x_α and y_α are the positions of the first and last appearance of α in F, respectively. So the whole F can be effectively compacted and represented as a set of $|\Sigma| + 1$ pairs, as illustrated in Fig. 3(b).

Denote by α_j the jth appearance of α in s. Assume that $rk_F(\alpha_j) = i$. Then, the position where α_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 α_j is assigned i if it is the ith appearance among all the appearances of α 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 α_j in L.

This equation is due to the rank correspondence between F and L. (See [6, 10] for a detailed discussion. Also see Fig. 3(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 us 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 $F_a = <a, [2, 5]>$. So $L' = L[2 .. 5]$, as illustrated in Fig. 4(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 Fig. 4(a) and Fig. 4(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 Fig. 4(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.

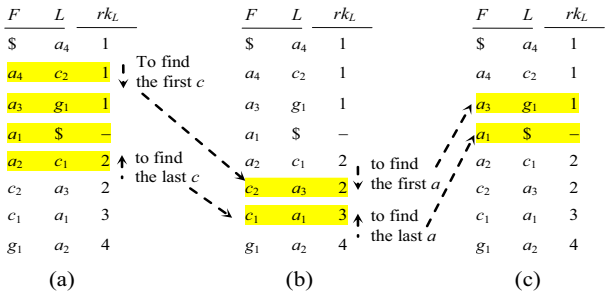

Figure 4. Sample trace

In the following, we will use $search(\alpha, L_z)$ to represent a search of L_z to find the first and the last rank of α (denoted respectively as i and j) within L_z, and return $<\alpha, [i, j]>$ as the result, where z represents a subrange in F_β for some character β and L_z represents a segment within L corresponding to z.

$$search(\alpha, L_z) = \begin{cases} <\alpha, [i, j]>, & \text{if } \alpha \text{ appears in } L_z; \\ \phi, & \text{otherwise.} \end{cases} \quad (4)$$

C. Construction of BWT arrays

It is not necessary to construct a BWT-array by rotating s, but by using a simple relationship between it and the corresponding suffix array [14] for s, as described below.

Let $s = a_0a_1 \ldots a_{n-1}$, ended with \$ (i.e., $a_i \in \Sigma$ for $i = 0, \ldots, n - 2$, and $a_{n-1} = \$$). Let $s[i] = a_i$ ($i = 0,1, \ldots, n - 1$) be the ith character of s, $s[i.. j] = a_i \ldots a_j$ a substring and $s[i .. n - 1]$ a suffix of s. Then, the suffix array J of s is a permutation of integers 0, ..., $n - 1$ such that $J[i]$ is the start position of the ith smallest suffix. The relationship between J and the BWT array L can be determined by formula (5) given below.

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

$$\begin{cases} L[i] = \$, & \text{if } J[i] = 0; \\ L[i] = s[J[i] - 1], & \text{otherwise.} \end{cases} \quad (5)$$

III. MAIN ALGORITHM

In this section, we present our algorithm to search a bunch of patterns against a target s. Its main idea is to organize all the pattern strings into an automaton A and search A against L to avoid any possible redundancy. First, we present the concept of automata in Subsection *A*. Then, in Subsection *B*, we discuss our basic algorithm for the task. We improve this algorithm in Section V.

A. Automaton over Pattern Strings

Let s be a target string, a very long string $\in \Sigma^*$ (for a DNA database, $\Sigma = \{A, T, C, G\}$). Let $R = \{r_1, \ldots, r_m\}$ be a set of patterns with each r_j being a short string $\in \Sigma^*$. The problem is to find, for every r_j's ($j = 1, \ldots, m$), all their occurrences in s.

A simple way to do this is to check each r_j against s one by one, for which different string searching methods can be used, such as suffix trees, BW-transformation [6], and so on. Each of them needs only a linear time (in the size of $\sum r_j$) to find all occurrences of r_j in s. However, in the case of very large m, which is typical in the new genomic research, one-by-one search of patterns against s 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 an automaton structure A and search A against s with the BW-transformation being used to check the string matching. For this purpose, we will first attach \$ to the end of each s ($i = 1, \ldots, n$) and construct $BWT(s)$. Then, attach \$ to the end of each r_j ($j = 1, \ldots, m$) to construct a prefix tree $T = pt(R)$ over R as below.

If $|R| = 0$, $pt(R)$ is, of course, empty. For $|R| = 1$, $pt(R)$ is a single node. If $|R| > 1$, R is split into $|\Sigma| = k$ (possibly empty) subsets $R_1, R_2, \ldots, R_k$ so that each R_i ($i \in \{1, \ldots, k\}$) contains all those strings with the same first character $\alpha_i \in \Sigma \cup \{\$\}$. The subtrees: $pt(R_1), \ldots, pt(R_k)$ are constructed in the same way except that at the lth step, the splitting of sets is based on the lth characters in the sequences. They are then connected from their respective roots to a single node to create $pt(R)$.

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

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

For these strings, a prefix tree T can be constructed as shown by the solid lines in Fig. 5. In T, v_0 is a virtual root while any other node v corresponds to a *real* character, labelling the edge e from v's parent to v and denoted $l(e)$. Therefore, all the characters on a path from the root to a leaf spell a pattern. For instance, the path from v_0 to v_8 corresponds to the third pattern r_3 = ACAGC\$. Note that each leaf node v is labelled with \$ and associated with a *pattern identifier*, denoted as $\gamma(v)$.

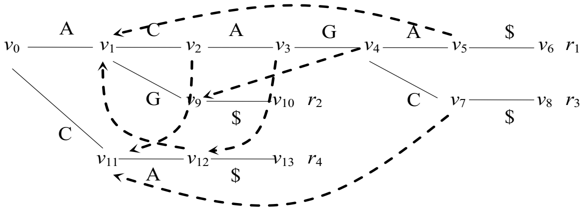

Figure 5. A trie and the corresponding automaton

In such a prefix tree, we define the label of a node v as the concatenation of edge labels on the path from the root to v, and denote it by $P(v)$. Then, we define a *failure function* $f(v)$ ($v \in T\backslash\{v_0\}$), which gives the node entered at a mismatch. That is, $f(v)$ is the node labeled by the longest proper suffix w of $P(v)$ such that w is a prefix of some pattern, as illustrated by the dashed arrows in Fig. 5. For example, $f(v_3) = v_{12}$ is represented by the dashed arrow from v_3 to v_{12}. We have this since $P(v_{12})$ = ‘CA’ is a suffix of $P(v_3)$.

Then, $T \cup \{f(v) \mid v \in T\backslash\{v_0\}\}$ makes up an automaton [1].

B. Integrating BWT Search with Automaton Search

It is easy to see that exploring a path in a prefix tree T over a set of patterns R corresponds to scanning a pattern $r \in R$. If we explore, at the same time, the array $L = BWT(\bar{s})$ established over a *reversed* target string $\bar{s}$, we will find all the occurrences of r (without \$ involved) in s (which is equivalent to searching $\bar{r}$ against $BWT(s)$.) Then, a depth-first searching of T against L will find all the occurrences of all patterns. In this process, the failure function can be used to speed up the computation as follows:

1. Each encountered node in T will be marked.
2. Let v be a node currently encountered in T. If $f(v) = u$ is not marked, we will search along a path bottom-up in T, starting from u: $u = u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_k$ such that u_i is a child of u_{i-1} ($i = 2, \ldots, k$) and u_k is a direct child of the root or a node associated with a range by the following step.
3. Let $v = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be the corresponding path starting from v. Let $[x_i, y_i]$ be the range found for $l(v_{i-1} \rightarrow v_i)$. We will attach it to u_i, denoted as $g(u_i)$.

The above process is denoted as *rangeAttach*(v, u). Its purpose is to avoid any repeated work. Each time we explore an edge $u \rightarrow v$ in T, we will search a segment L' within L to find a subrange for $\alpha = l(u \rightarrow v)$. If $g(v)$ is available, we will not search the whole L', but part of it as follows:

i) Let $g(v) = [x, y]$. Search $L[x' .. x - 1]$ to find the first appearance x'' of α, where $L[x']$ is the first element of L'. If α cannot be found, x should be the beginning position of the subrange to be found. Otherwise, it is x''.

ii) Search $L[y + 1 .. y']$ to find the last appearance y'' of α, where $L[y']$ is the last element y'' of L'. If α cannot be found, y should be the ending position of the subrange to be found. Otherwise, it is y''.

This is the main benefit brought by the failure function. To emphasize its difference from *search*( ), we denote the process as *searchI*(α, L', $g(v)$). See Fig. 6 for illustration.

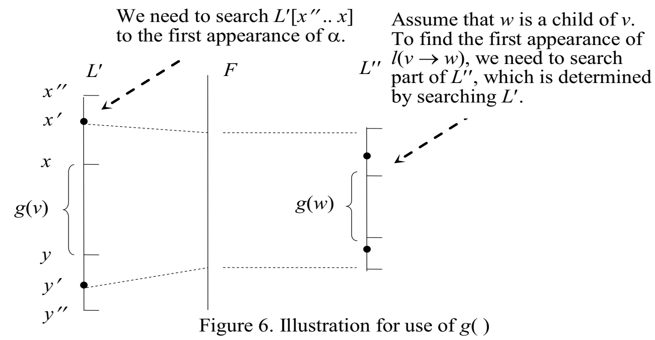

Figure 6. Illustration for use of $g(\,)$

According to the above discussion, we give 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 pair $<v, [a, b]>$ with v being a node in T and $a \leq b$, used to indicate a sub-segment to be searched to find the first and last appearances of $l(u \rightarrow v)$ in it. For example, when searching the tree shown by the solid lines in Fig. 5 against the L array shown in Fig. 3(a), we will have an entry like $<v_1, [1, 8]>$ in S to represent a sub-segment $L[1 .. 8]$ to be searched to find a subrange for $l(v_0 \rightarrow v_1)$ = 'A'. In addition, for technical convenience, we use ε to represent the *empty* character and set $search(\varepsilon, L') = L'$ for any sub-segment L' within L. We also assume that the character associated with the edge from the root's parent to the root is ε.

ALGORITHM *patternSearch*(*A*, *LF*)

```
begin
1.  v ← root(A);  ℜ ← Φ;
2.  push(S, <v, [1, |s|]>);
3.  while S is not empty do {
4.    <v, [a, b]> ← pop(S); let u be the parent of u; α ← l(u → v);
5.    if g(v) not available then [i, j] ← search(α, [a, b]);
6.    else [i, j] ← searchI(α, [a, b], g(v));
7.    let v_1, …, v_k be the children of v;
8.    for l = k downto 1 do {
9.      if v_l is the parent of a leaf then ℜ ← ℜ ∪ {<γ(v_l), α, i, j>};
10.     else{rangeAttach(v_l, f(u)); push(S, <v_i, [x_α + i - 1, x_α + j - 1]>); }
11.  }
12. }
end
```

In the algorithm, we first push $<root(A), [1, |s|]>$ into stack S (lines 1 – 2), and set the result $\Re$ to be Φ. Then, we go into the main **while-loop** (lines 3 – 12), in which we will first pop out the top element from S, stored as a pair $<v, [a, b]>$ (line 4). Then, we will search $L[a .. b]$ to find the subrange for $l(u \rightarrow v)$, where u is the parent of v, by executing *searchI*( ) or *search*( ), depending on whether $g(v)$ is available (see lines 5 – 6). Next, for each child v_l ($l = 1, \ldots, k$ for some k) of v, we will push $<v_l, [x_\alpha + i - 1, x_\alpha + j - 1]>$ into stack S if v_l is not the parent of a leaf node, where $[i, j]$ represents a subrange in F_α found by using *searchI*( ) or *search*( ). Otherwise, all the occurrences of $\gamma(v_l)$ have been found, which are represented by α, i, and j, i.e., $F[x_\alpha + i - 1 .. x_\alpha + j - 1]$, from which all those occurrences can be easily determined.

IV. IMPROVEMENTS

In the algorithm discussed in the previous section, each time only for a single character α part of L is searched to determine its sub-segment within it. However, we can manage to search the segment for multiple characters in one scan, i.e., for all the characters labeling the different edges going out of a certain node. To this end, we need to make the following changes:

- The characters in Σ will be represented as integers. For example, we can use 1, 2, 3, 4, 5 to represent *A*, *C*, *G*, *T*, \$ in a DNA sequence.
- Each entry in stack S is still a pair $<v, [a, b]>$. But $[a, b]$ is now a sub-segment in L found for $l(u \rightarrow v)$, where u is the parent of v.

Let $v_1, \ldots, v_k$ be the children of v. What we want is to find all the sub-segments in L for each $\alpha_l = l(v \rightarrow v_l)$ ($l = 1, \ldots, k$) in one scan of $L[a .. b]$. For simplicity, however, only the process to find the first appearances of α_l's is explained. For this, the following data structures will be used:

- 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 v_l of v such that $l(v \rightarrow v_l) = i$; otherwise, $B_v[i] = 0$.
- c_i: a variable associated with $i \in \Sigma$ to record the first appearances of i during a search of $L[a .. b]$.

By using the above data structures, the task to find the first appearance of all α_l's can be done as follows:

- Let $g(v_l) = [x_l, y_l]$ ($l = 1, \ldots, k$). Denote $x = max\{x_1, \ldots, x_k\}$. (If for any v_l $g(v_l)$ is not available, x is set to be b.)
- Search $L[a .. x - 1]$ from the start to the end. For each encountered entry $L[j]$ ($a \le j \le x - 1$), we will check whether $B_v[L[j]] = 1$. If it is the case, store j in $c_{L[j]}$ and change $B_v[L[j]]$ to 0 (which guarantees that for each character, only the first encountered j is recorded.)

See Fig. 7 for illustration.

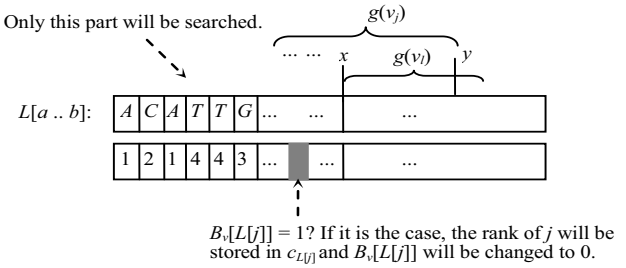

Figure 7. Illustration for multi-character checking

The last appearances of all α_l's can be found in a similar way. In the algorithm below, these two procedures are referred to as *firstAp* (v) and *lastAp* (v), respectively.

ALGORITHM *pS*(T, LF, β)

begin

1. $v \leftarrow root(T)$;
2. $push(S, <v, [1, |s|]>)$;
3. **while** S is not empty **do** {
4. $<v, a, b> \leftarrow pop(S)$; $\alpha \leftarrow l(u \rightarrow v)$; $c \leftarrow rk_L(a)$; $d \leftarrow rk_L(b)$;
5. let $v_1, \ldots, v_k$ be all those children of v, which are labeled with \$;
6. let $u_1, \ldots, u_j$ be all the rest children of v;
7. **for** each $i \in \{1, \ldots, k\}$ **do** $\{\mathcal{R} \leftarrow \mathcal{R} \cup \{<\gamma(v_i), \alpha, c, d>\}$;
8. let $l(v \rightarrow u_l) = \alpha_l$ ($l \in \{1, \ldots, j\}$);
9. call *firstAp*(v) to find the ranks of the first appearances of $\alpha_1, \ldots, \alpha_j$, respectively: $r(u_1), \ldots, r(u_j)$;
10. call *lastAp*(v) to find the ranks of the last appearances of $\alpha_1, \ldots, \alpha_j$, respectively: $r'(u_1), \ldots, r'(u_j)$;
11. **for** $l = j$ **downto** 1 **do**
12. $\{ push(S, <u_l, [x_{\alpha_j} + r(u_l) - 1, x_{\alpha_j} + r'(u_j) - 1]>) \}$;
13. }

end

The main difference of the above algorithm from *patternSearch*( ) 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, *firstAp*(v) is called to scan part of L only once (while in *patternSearch*( ) this has to be done once for each different child.) See line 9. Similarly, to find the ranks of the last appearances of these labels, another part of L is also scanned only once by calling *lastAp*(v). See line 10. All the other operations are almost the same as in *pattern-Search*( ). (For ease of understanding, the use of failure functions is not included.)

V. 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),
- *Automaton-BWT* (aBWT for short, discussed in this paper),
- *Improved Automaton-BWT* (iaBWT 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 [4] while all the other three algorithms are implemented by ourselves. All of them are able to find all occurrences of every read (short DNA sequence) 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.

For the tests, five reference genomes are used:

Table 1: 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

All the pattern strings are created by simulating reads from the five genomes shown in Table 1, with varying lengths and amounts. It is done by using the *wgsim* program included in the *SAMtools* package [13] with default model for single read simulation.

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. In Fig. 8(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 aBWT and iaBWT, the use of tries enables us to avoid repeated checking for similar reads.

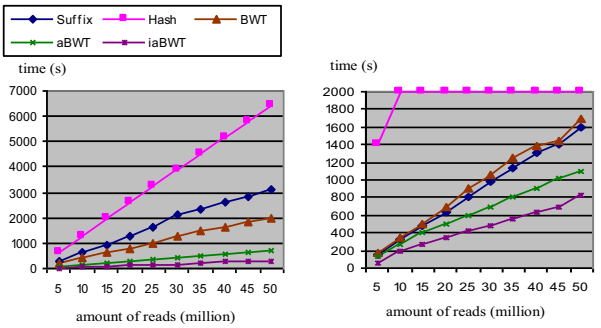

Figure 8: Test results on very amount of reads

In these two figures, the time for constructing automata over reads is not included. It is because in the biological research an automaton 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 automaton can be established very fast as demonstrated in Table 2, in which we show the times for constructing automata over different amounts of read.

Table 2: Time for trie construction over reads of length 100 bps

No. of reads	30M	35M	40M	45M	50M
Time forAuto. Con.	51.9s	65s	83s	97s	113s

The difference between aBWT and iaBWT is due to the different number of BWT array accesses as shown in Table 3. 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 aBWT) or a set of characters from more than one read (by iaBWT).

Table 3: 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

VI. CONCLUSION

In this paper, a new method to search a large volume of pattern strings against a single long target string has been proposed, aiming at efficient next-generation sequencing in DNA databases. The main idea is to combine the search of automata constructed over the patterns and the search of the BWT indexes over the target. By using the failure functions, the sizes of sub-segments of a BWT array to be searched can be dramatically decreased. In addition, the so-called multiple-character checking has been introduced, which reduces the multiple scanning of a BWT array to a single search of it. Experiments have been conducted, which show that our method improves the running time of the traditional methods by an order of magnitude or more.

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] S. Bauer, M.H. Schulz, P.N. Robinson, gsuffix: http:://gsuffix. Sourceforge.net/, retrieved: April 2016.

[5] 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.

[6] 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, 1994, retrieved: 2016.

[7] 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.

[8] 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.

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

[10] B. Langmead, "Introduction to the Burrows-Wheeler Transform," www.youtube.com/watch?v=4n7N Pk5lwbI, retrieved: April 2016.

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

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

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

[14] 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.

[15] 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.