A Tree Approach for Solving the Exact Multiple String Matching Problem

Yi-Kung Shieh, Shyong-Jian Shyu, Chin-Lung Lu and Richard Chia-Tung Lee
Department of Computer Science
Nation Tsing Hua University, Hsinchu, Taiwan
d9762814@oz.nthu.edu.tw, sjshyu@mail.mcu.edu.tw, cllu@cs.nthu.edu.tw and rctlee@rctlee.cyberhood.net.tw

Abstract

In this study, we deal with the exact multiple string matching problem. Given a text T and a set of r patterns $P_1, P_2, \ldots, P_r$, find the ending positions of all sub-stringss in T which is equal to a pattern. By transforming all sub-strings of T into a reference tree such that each internal node stores a reference string, the exact multiple pattern matching problem can be efficiently solved by searching patterns in the tree via the guidance of the reference strings. We design elegant algorithms to construct the reference tree and to search patterns in the tree in this paper. The experiments involving problem instances from the DNA sequences and the English language are conducted to compare the performances of our approach and the well-known Burrows-Wheeler Transform (BWT) algorithm. The computational results demonstrate the advantage of our approach over BWT. In spite of the simplicity, the performance of our approach is quite efficient and robust.

1. Introduction

The exact string matching problem is a classic string problem. We are given a text string $T = t_1t_2\ldots t_n$ and a pattern string $P = p_1p_2\ldots p_m$. Each character in text or pattern string is a character or a symbol. The text string T can be denoted as $T(1, n)$, and thus a sub-string of T can be denoted as $T(i, j) = t_it_{i+1}\ldots t_j$ where $1 \leq i \leq j \leq n$. The exact string matching problem is to determine whether and where P appears in T. Exact string matching is widely used in DNA and English language text searching.

Comprehensive reviews of string matching algorithms can be found in [6, 7, 8, 10, 19, 22]. We may classify these algorithms into three categories:

(1) Algorithms without pre-processing: Convolution Algorithm [9].
(2) Algorithms with pre-processing on pattern strings: MP Algorithm [17], KMP Algorithm [13], Boyer-Moore Algorithm [1], Horspool Algorithm [11], Colussi Algorithm [5], Reverse Factor Algorithm [14], Simon Algorithm [20], Smith Algorithm [21], WM Algorithm [24], BNDM Algorithm [18] and Lecroq Hashing Algorithm [15].
(3) Algorithms with pre-processing on text string: Suffix Tree Method [23], Suffix Array Method [12] and Burrows-Wheeler Transform Algorithm [2] (BWT, for short).

For the exact multiple string matching problem, we expect that for a single text string, there will be a large number of pattern strings. For instance, the Old Testament of the Bible is a popular book and many researchers will try to find where certain phrases appear in it. Another application is the DNA searching. For a given DNA sequence, many researchers will try to determine whether some pattern strings appear in it or not. In such condition, it is worthwhile to conduct a pre-processing on the text string such that the later searching for patterns in the text will be very fast.

In this paper, we are interested in the exact multiple string matching problem. Therefore, we will compete with the algorithm in Category 3. Although the time complexity of the pre-processing phase for all algorithms in Category 3 are $O(n)$, those of the searching phase for the suffix tree method, suffix array method and BWT are $O(m+occ)$, $O(m+\log n+occ)$ and $O(m)$, respectively, where occ is the number of occurrences of the found patterns. Since BWT is more efficient than the other two methods, we shall only consider the BWT as our comparison basis.

In Section 2, we briefly introduce the BWT algorithm. Our approach for the single string matching problem by constructing a reference tree and searching the tree is informally explained in Section 3, and then we define the reference tree and present our pre-processing and searching algorithms in Section 4. Experimental results are shown in Section 5. Finally, we conclude this work in Section 6.

2. The Burrows-Wheeler Transform to the Exact String Matching Problem

The Burrows-Wheeler Transform algorithm (BWT for short) [2, 4] was first used for data compression.

Li and Durbin [16] utilized it to solve the exact string matching problem.

Given a text string, we first add a special symbol \$ onto the end of the string and start rotating the string. For example, given T = gtcagtc\$, the rotation of T is tcagtc\$g. We then rotate the string to obtain the following strings:

1. gtcagtc\$
2. tcagtc\$g
3. cagtc\$gt
4. agtc\$gtc
5. gtc\$gtca
6. tc\$gtcag
7. c\$gtcagt
8. \$gtcagtc

These rotated strings are sorted lexicographically, and then the result is as follows:

1. \$gtcagtc
2. agtc\$gtc
3. c\$gtcagt
4. cagtc\$gt
5. gtc\$gtca
6. gtcagtc\$
7. tc\$gtcag
8. tcagtc\$g

The above array is denoted as the sorted rotation array. We pick up all of the last character of the above strings from top to bottom. The resulting string is called the BWT of T which is cctta\$gg.

We can see that the sorted rotation array gives each suffix in T an index. For instance, consider the suffix tcagtc\$ in T. It is indexed as 8 in the sorted rotation array. For the suffix cagtc\$ in T, it is indexed as 4 in the array. We shall see that this indexing scheme is quite useful for us to do string matching.

We define the following three functions:

(1) *Count*(α) is the number of characters smaller than α in T lexicographically where α is a character in alphabet of T. Table 1 gives the *Count* function for the characters in the alphabet of T = gtcagtc\$.

Table 1. The *Count* function of T = atcatg\$

α	\$	a	c	g	t
Count(α)	0	1	2	4	6

(2) The range of the sorted rotation array of a pattern P', denoted as $SA(P') = (s, e)$, where s is the starting row of the sorted rotation array whose prefix contains P' and e is the ending row of the sorted rotation array whose prefix contains P'. For instance, SA(a) = (2, 2) and SA(gt) = (5, 6). Note that the range of sorted rotation array includes the whole BWT of T initially, i.e. SA(" ") = (1, 8).

(3) *Precede*(i, α) is the number of character α occurring in the prefix with length i of BWT of T. For instance, let i = 6 and α = a. Then *Precede*(i, α) = *Precede*(6, a) = 1. Table 2 displays the *Precede* function for the BWT of T.

Table 2. The *Precede* function for cctta\$gg.

	0	1	2	3	4	5	6	7	8
\$	0	0	0	0	0	0	1	1	1
a	0	0	0	0	0	1	1	1	1
c	0	1	2	2	2	2	2	2	2
g	0	0	0	0	0	0	0	1	2
t	0	0	0	1	2	2	2	2	2

The BWT can be used to solve the exact string matching problem. The pattern string P may be considered to be $P = \alpha P'$. Let us assume that we have constructed the sorted rotation array and already found $SA(P') = (s, e)$. Then we can determin the range of $P = \alpha P'$ based upon the following equation:

$$SA(P) = SA(\alpha P') = (s', e') = \\ (Count(\alpha)+Precede(s-1, \alpha)+1, Count(\alpha)+Precede(e, \alpha))$$

For example, assume that P = gtc. Note that SA(" ") = (1, 8) initially.

(1) For the shortest suffix $P(3, 3)$ = c, we have:
SA(c) = (s', e')
= ($Count$(c)+$Precede$(s–1, c)+1, $Count$(c)+$Precede$(e, c))
= ($Count$(c)+$Precede$(1–1, c)+1, $Count$(c)+$Precede$(8, c))
= (2+0+1,2+2)
= (3, 4)

(2) For the suffix $P(2, 3)$ = tc = t$P(3, 3)$, we have:
SA(tc) = SA(tP(3, 3)) = (s', e')
= ($Count$(t)+$Precede$(s–1, t)+1, $Count$(t)+$Precede$(e, t))
= ($Count$(t)+$Precede$(3–1,t)+1, $Count$(t)+$Precede$(4, t))
= (6+0+1, 6+2)
= (7, 8)

(3) For the suffix $P(1,3)$ = gtc = g$P(2, 3)$, we have:
SA(gtc) = SA(gP(2, 3)) = (s', e')
= ($Count$(g)+$Precede$(s–1, g)+1, $Count$(g)+$Precede$(e, g))
= ($Count$(g)+$Precede$(7–1, g)+1, $Count$(g)+$Precede$(8, g))
= (4+0+1, 4+2)
= (5, 6)

The whole pattern has been considered and found the range to be from 5 to 6. That is, the prefixes with the length of P from the 5th row of sorted rotation array to the 6th sorted rotation array are equal to P.

Let Σ be the alphabet of T and P and σ be the size of Σ. The time complexity for obtaining the sorted rotation array and the two auxiliary functions is $O(n)$ and the space complexity is $O(n\times\sigma)$. The time complexity of searching is $O(|P|)$, where $|P|$ is the length of pattern P. The biggest problem for the BWT method is the space required. If n is great, such as 16M and σ is 63 for English language, the *Precede* function will be too large for ordinary personal computers.

The Skew algorithm [12] constructs the sorted rotation array more efficiently. Chen [3] further improved the Skew algorithm. In [4], the authors modified the BWT space complexity by proposing a data compression method. The space complexity is reduced from $O(n\times\sigma)$ to $O(n\log\sigma)$-bit but the searching time is increased to $O(|P|+(\log_{\sigma}n+occ)\log n/\log\log n)$ where occ is the number of the occurrences of the found patterns. Note that the time complexity of searching for original BWT [16] is only $O(|P|)$.

3. An Informal Introduction of Our Reference String Approach

Our reference string approach can be informally explained as follows: Suppose we have a set of input strings with the same length m and a test string x, also with length m. We like to find strings in the input strings which are exactly equal to string x. This approach consists of two stages: The pre-processing phase and the searching phase. Let us first introduce the pre-processing phase.

The pre-processing essentially constructs a reference tree. The root of the tree, denoted as N, contains a pointer $PS(N)$ pointing to the set $S(N)$ which contains all input strings and a reference string, denoted as $RF(N)$, as shown in Figure 1.

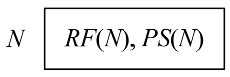

Figure 1. The root node N in the reference tree

Suppose our input strings are {gcca, ctac, ctaa, acgt, ttgc, ccac, ctgc, tgta, acta, ggac, cggc, ctag} and we choose ctac as the reference string. Then the root node of the reference tree will be shown in Figure 2.

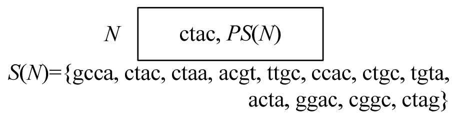
N
ctac, PS(N)
S(N)={gcca, ctac, ctaa, acgt, ttgc, ccac, ctgc, tgta, acta, ggac, cggc, ctag}

Figure 2. The root node N with the reference string ctac

We calculate the distances between these strings of $S(N)$ and the reference string $RF(N)$. The measurement of the distance between two strings is Hamming distance. The definition of Hamming distance between two strings with equal length is the number of positions at which the corresponding characters or symbols are different. All of the distances are positive integers, ranging from 0 to m.

We now expand the root node. Let N-i denote the ith child node of N. N-i contains a pointer pointing to all of the strings of $S(N)$ whose distances with $RF(N)$ are i. This set is denoted as $S(N$-$i)$. Each node N-i also contains a reference string $RF(N$-$i)$ which is selected from $S(N$-$i)$. If the size of $S(N$-$i)$ is too small, we terminate the expanding; otherwise, we recursively expand the tree with $S(N$-$i)$ as the root node. Figure 3 illustrates the above discussion.

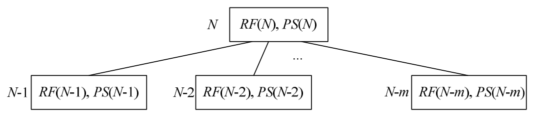

Figure 3. The nodes in level 1 and level 2 of the reference tree

For our input strings and the reference string, we will have the reference tree expanded as shown in Figure 4. The complete reference tree is shown in Figure 5.

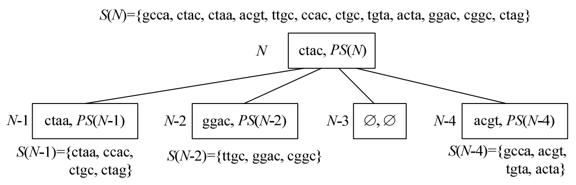
S(N)={gcca, ctac, ctaa, acgt, ttgc, ccac, ctgc, tgta, acta, ggac, cggc, ctag}
N
ctac, PS(N)
N-1
ctaa, PS(N-1)
N-2
ggac, PS(N-2)
N-3
∅, ∅
N-4
acgt, PS(N-4)
S(N-1)={ctaa, ccac, ctgc, ctag}
S(N-2)={ttgc, ggac, cggc}
S(N-4)={gcca, acgt, tgta, acta}

Figure 4. The nodes in level 1 and level 2 with the reference strings

After the reference tree has been constructed, we search the test string x in this reference tree. The search starts from the root node. In root node N, the distance d between its reference string $RF(N)$ and x is calculated. If d is 0, x must be identical with $RF(N)$.

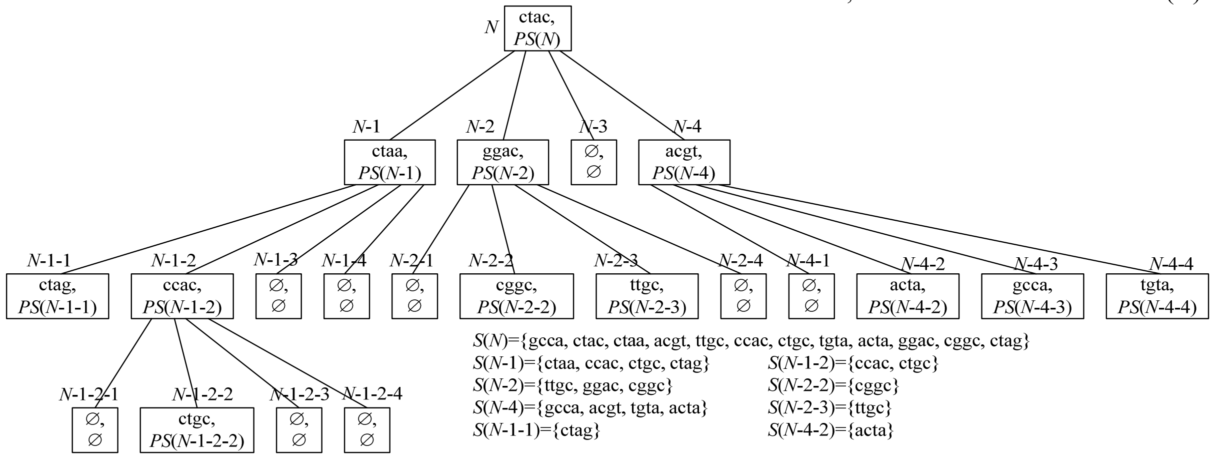

Figure 5. The complete reference tree of the example

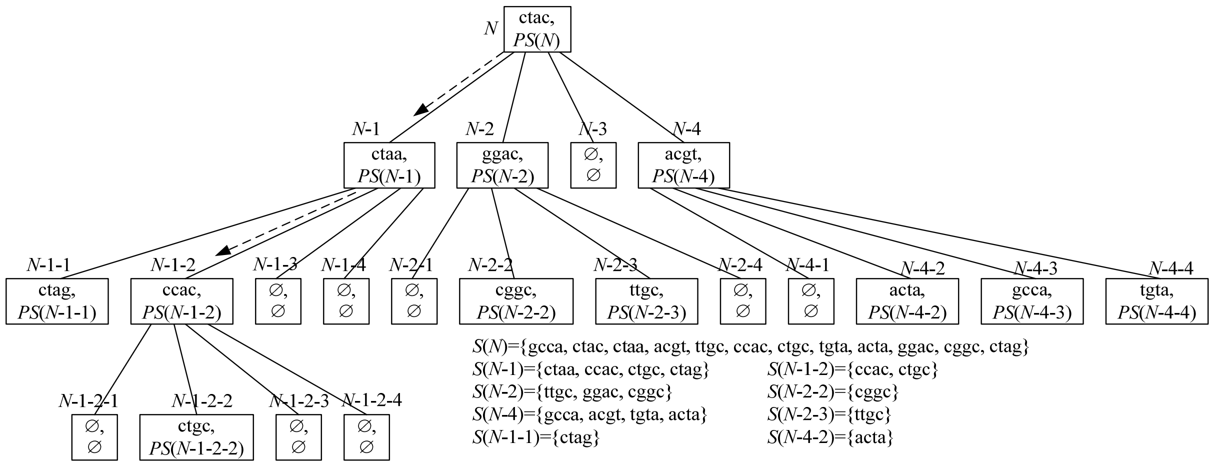

Figure 6. The search for the test string x = ccac in the reference tree

Then we get the solution and stop searching the reference tree. If d is greater than 0, the search would be continued to its sub-trees. The possible solution only exists in the sub-tree rooted by the child node N-d because the distances between $RF(N)$ and strings in $S(N\text{-}d)$ all are d. Hence, we only search the child node N-d recursively.For example, assume that the test string is x = ccac and the reference tree is shown in Figure 5. The search starts from the root node N. The distance between reference string $RF(N)$ = ctac and x = ccac is 1. We continue searching the sub-tree rooted by N-1. In the node N-1, the distance between reference tree $RF(N\text{-}1)$ = ctaa and x = ccac is 2, and then we search the sub-tree rooted by N-1-2. In the node N-1-2, the distance between reference string $RF(N\text{-}1\text{-}2)$ = ccac and x = ccac is 0. That is, we get the solution because the reference string $RF(N\text{-}1\text{-}2)$ is equal to the test string. The above searching procedure is illustrated in Figure 6.

Once the reference tree with reference strings has been built, the subsequent search for a string (or searches for multiple strings) would be very efficient. To solve the problem considered, we shall give formal descriptions and implementation details of our reference tree in the next section.

4. Applying Reference String Approach to Exact String Matching Problem

The exact string matching problem is defined as follows: We are given a text string T of length n and a pattern string P of length m. Our work is to determine whether P appears in T and if does, where it appears. We observe that determining whether P is in T is less efficient that determining whether a prefix of P of length, say ℓ ($< m$), is in T. Therefore, we may compare first $T(i, i+\ell-1)$ for $1 \le i \le n-m+1$ with the prefix of the pattern string, also of length ℓ. If they are identical, we continue to compare the rest of strings. Otherwise, there would be no further comparison. This will save our searching time. Such idea inspires our development of the reference strings and reference tree for T. Note that the length ℓ has to be shorter than m.

The reference tree is formally described as follows:

(1) The root node is denoted as N.
(2) For every node X, there is a corresponding $S(X)$ where each element of $S(X)$ is a sub-string of length ℓ in T. Note that in the root node N, $S(N)$ consists of sub-strings $T(i, i+\ell-1)$'s where $1 \le i \le n-m+1$.
(3) For every node X, it contains a pointer $PS(X)$ pointing to the linked list storing the starting positions of the strings in $S(X)$.
(4) For every internal node X, there is a chosen reference string $RF(X)$ from $S(X)$. X stores the starting position of $RF(X)$.
(5) For every internal node X, the child nodes of X are X-0, X-1, ..., X-ℓ where $S(X\text{-}i)$ contains all strings of $S(X)$ whose distance to $RF(X)$ are i.
(6) For every node X-0, the strings in $S(X\text{-}0)$ are equal to $RF(X)$. X-0 is a leaf node because as soon as the searching reaches it, the sub-strings which are equal to the ℓ prefix of the pattern have been found.
(7) For any node X, if the size of $S(X)$ is smaller than or equal to a pre-specified parameter k ($< n-m+1$), X is a leaf node because we will simply conduct an exhaustive search on all strings of $S(X)$.
(8) In a leaf node, there is no reference string.

Let τ denote a reference tree and τ_I denote the ith sub-tree of τ where $0 \le i \le \ell$. The pre-processing algorithm is as follows:

Algorithm Pre-processing

Input: Text string T, length m, length ℓ and parameter k

```
Output: The reference tree $\tau$ with respect to $T$, $\ell$ and $k$
Let $S$ be the set containing sub-strings $T(i, i+\ell-1)$'s in
    $T$ where $1 \le i \le n-m+1$
Let $PS$ be the pointer pointing to the linked list
    consisting of the starting positions of the strings in $S$
Call $\tau = ConstructRefTree(T, PS, \ell, k)$
return $\tau$
```

```
Algorithm Constructing Reference Tree
Input:  Text string $T$, pointer $PS$ pointing to the list
        consisting of the starting positions of the
        strings in $S$, length $\ell$ and parameter $k$
Output: The reference tree $\tau$ with respect to $T$, $\ell$ and $k$
$ConstructRefTree(T, PS, \ell, k)$
Follow $PS$ to obtain $S$
if ($|S| \le k$) then
    Create a leaf node $X$ and set $PS(X)$ as $PS$
else        // $|S| > k$
    Create an internal node $X$
    Arbitrary choose a string from $S$ to be the
        reference string $RF(X)$ whose starting position
        in $T$ is stored in $X$
    for (each $i \in [0, \ell]$) do $S(X\text{-}i) = \varnothing$
    for (each position $j$ in the list pointed by $PS(X)$) do
        Compute distance $d$ between $RF(X)$ and $T(j, j+\ell-1)$
        Add $T(j, j+\ell-1)$ into $S(X\text{-}d)$
    endfor
    for (each $i \in [0, \ell]$) do
        Create a link list to store the starting positions
            of the strings in $S(X\text{-}i)$ and set $PS(X\text{-}i)$ as the
            pointer pointing to the list
    endfor
    Create a leaf node $X$-0, pointed by $\tau_0$, to store $PS(X\text{-}0)$
    for (each $i \in [1, \ell]$) do
        if ($|S(X\text{-}i)| > 0$) then
            $\tau_i = ConstructRefTree(T, PS(X\text{-}i), \ell, k)$
        else
            $\tau_i$ is null
        endif
    endfor
    // $\tau_i$ is the pointer pointing to the ith sub-tree of
    // $X$ rooted by $X$-$i$
    Set $\tau_0, \tau_1, \dots, \tau_\ell$ (rooted by $X$-0, $X$-1, ... , $X$-$\ell$,
        respectively) as the sub-trees of $X$
endif
return $\tau$ (the pointer pointing to the tree rooted by $X$)
```

During the tree construction, every internal node distributes the strings of $S(X)$ (whose starting positions in T are stored as a linked list pointed by $PS(X)$) into $S(X\text{-}0)$, $S(X\text{-}1)$, ... , $S(X\text{-}\ell)$ according to their distances to $RF(X)$. The sub-trees of X rooted by X-1, X-2, ... , X-ℓ having strings of $S(X\text{-}1)$, $S(X\text{-}2)$, ... , $S(X\text{-}\ell)$ are built in a recursive way. After the tree has been constructed, every internal node X only stores the starting position of $RF(X)$. The ℓ prefixes of all sub-strings of length m in T are actually distributed into the corresponding leaves and stored by their starting positions in the corresponding lists. Each leaf node X merely stores the pointer $PS(X)$. The space complexity of the reference tree is $O(n)$.

In an internode node X, the probability that a string in $S(X)$ is equal to $RF(X)$ is $(1/\sigma)^\ell$ where σ is the size of alphabet of T. The probability that a string in $S(X)$ is different to $RF(X)$ shall be $1-(1/\sigma)^\ell$. We expect the character comparisons for construction of the reference tree to be $O(n\times\ell\times(1-(1-q_0)^{h-1})/q_0)$ where h is the height of the reference tree and $q_0 = (1/\sigma)^\ell$.

The searching in the reference tree can be informally described as follows:

Step 1: Let $X = N$ where N is the root node of the reference tree.

Step 2: Compute the distance d between the prefix with length ℓ of the pattern and $RF(X)$.

Step 3: If d is 0, compare the rest of the pattern with $T(i+\ell, i+m-1)$ for each i in the list pointed by $PS(X\text{-}0)$ to obtain solutions and exit.

Step 4: If d is not 0 and child node X-d is an internal node, let $X = X\text{-}d$ and go to Step 2.

Step 5: If d is not 0 and X-d is a leaf node, exhaustively search the strings in $PS(X\text{-}d)$ to get solutions and exit.

In a leaf node X-t ($t \ne 0$) where $T(i, i+\ell-1)$ is different from reference string $RF(X)$, we have to handle a *special string matching problem* which is to determine whether or not $T(i, i+\ell-1)$ is equal to the pattern string for each i in the list pointed by $PS(X\text{-}t)$. Any string matching algorithm can be applied to verify these strings. We simply compare the two strings character by character until any mismatch occurs or they exactly match. This method is simple and very efficient.

We search the pattern from the root node of the reference tree. The searching algorithm is as follows:

```
Algorithm Searching
Input:  Text string $T$, reference tree $\tau$, pattern $P$,
        length $\ell$ and length $m$
Output: Ending positions of all occurrences of $P$ in $T$
$Searching(T, \tau, P, \ell, m)$
$R = \varnothing$      // solutions will be stored into set $R$
Let $X$ be the root node of $\tau$
Compute the distance $d$ between $RF(X)$ and the
    prefix with length $\ell$ of $P$
if ($d = 0$) then
    Obtain pointer $PS(X\text{-}0)$ from child node $X$-0
    for (each position $j$ in the list pointed by $PS(X\text{-}0)$) do
        if ($T(j+\ell, j+m-1) = P(\ell+1, m)$) then $R = R \cup \{j+m-1\}$
    endfor
else
    if (child node $X$-$d$ is empty) then $R = \varnothing$
    else if ($X$-$d$ is a leaf node) then
    // perform the special string matching
        Obtain pointer $PS(X$-$d)$ from child node $X$-$d$
        for (each position $j$ in the list pointed by $PS(X$-$d)$) do
            if ($T(j, j+m-1) = P$) then $R = R \cup \{j+m-1\}$
        endfor
    else        // $X$-$d$ is an internal node
        $R = R \cup Searching(T, \tau_d, P, \ell, m)$
        // $\tau_d$ is the sub-tree rooted by $X$-$d$
    endif
endif
return $R$
```

The time complexity of determining whether the prefix with length ℓ of P is equal to the reference string of any internal node in the reference tree is $O(h \times \ell)$. In leaf node X-0 where each string $T(i, i+\ell-1)$ for each i in the list pointed by $PS(X$-$0)$ is equal to $RF(X)$, the number of strings is $O(n \times q_0)$ where $q_0 = (1/\sigma)^{\ell}$. The worst case of the number of character comparisons for verifying these strings is $O(n \times q_0 \times |P|)$. In leaf node X-t ($1 \leq t \leq \ell$) where each string $T(j, j+\ell-1)$ for each j in the list pointed by $PS(X$-$t)$ is different from $RF(X)$, the worst case of the number of character comparisons for verifying the strings stored in this leaf is $O(k \times |P|)$.

Our approach can be used to tackle the exact multiple string matching problem in which r patterns denoted as $P_1, P_2, \ldots, P_r$ are considered. For each pattern, we have to find the ending positions of all occurrences in T. Let m_i be the length of P_i and $\ell_{\min}$ be the shortest length among all patterns. Given a text T, length ℓ and parameter k, we first construct the reference tree by Algorithm Pre-processing where ℓ must be shorter than $\ell_{\min}$. Then, whether and where P_i is in T can be easily determined by searching the reference tree via the reference strings for $1 \leq i \leq r$.

As previously discussed, the worst case of the number of character comparisons for the special string matching in a leaf node is $O(k \times |P_i|)$ where $1 \leq i \leq r$. However, in practical implementations, the number of character comparisons needed is much smaller than $k \times |P_i|$. The performances of our approach and the BWT algorithm for the problem considered are tested and compared in the following section.

5. Experimental Results

The parameters in our approach include the alphabet Σ, length n of text T, size r of patterns $P_1, P_2, \ldots, P_r$, length m_i of pattern P_i for $1 \leq i \leq r$, length ℓ (such that $\ell+1$ is the number of the children for each internal node in the reference tree) and k. As mentioned, in computational biology the researchers often need to search established lists of short motifs in specific segment of DNA sequence. Or, we would like to find inspiring phrases in the Bible or classic novels/speeches. Thus, the alphabets in our experiments include the DNA sequence ($\sigma = 4$) and English language ($\sigma = 63$). Specifically, we designed three experiments to test our approach for various problem instances:

(1) the real DNA sequence;
(2) the Bible; and
(3) the real DNA sequence with longer lengths of texts.

For the first two experiments, the lengths of the texts were set as 1M (10^6) and 4.04M, respectively, and the size of the patterns was $r = 10^4$. Each pattern was randomly selected to be a sub-string of the text. That is, each pattern appears in the text at least once. The lengths of the r patterns may be different from each other. In addition, different pattern groups for different purposes (e.g., finding short/long motifs, searching simple/compound phrases, etc.) may also be different. Thus, we tested ten pattern groups with lengths: 100±20%, 200±20%, ... , 1000±20%, respectively. Note that the range of the lengths of the first group is [80, 120], while that of the last group is [800, 1200]. For each pattern group, 1000 independent pattern sets (with $r = 10^4$ each) were tested. The results for each group reported in the following are the average outcomes of these 1000 independent runs. Based on our preliminary and extensive experiments, the length ℓ was set to be 20 and $k = \ell \times 20 = 400$.

Regarding the third experiment, we prepared eight texts with lengths 50M, 100M, 150M, ... , 400M and the pattern group with length 100±20%. Likewise, each pattern appears in the text at least once. For the pattern group, 100 independent pattern sets were tested, and the average results of these 100 independent runs are reported.

For the purpose of comparison, the famous BWT algorithm was also implemented. Our experimental platform is a personal computer with a 2.4GHz CPU (Intel Q6600) and 4GB main memory. The operating system is CentOS 6.5 64-bit. Our and BWT algorithms were implemented in C language and complied by GUN Compiler Collect (gcc) 4.4.7 with the level 3 of its optimization options (-O3). The computational results of our approach and BWT for the three experiments are reported as follows one by one.

Experiment 1.

We downloaded the sequence of Drosophila Miranda from National Center for Biotechnology Information (NCBI). The gi number is 1036192274 and the ref number is NC_030304.1. The symbols N's were eliminated, and then the length of the whole sequence was about 32M. We simply chose the prefix with length 1M as our test text.

Let RT denote our approach using ℓ = 20. Let t_p(BWT) and t_p(RT) be the pre-processing time (in seconds) of BWT and RT, respectively, for a certain problem instance. Table 3 summarizes t_p(BWT), t_p(RT) and t_p(BWT)/t_p(RT) for each of the ten pattern groups where the better value between t_p(BWT) and t_p(RT) is shown in bold.

Table 3. Comparison on pre-processing time of BWT and RT in Experiment 1

Pattern length	t_p(BWT)	t_p(RT)	t_p(BWT)/t_p(RT)
100±20%	0.471343	**0.178984**	2.633437
200±20%	0.476015	**0.179076**	2.658173
300±20%	0.482690	**0.179114**	2.694876
400±20%	0.488278	**0.179034**	2.727292
500±20%	0.494930	**0.178998**	2.765003
600±20%	0.499993	**0.179049**	2.792493
700±20%	0.505980	**0.179076**	2.825504
800±20%	0.512138	**0.178998**	2.861138
900±20%	0.517186	**0.178910**	2.890761
1000±20%	0.522407	**0.179021**	2.918133

It is easily seen from Table 3 that the pre-processing time t_p(RT) is shorter than t_p(BWT) for every pattern group tested. The pre-processing time of our approach is more efficient than that of BWT in this experiment.

Let t_s(BWT) and t_s(RT) be the searching time (in seconds) of BWT and RT for a certain problem instance. Table 4 presents t_s(BWT), t_s(RT) and t_s(BWT)/t_s(RT) for the ten pattern groups tested. It is obvious that RT outperforms BWT as well in terms of the searching time. When the lengths of the pattern groups grow, the increment of t_s(RT) is rather slight; while that of t_s(BWT) becomes relatively magnificent. The search by way of the reference strings in our reference tree is more efficient than that by BWT, especially for long patterns.

Table 4. Comparison on searching time of BWT and RT in Experiment 1

Pattern length	t_s(BWT)	t_s(RT)	t_s(BWT)/t_s(RT)
100±20%	0.163133	**0.042020**	3.882270
200±20%	0.314126	**0.045183**	6.952305
300±20%	0.470059	**0.048036**	9.785557
400±20%	0.618686	**0.049916**	12.394543
500±20%	0.770425	**0.052571**	14.654943
600±20%	0.920703	**0.055937**	16.459642
700±20%	1.071062	**0.058261**	18.383859
800±20%	1.221137	**0.058369**	20.920985
900±20%	1.370734	**0.063551**	21.569039
1000±20%	1.520726	**0.064559**	23.555600

Figure 7 illustrates the total time spent by BWT and RT for the ten pattern groups. As the lengths of the patterns increase, the performance of BWT degrades, while that of our approach is quite robust. The advantage of our algorithm over BWT is appealing in this experiment.

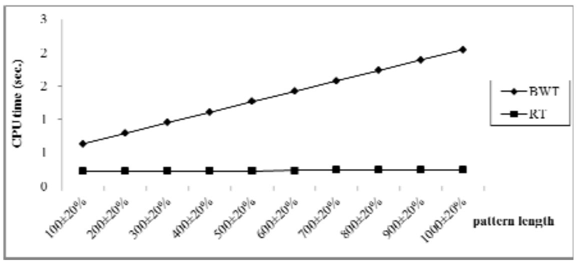

Figure 7. Comparison on total time spent by BWT and RT (real DNA for n = 1M)

Experiment 2.

We downloaded the King James version of the Bible (Old Testament) from the web-site String Matching Researching Tool (SMART) to be the test text. The length of the strings in this version is about 4.04M with σ = 63.

Table 5 exhibits the pre-processing times t_p(BWT) and t_p(RT) as well as the ratio t_p(BWT)/t_p(RT) for the ten pattern groups. From Table 5, it can be seen that t_p(RT) is again shorter than t_p(BWT) for every pattern group tested. The efficiency of constructing the reference tree can thus be convinced.

Table 5. Comparison on pre-processing time of BWT and RT in Experiment 2

Pattern length	t_p(BWT)	t_p(RT)	t_p(BWT)/t_p(RT)
100±20%	5.173900	**1.100900**	4.699700
200±20%	5.253240	**1.102420**	4.765189
300±20%	5.337910	**1.101530**	4.845905
400±20%	5.419230	**1.100650**	4.923663
500±20%	5.503010	**1.100160**	5.002009
600±20%	5.574120	**1.100380**	5.065632
700±20%	5.653880	**1.101080**	5.134849
800±20%	5.730030	**1.100240**	5.207982
900±20%	5.811140	**1.101190**	5.277146
1000±20%	5.884700	**1.100580**	5.346908

Table 6 displays the searching times t_s(BWT) and t_s(RT) as well as the ratio t_s(BWT)/t_s(RT) for the ten pattern groups. We realize from Table 6 that RT outperforms BWT and the superiority of RT over BWT increases as the lengths of the pattern groups grow.

Table 6. Comparison on searching time of BWT and RT in Experiment 2

Pattern length	t_s(BWT)	t_s(RT)	t_s(BWT)/t_s(RT)
100±20%	0.285199	**0.076977**	3.704990
200±20%	0.553022	**0.080230**	6.892958
300±20%	0.820660	**0.081806**	10.031783
400±20%	1.088120	**0.082409**	13.203898
500±20%	1.353990	**0.085557**	15.825590
600±20%	1.619080	**0.088429**	18.309378
700±20%	1.882580	**0.091258**	20.629205
800±20%	2.143070	**0.091729**	23.363059
900±20%	2.400370	**0.094309**	25.452184
1000±20%	2.663520	**0.096227**	27.679549

Figure 8 displayed the total time spent by BWT and RT for the ten pattern groups. When the lengths of the patterns increase, the performance of BWT degrades, while that of our approach is still quite stable. The computational consequence is similar to that of Experiment 1.

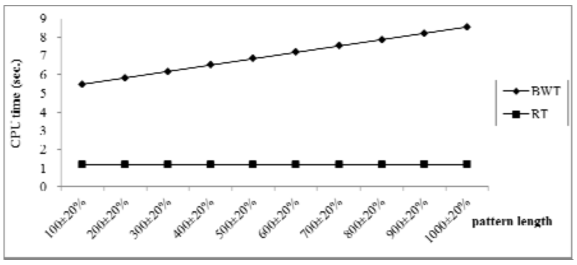

Figure 8. Comparison on total time spent by BWT and RT (the Bible data)

Experiment 3.

From the consequences of Experiments 1 and 2, we know that our approach is elegant and better than BWT (under (n, σ) = (1M, 4) and (n, σ) = (4.04M, 63)) especially for searching long patterns. In this experiment, we would like to know the performances of BWT and our approach for longer texts. We set $r = 10^4$ as well and fixed the range of the lengths of patterns as [80, 120] (instead of long patterns, which favor our approach). We downloaded the first and second chromosomes of Homo sapiens from NCBI with gi numbers 568815364 and 568815352 as well as the ref numbers NT_077402.3 and NT_005334.17, respectively. The symbols N's were eliminated, and then we concatenated them into a long sequence of length about 470M. We chose the texts to be the prefixes of lengths 50M, 100M, 150M, 200M, 250M, 300M, 350M and 400M of the long sequence. For a certain text, patterns were randomly selected from the text.

Table 7 shows the pre-processing times t_p(BWT), t_p(RT) and the ratio t_p(BWT)/t_p(RT). When the length of the text increases, both t_p(RT) and t_p(BWT) increase. The growth of t_p(RT) is gentle, while that of t_p(BWT) becomes serious. Let us observe t_p(BWT)/t_p(RT). It tends to be larger when the text grows longer. For n = 50M, t_p(RT) is three times faster than t_p(BWT); whereas, for n = 350M, it becomes 10 times faster. In fact, under our test platform, when the length of the text becomes longer than 150M, the space requirement for recording its auxiliary tables obliges BWT to access the external memory. This phenomenon could be better explained via the information of t_s(BWT) in the subsequent Table 8. Note that Table 7 does not record the information of t_p(BWT) for n > 350M owing to the lengthy response time.

Table 7. Comparison on pre-processing time of BWT and RT in Experiment 3

Text length	t_p(BWT)	t_p(RT)	t_p(BWT)/t_p(RT)
50M	63.178	**20.039**	3.152711
100M	135.909	**44.195**	3.075205
150M	230.356	**70.065**	3.287771
200M	585.023	**97.215**	6.017802
250M	948.621	**124.916**	7.594071
300M	1273.720	**161.797**	7.872334
350M	2115.930	**201.974**	10.47625
400M	-	**293.387**	-

-: No record.

The results of the searching times are listed in Table 8. Both t_s(BWT) and t_s(RT) grows as n increases. However, t_s(BWT)/t_s(RT) is about 1 for n = 100M; while, it turns to be about 28 for n = 150M and up to 3649 for n = 200M. The dramatic variance is due to the requirement of external memory of BWT in our platform. In the extreme case of n = 350M, t_s(BWT) is slower than t_s(RT) about 8333 times.

Table 8. Comparison on searching time of BWT and RT in Experiment 3

Text length	t_s(BWT)	t_s(RT)	t_s(BWT)/t_s(RT)
50M	0.159	**0.136**	1.168993
100M	0.161	**0.160**	1.005254
150M	5.238	**0.183**	28.68003
200M	756.697	**0.207**	3649.353
250M	1473.220	**0.234**	6295.247
300M	2001.190	**0.316**	6326.153
350M	2897.040	**0.348**	8333.664
400M	-	**0.491**	-

Figure 9 illustrates the total times spent by BWT and RT for the eight texts of different lengths. The superiority of our approach is computationally demonstrated here. It is obvious that the time of our approach increases steady when the length of text increases; while BWT may increase dramatically owing to the requirement of the external memory.

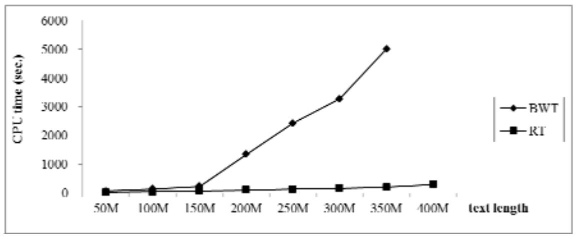

Figure 9. Comparison on total time spent by BWT and RT (for different text lengths)

6. Concluding Remarks

To deal with the exact multiple string matching problem, we design simple and elegant algorithms to construct the sub-strings of the text as a reference tree and search the patterns in the tree. Each internal node X only stores the staring position of one reference string of length ℓ selected from T and has ℓ+1 sub-trees rooted by nodes X-0, X-1, ... , X-ℓ. All sub-strings in T are deliberately distributed into these sub-trees according to the reference strings in the internal nodes and physically stored by their starting positions in the leaf nodes. Specifically, any sub-string starting at position a in T would be distributed from X into X-d if the distance between $T(a, a+\ell-1)$ and $RF(X)$ is d in a recursive way. It is simply stored as a in some leaf node in the form of X-0, or X-t ($1\leq t\leq\ell$) whose size is no greater than k. Such a simple idea makes the searches of the patterns easy and efficient. By comparing the ℓ prefix of the pattern to the reference string in the internal node, we know which among the ℓ+1 child nodes should be further searched. When the search goes from the root to some leaf, a special string matching (for the rest of the pattern if its ℓ prefix equal to some reference string; or the whole pattern otherwise) is performed to determine the solutions. The space complexity is $O(n)$. The number of character comparisons is $O(n\times\ell\times(1-(1-q_0)^{h-1})/q_0)$ in the per-processing phase where h is the height of the reference tree and $q_0 = (1/\sigma)^{\ell}$. In the searching phase, a pattern P needs $O(h\times\ell)$ time to determine whether its prefix is equal to any reference string and uses $O(k\times|P|)$ to deal with the special string matching in the leaf node.

With proper arrangements of ℓ and k, our reference string approach delivers pleasing performance on solving the exact multiple string matching problem. In our experiments including the data sets from the real DNA sequences and the Bible, our approach outperforms the well know BWT algorithm for both the pre-processing (tree constriction) and searching phases. According to the computational results of our experiments, our approach is more efficient than BWT. As the length of the text increases, the performance of our approach degrades gracefully in terms of time and space; whereas, that of BWT may reduce dramatically owing to the access of the external memory to meet its large space requirement.

We emphasize that our approach, in spite of its simplicity, is advantageous and competitive for the exact multiple string matching problem. The formal analysis for the relationship between the height of the tree and parameter k and the comparison of the numbers of character comparisons between the theoretical and computational outcomes are presented in the appendix. How to find a suitable (or even the best) setting of ℓ and k in terms of the given texts with various lengths (n) and patterns with different sizes (r) and lengths (m_i) is worthy of further investigation. It is also interesting to tackle the approximate string matching problem by extending our reference string approach.

References

[1] Boyer, R.S. and Moore, J.S. (1977) A fast string searching algorithm. *Communications of the ACM*, 22, 762-772.

[2] Burrows, M. and Wheeler, D.J. (1999) A block sorting lossless data compression algorithm. Technical Report 124. Digital Equipment Corporation, USA.

[3] Chen, T.W. and Lee, R.C.T (2016) Application of the BWT method to solve the exact string matching problem, *Proceedings of the 33rd Workshop on Combinatorial Mathematics and Computation Theory*, Taipei, Taiwan, 13-14 May, pp. 52-57, Tsinghua University Press, Hsinchu.

[4] Chien, Y.F., Hon, W.K., Shah, R., Thankachan, S.V. and Vitter, J.S. (2015) Geometric BWT: Compressed text indexing via sparse suffixes and range searching. *Algorithmica*, 71, 258-278.

[5] Colussi, L. (1991) Correctness and efficiency of the pattern matching algorithms. *Information and Computation*, 95, 225-251.

[6] Crochemore, M., Hancart, C. and Lecroq, T. (2007) *Algorithms on Strings*. Cambridge University Press, New York.

[7] Crochemore, M. and Rytter, W. (1994) *Text Algorithms*. Oxford University Press, New York.

[8] Crochemore, M. and Rytter, W. (2002) *Jewels of Stringology*. World Scientific Press, Singapore.

[9] Fischer, M.J. and Paterson, M.S. (1974) String-matching and other products. *Proceedings of the 7th SIAM–AMS Complexity of Computation*, Providence, RI, 18-19 April, pp. 113-125. American Mathematical Society, Providence.

[10] Gusfield, D. (1997) *Algorithms on Strings,*

Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York.

[11] Horspool, R.N. (1980) Practical fast searching in strings. *Software: Practice and Experience*, 10, 501-506.

[12] Kärkkäinen, J. and Sanders P. (2003) Simple linear work suffix array construction. *Proceedings of the 30th international conference on Automata, language and programming*, Eindhoven, Netherlands, 30 June - 04 July, pp. 943-955, Springer-Verlag, Berlin.

[13] Knuth, D.E., Morris, J.H. and Pratt, V.R. (1977) Fast pattern matching in strings. *SIAM Journal on Computing*, 6, 323-350.

[14] Lecroq, T. (1992) A variation on the Boyer-Moore algorithm. *Theoretical Computer Science*, 92, 119-144.

[15] Lecroq, T. (2007) Fast exact string matching algorithms. *Information Processing Letters*, 102, 229-235.

[16] Li, H. and Durbin, R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25, 1754-1760.

[17] Morris, J.H. and Pratt, V.R. (1970) A linear pattern-matching algorithm. Technical Report 40. University of California, Berkeley.

[18] Navarro, G. and Raffinot, M. (2000) Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Experimental Algorithmics*, 5, 1-36.

[19] Navarro, G. and Raffinot, M. (2002) *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York.

[20] Simon, I. (1993) String matching algorithms and automata. *Proceedings of 1st American Workshop on String Processing*, Belo Horizonte, Brazil, 13-15 September, pp. 151-157, Springer-Verlag, London.

[21] Smith, P.D. (1994) On tuning the Boyer-Moore-Horspool string searching algorithm. *Software: Practice and Experience*, 24, 435-436.

[22] Szpankowski, W. (2001) *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons, Inc., New York.

[23] Ukkonen, E. (1995) On-line construction of suffix trees. *Algorithmica*, 14, 249-260.

[24] Wu, S. and Manber, U. (1994) A fast algorithm for multi-pattern searching. Technical Report TR-94-17. Department of Computer Science, University of Arizona, USA.