Application of the BWT Method to Solve the Exact String Matching Problem

T. W. Chen and R. C. T. Lee
Department of Computer Science
National Tsing Hua University, Hsinchu, Taiwan
chen81052084@gmail.com

Abstract

In this paper, we first introduce the BWT Method to solve the exact string matching problem. The critical technology of applying the BWT is the algorithm to construct the BWT. If we use an ordinary sorting algorithm to construct the BWT, the BWT Method will be only theoretically interesting and not practically feasible. We will introduce our method to construct the BWT. As explained, our method is easy to understand, easy to implement and efficient. For a text string with length 10M, our method only needs 15 seconds. In this paper, we also show that the BWT Method is exceedingly efficient as compared with several other exact string matching algorithms.

1 Introduction

In this paper, we focus on the exact string matching problem. We are given a text string T and a pattern string P. Our job is to determine whether P appears in T and if it does, the location of T where P appears. Many algorithms have been designed to solve this problem. A comprehensive review of some famous algorithms, such as the Convolution Method ([5], [16] and [6]), Reverse Factor Method([4] and [14]), the Suffix Tree Method ([17], [11] and [15]), the Suffix Array Method [12], the KMP Method ([13] and [9]), the Boyer and Moore Method [1], the Horspool Method [7] and the BWT Method can be found ([2] and [3]). Among all of the algorithms [10], the Suffix Tree Method, the Suffix Array Method and the BWT Method need a pre-processing. After the pre-processing is done, the exact string problem can be very efficiently solved. Our research focuses on the pre-processing part of the BWT method. We propose an efficient algorithm to construct the BWT.

2 The BWT of a Text String T

In the exact string matching problem, we are given a text string $T = t_1 t_2 \dots t_n$ and a pattern $P = p_1 p_2 \dots p_m$. Our job is to determine whether P appears in T and if it does, the location where it appears. One method of the exact string method is the suffix tree approach. We may also use the suffix array approach. For instance, consider the text string $ggtccagaacca\$$. The suffixes of this string are $ggtccagaacca\$$, $gtccagaacca\$$, $tccagaacca\$$, $ccagaacca\$$, $cagaacca\$$, $agaacca\$$, $gaacca\$$, $aacca\$$, $acca\$$, $cca\$$, $ca\$$, $a\$$ and $\$$.

If a pattern P appears in T, it must be a prefix of one of the suffixes. The suffix array approach uses a special binary searching algorithm to solve the exact string matching problem.

In this section, we shall introduce the Burrow Wheeler Transform (BWT for short) which uses the suffix array, but with a much more efficient searching algorithm. Before giving the formal definition of the transform, we shall use an example to explain the main ideas behind it. Consider the following string:

$$T = ggtccagaacca\$$$

Given a string $T = t_1 t_2 \dots t_n$, a rotation of T is $T = t_2 \dots t_n t_1$. For instance, let $T = ggtccagaacca\$$. Then the rotation of T is $gtccagaacca\$g$. If we further rotate this string, we will obtain $tccagaacca\$gg$. We then rotate the string to obtain the following strings:

Table 1

1	$ggtccagaacca\$$
2	$gtccagaacca\$g$
3	$tccagaacca\$gg$
4	$ccagaacca\$ggt$
5	$cagaacca\$ggtc$
6	$agaacca\$ggtcc$
7	$gaacca\$ggtcca$
8	$aacca\$ggtccag$
9	$acca\$ggtccaga$
10	$cca\$ggtccagaa$

11	$ca\$ggtccagaac$
12	$a\$ggtccagaacc$
13	$\$ggtccagaacca$

The next step is to sort the above strings alphabetically. The result is as follows:

Table 2

row	$SA(T)$	rotation
1	13	$\$ggtccagaacca$
2	12	$a\$ggtccagaacc$
3	8	$aacca\$ggtccag$
4	9	$acca\$ggtccaga$
5	6	$agaacca\$ggtcc$
6	11	$ca\$ggtccagaac$
7	5	$cagaacca\$ggtc$
8	10	$cca\$ggtccagaa$
9	4	$ccagaacca\$ggt$
10	7	$gaacca\$ggtcca$
11	1	$ggtccagaacca\$$
12	2	$gtccagaacca\$g$
13	3	$tccagaacca\$gg$

We pick up all of the last characters of the above strings in Table 2 from top to bottom. The resulting string is $BWT(T) = acgacccata\$gg$. Note that, in Table 2, the numbers on the second column are the starting positions of the suffixes of T, denoted as $SA(T)$. In our example, $SA(T) = \{13,12,8,9,6,11,5,10,4,7,1,2,3\}$. The relationship between $SA(T)$ and $BWT(T)$ is as follows:

$$BWT(T)_i = T[n], \text{ if } SA[i] = 1$$
$$BWT(T)_i = T[SA[i] - 1], \text{ otherwise}$$

3 The Backward Search

Consider row 2 in the BWT matrix in Table 2. Row 2 starts with a and $BWT(T)_2 = c$. This actually means that there exists a suffix starting with ca. For every string $S = s_1 s_2 \dots s_k$, there are two corresponding $s(S)$ and $e(S)$. $s(S)$ and $e(S)$ are the first row of the BWT matrix which starts with S and the last row of BWT matrix which starts with S respectively. For instance, $S = cca$, $s(S) = 8$ and $e(S) = 9$. Thus, our job is to find the $s(S)$ and $e(S)$ for a given pattern S.

To apply the backward search of the BWT to solve the string matching problem, two functions have to be available: $Count(x)$ and $Precede(i, x)$. The function $Count(x)$ is the total number of characters in T that are lexicographically smaller than x. For our example, $T = ggtccagaacca\$$, $Count(a) = 1$, $Count(c) = 5$, $Count(g) = 9$ and $Count(t) = 12$. $Count(c) = 5$ because four a's and one \$ appear in T. This means that any suffix starting with c cannot be in row 1 to row 5 of the BWT matrix. It must start from row 5+1=6 in the BWT matrix. Similarly, $Count(t) = 12$. Any suffix starting with t must start from row 12+1=13. The function $Precede(i, x)$ is the number of character x in $BWT(T)_1$ to $BWT(T)_{i-1}$. In our example, $BWT(T) = acgacccata\$gg$, $Precede(6, a) = 2$, $Precede(8, a) = 2$ and $Precede(8, g) = 1$. The meaning of $Precede(6, a) = 2$ is that from row 1 to row 5 of BWT matrix, there are two rows ending with a. These two functions $Count(x)$ and $Precede(i, x)$ can be stored in matrixes in pre-processing time.

Next, given the range $[s(y), e(y)]$ of a string y, computing the range $[s(xy), e(xy)]$ for the string xy for any character x can be done by the following formulas:

$$s(xy) = Count(x) + Precede(s(y), x) + 1$$
$$e(xy) = Count(x) + Precede(e(y) + 1, x)$$

Initially, y is an empty string $\emptyset$. Therefore, $s(y)$ and $e(y)$ are initialized as 1 and n respectively because every rows in the BWT matrix are started with an empty string. For example, given a pattern $P = gt$, we start with $x = t$ and $y = \emptyset$. Since $Count(t) = 12$, $Precede(1, t) = 0$ and $Precede(14, t) = 1$, $s(t) = 12 + 0 + 1 = 13$ and $e(t) = 12 + 1 = 13$. Next, we can use the results of previous step to find the $s(gt)$ and $e(gt)$. By pre-processing, we know $Count(g) = 9$, $Precede(s(t), g) = Precede(13, g) = 2$ and $Precede(e(t) + 1, g) = Precede(14, g) = 3$. Therefore, we can find that $s(gt) = 9 + 2 + 1 = 12$ and $e(gt) = 9 + 3 = 12$. This means that we have found one solution in row 12 of the BWT matrix. Finally, we conclude that string gt appears at location 2 of text T since the starting position of row 12 is 2, which is recorded in $SA(T)$. Next, we present a case, $P = agt$, that does not exist. Let $y = gt$ and $x = a$. Previously, we have found that $[s(y), e(y)] = [s(gt), e(gt)] = [12, 12]$. By pre-processing, we know $Count(a) = 1$, $Precede(s(gt), a) = Precede(12, a) = 4$ and $Precede(e(gt) + 1, a) = Precede(13, a) = 4$. Then, $s(agt) = Count(a) + Precede(s(gt), a) + 1 = 1 + 4 + 1 = 6$ and $e(agt) = Count(a) + Precede(e(gt) + 1, a) = 1 + 4 = 5$. Since $e(agt) < s(agt)$, we conclude that string agt does not exist in T.

We can see that the backward search of the BWT is exceedingly fast and it's time-complexity is proportional to the length of the text and the worst case time-complexity is $O(m)$. The critical problem is how to find the $BWT(T)$. A straight forward method is to sort all of the rotated suffixes. Suppose the length of the text is 10 million long. Then, we have 10 million strings to sort. The authors of the paper [9] proposed an efficient method to obtain the suffix array. This method, which may be called the KS Method, can be easily modified to construct the $SA(T)$ in linear time. The KS method is hard to understand and hard to implement. In the next section, we introduce our method to construct the $BWT(T)$ which is based on the KS method.

4 Our Method to Construct $BWT(T)$

The main idea of our method is that when we compare two strings, we may compare the first three characters. If they are not identical, the comparison is already done. If they not identical, we just continue to compare the next three characters. This decreases a lot of time.

In our algorithm, we will use SA array for $\{x_a, x_{a+1}, \dots, x_b\}$, denoted as $SA\{x_a, x_{a+1}, \dots, x_b\}$. If $SA[i] = j$, the i-th smallest element of $\{x_a, x_{a+1}, \dots, x_b\}$ is x_j. We also use a term rank. If x_j is the k-th smallest element of $\{x_a, x_{a+1}, \dots, x_b\}$, the rank of x_j is k.

Instead of sorting the suffixes, we may sort the prefixes of the suffixes. For instance, consider suffix gtcca$ and suffix ggtcca$. Let us only compare their prefixes with which contain three charcaters. Such prefixes for gtcca$ and ggtcca$ are gtc and ggt. We compare these two prefixes and can see easily that ggt is alphabetically smaller than gtc. Therefore we can immediately conclude that suffix ggtcca$ is alphabetically smaller than suffix gtcca$.

We shall call a string with three characters a "triple." We now first add two 0's to our input string. That is, our string becomes:

$$T = ggtccagaacca\$00$$

Table 3 shows all the triples of $T = ggtccagaacca\$$. We then use radix sort to sort these triples. After radix sort, we can obtain the sorted triples which are shown in Table 4.

Consider the triples in Table 4. For $i = 6, SA[6] = 11$. This means that the 6-th smallest triple is $triple_{11}$. Note that for this case, the ranks of $triple_4$ and $triple_{10}$ are both 8.

Table 3

Index	$triple_i$
1	ggt
2	gtc
3	tcc
4	cca
5	cag
6	aga
7	gaa
8	aac
9	acc
10	cca
11	$ca\$$
12	$a\$0$
13	$\$00$

Table 4

Index	SA		$rank$
1	13	$\$00$	1
2	12	$a\$0$	2
3	8	aac	3
4	9	acc	4
5	6	aga	5
6	11	$ca\$$	6
7	5	cag	7
8	4	cca	8
9	10	cca	8
10	7	gaa	10
11	1	ggt	11
12	2	gtc	12
13	3	tcc	13

If the ranks of two $triple_a$ and $triple_b$ are the same, we continue to compare the rank of $triple_{a+k}$ and $triple_{b+k}$, k is initialized as 3. If the ranks of $triple_{a+k}$ and $triple_{b+k}$ are still the same, we set $k = k + 3$ and compare again until we can find which one is smaller. Since $triple_4$ and $triple_{10}$ are identical, we know that we must compare $triple_{4+3} = triple_7$ and $triple_{10+3} = triple_{13}$. The rank of $triple_7 = 10$ and that of $triple_{13} = 1$. We say that the rank of $triple_7$ is larger than that of $triple_{13}$. Since the rank of $triple_7$ is larger than the rank of $triple_{13}$, we may say that the rank of $triple_4$ is larger than that of $triple_{10}$. Thus we update their ranks and obtain the following table:

Table 5

Index	SA		$rank$
1	13	\$00	1
2	12	a\$0	2
3	8	aac	3
4	9	acc	4
5	6	aga	5
6	11	ca\$	6
7	5	cag	7
8	10	cca	8
9	4	cca	9
10	7	gaa	10
11	1	ggt	11
12	2	gtc	12
13	3	tcc	13

Now, the ranks are all different. We then use the following formula

$$BWT(T)_i = T[n], \text{if } SA[i] = 1$$
$$BWT(T)_i = T[SA[i] - 1], \text{otherwise}$$

to obtain the $BWT(T) =$ acgaccctaa\$gg . The algorithm of our method is as follows:

The Algorithm of our Method
Input: a text T with n characters
Output: $BWT(T)$

For i from 1 to n **do**
  $triple_i = t_i t_{i+1} t_{i+2}$.
End
Radix sort the triples and produce the $SA\{triple_1, triple_2, \dots, triple_n\}$.
Set $rank_{SA[1]} = 1$.
For i from 2 to n **do**
  if $triple_{SA[i]} = triple_{SA[i-1]}$
    $rank_{SA[i]} = rank_{SA[i-1]}$
  Else
    $rank_{SA[i]} = i$
End
For i from 2 to n **do**
  $k_i = 3$
End
While some ranks are identical **do**
For each (a, b) (such that the rank of $triple_{SA[a]}$ equals to that of $triple_{SA[b]}$ and b − a is the maximum) **do**
    Sort $rank_{SA[a]+k}$ to $rank_{SA[b]+k}$ to produce $SA\{rank_{SA[a]+k}, rank_{SA[a+1]+k}, \dots, rank_{SA[b]+k}\}$;
    $SA\{triple_a, triple_{a+1,\dots,} triple_b\} = SA\{rank_{SA[a]}, rank_{SA[a+1]}, \dots, rank_{SA[b]}\}$;
    For i from $a+1$ to b **do**
      if $rank_{SA[i]+k} = rank_{SA[i-1]+k}$
        $rank_{SA[i]} = rank_{SA[i-1]}$
      Else
        $rank_{SA[i]} = i$
      $k_i = k_i + 3$
    End
  End
End of while
$BWT(T)_i = T[n]$, if $SA[i] = 1$
$BWT(T)_i = T[SA[i] - 1]$, otherwise

5 Experiments

The environment of our experiment is on the Windows 7 64-bit computer which equipped with two 3.20GHz cores Intel(R) Core(TM) i5-4570 CPU, 8GB RAM and 1000GB hard disk. The experiment results are the average of 100 randomly generated texts.

- Experiment 1: A Comparison of Our Method and Ordinary Method to Construct the BWT.

  By ordinary method to construct the BWT, we mean the method of using some $O(n \log n)$ sorting method. That is, we compare two complete suffixes. If the text string is very long, there will be a large number of long suffixes. Therefore this kind of method will be highly inefficient. Table 5 shows the experimental results. The vertical dimension indicates the methods, the horizontal dimension indicates the length of input text T, and the cells in the table contain the execution time of each conditions. The experiment results are the average of 100 randomly generated texts. From this result, we can see that our method is much more efficient than an ordinary method.

Table 5

Length / Method	10K	100K	1M	10M
Our method	0.0012	0.158	0.29	15.7
Ordinary Sorting	0.12	34	5051	69243

- Experiment 2: Comparison of the searching Time of the BWT Method and the Searching Time of Brute-Force Method.

  In the experiment, we compare Brute-Force Method and the BWT Method. Given a text $T = t_1 t_2 \dots t_n$ and a pattern $P = p_1 p_2 \dots p_m$, the Brute-Force Method compares $t_i t_{i+1} \dots t_{i+m}$ and $p_1 p_2 \dots p_m$ for i from 1

to $n-m$. In this experiment, we randomly generated a text string and 1000 randomly generated patterns. Table 6 shows the result of searching time in microseconds. The vertical dimension indicates the length of text T and the searching algorithms, and the horizontal dimension indicates the length of the pattern P. We can see that the BWT Method is much more efficient than the Brute-Force Method. Besides, it is interesting to note that for a fixed text length, the searching time of the BWT Method is independent of the pattern length.

Table 6

T \ P		50	100
10K	BWT	0.000001	0.000001
	Brute-Force	0.0024	0.0045
100K	BWT	0.000001	0.000001
	Brute-Force	0.023	0.045
1M	BWT	0.000001	0.000001
	Brute-Force	0.23	0.45
10M	BWT	0.000002	0.000002
	Brute-Force	2.33	4.58

- Experiment 3: Comparison of the Searching Time of the BWT Method and the Searching Time of Some Other Methods.

  In this experiment, we tested the KMP Method, Reverse Factor Method and the BWT Method. The result is shown in Table 7. The vertical dimension indicates the length of text T and the searching algorithms, and the horizontal dimension indicates the length of the pattern P. As can be seen, the BWT method is much faster than the other methods.

Table 7

T \ P		10	50	100
100K	BWT	0.0002	0.0002	0.0002
	KMP	0.1	0.1	0.1
	RF	0.023	0.007	0.006
1M	BWT	0.0002	0.0002	0.0002
	KMP	1	1	1
	RF	0.23	0.06	0.03
10M	BWT	0.0002	0.0002	0.0002
	KMP	10	10	10
	RF	2.37	0.63	0.36

6 Concluding Remarks

By examining the results of Experiment 1, Experiment 2 and Experiment 3, we can conclude that the BWT Method is suitable for multiple pattern searching. This is the case for all exact string algorithms which need pre-processing, such as the suffix tree method and the suffix array method. The pre-processing always takes some time. But once it is done, it is done. It is common these days that some research organization constructs a huge data base of text strings and it will be available for researchers around the world. In such a situation, pre-processing is worthwhile. Of course, we do not like the pre-processing time to be exceedingly long. The pre-processing of the BWT Method is to construct the BWT for a given text. From the result of Experiment 1, we can see that our method to construct the BWT Method is not too long. For instance, for a text string with 10M length, the time needed to construct the BWT is only 15.7 seconds. Our experience tells us that this pre-processing method is much better than that for the suffix tree Method, for instance. Not mentioning the time needed to construct the suffix tree, the memory needed to store the suffix tree is much larger than that needed for the BWT Method.

The contribution of our work is to present an efficient algorithm to construct the BWT and thus make the BWT Method feasible. Our experimental results showed that the BWT Method is much more efficient than any other exact string matching algorithm.

7 Future Works

For future works, we will make a study of some research done on the searching algorithms of the BWT approach.

We will also apply the BWT Method to solve the repeating group finding problem. This problem is defined as follows: We are given a string T. The problem is finding all repeating sub-strings in T under the condition that the lengths of the substrings are all larger than a threshold because short repeating groups are not meaningful. This problem is quite useful for biological research. In [8], Kung used the dynamic programming method to solve the problem. We believe that our method to construct the BWT can be used too.

References

[1] Boyer, R. S. and Moore, J. S., A Fast String Searching Algorithm, Communications of ACM, Vol. 20, No. 10, 1977, p.p. 762–772.

[2] Burrows, Michael and Wheeler, David J.. A block sorting lossless data compression algorithm. *Technical Report124, Digital Equipment Corporation, 1994*.

[3] Ferragina P. and Manzini G.. Opportunistic data structures with applications. *Proceedings of the 41st Symposium on Foundations of Computer Science, 2000*.

[4] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W. and Rytter, W., Speeding Up Two String-matching Algorithms, Algorithmica, Vol. 12, 1994, pp. 247-267.

[5] Fischer, M. M. and Paterson, M. S., String-Matching and Other Products, SIAM-AMS Proceedings, Vol. 7., 1974, pp. 113-125 (In "Complexity of Computation", R.M. Karp.)

[6] Hou, K. W., The Discrete Convolution Method on Solving the Exact String Matching Problem, MS Thesis, 2012, Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan.

[7] Horspool, R. N., "Practical Fast Searching in Strings", Software Practice and Experience, Vol. 10, 1980, pp. 501-506.

[8] Kung, B. L. and Lee, R. C. T. On the Repeating Group Finding Problem. Takming University of Science and Technology, Taipei, Taiwan.

[9] Knuth, D. E., Morris, J. H. and Pratt, V. R., Fast Pattern Matching in Strings, SIAM Journal on Computing, Vol. 6, No.2, 1977, pp. 323-350.

[10] Lee, R. C. T., Chen, K. H., Lu, C. W. and Ou, C. S. and Shieh, Y. K. Introduction to String Matching Algorithms, Lecture Notes, National Tsing Hua University, Hsinchu, Taiwan 300.

[11] McCreight, E. M., A Space-Economical Suffix Tree Construction Algorithm, Journal of the ACM, Vol. 23, 1976, pp. 262-272.

[12] Manber, U. and Myers, G., Suffix Arrays: A New Method for On-line String Searches, SIAM Journal on Computing, Vol. 22, 1993, pp. 935-948.

[13] Morris, J. H. and Pratt, V. R., A Linear Pattern-matching Algorithm, Technical Report 40, University of California, Berkeley, 1970.

[14] Raffinot, M., On the Multi Backward Dawg Matching Algorithm, Proc. The 4th South American Workshop on String Processing, 1997, pp. 149-165.

[15] Ukkonen, E., On-line Construction of Suffix Trees, Algorithmica, Vol. 14, 1995, pp. 249-260.

[16] Wu, B. H., Convolution and Its Application to Sequence Analysis, MS Thesis, 2004, National Chi Nan University, Puli, Nantou, Taiwan.

[17] Weiner, P., Linear Pattern Matching Algorithms, 14th Annual IEEE Symposium on Switching and Automata Theory, 1973,pp. 1–11.