Backward Exact String Searching Strategy

Ebaa Fayyoumi* and Ahmed Al-Jaber*

Received on Oct. 8, 2002 Accepted for publication on April 9, 2003

Abstract

A new strategy is presented which finds all occurrences of one given string within another. It is worth mentioning that there are no studies in literature related to change the direction of text scan by starting the match of the pattern at the end of the text, often allows the algorithm to proceed faster. Performance of this method was measured through applying some of exact string searching algorithms and their backward on different text sizes. It affords an improvement of the time required by reducing it up to 5.6%.

Keywords: String searching; Text editing; Information retrieval; Boyer-Moore-Horspool Algorithm; Raita Algorithm; Cycle Algorithm.

1. Introduction

String searching is an important component of many problems, including text editing, information retrieval and symbol manipulation. The string searching or string-matching problem consists of finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over same alphabet. Let $pat$ be the $i^{th}$ character in the pattern string $Pat = pat_0...pat_m$, of length $m$, and let $text$ be the $j^{th}$ character in the text string $Text = text_0...text_n$, of length $n$ which is fairly larger than $m$.

There are many algorithms that focus on this problem [1-24]. However, string searching requires two kinds of solutions depending on which string, the pattern or the text is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem [8,4,5,13]. The notation of indexes realized by tree or automata is used in the second kind of solutions [15], this study deals with the first kind only.

The best way to understand how string matching algorithm works, is to imagine that there is a window on the text. This window has the same length as the pattern "pat". This window is first aligned with the left end of the text and then the string matching algorithm scans if the characters of the window match the characters of the
pattern (this specific work is called attempt). After each attempt the window is shifted to right over the text until it goes over the right end of the text (this mechanism is usually called the sliding window mechanism). A string-matching algorithm is a succession of attempts and shifts. The aim of a good algorithm is to minimize the work done during each attempt and to maximize the length of the shift [13].

Most of the previous studies in exact string searching algorithms differ in the way of performing the comparison between pattern characters and text characters at each attempt. Four categories arise: the most natural way to perform the comparisons is from left to right, which is the reading direction, such as the "Shift-Or Algorithm" [1] and "KMP Algorithm" [12]. Generally, the second category leads to the most practical algorithms via performing the comparison from right to left as described in "Boyer-Moore Algorithm" [7] and "Turbo-Boyer-Moore Algorithm" [8]. But the best theoretical bounds are reached when comparisons are done in a specific order as in "Two-Way String-Matching Algorithm" [9] and "Time-Space-Optimal String Matching Algorithm" [10]. Finally, some algorithms are not relevant for which the order in which the comparisons are done such as "Quick Search Algorithm" [20], "Raita Algorithm" [17], "Boyer-Moore-Horspool Algorithm" [21], and the most recently one is Cycle Algorithm [14].

This article consists of five sections. An introduction has been introduced with fine details in first section. Second section explains the Cycle Algorithm with an example, while section three concentrated in the new methodology. The resulting codes are presented in the forth section. Finally the conclusion and further work are added.

2. Cycle Algorithm.

This algorithm finds the exact occurrences of a pattern \( pat_{n} \ldots pat_{1} \) in the text \( text_{n} \ldots text_{1} \). Cycle Algorithm treats the pattern as a cycle logically, this means that there is no fixed order of comparison. At the beginning of the search process, the algorithm chooses the first character in the pattern to be compared first. In each checking step, it always starts from comparing the mismatch character in the last step. When the comparison successfully turns a round in one checking step, a complete match is found, if there is a mismatch, the character that is next to the right most character of the current substring should be chosen for checking. If that character doesn't occur in the pattern, the skip distance is \( m+1 \), and this is the maximum distance that this algorithm can move. Otherwise, the text is shifted so as to align that character with its right most occurrence in the pattern character [14].

The Cycle Algorithm is based on the idea of Smith's adaptive method [22]. The mismatched character should be given a high priority in the next checking step. The difference of the two methods is that the information of mismatched results is used in a statistical way in Smith's adaptive method, on the other hand, the result comparing
Backward Exact String Searching Strategy

characters is chosen in a self-adaptive way in Cycle method. The character, which is most difficult to be matched with in the pattern, will be most frequently chosen to compare first [22,14].

Example 1

Assuming the text string is "ABCDCDFBCDRBCDFABCDFABCDR", and the pattern string is "ABCDR". The Cycle Algorithm is used in this example, because it is faster than other algorithms.

According to the given pattern, the value of skip array is

Skip ['A'] = 5
Skip ['B'] = 4
Skip ['C'] = 3
Skip ['D'] = 2
Skip ['R'] = 1

Otherwise, the value of the skip array is m+1 = 6.

ABCDCDFBCDRBCDFABCDFABCDR

ABCDR

At the beginning, the cycle starts the comparison from left to right. It can be noticed that there is a match between the character pat_0 to pat_3 with the corresponding character in the text. There is a mismatch at pat_4 with the corresponding character 'C' at text_4. The number of characters passed for comparisons is five, in addition, four comparisons are needed to test whether it reaches the end of the pattern or not. According to the skipping step, since the character 'D' at text_5 occurs in the pattern, two locations right are required to align text_5 with pat_5. The result is displayed as

ABCDCDFBCDRBCDFABCDFABCDR

ABCDR

Since the mismatch occurs at pat_4, the comparison starts from that position. There is a mismatch at pat_4. To align text_5 with pat_4, four positions are needed to moved. One character comparison is needed. The result is depicted as

ABCDCDFBCDRBCDFABCDFABCDR

ABCDR

Starting the comparison at pat_4, there is a match. Now reaching the end of the pattern, the comparison as a logical circle Pat_0 must be continued. There is a mismatch at pat_4. Two character comparisons and one logical end comparison are needed. Moving four positions to align text_11 with pat_11. The result is shown as

ABCDCDFBCDRBCDFABCDFABCDR

ABCDR

419
There is a mismatch at \( \text{pat}_0 \). Only one comparison is needed. Moving five positions. The result is as follows

\[
\text{ABCD CDCFBC DRBCDFABCDFA BCDR}
\]

It can be noticed that there is a match between \( \text{pat}_0 \) to \( \text{pat}_3 \) with the corresponding text characters. There is a mismatch at \( \text{pat}_4 \). Five comparisons and four logical end comparisons are needed. In addition to five positions are required to be shifted. The result is as illustrated.

\[
\text{ABCD CDCFBC DRBCDFABCDFA BCDR}
\]

Finally, there is a match between each character in the pattern with the corresponding characters in the text. So, the pattern is found. Ten comparisons are required, including five for character comparison and five for the logical end comparisons. The total number of character comparison required is 19, while the total number of logical end comparisons is 14. But the total number of shift is 6.

3. **Backward String Searching Strategy**

It is more conventional to start scanning the text in exact string searching algorithms from the beginning. Consequently, none of these algorithms starts scanning form the end of the text. This study reflects the novel idea of changing the direction of text scan and evaluates its effect in the pattern pre-process phase, number of comparison and shift, and also running time.

Many algorithms try to improve the length of the shift through matching some suffixes of the pattern [7,17,14]. In rapport of that it is possible to invert all string searching algorithms to improve the length of the shift via matching some prefixes of the pattern by scanning the character of the window from right to left. Three algorithms have been adapted from literature for testing and to prove its validation where each algorithm has its own way in searching [21, 2, 17, 14]

In pre-process phase, the skip array does not express the right most occurrences of character in the pattern, on the contrary of that the skip array has to express the left most occurrences of character in the pattern in order to conserve the correctness of algorithm.

As first impression, it may appear that pre-processing the pattern from left to right equals to pre-process from right to left. However, the following example gives a deep analyzing of the difference between pre-processes the pattern from right to left and from left to right.
Example 2

Assume there exist three patterns "ABCD\textsuperscript{R}\textsuperscript{K}B\textsuperscript{K}\textsuperscript{K}B\textsuperscript{C}\textsuperscript{C}K" need to be processed from right to left and from left to right.

1) Pattern "ABCD\textsuperscript{R}\textsuperscript{K}B\textsuperscript{K}\textsuperscript{K}B\textsuperscript{C}\textsuperscript{C}K"

Pre-process the pattern from right to left

\begin{align*}
\text{Skip}'A' &= 4 \\
\text{Skip}'B' &= 3 \\
\text{Skip}'C' &= 2 \\
\text{Skip}'D' &= 1 \\
\text{Skip}'R' &= 5 \\
\end{align*}


Pre-process the pattern from left to right

\begin{align*}
\text{Skip}'A' &= 5 \\
\text{Skip}'B' &= 1 \\
\text{Skip}'C' &= 2 \\
\text{Skip}'D' &= 3 \\
\text{Skip}'R' &= 4 \\
\end{align*}

Total number of Shift = 15.

2) Pattern "EKKKCB"

Pre-process the pattern from right to left

\begin{align*}
\text{Skip}'E' &= 5 \\
\text{Skip}'K' &= 2 \\
\text{Skip}'C' &= 1 \\
\text{Skip}'B' &= 6 \\
\end{align*}

Pre-process the pattern from left to right

\begin{align*}
\text{Skip}'E' &= 6 \\
\text{Skip}'K' &= 1 \\
\text{Skip}'C' &= 4 \\
\text{Skip}'B' &= 5 \\
\end{align*}

Total number of Shift = 14.

3) Pattern "ABCKC"

Pre-process the pattern from right to left

\begin{align*}
\text{Skip}'A' &= 5 \\
\text{Skip}'B' &= 1 \\
\text{Skip}'C' &= 2 \\
\text{Skip}'K' &= 4 \\
\end{align*}

Pre-process the pattern from left to right

\begin{align*}
\text{Skip}'A' &= 4 \\
\text{Skip}'B' &= 3 \\
\text{Skip}'C' &= 1 \\
\text{Skip}'K' &= 5 \\
\end{align*}

Total number of Shift = 13.

The difference appears only when there is a repetition in the pattern character. Otherwise, the total number of shift will be equal. During the searching phase the algorithm starts at the end of the text to find an occurrence or all occurrences of the pattern in the text by parsing the characters of the window from the left to right, instead of parsing form right to left. It can be useful if the text is large and there exist
a prior knowledge about the frequency of each character. Then the high frequency character takes the large amount of skip from two auxiliary tables which are computed in Preprocessing phase.

Here Boyer-Moore-Horspool Algorithm, Raita Algorithm, and Cycle Algorithm are inverted as shown in figures 1, 2, and 3.

```c
char pat[m] 
char text [m]
int skip [alpha+1]
int n; {text length}
int m; {pattern length}
    int poss ; { number of the pattern occurrence }

1. // Preprocess Phase
2.
3. for (i=0 ; i< alphasize ; i++)
4.    L_occ[i]= m;
5.
6.     j=1;
7. for (i= m-1 ; i>=0 ;i--)
8.    {   L_occ[pat[i]]= m-j;  j++;}
9.
10. //Searching Phase
11.
12. i= n-m;
13. j= m-1;
14. pos=1;
15. while (i>= 0)
16.    {
17.        k = i;
18.        j = 0;
19.        while ((text[k] == pat[j])&&(j<m))
20.           { k++;  j++; }
21.        if (j == m)
```
Backward Exact String Searching Strategy

```
22. { position[pos] = i; pos++; }
23. i = i - L_occ[ text[i] ];
24. }
```

**Figure 1.** Backward Boyer-Moore-Horspool Algorithm (BBMHA)

```c
char pat[m] }

char text [m]
int skip [alpha+1]
int n: {text length}
int m: {pattern length}
int poss : { number of the pattern occurrence }
1. // Preprocess Phase
2.
3. for (j=0 : j<alphasize : j++)
4.    skip[j] = m :
5.
6. int jj=0;
7. for (j=m-1 : j>0 : j--)
8.    { skip[pat[j]] = m-jj-1 ; jj++ ; }
9.
10. //Searching Phase
11.
12. poss = 0;
13. i = n- m;
14. j =0 ;
15. while (i >=0)
16. {
17.    if (text[i]==pat[0] )
18.     if (text[i+m-1]==pat[m-1])
19.     {
20.         for ( j=1, k=i0+1 : j<m-1 ; k++ , j++)
21.             if (text[k]!= pat[j]) break; }
22.         if (j==m-1)
23.             {poss++; position[poss]=i; }
24.     }
25. }
26. i = i - skip[text[i]];  // End while;
27. }
```

**Figure 2.** Backward Raita Algorithm (BRA)

```c
char pat[m] }

char text [m]
int skip [alpha+1]
int n: {text length}
int m: {pattern length}
int poss : { number of the pattern occurrence }
```
Figure 3. Backward Cycle Algorithm (BCA)

Example 3

Assuming the text string is "ABCDCDFBCDCFRBCDFABCDFABCDR", and the pattern string is "ABCDR". The Backward Cycle Algorithm is used in this example. According to the given pattern, the value of skip array is.

Skip['A'] = 5
Skip['B'] = 4
Backward Exact String Searching Strategy

Skip ['C'] = 3
Skip ['D'] = 2
Skip ['R'] = 1

Otherwise, the value of the skip array is m+1 = 6.

At the beginning, the backward algorithm starts the comparison from left to right at the end of the text as

ABCD

It can be noticed that there is a match between each character in the pattern with the corresponding characters in text, so the pattern is found. Five comparisons is required to detect an occurrence of the pattern. According to skipping step, six locations have to be moved left to align text9 with pat8 since the character 'F' at text9 doesn't occur in the pattern. The result is depicted as the following

ABCD

There is a mismatch at pat0 with the corresponding character 'F' at text14 so one comparison is needed, in addition to fourth position need to be moved left since the character 'D' occurs in the pattern. The result is shown as the following

ABCD

Since the mismatch in previous step occurs at pat0 the comparison starts from the same position. Where a mismatch found at pat0. To align text0 with pat5 five positions have to be moved left, one comparison is needed. The result is displayed as the following

ABCD

Starting the comparison at pat0 there is a mismatch. One comparison is needed with four positions have to be moved left, relative to 'D' text character as illustrated below.

ABCD

425
It can be noticed that there is a mismatch between text$_2$ and pat$_5$, so one comparison is needed beside two position moves, to the left as it is depicted below.

ABCD
DFBCDR
BCDFABCDFABCDR
ABCDR

Finally, there is a match between the character pat$_6$ to pat$_3$ with the corresponding text characters but there is a mismatch at pat$_4$ with the corresponding text character 'C' at text$_4$. The number of comparison passed is five.

The previous example shows that both top down and bottom up of text scan has the same number of shift. However the total numbers of comparison and logical end comparison have been reduced dramatically to 14, and 9 instead of 19 and 14 respectively.

4. Experimental Results

Boyer-Moore-Horspool, Raita Algorithm, and Cycle Algorithm were compared with their backward algorithms. These algorithms are implemented using C language, due to it's usage as a system language and also most of the previous string searching algorithms use it [8,14].

These algorithms have been applied on three different English texts extracted from the Internet, online papers, and documents in different fields such as health, sports, computer, architecture, CNN news...etc.

1. The first text size was about 0.5 million characters (exactly 530,440 characters).
2. The second text size was about 1.0 million characters (exactly 1,111,842 characters).
3. The third text size was about 2.0 million characters (exactly 2,110,613 characters).

A C program was designed to select randomly 560 patterns vary in lengths from 3 to 30. Each pattern length has 20 different patterns with different number of occurrences. This program is applied on the above three texts thereafter we call text1, and text3. Another program is designed also to select 1,120 patterns that differ in length from 3 to 30. Each pattern length has 40 different patterns with various numbers of occurrences.

The search cost is measured using three criteria: total time, average number of comparisons, and average number of shifts to find all occurrences of all patterns in each text. The time during the pre-processing phase of pattern is taken each time as
Backward Exact String Searching Strategy

an average over 13 searches. The tests were done on Pentium 1.32 RAM microcomputer.

Table 1 depicts an improvement in comparison through changing the direction of text scan. ranges from 0.96% to 1.933% depending on the size of the text and the type of the algorithm. Also there is an improvement in the average number of shift in BCA ranges from 1.659% to 2.0%, while in the other algorithm the average number of shifts is almost the same as shown in table2. Generally, the improvement in running time is a reasonable result obtained from the improvement in comparisons and shifts as shown in table3 the time improvement reaches (4.18%) using BBMHA, (3.94%) using BRA, and (5.69%) using BCA.

Figures 4, 5 and 6 show the improvement in the number of comparisons, the number of shifts, and running the time, respectively.

It can be noticed that Backward Cycle Algorithm achieved the highest improvement reached to 5.618% in the average running time. 2.193% in the average number of comparison, and 2.257% in the average number of shifts. This may be attributed to which character is chosen to determine the amount of shift. In other words, Cycle Algorithm chooses the next text character that corresponds to the last character in the pattern to determine amount of shift, so the maximum shift length is equal to m+1 [14]. But Boyer-Moore-Horspool Algorithm and Raita Algorithm choose the text character that corresponds to the last character in the pattern to determine amount of shift, therefore the maximum shift length is equal to m [8,23].

Table 1. Average of averages of total number of comparison and percent improvement of Top down and Bottom up through out different text lengths.

<table>
<thead>
<tr>
<th>Algorithm*</th>
<th>Text1 Improvement%</th>
<th>Text2 Improvement%</th>
<th>Text3 Improvement%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMHA</td>
<td>194621</td>
<td>398422</td>
<td>75993</td>
</tr>
<tr>
<td>BBMHA</td>
<td>19783</td>
<td>393388</td>
<td>751121</td>
</tr>
<tr>
<td>RA</td>
<td>128278</td>
<td>266417</td>
<td>50161</td>
</tr>
<tr>
<td>BRA</td>
<td>126906</td>
<td>262859</td>
<td>502076</td>
</tr>
<tr>
<td>CA</td>
<td>118759</td>
<td>243577</td>
<td>497121</td>
</tr>
<tr>
<td>BCA</td>
<td>16884</td>
<td>238021</td>
<td>458185</td>
</tr>
</tbody>
</table>

* BMHA denoted to Boyer-Moore –Horspool Algorithm , RA to Raita Algorithm, and CA to Cycle Algorithm

** Text size equals 0.5, 1.0 and 2.0 million character respectively.
Figure 4 shows a percent of improvement in bottom up in the average of total number of comparison.

Table 2. Average of averages of total number of shift, and percent improvement of Top down and Bottom up throughout different text lengths

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Text1</th>
<th>Improvement %</th>
<th>Text2</th>
<th>Improvement %</th>
<th>Text3</th>
<th>Improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMHA</td>
<td>62461</td>
<td>-0.172</td>
<td>130081</td>
<td>-0.142</td>
<td>247852</td>
<td>-0.161</td>
</tr>
<tr>
<td>BBMHA</td>
<td>62569</td>
<td></td>
<td>130266</td>
<td></td>
<td>248252</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>62461</td>
<td>-0.172</td>
<td>130082</td>
<td>-0.142</td>
<td>247852</td>
<td>-0.161</td>
</tr>
<tr>
<td>BRA</td>
<td>62569</td>
<td></td>
<td>130266</td>
<td></td>
<td>248252</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>56910</td>
<td>1.659</td>
<td>118052</td>
<td>2.257</td>
<td>225445</td>
<td>2.00</td>
</tr>
<tr>
<td>BCA</td>
<td>55966</td>
<td></td>
<td>115388</td>
<td></td>
<td>220937</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 shows a percent of improvement in bottom up in the average of averages of total number of shift.
Backward Exact String Searching Strategy

Table 3. Average of averages of total time (second) and percent improvement of Top down and Bottom up through out different text lengths.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Text1</th>
<th>Improvement</th>
<th>Text2</th>
<th>Improvement</th>
<th>Text3</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
</tr>
<tr>
<td>BMHA</td>
<td>9.1913</td>
<td>1.488</td>
<td>21.013</td>
<td>4.175</td>
<td>69.09</td>
<td>2.828</td>
</tr>
<tr>
<td>BBMHS</td>
<td>9.0546</td>
<td>3.939</td>
<td>20.136</td>
<td>67.136</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>8.8535</td>
<td>17.891</td>
<td>1.859</td>
<td>68.043</td>
<td>2.928</td>
<td></td>
</tr>
<tr>
<td>BRA</td>
<td>8.5047</td>
<td>17.559</td>
<td>66.051</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>7.2767</td>
<td>5.618</td>
<td>15.556</td>
<td>5.013</td>
<td>59.738</td>
<td>5.618</td>
</tr>
<tr>
<td>BCA</td>
<td>6.8679</td>
<td>14.776</td>
<td>56.382</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 shows a percent of improvement in bottom up in the average of averages of total time (second).

5. Conclusion and Further Work

Searching Algorithms of a new backward strategy developed based on the three exact string searching algorithms BMHA, RA, and CA. The new developed algorithms process the text in the converse of its original algorithms, while the pre-
processing of the pattern use the same heuristic tables used in the original algorithms, with size doesn’t exceed the alphabet size.

Many experiments were designed to compare these algorithms with their original searching algorithms according to the three main factors: numbers of comparisons, number of shifts, and total time of execution. As a result of changing the direction of the text scan, we can conclude: First, the new developed algorithms give a better performance than the original algorithm. The running time has reduced in all adapted algorithms to provide more improvement in the range of 1.49-5.6%.

Second, the backward cycle algorithm achieves the best performance time among all other algorithms in case of large text size (up to 5.618). Third, the running time is more accurately governed by the number of comparisons rather than the number of shifts as in table1 and table3.

This could be justified due to two reasons. One of them, by changing the direction of the text scan not the same character was compared through scanning the text from the end of the text. In addition, this will result in supplying with a different amount of shifts. In other words it is a language dependency factor. The other reason is the changing of the direction in the pre-process step which attributed significantly in the search process; see Example2.

Finally, it is worth mentioning that searching in this new strategy is useful to find a specific pattern wherever is the position you start the search process in a given text. It searches from the starting position in the given text to the end using forward scan, while backward scan to search from the starting position of the search to the beginning of the text, it is more important to give a large text size for the backward scan, in order to reduce the cost of the searching. This is because backward search is more efficient than forward search as shown in the tables [8,12,20]. Future research could be directed towards the parallelizations of the three algorithms, or to investigates the behavior of these algorithms under a different language, such as Arabic.

طريقة البحث في النص في اتجاه عكسي

إباء القيومي وأحمد الجابر

ملخص

تطرق هذا البحث إلى استخدام أسلوب عكسي في عملية البحث في النص لتحديد المواقع التي يتواجد بها نص جزئي. وكذلك إعادة بناء الخوارزميات المعروفة بحيث تستخدم أسلوب عكسي في مسار الخوارزميات الجديدة مقارنة مع الخوارزميات المعروفة في هذا المجال. وتتبين من خلال التجارب المعمولة على نصوص مختلفة الأطوال (نصف مليون حرف إلى مليون حرف) حيث أخذت من مواقع مختلفة على الشبكة بأن هذا الأسلوب أعطي فعالية أكثر من الخوارزميات الأصلية بحيث أنه يقلل من الوقت التنفيذي للخوارزمية بمقدار يصل إلى 5.6% من وقت الخوارزمية الأصلية في بعض الحالات.
Backward Exact String Searching Strategy

References


