An Intelligent and Efficient Matching Algorithm to Finding a DNA Pattern

Mahmoud Moh'd Mhashi
Computers and Information Technology
Tabuk University
Email: mmhashi@ut.edu.sa

Abstract
The objective of string matching is to search and find all the occurrences of the text pattern $\text{Pat}$ with size $m$ characters in the text $\text{Text}$ with size $n$ characters, where $n >> m$. The string matching algorithm is so fundamental that most computer programs use it, and it continues to receive great attention due to various applications in text manipulation and other applications, especially in biological sequence analysis and comparison. The main goal of this research is to develop a new enhanced skipping algorithm with intelligent issues that outperforms the algorithms introduced in the literature. The string matching algorithms consist of two main operations: checking and skipping. Traditionally, authors try to reduce the number of comparisons through the checking operation. The new algorithm in this paper decreases the number of comparisons by decreasing the number of checking operations through an increase in the shift distance. This enhances the searching response time. The proposed algorithm is an efficient algorithm that can be used to search for exact occurrences of long patterns in DNA patterns.

1. Introduction

Exact string searching is one of the most important problems that has been investigated; studies range from finding the shortest string in DNA sequencing [1-4] to searching for occurrences of a pattern. In general, string searching algorithms deal with searching the occurrences of a string (the pattern) of size $m$ in a string (the text) of size $n$ (where $n \geq m$) [5-7]. In the literature, many exact string searching and pattern matching algorithms were introduced and their performances were investigated against classical exact string searching algorithms such as the Naïve (brute force) algorithm and Boyer-Moore-Horpeool(BMH) algorithms. Some of these algorithms preprocess both the text and the pattern [8] while others preprocess the pattern [9-12].

The exact string searching problem consists of two major steps: checking and skipping, the checking step itself consists of two phases:

1) A search along the text for a reasonable candidate string
2) A detailed comparison of the candidate against the pattern to verify the potential match.

Some characters of the candidate string must be selected carefully in order to avoid the problem of repeated examination of each character of text when patterns are partially matched; the fewer the number of character comparisons in the checking step the better the algorithm is. There are different algorithms that can be used to check if the characters in the text match with the corresponding characters in the pattern [13-21]. After the checking step, the skipping step shifts the pattern to the right to determine the next position in the text where the substring text can possibly match with the pattern.
This research will focus on searching for exact occurrences of long patterns in the DNA alphabet. Searching for DNA sequence is studied by many authors in the literature [22-26]. Molecular biologists often search for the important information from the databases in different directions of different uses [27]. A DNA sequence can be abstracted as a string over an alphabet of 4 basic characters, (A, C, G, and T). DNA sequences can be very long. For example, the total size of a human genome (DNA) is around 3Gbp. Researchers tend to search patterns in DNA data frequently [28]. So, an efficient searching tool which is enough to allow users to locate patterns in large DNA data is desired. Developing a fast and efficient algorithm to find all occurrences of DNA patterns is important to the different applications using exact matching. The main goal is to develop a new enhanced checking and skipping algorithm where intelligent issues outperform the algorithms that are introduced in the literature.

2. The Naïve or (Brute Force) Algorithm

Let us assume that the target sequence is an array Text[n] of n characters and the pattern sequence is the array Pat[m] of m characters, then a Naïve approach to the problem would be:

```c
void Naïve (char *Pat, int PatLength, char *Text, int TextLength)
{
    for (int TextIx = 0; TextIx <= TextLength - PatLength; TextIx++)
    {
        int PatIx = 0;
        while (Text[TextIx + PatIx] == Pat[PatIx])
        {
            if (PatIx == PatLength -1)
            {
                cout << "\n Occurrence at " << TextIx << " to " << TextIx + PatIx;
                break;
            }
            else PatIx++;
        }
    }
    return;
}
```

In the outer loop, Text is searched for occurrences of the first character in Pat. In the inner loop, a detailed comparison of the candidate string is made against Pat to verify the potential match. The algorithm has a worst case time of \(O(nm)\), where \(O(nm)\) is the number of comparisons performed by the algorithm to find all the occurrences of Pat with size m characters in the Text with size n characters. The worst case in this algorithm
occurs when we get a match on each of the \( n \) Text characters and at each position we may need to perform \( m \) comparisons.

3. Multiple Reference Character Algorithms (MRCA4m)

A string matching algorithm is a succession of checking and skipping, where the aim of a good algorithm is to minimize the work done during each checking and to maximize the length distance during the skipping. An algorithm can skip to the next position in the text without missing any pattern occurrence. Most of the string matching algorithms preprocess the pattern before the search phase to help the algorithm to maximize the length of the skips. The preprocessing phase in this new MRCA4m algorithm helps in increasing the performance of maximizing the length of the skips.

In order to test the role of skipping distance on the number of checking operations, the MRCA algorithm in [29] is enhanced. MRCA algorithm has five reference characters, including three static and two dynamic references. Two new versions of MRCA algorithms are developed. The first version of MRCA (let us call it MRCA2m) has three reference characters, including two static characters and one dynamic character. The second version of MRCA (let us call it MRCA4m) has seven reference characters, including four static reference characters and three dynamic characters. So, the number of reference characters in MRCA2m, MRCA, and MRCA4m is three, five, and seven reference characters, respectively. The shift distance ranges from \( 2m+1 \) characters in MRCA2m, \( 3m+1 \) characters in MRCA, and \( 4m+1 \) characters in MRCA4m. Of course, increasing the shift distance in the skip operation decreases the number of checking operations. This is, in turn, reduces the number of comparisons required to find all the occurrences of patterns in the text and decrease the elapsed time. A case study will be conducted to test the above issues and will be explained in the next section. A brief description of MRCA follows.

As mentioned earlier, CSA has five reference characters. The Text pointer \( TextIx \) always points to the character, which is next to the character corresponding to the last character in \( Pat \) and the reference character \( ref \) always points to the character that corresponds to the last character in \( Pat \) (i.e. \( ref = TextIx - 1 \)). Now let \( ref1 = TextIx \), then the reference character \( ref2 \) can be calculated from \( ref \) or \( ref1 \), where \( ref2 \) can be found as “\( ref2 = TextIx + m - 1 \)” or “\( ref2 = TextIx + m \)” depending on the existence of \( ref \) or \( ref1 \) in \( Pat \), where \( ref2 = TextIx + m - 1 \) during the checking step if the character at \( ref \) doesn’t exist in \( Pat \), or \( ref2 = TextIx + m \) after the checking step if the character at \( ref1 \) doesn’t exist in \( Pat \). The above formulas may be illustrated in the following example:

<table>
<thead>
<tr>
<th>Text:</th>
<th>K</th>
<th>K</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pat:</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the above, Text and Pat assume that the pointers \( TextIx \) and \( ref1 \) point to the character ‘D’, the pointer \( ref \) points to the character ‘C’, and \( m = 3 \), in this case since the character ‘C’ at \( ref \) doesn’t exist in \( Pat \), then \( ref2 = TextIx + m - 1 = 3 + 3 - 1 = 5 \). So, start counting from zero, \( ref2 \) points to the character ‘F’ in Text. The occurrence of character ‘D’ at \( ref1 \) must be examined whether \( Pat \) occurs in Text or not, and since ‘D’ doesn’t occur in \( Pat \) then \( ref2 = TextIx + m = 3 + 3 = 6 \), hence, \( ref2 \) points to the character ‘G’ in Text.
In addition to that, MRCA pre-processes the pattern to produce two different arrays, namely skip and pos, each array has a length equals to the alphabet size. The skip array is used when the reference character ref1 exists in Pat, it expresses how much the pattern is to be shifted forward after the checking step. While the pos array defines where each one of the different reference characters ref1, ref2, ref_ref1, or ref_ref2 is located in Pat, if any one of them exists in Pat, where the two dynamic pointers ref_ref1 and ref_ref2 can be calculated from the two static pointers ref1 and ref2 respectively. In particular, the dynamic pointer ref_ref1 which is calculated at the skipping step if ref1 occurs in Pat can be found as

$$\text{ref_ref1} = \text{ref1} + m - \text{pt}$$

where m is the Pat length and pt is the location of ref1 in Pat (i.e., pt = pos[Text[ref1]]), and the dynamic pointer ref_ref2 which is calculated and used only during checking step if ref doesn’t occur in Pat or after the checking step if ref1 doesn’t occur in Pat can be found as

$$\text{ref_ref2} = \text{ref2} + m - \text{pt1}$$

where pt1 determines where ref2 is located in Pat. The above formulas can be illustrated in the following examples:

**Example 2: Calculating ref_ref1**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text:</td>
<td>K</td>
<td>K</td>
<td>G</td>
<td>E</td>
<td>K</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>Pat:</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming the checking step is performed on the above Text and Pat, then the reference character will be character ‘E’ at location 3 in Text, and since ‘E’ at ref1 occurs at location 1 in Pat, then pt = pos[Text[ref1]] = pos[‘E’] = 1 (note that for the case of counting position the counting starts from 1 not zero), and ref_ref1 can be found as

$$\text{ref_ref1} = \text{ref1} + m - \text{pt} = 3 + 3 - 1 = 5$$

(i.e. ref_ref1 points to the character ‘F’ in Text at location 5).

**Example 3: Calculating ref_ref2 when the character at ref doesn’t occur in Pat**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text:</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>Pat:</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since ref doesn’t occur in Pat, then ref2=5 (calculated above in Example 1). Consequently, ref2 + m - pt1, where m = 3 and pt1 = pos[Text[ref2]] = pos[Text[‘F’]] = 1. Hence, ref_ref2 may be found as

$$\text{ref_ref2} = 5 + 3 - 1 = 7$$

In such a case, the alignment will be with ‘H’ at location 7 in Text, however, the letter ‘H” doesn’t occur in Pat, so, TextIx will move forward 7 locations to point to the character ‘G’ at position 10.

**Example 4: Calculating ref_ref2 after the checking step if ref1 doesn’t exist in Pat**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text:</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>E</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Pat:</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let Text and Pat occur as shown above. Since the character at ref occurs in Pat and there is a mismatch then ref1 will be considered as a basis for calculating ref2, and since ref1 doesn’t occur in Pat then ref2 = TextIx + m = 6, and pt1 = pos[Text[ref2]] = pos[Text[‘I’]] = 0. Hence, ref_ref2 may be found as

$$\text{ref_ref2} = 6 + 3 - 0 = 9$$

In addition, the TextIx pointer will be shifted forward 10 positions to align with the letter
Text[ref_ref2] = Text[‘L’] = 3m+1 positions, which is the maximum shift distance that this algorithm can skip with only two-character-access. As a result, the pointer TextIx will point to the letter ‘M’ at position Text[13], and the result will be as follows:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text:</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>M</td>
</tr>
<tr>
<td>Pat:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>D</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

Based on that, The MRCA algorithm that reflects the above issues is as follows:

1) void PreProcessPat(char *Pat, int PatLength, int *pos, int *skip)

   {
   
   2) char c;

   3) /* Fill tables with initial values */

   4) for(int j = 0; j<ASIZE; j++) {

   5)   pos[j]=0; skip[j] = 2*PatLength;

   }

   6) /* Compute shift distance and position of characters in Pat*/

   7) for( j=0; j<PatLength; j++) {

   8)   c = Pat[j]; pos[c]= j +1; skip[c] = 2 * PatLength - j -1;

   }

   }

9) void ECSA(char *Pat, int PatLength, char *Text, int TextLength, int *pos, int *skip)

   {

10) int TextIx, PatIx, last_mismatch, z;

11) int pt, pt1, ref, ref1, ref_ref1, ref2, ref_ref2;

12) int infix[ASIZE] = {0};

13) /* Update infix table according to the first character in Pat */

14) infix[Pat[0]] = 1; last_mismatch =0; TextIx = PatLength;

15) // Start Searching operation

16) while(TextIx<=TextLength+1)

   { // Checking step: Check first the occurrence of the character at the previous mismatch.

17)   if(Text[TextIx - PatLength + last_mismatch] == Pat[last_mismatch])
{ 
18)   // Check now the character in Text that corresponds the first character in Pat;
19)   if(infix[Text[TextIx - PatLength]])
20)   {  // Check the occurrence of Pat in Text from right to left excluding first character
21)     for( z = 0, PatIx = PatLength - 1; PatIx; PatIx-- )
22)       if(Text[TextIx - ++z] != Pat[PatIx])
23)         last_mismatch = PatIx;
24)       goto next;
25)   }
26)   cout<<"\nAn occurrence at location ",TextIx-PatLength," to ",TextIx 1<<endl;
   }
}

26)   // Start the skipping part
27) next:
28)   ref1 = TextIx;       pt = pos[Text[ref1]];
29)   if ( !pt )
30) {
31)     ref2 = ref1 + PatLength; pt1 = pos[Text[ref2]];
32)     if ( !pt1 ) {
33)       ref_ref2 = ref2 + PatLength - pt1;
34)       TextIx += 3 * PatLength - pt1 + 1 - pos[Text[ref_ref2]];
35)     } else {
36)       TextIx += 2 * PatLength - pt1 + 1;
37)     } else {
38)       ref_ref1 = ref1 + PatLength - pt;
39)       TextIx += skip[Text[ref_ref1]] - pt + 1;
The first new version of MRCA is MRCA2m. As mentioned earlier, MRCA2m has skipping distance $2m+1$. The algorithm MRCA2m is as the same as the algorithm MRCA except the segment of code starting at line 26. The new version from line 26 follows:

```c
26)   // Start the skipping part
27)   next:
28)   ref1 = TextIx;
29)   if ( !pos[Text[ref1]])
    {
30)     ref2 = ref + PatLength;
31)     pt1 = pos[Text[ref2]];
32)     TextIx += 2 * PatLength - pt1;
    }
33)   else
    {
34)     ref_ref1 = ref1 + PatLength - pt;
35)     TextIx += skip[Text[ref_ref1]] - pt + 1;
    }
36)   return;
}
```

The second new version of MRCA is MRCA4m. As mentioned earlier, MRCA4m has skipping distance ranges from one character to $4m+1$ characters. The segment of code of MRCA4m starting from line 26 in MRCA follows:

```c
26)   // Start the skipping part
27)   next:
28)   ref1 = TextIx;
29)   pt = pos[Text[ref1]];
30)   if ( !pt)
    {
```
4. Illustration and Discussion

The three algorithms MRCA2m, ECSA, and MRCA4m were implemented and compared on DNA sequence of 4 basic characters, (A, C, G, T), with a size of more than 1.8 mega characters. The algorithms executed using Intel(R) Pentium(R) 4 PC with CPU speed 2.40GHz, 246MB RAM, and Windows XP professional operating system, and a program was designed in C++ to select randomly 6 groups. Each group of patterns consists of 3000 patterns. The number of characters in patterns ranges from 6 to 50 characters. The average number of occurrences ranges from 0 to 2346. The cost of the searching process to find all the occurrences of the different patterns in each group in Text is measured by finding 1) the search clock time, where the total clock time includes the preprocessing clock time of patterns, and 2) the average number of checking operations (i.e., the first main operation in string searching algorithms) required to find each group of patterns. The second
measurement is important to give us an indication about the percentage of reduction in the number of comparisons. Table (1) shows the clock time required to find each group of patterns.

Table 1: The clock time (seconds) required by MRCA2m, MRCA, and MRCA4m to find all the occurrences of each group of patterns

<table>
<thead>
<tr>
<th>Group #</th>
<th>3000 Patterns with Length In character</th>
<th>Number of Occurrences</th>
<th>Clock time in seconds</th>
<th>Improvements of latest version MRCA4m vs. MRCA2</th>
<th>ECSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Naive 2m 2m+1</td>
<td>MRCA2m 2m+1</td>
<td>ECSA 3m+1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>0</td>
<td>22.1 7.75 6.61 5.39</td>
<td>15%</td>
<td>31%</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
<td>22.4 8.44 7.25 6.19</td>
<td>14%</td>
<td>27%</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2346</td>
<td>22.5 9.94 8.72 7.84</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0</td>
<td>22.3 2.78 2.13 1.44</td>
<td>24%</td>
<td>48%</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>0</td>
<td>22.1 3.06 2.88 2.56</td>
<td>6%</td>
<td>16%</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>928</td>
<td>24.3 15.33 13.47 13.19</td>
<td>12%</td>
<td>14%</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>0</td>
<td>22.3 1.52 1.03 0.81</td>
<td>32%</td>
<td>47%</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>0</td>
<td>22.4 2.31 1.89 1.70</td>
<td>18%</td>
<td>26%</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>127</td>
<td>31.8 13.52 12.50 12.20</td>
<td>8%</td>
<td>10%</td>
</tr>
</tbody>
</table>

One can notice from Table (1) that the number of occurrences is zero. This is because the performance of these versions of the algorithm ECSA needs to be tested. The characters in patterns in groups 1, 4, and 7 are completely different from the characters in Text. About one third of the characters in the patterns in groups 2, 5, and 8 are exists in Text. Finally, the characters in groups 3, 6, and 9 are 100% exist in Text. Each group consists of 3000 patterns. The time required to find all the occurrences of each group ranges from 1.515 seconds (group 7) to 15.38 seconds (group 6), 1.031 seconds to 13.468 seconds, and 0.812 seconds to 13.187 seconds by the algorithms MRCA2m, ECSA, and MRCA4m, respectively. Furthermore, one can see from Table (1) that the algorithm MRCA4m (4m+1) reduces the clock time required by MRCA2m and ECSA algorithms by 6.11% to 31.95% and 9.71 (group 9) to 48.33% (group 4). In order to show the significant of the time required by these versions of MRCA, the naive algorithm is included in Table (1). The time required to find all the occurrences of each group of patterns by the naïve algorithm ranges from 22.1 seconds to 31.8 seconds. Figure (1) show the time required to find each group of patterns by the different algorithms.
Fig. 1: The clock time required by MRCA2m, ECSA, and MRCA4m algorithms to find all the occurrences of patterns in each group.

Table 2: The number of checking operation required by MRCA2m, ECSA, and MRCA4m to find all the occurrences of each group of patterns

<table>
<thead>
<tr>
<th>Group #</th>
<th>Patterns with Length In character</th>
<th>Number of Occurrences</th>
<th>Number of Checking operations</th>
<th>Improvements of latest version MRCA4m vs. MRCA2m</th>
<th>ECSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MRCA2m 2m+1</td>
<td>ECSA 3m+1</td>
<td>MRCA4m 4m+1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>0</td>
<td>138708</td>
<td>86283</td>
<td>65577</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
<td>152076</td>
<td>99498</td>
<td>78764</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6517</td>
<td>163794</td>
<td>118503</td>
<td>100550</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0</td>
<td>32028</td>
<td>20755</td>
<td>15493</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>0</td>
<td>40160</td>
<td>34588</td>
<td>30629</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>4354</td>
<td>224542</td>
<td>210939</td>
<td>207656</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>0</td>
<td>16367</td>
<td>10673</td>
<td>7936</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>0</td>
<td>23683</td>
<td>22343</td>
<td>21938</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>3126</td>
<td>214086</td>
<td>201129</td>
<td>193386</td>
</tr>
</tbody>
</table>

Table (2) presents the three algorithms with the number of checking operations required to find all the occurrences of Pat in Text for all the different groups. The information of number of groups, pattern length, and
the number of occurrences in this table are the same as in Table (1). The number of checking operations ranges from 16367 (group 7) to 224542 (group 6), 10673 to 210939, and 7936 to 207656 by the algorithms MRCA2m, ECSA, and MRCA4m, respectively. Furthermore, from Table (2) the algorithm MRCA4m (4m+1) reduces the percentage of number of checking operations required by MRCA2m and ECSA algorithms by the ranges from 7% (group 8) to 53% (group 1) and from 2% (group 8) to 26% (group 7).

From the analysis of the discussed algorithms, it is clear that the new enhanced algorithm MRCA4m outperforms the other algorithms, especially when the Pattern does not occur in Text (53%). This is due to the long shift distance (4m+1). This decreases the number of checking operation, and in turn reduces the number of basic comparison operations. However, when the characters in Pat exist in Text, the performance of the new algorithm is not high as that in the previous case, especially when the pattern is long a little bit (group 6 and group 9). Most of the words, sentences, and patterns have, prefix, suffix, infix, or may be a combination of two of them. In such a case a lot of not necessary comparisons performed till the algorithm decides that there is no mismatch. This decreases the performance of the new algorithm.

6. Conclusions

A new exact string searching algorithm MRCA4m was developed, and its performance was compared with two algorithms, namely, MRCA2m, and MRCA. The new algorithm increases the performance of the skipping phases needed for the string searching process. In the skipping phase MRCA4m focuses on increasing the shift distance. The search clock time and counting the number of checking phase operation criteria were used in a case study to compare the performance of MRCA4m against MRCA2m and MRCA. The results showed that:

1) Using MRCA4m improved the clock time required by MRCA2m and MRCA by 6% to 32% and 10% to 48% respectively.

2) Using MRCA4m improved the number of checking operation required by MRCA2m and MRCA by 7% to 53% and 2% to 26% respectively.

3) Decreasing the pattern length decreases the system performance.

4) The performance of the new algorithm is positively distinguished in case that the percentage of existence of the characters of patterns in Text is low. In such a case, if there is some applications required that situation (i.e., rare existence characters of Pat in Text), it is possible to increase the shift distance more and more, and in turn, increasing the performance.

5) Increasing the shift distance has a major effect on reducing the number of checking operations, and in turn decreases the number of comparisons and increasing the system performance.

References


