The effect of multiple reference characters on detecting matches in string-searching algorithms

Mahmoud Moh’d Mhashi*†

Faculty of Science, Department of Information Technology (IT), Mu’tah University, Mu’tah, Al-Karak, 61710, Jordan

SUMMARY
The effect of multiple reference characters and the condition types on the performance of exact string-searching algorithms is tested. In order to perform such a test a new algorithm called the Multiple Reference Characters Algorithm (MRCA) is developed. An experiment is performed using English text; the results are compared with the known string-matching algorithms called Boyer–Moore–Horspool (BMH) and Straight Forward (Naïve). With the MRCA algorithm, the shift distance is increased up to $3m + 1$ positions in comparison with exactly one position in the Naïve algorithm and up to $m$ positions in BMH. Furthermore, by using the new algorithm MRCA, the results suggest that the evaluation criteria of the average number of comparisons, the average number of shifts, and the clock time required by BMH are improved up to 73.1%, 64.7%, and 49.6%, respectively. The same evaluation criteria required by Naïve are improved by MRCA up to 98.1%, 98%, and 94.7%, respectively. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: string-searching; pattern matching; checking and skipping; condition type; multiple references

1. INTRODUCTION

The string matching problem is one of the most important areas that has been studied in computer science, ranging from the command ‘s’ in emacs [1] to finding the shortest common superstring in DNA sequencing [2]. Furthermore, searching for occurrences of a pattern occurs in text editors in connection with a ‘Find’ command.

The general problem deals with a string (the text) of size $n$ and searches it for all occurrences of another, shorter string (the pattern) of size $m$ (usually $n \geq m$) [3–5], where $n$ is the text length and $m$ is the pattern length. There are extensive algorithms on string searching and pattern matching.
Some of these algorithms preprocess both the text and the pattern [6] while other algorithms need only to preprocess the pattern [7–9].

One of the most important operations in exact string searching algorithms is the skipping step, which shifts the window and the pattern to the right or determines the next position in Text where the substring text can possibly match with Pat when the reference character exists in Pat or not. The reference character is a text character in Text chosen as the basis for the shift according to the shift table. Some of the searching algorithms use only one reference character [10] while others use two reference characters [11]. Some of the references may be static references and others may be dynamic references, as outlined in this paper.

Additionally, a variety of computers revealed wide variations between pattern matching algorithms on different computers. Careful attention also needs to be given when attempting to compare algorithms on the basis of character comparison and memory references [12]. Some algorithms presented three types of character comparisons [13] and this is another major emphasis of this paper.

2. THE NAÏVE OR (BRUTE FORCE) ALGORITHM

Let us say the target sequence is an array Text[n] of n characters (i.e. n is text length) and the pattern sequence is the array Pat[m] of m characters (i.e. m is the pattern length). A naive 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 Occurence 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 $O(nm)$, because in the worst case we may get a match on each of the n Text characters and at each position we may proceed to complete m comparisons.

Assume that we are given the following Text and Pat:

| Text: | A | C | C | D | E | F | C | F | X | G | H | C | F | B | C | F | B |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pat:  | C | F | X |

Searching process: Loop 1:
Comparison starts from left to right →→
(Pat[j] = ‘C’) ≠ (Text[i] = ‘A’). Skipping right one position produces Loop 2:

\[
\begin{array}{cccccccccccccccc}
C & F & X & Pat[j] \\
\neq
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & Text[i]
\end{array}
\]

(Pat[j] = ‘C’) == (Text[i] = ‘C’). (Pat[j + 1] = ‘F’) ≠ (Text[i + 1] = ‘C’). Loop 3 is similar to this loop in that it needs two character comparisons to find the mismatch. Proceeding to Loop 4:

\[
\begin{array}{cccccccccccccccc}
C & F & X & Pat[j] \\
\neq
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & Text[i]
\end{array}
\]

(Pat[j] = ‘C’) ≠ (Text[i] = ‘D’). The next two loops (Loop 5 and Loop 6) are similar to this loop in the sense that each needs one character comparison to find the mismatch. Going to Loop 7:

\[
\begin{array}{cccccccccccccccc}
C & F & X & Pat[j] \\
\neq
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & Text[i]
\end{array}
\]

Each character in Pat matches the corresponding character in Text. There is an occurrence at location 6–8. Each one of the next four loops needs one character comparison to find the mismatch. Proceeding to Loop 12:

\[
\begin{array}{cccccccccccccccc}
C & F & X & Pat[j] \\
\neq
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & Text[i]
\end{array}
\]

(Pat[j] = ‘C’) == (Text[i] = ‘C’). (Pat[j + 1] = ‘F’) == (Text[i + 1] = ‘F’). (Pat[j + 2] = ‘X’) ≠ (Text[i + 2] = ‘B’). Each one of the next two loops needs only one character comparison to find the mismatch. Going to Loop 15:

\[
\begin{array}{cccccccccccccccc}
C & F & X & Pat[j] \\
\neq
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & Text[i]
\end{array}
\]

(Pat[j] = ‘C’) == (Text[i] = ‘C’). (Pat[j + 1] = ‘F’) == (Text[i + 1] = ‘F’). (Pat[j + 2] = ‘X’) ≠ (Text[i + 2] = ‘B’). Moving one position ends the searching process. To find all the occurrences of Pat in Text, 23 character comparisons are needed, in addition to 55 number comparisons (i.e. the total number of comparisons is 78).

3. BOYER–MOORE–HORSPOOL ALGORITHM

Horspool is one of many authors who have extended the BM algorithm [1]. Horspool proposed the Boyer–Moore–Horspool (BMH) [14] algorithm, which is regarded as the best general-purpose string-searching algorithm. The algorithm scans the characters of the pattern from right to left beginning with
the rightmost character. In the case of a mismatch (or a complete match of the whole pattern), it uses only a single auxiliary skip table indexed by the mismatching text symbols. If the reference character (the character in Text that corresponds to the last character in Pat) does not occur in Pat it is possible to skip forward by \( m \) positions (the pattern length) and repeat the examination. The following example explains the algorithm. Assume that we are given the following Text and Pat:

Text: 
[\[\begin{array}{cccccccccccc}
A & C & C & D & E & F & C & F & X & G & C & F & B & C & F & B \\
\end{array}\]\]

Pat: 
[\[\begin{array}{cccccccccccc}
C & F & X \\
\end{array}\]\]

The construction of the skip table for the pattern is as follows:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>\ldots</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>\ldots</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Any character that is not in the pattern will produce a shift distance equal to \( m \), where \( m = 3 \). Any character in the pattern \( \text{Pat}[j] \) produces a shift distance equal to \( m - j \), where \( j = 0 \), 1, and 2.

Searching process: Loop 1:

\[
\begin{array}{cccccccccccc}
C & F & X & \text{Pat}[j] \\
\neq \\
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & \text{Text}[i] \\
\end{array}
\]

\( (\text{Pat}[j] = \text{‘X’}) \neq (\text{Text}[i] = \text{‘C’}) \). Looking up in the skip table for character ‘C’ gives value 2. Skipping right two positions produces Loop 2:

\[
\begin{array}{cccccccccccc}
C & F & X & \text{Pat}[j] \\
\neq \\
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & \text{Text}[i] \\
\end{array}
\]

\( (\text{Pat}[j] = \text{‘X’}) \neq (\text{Text}[i] = \text{‘E’}) \). Looking up in the skip table for character ‘E’ gives value 3. Skipping right three positions produces Loop 3:

\[
\begin{array}{cccccccccccc}
C & F & X & \text{Pat}[j] \\
\neq \\
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & \text{Text}[i] \\
\end{array}
\]

\( (\text{Pat}[j] = \text{‘X’}) \neq (\text{Text}[i] = \text{‘F’}) \). Looking up in the skip table for character ‘F’ gives value 1. Skipping right one position produces Loop 4:

\[
\begin{array}{cccccccccccc}
C & F & X & \text{Pat}[j] \\
\neq \\
A & C & C & D & E & F & C & F & X & G & H & C & F & B & C & F & B & \text{Text}[i] \\
\end{array}
\]

Each character in \( \text{Pat} \) matches the corresponding character in Text. There is an occurrence at location 6–8. Looking up in the skip table for character ‘X’ gives value 3. Skipping right three positions produces Loop 5:
(Pat[j] = ‘X’) ≠ (Text[i] = ‘C’). Looking up in the skip table for character ‘C’ gives value 2. Skipping right two positions produces Loop 6:

(Pat[j] = ‘X’) ≠ (Text[i] = ‘B’). Looking up in the skip table for character ‘B’ gives value 3. Skipping right three positions produces Loop 7:

(Pat[j] = ‘X’) ≠ (Text[i] = ‘B’). Looking up in the skip table for character ‘B’ gives value 3. Skipping right three positions ends the searching process. In this example, for a pattern of three characters and a Text of 17 characters, only nine character comparisons have been performed, in addition to 25 number comparisons, to find all the occurrences of Pat in Text (i.e. the total number of comparisons is 34).

```c
void PreProcessBMH(char *Pat, int PatLength, int *skip_table)
{
    for(int i = 0; i < ASIZE; i++)
        skip_table[i] = PatLength;
    for(i = 0; i < PatLength - 1; i++)
        skip_table[Pat[i]] = PatLength - i - 1;
}

void BMH(char *Pat, int PatLength, char *Text, int TextLength, int *skip_table)
{
    char c;
    /* Preprocessing */
    PreProcessBMH(Pat, PatLength, skip_table);
    /* Searching */
    int TextIx = 0;
    while (TextIx <= TextLength - PatLength) {
        c = Text[TextIx + PatLength - 1];
        if (Pat[PatLength - 1] == c && memcmp(Pat, Text + TextIx, PatLength - 1) == 0)
            cout << nOccurrence at location " << TextIx << to location " << TextIx + PatLength - 1;
        TextIx += skip_table[c];
    }
}
```
4. MULTIPLE REFERENCE CHARACTERS ALGORITHM

As any string matching algorithm, the Multiple Reference Characters Algorithm (MRCA) finds all the occurrences of a pattern $Pat_0, \ldots, Pat_{n-1}$ in the text $Text_0, \ldots, Text_{m-1}$, where $m$ is the length of $Pat$ and $n$ is the length of $Text$. MRCA pre-processes the pattern to produce two different arrays $skip$ and $pos$. Each array has a length equal to the alphabet size. The code that is used to initialize the two arrays is presented in the MRCA algorithm in this section.

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. The $pos$ array determines where each one of the different reference characters $ref1, ref2, ref_1ref1, or ref_2ref2$ (they are explained below) is located in $Pat$, if any one of them exists in $Pat$.

MRCA has five reference characters, including three static references and two dynamic references. The $Text$ pointer $TextIx$ always points to the character which is next to the character corresponding to the last character in $Pat$. The reference character $ref$ always points to the character that corresponds to the last character in $Pat$. In other words, $ref = TextIx - 1$. Let $ref1 = TextIx$. The reference character $ref2$ can be calculated from $ref$ or $ref1$. The pointer $ref2$ equals either ‘$TextIx + m - 1$’ or ‘$TextIx + m$’. This value can be determined on the basis of the existence of $ref$ or $ref1$ in $Pat$. The value of $ref2$ is equal to ‘$TextIx + m - 1$’ during the checking step and if the character at $ref$ does not exist in $Pat$. Otherwise, the value of $ref2$ is equal to ‘$TextIx + m$’ after the checking step and if the character at $ref1$ does not exist in $Pat$. The examples below explain the above formulas:

**Example 1**

<table>
<thead>
<tr>
<th>$Text$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</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 $Pat$, the pointers $TextIx$ and $ref1$ point to the character ‘D’, the pointer $ref$ points to the character ‘C’, and $m = 3$. In this example, since the character ‘C’ at $ref$ does not exist in $Pat$, then $ref2 = TextIx + m - 1 = 3 + 3 - 1 = 5$. So, counting from zero, $ref2$ points to the character ‘F’. Assuming that the character ‘C’ in $Text$ is either the character ‘E’, ‘F’, or ‘G’ (i.e. any character exists in $Pat$), then the character at $ref$ occurs in $Pat$. In such a case, the checking step will be continued to find out the occurrence of $Pat$ in $Text$. After the checking step and whether $Pat$ occurs in $Text$ or not, the occurrence of character ‘D’ at $ref1$ will be examined. Since ‘D’ does not occur in $Pat$, then $ref2 = TextIx + m = 3 + 3 = 6$. So, $ref2$ points to the character ‘G’ in $Text$.

The two dynamic pointers $ref_1ref1$ and $ref_2ref2$ can be calculated during the skipping stage from the two static pointers $ref1$ and $ref2$, respectively. The dynamic pointer $ref_1ref1$ can be calculated according to the following formula: $ref_1ref1 = ref1 + m - pt$, where $m$ is the $Pat$ length and $pt$ determines where $ref1$ is located in $Pat$ (i.e. $pt = pos[Text[ref1]$]). The pointer $ref_1ref1$ can be calculated at the skipping step and if $ref1$ occurs in $Pat$. The dynamic pointer $ref_2ref2 = ref2 + m - pt1$, where $pt1$ determines where $ref2$ is located in $Pat$. The pointer $ref_2ref2$ is calculated and used only during the checking step if $ref$ does not occur in $Pat$ or after the checking step if $ref1$ does not occur in $Pat$. The next examples below explain the above formulas.
Example 2

<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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></td>
<td>A</td>
<td>B</td>
<td>G</td>
<td>E</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td><strong>Pat</strong></td>
<td>E</td>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example explains how \( \text{ref}_1 \) can be calculated. Assuming the checking step is performed in the above \( \text{Text} \) and \( \text{Pat} \), the reference character will be the character ‘E’ at location 3. Since ‘E’ at \( \text{ref}_1 \) occurs in \( \text{Pat} \), then \( \text{ref}_1 \) will be calculated. Since the character ‘E’ at \( \text{ref}_1 \) occurs at location 1 in \( \text{Pat} \), then \( \text{pt} = \text{pos}[	ext{Text}[\text{ref}_1]] = \text{pos}['E'] = 1 \) (the counting starts from 1 not 0). So, \( \text{ref}_1 = \text{ref}_1 + m - \text{pt} = 3 + 3 - 1 = 5 \). Therefore, \( \text{ref}_1 \) points to the character ‘F’ in \( \text{Text} \) at location 5. One could say why and how \( \text{ref}_1 \) developed. In the above example, if we were to apply the BMH algorithm, then the letter G at location 2 is the reference character and, according to the skip table, the character ‘G’ gives 3. After the skipping phase, the new picture will be as follows according to BMH:

<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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></td>
<td>A</td>
<td>B</td>
<td>G</td>
<td>E</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td><strong>Pat</strong></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 new algorithm, rather than making alignment with ‘G’ at position 2, one could make alignment with letter ‘F’ at position 5 and this is where \( \text{ref}_1 \) points.

Example 3

<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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></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>
<td>G</td>
</tr>
<tr>
<td><strong>Pat</strong></td>
<td>F</td>
<td>E</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example explains how \( \text{ref}_2 \) is calculated when the character at \( \text{ref} \) (position 2) does not occur in \( \text{Pat} \). Since \( \text{ref} \) does not occur in \( \text{Pat} \), then \( \text{ref}_2 \) will be calculated and it equals 5 (calculated above in Example 1). Consequently, \( \text{ref}_2 \) will be calculated and it equals ‘F’ at location 5 in \( \text{Text} \). The value of \( m \) equals 3 and \( \text{pt}_1 = \text{pos}[	ext{Text}['F']] = 1 \). So, the value of \( \text{ref}_2 = 5 + 3 - 1 = 7 \). In such a case, the alignment will be with ‘H’ at location 7 in \( \text{Text} \). However, the letter ‘H’ does not occur in \( \text{Pat} \). So, \( \text{Text} \) will move forward seven locations to point to the character ‘G’ at position 10. One might ask how and why we aligned with character ‘H’ at location 7.

Applying BMH to the above-mentioned example, the picture 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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></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>
<td>G</td>
</tr>
<tr>
<td><strong>Pat</strong></td>
<td>F</td>
<td>E</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One might use ‘F’ at location 5 for alignment and this is \( \text{ref}_2 \) in the new algorithm. Assuming that the alignment is made with ‘F’ at location 5, then the picture will be as follows:
So, one might use ‘H’ at location 7 in Text for alignment, and this is \( \text{ref}_2 \) in the new algorithm. Now aligning with ‘H’ at location 7, we get

<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>
</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>
<td>G</td>
</tr>
<tr>
<td>Pat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>E</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 4

This example illustrates how \( \text{ref}_2 \) can be calculated after the checking step and if \( \text{ref}_1 \) does not exist in \( \text{Pat} \). Let Text and \( \text{Pat} \) 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>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>
<td></td>
</tr>
</tbody>
</table>

Since the character at \( \text{ref} \) occurs in \( \text{Pat} \) and there is a mismatch, then \( \text{ref}_1 \) will be considered as a basis for calculating \( \text{ref}_2 \). Since \( \text{ref}_1 \) does not occur in \( \text{Pat} \), then \( \text{ref}_2 = \text{Text}_I + m = 6 \). The value of \( pt_1 \) equals \( \text{pos}[\text{Text}[\text{ref}_2]] = \text{pos}[\text{Text}[I]] = 0 \). So, the value of \( \text{ref}_2 \) equals \( \text{ref}_2 + m - pt_1 = 6 + 3 - 0 = 9 \). The TextIx pointer will be shifted forward 10 positions to align with the letter \( \text{Text}[\text{ref}_2] = \text{Text}[L] \) that equals \( 3m + 1 \) 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]. 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>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>
</tr>
</tbody>
</table>

How do we get to the above result? First of all, looking at the next table, the occurrence of \( \text{Pat} \) is checked and there is a mismatch. In such a case and by using the Naïve algorithm, we have to move forward one position. This leads to \( \text{ref}_1 \). Applying BMH on \( \text{ref}_1 \), then character ‘C’ in \( \text{Pat} \) will be moved to location 6. This leads to \( \text{ref}_2 \). Assume that we would like to align with character ‘I’ at location 6, then the character ‘C’ will be moved to location 9. This leads to \( \text{ref}_2 \). Now aligning with character ‘L’ at location 9 leads to moving TextIx to location 13 in Text, which is the same result as in the previous example.
So, the algorithm works as follows:
The existence of the character at ref in Pat will be checked first. There are two cases.

(1) The character at ref exists in Pat. In such a case, the existence of Pat in Text will be checked. After the checking step, the existence of ref1 in Pat will be examined. There are two cases.

(1.1) The character at ref1 does not exist in Pat. Then ref2 and ref reflux will be calculated. Next, the pointer TextIx will be moved forward to align with the character at ref reflux.

(1.2) The character at ref1 exists in Pat. Then ref reflux will be calculated. Then, the pointer TextIx will be moved forward to align with the character at ref reflux.

(2) The character at ref does not exist in Pat. Then ref2 and ref reflux will be calculated according to the pointer ref. Next, the pointer TextIx will be moved forward to align with the character at ref reflux.

The MRCA algorithm that reflects the above ideas is as follows:

```c
void PreProcessPat(char *Pat, int PatLength, int *pos, int *skip)
{
    char c;
    /* Fill tables with initial values */
    for(int j = 0; j < ASIZE; j++) {
        pos[j] = 0; skip[j] = 2*PatLength;
    }
    /* Compute shift distance and position of characters in Pat*/
    for(j = 0; j < PatLength; j++) {
        c = Pat[j];
        pos[c] = j + 1; skip[c] = 2 * PatLength - j - 1;
    }
}

void MRCA(char *Pat, int PatLength, char *Text, int TextLength, int *pos, int *skip)
{
    int TextIx, PatIx, k, pt, pt1, ref, ref1, ref reflux, ref reflux2;
    /* Pattern Preprocessing Phase*/
    PreProcessPat (Pat, PatLength, pos, skip);
    /* Searching Process */
```
TextIx = PatLength;
while(TextIx <= TextLength + 1) {
    ref = TextIx - 1;
    ref1 = TextIx;
    if (!pos[Text[ref]]) {
        ref2 = ref + PatLength;
        pt1 = pos[Text[ref2]];
        ref_ref2 = ref2 + PatLength - pt1;
        TextIx += 3 * PatLength - pt1 - pos[Text[ref_ref2]];
    }
    else {
        if (Text[TextIx - PatLength] == Pat[0]) {
            if (Text[ref] == Pat[PatLength - 1]) {
                for (PatIx = 1, k = TextIx - PatLength + 1; PatIx < PatLength - 1; k++, PatIx++) {
                    if (Text[k] != Pat[PatIx]) break;
                }
                if (PatIx == PatLength - 1) cout << "\n Occurrence at" << TextIx - PatLength << " to " << TextIx - 1 << endl;
            }
        }
        pt = pos[Text[ref1]];
        if (!pt) {
            ref2 = TextIx + PatLength;
            pt1 = pos[Text[ref2]];
            ref_ref2 = ref2 + PatLength - pt1;
            TextIx += 3 * PatLength + 1 - pt1 - pos[Text[ref_ref2]];
        } else {
            ref_ref1 = ref1 + PatLength - pt;
            TextIx += skip[Text[ref_ref1]] - pt + 1;
        }
    }
} // This is the end of else of main if statement
} // End while
} // End main function

The following example explains the algorithm overall. The Text and Pat that are used in the previous algorithms are also used here.

| Text: | A | C | C | D | E | F | C | F | X | G | H | C | F | B | C | F | B |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pat:  | C | F | X |

The construction of the skip table for the pattern follows:
Any character that is not in the pattern will produce a shift distance equal to $2m$, where $m = 3$. Any character in the pattern $Pat[j]$ produces a shift distance equal to $2 \times m - j - 1$, where $j = 0, 1,$ and 2.

The construction of the pos table for the pattern follows:

The number zero indicates that the character does not exist in $Pat$; otherwise the number indicates where the character is located in $Pat$.

Searching process: Loop 1:

$\leftarrow$—Comparison starts from first, last, and the rest from right to left.

Each character in $Pat$ matches the corresponding character in $Text$. There is an occurrence at location 6–8. The reference $ref 1$ points to character ‘G’ at location 9 and does not exist in $Pat$. The reference $ref 2$ must be calculated and it points to character ‘F’ at location 12. It exists in $Pat$ at location 2. The reference $ref \_ref2$ must be calculated and it equals 13, pointing to the character ‘B’. The pointer $TextIx$ must be moved forward eight positions to align with $ref \_ref2$ at location 13. Skipping right eight positions produces Loop 3:
5. EXPERIMENTAL RESULTS AND DISCUSSION

In this experiment, the three algorithms Naïve, BMH, and MRCA were implemented and compared using English text containing more than one mega characters (exactly 1 005 568 characters). The computer was an Intel(R) Pentium(R) 4 CPU 2.40 GHz, 246 MB RAM, running the Windows XP professional operating system.

A program was designed in C++ to randomly select 3000 patterns divided into 10 groups. Each group consists of 300 patterns. Table I presents the necessary information about the different groups. The pattern length ranges from 4 to 94 characters. The average number of occurrences ranges from 1 to 579. 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 average number of shifts;
2. the number of comparisons, including character comparison, number comparison, and character-access comparison;
3. the search clock time; this is the total clock time including the preprocessing clock time of patterns, in addition to the searching clock time.

Table I presents the results of the experiment. The average number of shifts is presented in Table I(A). The average number of shifts ranges from 1 005 475 to 1 005 565, 43 892 to 279 122, and 19 532 to 98 485 by using Naïve, BMH, and MRCA, respectively. Note that by using MRCA, the average number of shifts required by Naïve and BMH are reduced by between 90% (group 1) and 98% (group 8 to group 10) and between 55.1% (group 10) and 64.7% (group 1), respectively.

The average total number of comparisons is recorded in Table I(E). This average is distributed among the three types of conditions including character comparison (Table I(B)), number comparison (Table I(C)), and character-access comparison (Table I(D)). The last table is for MRCA only. The average total number of comparisons for the three types of conditions ranges from 2 154 989 to 2 178 989, 184 822 to 1 168 241, and 78 744 to 314 544 by using Naïve, BMH, and MRCA, respectively. By using MRCA, the average total number of comparisons required by Naïve and BMH is reduced by 85.5–96.4% and 56.5–73.1%, respectively.

Table I. A comparison is presented between Naïve, BMH and MRCA in terms of (A) average number of shifts, (B) average number of character comparisons, (C) average number of number comparisons, (D) average number of character-access comparisons, (E) average number of total comparisons for the three types, (F) clock time required to find each group, including the percentage of improvements.

<table>
<thead>
<tr>
<th>Group no.</th>
<th>Pattern length in characters</th>
<th>Average number of occurrences</th>
<th>Improvements in MRCA versus</th>
<th>Improvements in MRCA versus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Using Naïve</td>
<td>Using BMH</td>
<td>Using MRCA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Using Naïve (%)</td>
<td>Using BMH (%)</td>
<td>Using MRCA (%)</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1005 565</td>
<td>279 122</td>
<td>98 485</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>1005 555</td>
<td>107 091</td>
<td>39 792</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>1005 545</td>
<td>77 933</td>
<td>30 611</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>1005 535</td>
<td>65 564</td>
<td>26 651</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>1005 525</td>
<td>57 367</td>
<td>23 912</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>1005 515</td>
<td>53 381</td>
<td>22 513</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>1005 505</td>
<td>47 691</td>
<td>20 492</td>
</tr>
<tr>
<td>8</td>
<td>74</td>
<td>1005 495</td>
<td>44 966</td>
<td>19 532</td>
</tr>
<tr>
<td>9</td>
<td>84</td>
<td>1005 485</td>
<td>45 165</td>
<td>19 815</td>
</tr>
<tr>
<td>10</td>
<td>94</td>
<td>1005 475</td>
<td>43 892</td>
<td>19 717</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group no.</th>
<th>Using Naïve</th>
<th>Using BMH</th>
<th>Using MRCA</th>
<th>Improvements in MRCA versus</th>
<th>Improvements in MRCA versus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Naïve (%)</td>
<td>BMH (%)</td>
<td>MRCA (%)</td>
<td>Naïve (%)</td>
<td>BMH (%)</td>
</tr>
<tr>
<td>1</td>
<td>1080 862</td>
<td>863 533</td>
<td>107 565</td>
<td>90.1 (%)</td>
<td>87.5 (%)</td>
</tr>
<tr>
<td>2</td>
<td>1082 422</td>
<td>331 306</td>
<td>42 107</td>
<td>96.1 (%)</td>
<td>87.3 (%)</td>
</tr>
<tr>
<td>3</td>
<td>1087 208</td>
<td>242 128</td>
<td>32 444</td>
<td>97 (%)</td>
<td>86.6 (%)</td>
</tr>
<tr>
<td>4</td>
<td>1088 778</td>
<td>203 891</td>
<td>28 281</td>
<td>97.4 (%)</td>
<td>86.1 (%)</td>
</tr>
<tr>
<td>5</td>
<td>1089 495</td>
<td>177 860</td>
<td>25 159</td>
<td>97.7 (%)</td>
<td>85.9 (%)</td>
</tr>
<tr>
<td>6</td>
<td>1085 608</td>
<td>165 394</td>
<td>23 578</td>
<td>97.8 (%)</td>
<td>85.7 (%)</td>
</tr>
<tr>
<td>7</td>
<td>1090 699</td>
<td>148 237</td>
<td>21 627</td>
<td>98 (%)</td>
<td>85.4 (%)</td>
</tr>
<tr>
<td>8</td>
<td>1077 495</td>
<td>139 329</td>
<td>20 426</td>
<td>98.1 (%)</td>
<td>85.3 (%)</td>
</tr>
<tr>
<td>9</td>
<td>1083 460</td>
<td>140 182</td>
<td>20 815</td>
<td>98.1 (%)</td>
<td>85.2 (%)</td>
</tr>
<tr>
<td>10</td>
<td>1081 335</td>
<td>136 304</td>
<td>20 726</td>
<td>98.1 (%)</td>
<td>84.8 (%)</td>
</tr>
</tbody>
</table>

Table I. Continued.

<table>
<thead>
<tr>
<th>Group no.</th>
<th>Using Naïve</th>
<th>Using BMH</th>
<th>Using MRCA</th>
<th>(E) Average number of total comparisons by using the three algorithms</th>
<th>Improvements in MRCA versus Naïve (%)</th>
<th>BMH (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 161 723</td>
<td>1 168 241</td>
<td>314 544</td>
<td>85.5</td>
<td>73.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2 164 843</td>
<td>448 427</td>
<td>146 494</td>
<td>93.2</td>
<td>67.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2 174 415</td>
<td>328 386</td>
<td>118 918</td>
<td>94.5</td>
<td>63.8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2 177 555</td>
<td>276 650</td>
<td>105 857</td>
<td>95.1</td>
<td>61.7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2 178 989</td>
<td>240 983</td>
<td>95 467</td>
<td>95.6</td>
<td>60.4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2 171 215</td>
<td>224 021</td>
<td>90 230</td>
<td>95.8</td>
<td>59.7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2 181 397</td>
<td>201 089</td>
<td>82 840</td>
<td>96.2</td>
<td>58.8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2 154 989</td>
<td>188 722</td>
<td>78 744</td>
<td>96.4</td>
<td>58.3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2 166 919</td>
<td>190 030</td>
<td>80 466</td>
<td>96.3</td>
<td>57.7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2 162 669</td>
<td>184 822</td>
<td>80 385</td>
<td>96.3</td>
<td>56.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group no.</th>
<th>Using Naïve</th>
<th>Using BMH</th>
<th>Using MRCA</th>
<th>(F) Clock time (s) by using the three algorithms</th>
<th>Improvements in MRCA versus Naïve (%)</th>
<th>BMH (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 161 723</td>
<td>1 168 241</td>
<td>314 544</td>
<td>4.093</td>
<td>0.969</td>
<td>76.3</td>
</tr>
<tr>
<td>2</td>
<td>2 164 843</td>
<td>448 427</td>
<td>146 494</td>
<td>4.141</td>
<td>0.453</td>
<td>89.1</td>
</tr>
<tr>
<td>3</td>
<td>2 174 415</td>
<td>328 386</td>
<td>118 918</td>
<td>4.156</td>
<td>0.344</td>
<td>91.7</td>
</tr>
<tr>
<td>4</td>
<td>2 177 555</td>
<td>276 650</td>
<td>105 857</td>
<td>4.141</td>
<td>0.296</td>
<td>92.9</td>
</tr>
<tr>
<td>5</td>
<td>2 178 989</td>
<td>240 983</td>
<td>95 467</td>
<td>4.156</td>
<td>0.250</td>
<td>94.4</td>
</tr>
<tr>
<td>6</td>
<td>2 171 215</td>
<td>224 021</td>
<td>90 230</td>
<td>4.141</td>
<td>0.250</td>
<td>94.4</td>
</tr>
<tr>
<td>7</td>
<td>2 181 397</td>
<td>201 089</td>
<td>82 840</td>
<td>4.156</td>
<td>0.235</td>
<td>94.4</td>
</tr>
<tr>
<td>8</td>
<td>2 154 989</td>
<td>188 722</td>
<td>78 744</td>
<td>4.078</td>
<td>0.219</td>
<td>94.6</td>
</tr>
<tr>
<td>9</td>
<td>2 166 919</td>
<td>190 030</td>
<td>80 466</td>
<td>4.109</td>
<td>0.218</td>
<td>94.7</td>
</tr>
<tr>
<td>10</td>
<td>2 162 669</td>
<td>184 822</td>
<td>80 385</td>
<td>4.094</td>
<td>0.235</td>
<td>94.3</td>
</tr>
</tbody>
</table>

Total time 41.265 5.859 3.469 91.6 40.8
Figure 1. A comparison of Naïve, BMH, and MRCA algorithms. The clock time required to find all the occurrences of patterns in each group is plotted against group number.

From Table I(B), note that MRCA reduces the average number of character comparisons required by Naïve and BMH by 92.5–96.6% and 20.6–73.4%, respectively. This reduction is increased in the average number of number comparison (Table I(C)). MRCA reduces the average number of number comparisons required by Naïve and BMH by 90.1–98.1% and 84.8–87.5%, respectively.

Table I(D) presents the average number of character-access comparisons. This type of condition is used by MRCA only and needs much less time than the other two types of conditions. The average number of character-access comparisons required by MRCA ranges from 21 131 to 125 924.

Table I(F) also presents the clock time required to find each group of patterns. The clock time includes the time required for pre-processing the patterns and finding all the occurrences of Pat in Text. The time required by using Naïve, BMH, and MRCA ranges from 4.078 to 4.156 s, 0.328 to 1.921 s, and 0.218 to 0.969 s, respectively. By using MRCA, the clock time required by the algorithms Naïve and BMH are reduced by 76.3–94.7% and 31.5–49.6%, respectively.

Figure 1 presents the results of the clock time required to find all the occurrences of Pat in Text by using the three algorithms Naïve, BMH, and MRCA.

6. CONCLUSIONS AND FUTURE WORK

A new algorithm (MRCA) has been developed and compared with two algorithms, namely, Naïve and BMH. The two algorithms, BMH and MRCA, have a pre-processing step on the pattern only. An experiment was performed to evaluate the new algorithm MRCA. In addition to the time complexity, there are many different criteria used to compare the different algorithms, including:

1. the number of comparisons for each type of conditions;
2. the number of shifts; and
3. the running time.

On comparing Naïve and MRCA and according to the experiment, we have the following results.

1. The average number of shifts, character comparisons, and number comparisons required by Naïve are improved by MRCA according to the following ranges: 90–98%, 92.5–96.6%, and 90.1–98.1%, respectively.
(2) The average number of total conditions required by Naïve is improved by MRCA in the range from 85.5 to 96.4%.
(3) The clock time required by Naïve is improved by MRCA in the range from 76.3% to 94.7%.

On comparing BMH and MRCA, we have the following improvements.

(1) The average number of shifts, character comparisons, and number comparisons required by BMH are improved by MRCA according to the following ranges: 55.1–64.7%, 20.6–73.4%, and 84.8–87.5%, respectively.
(2) The average number of total conditions required by BMH is improved by MRCA in the range from 56.5 to 73.1%.
(3) The clock time required by BMH is improved by MRCA in the range from 31.5% (group 7) to 49.6% (group 1).

It is clear from these results that the MRCA algorithm gains its performance from more than one direction, including the following.

(1) The shift distance: by using Naïve, the shift distance is only one position. By using BMH, the shift distance ranges from 1 to \( m \) positions. By contrast, by using MRCA, it ranges from 1 to \( (3m + 1) \) positions. Increasing the shift distance has a major effect on reducing the different types of comparisons, especially the number comparison.
(2) The reference character: the Naïve algorithm has no reference character. The BMH has only one reference character. By contrast, MRCA has five reference characters, three of them are static and the other two are dynamic. Increasing the number of reference characters resulted in an increase in the shifted distance.
(3) The MRCA concentrates on using the condition type character-access (which needs 40% less time to be executed than the time needed by any other type of conditions) and reducing the number of conditions of other types.

As a result, during the skipping operation, using the multiple reference character increases the shift distance, converting the number comparisons, indirectly, into character-access type, and reducing the different types of conditions. The following questions then arise: during the checking operation, is it possible to convert and/or reduce the conditions of type character comparison into a condition of type character-access? How does this affect the system performance? Such questions need to be investigated in further studies.

ACKNOWLEDGEMENT

I am indebted to the referees for their useful comments.

REFERENCES