Journal of Computer Science 1 (3): 405-412, 2005
ISSN 1549-3636


# A Fast Approximate String Searching Algorithm

Mahmoud Mhashi, Adnan Rawashdeh and Awni Hammouri
Department of IT, Faculty of Science, Mu'tah University, Mu'tah, Karak, 61710, Jordan

**Abstract:** In both approximate and exact string searching algorithms, the shift distance at the skipping step plays a major role in the performance of string matching algorithms. A new algorithm called the Modified Character-Weight Algorithm (MWA) has been developed to test the effect of the shift distance on the performance of approximate string matching. An experiment was performed comparing the performance of the MWA with that of Mhashi's Character-Weight Algorithm (WA) using English text of size 1,005,077 characters. Using the average number of comparisons and the clock time as evaluation criteria, the MWA algorithm used only about 4% to 15% as many comparisons as the WA algorithm and about 10% to 35% of the clock time.

**Key words:** Approximate String Matching, Searching, Shift Distance, Condition Types, Character Access

## INTRODUCTION

String searching algorithms that are used to retrieve information based on a search criterion form one of the most important topics in computer science. One example of a widely used application for approximate string matching algorithms nowadays is the Internet. People using the Internet usually search for information by typing keywords (string patterns) into a search engine and then waiting for information to be returned. The matching process between the keyword input and the text to be searched can be accomplished in two ways: (I) Exact match [1-4], meaning that the passages returned will contain an exact match of the keyword input. (II) Approximate match [5-7], meaning that the passages returned will contain some part of the keyword input. More information may be returned and users must then filter information accordingly to find what they need. The contribution of this study falls under the category of approximate string matching algorithms.

In the present study, we are interested in designing an algorithm that finds all the occurrences of $Pat = P_1 \ldots P_m$ in $Text = T_1 \ldots T_n$ where $m < n$ with at most $k$ differences of type (a), (b) and (c) below. Different authors [8-10] have studied this problem. Three types of differences are distinguished [11]:

* The $i^{th}$ character in *Text* does not exist in *Pat*.
* The $k^{th}$ character in *Pat* does not exist in *Text*.
* A character of the pattern corresponds to a different character of the text (*i.e.,* a mismatch).

An intensive search for information is taking place on the Internet all around the word and one of the most crucial issues associated with approximate string-searching algorithms is performance. How efficiently information can be retrieved becomes an important issue. Mhashi [12] introduced an improved algorithm, known as the Character-Weight Algorithm (WA). The WA algorithm introduced improvements over a previous counterpart algorithm in terms of both relevance and performance. This study focuses on the performance issue. We have enhanced the performance of an existing algorithm, in terms of the clock time required for the search, the number of character comparisons and the number of positions shifted while moving on with the comparison process. The new developed algorithm will work for any character set.

The organization of the study is as follows: Section 2 describes the earlier algorithm WA and shows how it works through an example. Section 3 describes the new MWA algorithm and illustrates how it works using the same example. Section 4 contains a discussion of the two algorithms and illustrates the performance improvement of MWA over WA. Tables showing the performance improvement achieved in this research support the evaluation process. Finally, Section 5 contains the conclusions of our research work and outlines future work in this area.

## DESCRIPTION OF THE CHARACTER-WEIGHT ALGORITHM (WA)

The WA algorithm concentrates on both the relevance and the performance of retrieving all the occurrences of the pattern in the text. In this section, we address the performance issues (for more details of the full algorithm see [12]. The WA algorithm has three steps:

1. The preprocessing step: In this step, the pattern is preprocessed to determine the *important* and *unimportant* parts. For our experiment, the first two-thirds of the characters are considered to be the *important* part while the rest are considered the *unimportant* part. The result of this step is a table called the weight table. The*weight* table expresses

the importance of the different characters in the pattern *Pat*. In our case, the weight for each character in the *important* part is *k_mismatches* + 1 (i.e., if there is a mismatch at any position in the *important* part, then the algorithm decides that there is no occurrence at that location). The weight for each character in the rest of the pattern (*unimportant*) is one.

2. The checking step: The pattern is checked in the following order:
   * Last and first characters in the *important* part,
   * The rest of the characters in the *important* part from right to left and
   * The rest of the characters in Pat (*unimportant*) from left to right.
3. The skipping step: At this step, the straightforward algorithm is applied and the pattern is shifted forward one position in relation to the text when a match/mismatch is found.

A pseudo code description of the WA algorithm follows:

```
int new_k_mismatches (Text, n, Pat, m, weight, kmis)
char *Text;   // Text: Text[0] … Text[n-1]
int n;      // Text length
char *Pat;     // Pat: Pat[0] … Pat[m-1]
int m;     //  Pat length
int *weight;  // weight: weight [0] … weight[m-1]
int kmis;       // kmis: The maximum number of k mismatches allowed
int mis;  // the total current number of  mismatches
 int i, j, k;
// Preprocess
 (1)  pref_inf = m/3 * 2 -1 ;    suf = m – pref_inf - 1;
// prepare the weight table
for(i=0; i<=pref_inf; i++) weight[i] = kmis + 1;
for(i=pref_inf+1; i<m; i++) weight[i] = 1;
i = m - 1;
// Searching Process.
While (  i <= n-1 + kmis)  { // Outer-loop, as long there is text >=m-kmis, continue searching
   mis = 0; j = 0;
         // test the last and first characters in the important part first
(4)     if ( Text[ i + suf]  != pat[pref_inf]  || text [ i – m + 1  ] != pat [ j ] )
(5)          mis +=  weight[j];
(6)     else  // test remaining important part characters from right to left
       ( for j = pref_inf – 1 , k = i – suf -1 ;  j > 0;  k--,  j-- )
(8)             if ( text[k] != pat[j] ) {
(9)                   mis += weight[j];   break;  // a mismatch found, no need to continue
                      }
(10)   if  ( !mis)  // test unimportant part characters from left to right.
(11)        for ( j = pref_inf + 1; k = i – suf + 1; j < m;  j++,  k++ )
(12)            if ( text[k] != pat[j] ) {
(13)                  mis += weight[j];
(14)                  if ( mis > kmis)   break;  // There is no occurrence
                }
(15)  if (j == m) report that there is an occurrence at i – m –1 to i
(16)   i++;  // move to the next location
      }
```

The following example demonstrates how the WA works:

**Example:** Let the number of mismatches *kmis* = 1. Let *Text* and *Pat* be as follows:

**Text:** PPEEPPPIIDPPIDPPIISE
**Pat:** PPEESS

Assume that "PPEE" is the *important*-part and "SS" is the *unimportant*-part. The comparison starts at character 'E' at $Text_3$ (the last character in the *important* part), then $Text_0$ (first character) followed by $Text_2$, then $Text_1$ respectively (remaining characters compared right-to-left). Since there is no mismatch for the *important* part the algorithm moves to the *unimportant* part, $Text_4$ and $Text_5$ (the *unimportant* part is compared from left to right). The algorithm detects two mismatches (making *mis* = 2), which is greater than the specified criterion (*kmis* = 1). Six character comparisons are needed. Thus the algorithm moves forward one position.

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

The algorithm detects one mismatch at character 'P' at $Text_4$, in one character comparison. Thus it will move ahead one position.

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

Another mismatch is detected at $Text_5$, in a one-character comparison. Moving ahead two positions, we get:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

Another mismatch is detected at $Text_6$, in four character comparisons. After moving one location, the result as follows:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

As in the first step above, two mismatches detected at $Text_9$ and $Text_{10}$ and six character comparisons are needed. Thus moving ahead four positions (one movement at a time), we get:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

One mismatch is detected at $Text_9$ and five character comparisons are needed. There is a mismatch at the next three positions, thus we can move ahead four positions. We get:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

Up to this point, five character comparisons are needed to detect one mismatch at $Text_{13}$, thus we can move ahead one position.

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

At this point, each character in *Text* at the *important* part is matched with the corresponding character in *Pat*. There is only one mismatch at character 'E' at $Text_{19}$ (i.e., *mis* ≤ *kmis*), thus we have found one occurrence of *Pat* in *Text*. Six character comparisons are needed. Moving forward one position, we get:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

One character comparison is needed to detect one mismatch at $Text_{18}$. Moving forward one position ends the searching process, because the number of characters left in *Text* is less than *m* – *kmis*, where *m* is the number of characters in the pattern. The total number of character comparisons needed is 35.

## THE MODIFIED CHARACTER-WEIGHT ALGORITHM (MWA)

Like any approximate string matching algorithm, MWA finds all the relevant occurrences of a pattern $Pat_0,\ldots,$ $Pat_{m-1}$ in the text $Text_0 \ldots Text_{n-1}$ with a maximum of *k_mismatches*. It preprocesses the pattern to divide the pattern into two parts (*important* and *unimportant*) and to produce three different arrays *skip*, *position* and *sign*. The length of the first part is called *pref_inf*. It is equal to *m* / 3 * 2 - 1. In other words, it equals to two-thirds of m, where *m* is equal to the *Pat* length. The length of the second part is called *suf*. It is equal to *m-pref_inf* - 1. In this study, we assumed that the first part (with length *pref_inf*) is the *important* part. Generally speaking, the *important* part might be any segment of the pattern.

The *skip* array records how much the pattern is to be shifted forward in relation to the text when a match/mismatch is found and the reference character in the text exists in the pattern. In this algorithm, there is only one dynamic reference character.

Let i0 = $Text_j$, where $Text_j$ is the character that corresponds to the last character in the pattern. Let *base* = *j* - *suf* be the position of the base character in the text that corresponds to the last character in the *important* part in Pat. From the *base* character, the position of the dynamic reference character (*ref*) of MWA can be calculated. If the base character occurs in *Pat*, then *ref* is calculated according to the following formula:

$$ref = (\mathbf{base} + \mathbf{1}) + (\mathbf{pref\_inf} + 1) - \mathbf{pT},$$

where *base* is defined above, pref_inf is the length of the *important* part in *Pat* and *pT* = pos[*Text*[*base*]], the position of the base character in the *Text*.

**Example**: In order to motivate these formulas, let us consider this example.

Text: ABCDEFGH
Pat: CDEFGH

In this example, j = 5 at character 'F' in *Text*. The value of *pref_inf* is 3 and the value of *suf* is 2. The *base* reference equals 3 at character 'D' in *Text* and it occurs at position *pT* = 2 in *Pat*. The position of the reference character *ref* = 3 + 3 + 1 – 2 = 5. So, the character 'F' in *Text* is the reference character. The location of *ref* ranges from *base* to *base* + *pref_inf* + 1.

The shift distance *d* depends on the occurrence of *ref* in *Pat*. If *ref* does not occur in *Pat*, then *d* = 2 * (*pref_inf* +1) – *pT* + 1 (i.e., *d* ranges from *pref_inf* +1 to 2 * *pref_inf* +1). If *ref* occurs in *Pat*, then the shift distance *d* = *skip*[*Text*[*ref*]] – *pT* +1. In other words, *d* ranges from 1 to 2 * *pref_inf* – 1.

The *sign* array tells whether the reference character in *Text* exists in the given pattern or not. Each location in this array holds one value, either zero (i.e., *ref* does not occur in *Pat* and there is a mismatch) or one (i.e., *ref* character occurs in *Pat*, but it is not necessary that there is a match). So, the algorithm works as follows:

* Checking step: The existence of *base* in *Pat* is checked first. If it does not exist, move to the skipping step. Otherwise, there is a possibility of an occurrence. In such a case, the last and first characters of the *important* part will be checked first followed by the rest of the *important* part from right to left. If there is any mismatch at the *important* part, then move to the skipping step. Otherwise, the *unimportant* part will be checked from right to the left. The number of mismatches will be counted. If the number of mismatches is less than or equal to k_mismatches, then an occurrence will be reported. Otherwise, there is no occurrence and we move to the skipping step.
* After the checking step, the reference character is determined according to the character that is next to the *base* character. Next, the shift distance is determined according to the occurrence of *ref* in *Pat*. Alignment is carried out and followed by the *checking* step. The pseudo code for the MWA algorithm that reflects the above ideas follows:

```
void MWA (char *Text, int n, char *Pat, int m)
/* where:
n is the Text length
Text: Text[0] … Text[n-1]
m is the  Pat length
Pat: Pat[0] … Pat[m-1] */
int  sign[ALPHABET_SIZE];   /* sign: sign[0] … sign[ALPHABET_SIZE - 1] */
int skip[ALPHABET_SIZE];   /* skip: skip[0] … skip[ALPHABET_SIZE - 1] */
int pos[ALPHABET_SIZE];    pos: pos[0] … pos[ALPHABET_SIZE - 1] */
    {
       pref_inf = m / 3 * 2 - 1;
       suf = m - pref_inf - 1;
     // initialize the arrays pos, skip and sign
       for(j=0; j<ALPH_SIZE; j++) {
pos[j]=0;       skip[j] = 2 * (pref_inf + 1);          sign[j]=0;
       }
     // Update the arrays according to Pat
       for(j=0; j<=pref_inf; j++) {
 pos[pat[j]]=j+1;       skip[pat[j]] = 2 * (pref_inf + 1) - j - 1;      sign[pat[j]] = 1;
       }
     //  Start Searching Here
       i0  = m - 1;  base = i0 – suf;
       while(i0 <= n - 1 + kmis) {
mis = 0;
if (!sign[text[base]]) {   //  skip if base character doesn't occur in Pat
            pT = pos[text[base]];
            ref = base + (pref_inf + 1) - pT;
            if (!sign[text[ref]])
                  i0 += 2 * (pref_inf + 1) - pT;
            else
i0 += skip[text[ref]] - pT;
         }
```

```
else {  /*  Test the possibility of occurrence */
       // test the last and the first characters
           if( text[base] != pat[pref_inf] || text[i0 - m_minus_1] != pat[0]  ) goto MOVE;
           else {
  // test the rest of important part
  for (j = pref_inf -1 , k = base -1; j > 0; k--, j--)
                      if ( text[k] != pat[j] )  goto MOVE;
                   // Test the unimportant part
                   for (j = pref_inf + 1 , k = base + 1; j < m; j++, k++) {
                  if ( text[k] != pat[j] ) mis += wight[j];
      if ( mis > kmis ) goto MOVE;
                   }
                   /* Report a match at i0 – m . . . i0-1   */
           }
}
MOVE:
      pT = pos[text[base + 1]];
      ref = (base + 1) + (pref_inf + 1) - pT;
      if (!sign[text[ref]])
           i0 += 2 * (pref_inf + 1) - pT + 1;
      else
i0 += skip[text[ref]] - pT + 1;
}  // End while

     } // Function end
```

**Worst Case and Average Case Analysis:** Let $W(n)$ be the number of comparisons required in the worst case to find all the occurrences of *Pat* in *Text*. Such worst case can be found when:

* The character at (*base* + 1) occurs at the last position in the *pref_infx* part (i.e., important part) and
* The *K*_mismatches are found at the last *K* positions in the *unimportant* part (suffix part).

This causes one skip position and *m* comparisons in each iteration. So, $W(n) = m*(n–m+1 + K\_mismatches)$ comparisons. Thus, the number of shifts with the worst case is equal ($n – m + K\_mismatches + 1$).

Regarding the average case, let $A(n)$ be the number of shifts performed in the average case. The shift distance in each iteration will be equal to either 1, 2, 3, …, $2(2m/3)+1 = 4m/3 + 1$. Assume the shift distance is equally likely to be in any particular iteration that equals to $1/((4m/3)+1)$. Then,

$$A_i(n) = 1/((4m/3)+1) * \sum_{i=1}^{4m/3+1} i = (2m/3+2)/2$$
$$= (4m+6)/3$$

**Example:** The same text and pattern used in the previous example are used here.

**Text:** PPEEPPPIIDPPIDPPIISE
**Pat:** PPEESS

As in the previous example, "PPEE" in *Pat* is the *important* part and "SS" is the *unimportant* part. According to the given pattern, the values of the *sign* array are:

*sign*['P'] = *sign*['E'] = *sign*['S'] = 1.

Otherwise, the initial value of the *sign* array is zero. At the beginning and in iteration number 1, MWA starts the comparison at the *important* part at the last character and then the first, followed by the rest from right to left (i.e., $Text_3$, $Text_0$, $Text_2$ and, $Text_1$ respectively). Since there is no mismatch for the *important* part, the algorithm moves to the *unimportant* part, $Text_4$ and $Text_5$ (*unimportant* part compared from left to right). The algorithm detects two mismatches (making *mis* = 2) and that is greater than the specified criterion (*kmis* = 1). This is the end of the checking step. Six character comparisons and one character access are needed. In order to complete the skipping step, *ref* must be calculated. $pT = pos['P'] = 2$, where 'P' at $Text_4$. The value of $ref = (base + 1) + (pref\_inf + 1) – pT = 4 + 4 – 2 = 6$. Thus the algorithm moves forward five positions to align $Text_6$ with $Pat_1$. The result is as follows:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

The checking step starts again. The existence of the base character at $Text_8$ in *Pat* is tested. According to the *sign* array, it occurs in *Pat*. There is a match between each character in *Text* with the corresponding characters in the *important* part of *Pat*. Moving to the *unimportant* part, the algorithm detects two mismatches, which is greater than *kmis* = 1. Six character comparisons and one character access are

needed. The *ref* must be calculated. The character 'D' at $Text_9$ will be used as reference for calculating *ref*. $pT = pos$['D'] = 0. The value of $ref = (base + 1) + (pref\_inf + 1) - pT = 9 + 4 - 0 = 13$. Since $Text_{13}$ doesn't occur in *Pat*, the algorithm moves forward nine positions. The result is as follows:

**Text:** PPEEPPPEEDPPEDPPEESE
**Pat:** PPEESS

The existence of the base character at $Text_{17}$ in *Pat* is tested. According to the *sign* array, it occurs in *Pat*. There is a match between each character in *Text* with the corresponding characters in the *important* part of *Pat*. Moving to the *unimportant* part, the algorithm detects one mismatch, but that number is less than or equal to *kmis* = 1, thus, there is an occurrence of *Pat* in *Text*. Six character comparisons and one character access are needed. Moving to the next position ends the search process since the number of characters left in the text is less than the *Pat* length. So, the number of shifts is three and the total number of character comparisons is 18 in addition to 3 character accesses and 4 number comparisons.

## RESULTS AND DISCUSSION

In this experiment, the two algorithms WA and MWA were implemented and compared on English text with size exactly 1,005,077 characters. A program was designed and implemented in C++ to select randomly 3000 patterns divided into 10 groups. Each group consists of 300 patterns. All the information about the different groups can be seen in Table 1 (A). The cost of the search process is measured by finding the average number of shifts, the clock time and the average number of conditions with the different types including character access, character-comparisons and number comparisons to find all the occurrences of the patterns in each group in *Text*. The clock time measure includes the preprocessing of the patterns in the two algorithms. The results of the experiment can be seen in Table 1.

In Table 1(A), the number of groups, the pattern lengths, the number of k-mismatches and the average number of occurrences are presented. The pattern length ranges from 11 to 92 characters. The number of k-mismatches ranges from 5 to 23. The average number of occurrences ranges from 1.63 (group 8) to 105.84 (group 1).

The clock time required to find each group of patterns is recorded in Table 1 (B). By using WA, the clock time required ranges from 2.487 seconds (group 4) to 2.625 seconds (group 2). By using MWA, it ranges from 0.235 seconds (group 9) to 0.906 seconds (group 1). By using MWA, the clock time required by the algorithm WA is reduced by 65.29% (group 1) to 90.83% (group 9).

The average number of comparisons required to find the occurrences of patterns in *Text* is recorded in Table 1 (C). From the table, it is clear that MWA uses three types of conditions, including character accesses, number comparisons and character comparisons. On the other hand, WA uses only two types of conditions, including number comparisons and character comparisons. The character accesses are of the following form:

```
if (!sign[Text[ref]]); // see MWA algorithm.
```

The second type of condition is number-comparisons of the following form:

```
if(i0 <= n - 1 + kmis); // see both algorithms.
```

The third type of condition is character comparisons of the following form:

```
if ( Text[k] == Pat[j]); // see both algorithms.
```

In order to test if there is any difference among these three types of conditions, in terms of clock time execution, each type was repeated $10^7$ times. It was found that the clock time needed to execute a condition of type character comparison is reduced to 2% of that used by number comparison and 41% of that used to do a character access.

Looking at Table 1 (C), one can see that WA uses only number comparisons and character comparisons. The total number of comparisons for both conditions ranges from 3,025,097 (group 10) to 3,029,718 (group 1). While using MWA, it ranges from 118,433 (group 10) to 434,553 (group 1). The new algorithm MWA reduces the total average number of comparisons required by WA by 85.66% (group 1) to 96.09% (group 10).

The performance improvement of MWA comes from different factors, including:

* Shift distance: using MWA, the shift distance ranges from one position to 2 * *important* length. On the other hand, the shift distance in WA is one position only. Of course, increasing the shift distance reduces the number of shifts and in turn decreases the number of comparisons.
* Number of conditions: the average total number of conditions using WA is much larger than the average number of conditions using MWA (3,029,718 conditions vs. 4345,53 conditions). This result comes directly from the shift distance.
* Condition type: using MWA, about half of the average totals conditions of type character access, while all the conditions are of type character comparison and number comparison using WA (See Table 1 C). From the previous discussion, one can notice that the condition of type character access needs less clock time by 40% than the clock time required by either the condition of type character comparison or number comparison. This has a major effect on the performance of the two algorithms.

Table 1: A Comparison between MWA and WA for Each Group in Terms of 1) the Clock Time Required to Find Each Group 2) the Average Number of Comparisons 3) and 4) the Percentage of Improvements

| | (A) Pattern Information | | | (B) Average Clock Time (in Seconds) | | |
|---|---|---|---|---|---|---|
| | | | | Average clock time in seconds by using the two algorithms | | |
| Group No. | Pattern Length in characters | Number of K-mismatches | Ave. Num. of Occurrences | Using WA | Using MWA | Improvements of MWA vs. WA |
| 1 | 11 | 5 | 105.84 | 2.610 | 0.906 | 65.29% |
| 2 | 20 | 7 | 2.53 | 2.625 | 0.562 | 78.59% |
| 3 | 29 | 9 | 3.29 | 2.622 | 0.438 | 83.30% |
| 4 | 38 | 11 | 2.73 | 2.487 | 0.375 | 84.92% |
| 5 | 47 | 13 | 2.07 | 2.609 | 0.328 | 87.43% |
| 6 | 56 | 15 | 1.69 | 2.594 | 0.281 | 89.17% |
| 7 | 65 | 17 | 1.90 | 2.594 | 0.282 | 89.13% |
| 8 | 74 | 19 | 1.63 | 2.562 | 0.265 | 89.66% |
| 9 | 83 | 21 | 1.68 | 2.563 | 0.235 | 90.83% |
| 10 | 92 | 23 | 1.75 | 2.531 | 0.250 | 90.12% |

(C) Average Number of Comparisons Including Character Accesses, Number Comparisons and Character Comparisons. Note that WA Requires No Character Accesses

| | Ave. of Comparisons using WA | | | Ave. of Comparisons using MWA | | | | Total Improvement |
|---|---|---|---|---|---|---|---|---|
| Group No. | Number-Comp. | Character-Comp. | Total | Char. Access | Number Comp | Character Comp. | Total | of MWA Versus WA |
| 1 | 1018109 | 2011609 | 3029718 | 249582 | 102091 | 82880 | 434553 | 85.66% |
| 2 | 1016034 | 2010830 | 3026864 | 136270 | 58628 | 66836 | 261734 | 91.35% |
| 3 | 1015969 | 2011000 | 3026969 | 99948 | 44843 | 59585 | 204376 | 93.25% |
| 4 | 1015307 | 2010802 | 3026109 | 82590 | 37988 | 55089 | 175667 | 94.19% |
| 5 | 1016192 | 2010863 | 3027055 | 71089 | 33361 | 50824 | 155274 | 94.87% |
| 6 | 1017226 | 2010912 | 3028138 | 65240 | 31206 | 49751 | 146197 | 95.17% |
| 7 | 1017504 | 2011000 | 3028504 | 60107 | 29107 | 47311 | 136525 | 95.49% |
| 8 | 1015662 | 2010871 | 3026533 | 53932 | 26023 | 42707 | 122662 | 95.95% |
| 9 | 1014461 | 2010786 | 3025247 | 52187 | 25423 | 42968 | 120578 | 96.01% |
| 10 | 1014292 | 2010805 | 3025097 | 50581 | 24876 | 42976 | 118433 | 96.09% |

Table 2: A Comparison between WA and MWA with Text Size 1,907,938 Characters

| | (A) Pattern Information | | | (B) Average Clock Time (in Seconds) | | |
|---|---|---|---|---|---|---|
| | | | | Average clock time in seconds by using the two algorithms | | |
| Group No. | Pattern Length in characters | Number of K-mismatches | Ave. Num.of Occurrences | Using WA | Using MWA | Improvements of MWA vs. WA |
| 1 | 11 | 5 | 142.64 | 4.703 | 1.703 | 63.79% |
| 2 | 20 | 7 | 2.90 | 4.609 | 1.063 | 76.94% |
| 3 | 29 | 9 | 3.83 | 4.609 | 0.828 | 82.04% |
| 4 | 38 | 11 | 3.35 | 4.594 | 0.687 | 85.05% |
| 5 | 47 | 13 | 2.33 | 4.656 | 0.610 | 86.90% |
| 6 | 56 | 15 | 1.94 | 4.657 | 0.562 | 87.93% |
| 7 | 65 | 17 | 1.97 | 4.672 | 0.516 | 88.96% |
| 8 | 74 | 19 | 1.71 | 4.578 | 0.469 | 89.76% |
| 9 | 83 | 21 | 1.72 | 4.578 | 0.453 | 90.11% |
| 10 | 92 | 23 | 1.89 | 4.562 | 0.453 | 90.07% |

In order to test the effect of text size on the performance of the two algorithms, the text size has been increased from 1,005,077 characters into 1,907,938 characters. The experiment is repeated with the same specifications except the text size. Table 2 reflects the results.

In Table 2(A), the average number of occurrences ranges from 1.71 (group 8) to 142.64 (group 1). The clock time required to find each group of patterns is recorded in Table 1 (B). By using WA, the clock time required ranges from 4.562 seconds (group 10) to 4.703 seconds (group 2). By using MWA, it ranges from

0.453 seconds (group 9 and group 10) to 1.703 seconds (group 1). By using MWA, the clock time required by the algorithm WA is reduced by 63.79% (group 1) to 90.11% (group 9). This percentage of reduction is very close to the percentage of reduction (65.29% to 90.83%) for the text size 1,005,077 characters. This led us not to include the other associated information for the size 1907,938 in this study to avoid redundancy.

## CONCLUSION

A new approximate string searching algorithm, the Modified Weight Algorithm (MWA) has been developed and compared with a recent approximate string-searching algorithm called the Weight Algorithm (WA). The two algorithms preprocess the pattern only. An experiment was performed to evaluate the new algorithm MWA. The number of comparisons, the running time and the shift distance are the different criteria used to compare the different algorithms.

In comparison between WA and the new algorithm MWA, the results of the experiment suggest that: (I) the average number of comparisons required by WA (3,025,097 to 3,029,718) has been improved by MWA (118,433 to 434,553). So, the improvement ranges from 85.66% to 96.09%, so the new algorithm requires only 4% to 15% as many comparisons. (II) The clock time range required by WA (2.61 to 2.653 sec) has been improved by MWA to become (0.906 to 0.235 sec). Thus, the percentage of improvement ranges from 65.29% to 90.83% and the new algorithm clock times range from 10% to 35% of the old ones. Furthermore, using the MWA, the shift distance ranges from 1 to 2* *important*_part_length + 1 positions. On the other hand, using WA, the shift distance is one position only.

The MWA algorithm gains its performance from two main changes. One change is the use of character access conditions. All the conditions used by WA, whether number comparisons or character comparisons require high execution clock time. On the other hand, using MWA about half the conditions are of type character access, which need less clock time than the other types of conditions. This increases the performance of MWA in comparison with WA. In general, increasing the percentage of the required conditions of type character-access increases the performance of string searching algorithms.

The other direction is the shift distance (1 to 2* *important*_part_length + 1). This decreases the number of required conditions, which, in turn, decreases the clock time required to finish the task. However, in our experiments using MWA we considered the first two-thirds of the pattern as the *important* part. In practice, the *important* part might be any part of the pattern. The question arises, is it possible to determine the *important* part automatically. How does that affect the relevance of the occurrences? How does that affect the system performance? Such questions need to be investigated next.

## ACKNOWLEDGMENTS

The authors would like to thank Professor Martha Evens from Illinois Institute of Technology for her helpful comments and suggestions that reflected many improvements in the presentation of this study.

## REFERENCES

1. Fredriksson, K., 2003. Shift or string matching with super-alphabets, Information Processing Letters, 87: 201-204.
2. Apostolico, A. and Z. Galil, 1997. Pattern Matching Algorithms, Oxford University Press.
3. Mhashi, M., 2003. A Fast String Matching Algorithm using Double-Length Skip Distances. Dirasat J., University of Jordan, Jordan, 30: 84-92.
4. Fenwick, M., 2001. Fast string matching for multiple searches. Software-Practice and Experience, 31: 815-833.
5. Lecroq, T., 1998. Experiments on string matching in memory structures, Software-Practice and Experience, 28: 561-568.
6. Dermouche, A., 1995. A fast algorithm for string matching with mismatches, Information Processing Letters, 55: 105-110.
7. Park, K. and Z. Galil, 1990. An improved algorithm for approximate string matching, SIAM J. Comput., 19: 989-999.
8. Wu, S. and U. Manber, 1995. Fast text searching with errors, Comm. ACM, 35: 83-91.
9. Galil, Z. and K. Park, 1998. An improved algorithm for approximate string matching, in ICALP, pp: 394-404.
10. Cole, R. and R. HariHaran, Approximate String Matching: A faster simpler algorithm, In Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp: 463-472.
11. Grossi, R. and F. Luccio, 1989. Simple and efficient string matching with k mismatches. Information Processing Letters, 33: 113-120.
12. Mhashi, M., 2001. Character-Weight based k-mismatches and Relevancy. Al-Manarah, 7: 95-114.