Asian Journal of Information Technology 8 (1): 14-23, 2009
ISSN: 1682-3915

On Improving the Naïve String Matching Algorithm

Rami H. Mansi and Jehad Q. Odeh
Department of Computer Science, Al-Bayt University, Mafraq, Jordan

Abstract: String matching algorithms are essential components used in implementations of the practical software under most operating systems. It is important to any string matching algorithm to be able to locate quickly some or all occurrences of a user-specified pattern in a text. In this study, we propose three new exact single pattern matching algorithms. These are: FC-RJ (First Character-Rami and Jehad), FLC-RJ (first and last Characters-Rami and Jehad) and FMLC-RJ (first, middle and last Characters-Rami and Jehad). The proposed algorithms rely on utilizing new technique based on occurrence list. The proposed algorithms are analyzed and implemented as a part of the experimental simulation system (RJ-SMT). The extensive testing and comparisons with the Naïve (Brute Force) algorithm show that the proposed algorithms enhance execution time by 7.4, 16.2 and 20.6%, respectively.

Key words: String matching, naïve algorithm, occurrence list, complexity

INTRODUCTION

The string matching problem may define as finding one or more of the occurrences of a pattern P of length m in a text T of length n. It has been extensively studied and many techniques and algorithms have been designed to solve this problem. These algorithms are mostly used in information retrieval, bibliographic search, molecular biology and question answering applications (Lecroq, 2007; Wu *et al*., 2007).

String matching is a very important subject in the wider domain of text processing and its algorithms are the basic components used in implementations of practical software under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (Watson, 2002; Charras and Lecroq, 2004).

In many information retrieval and text-editing applications, it is necessary to be able to locate quickly some or all occurrences of a user-specified pattern of words and phrases in a text (Alqadi *et al*., 2007). Furthermore, string matching has many applications including database query, DNA and protein sequence analysis. Therefore, the efficiency of string matching has a great impact on the performance of these applications (Crochemore and Lecroq, 2003). Although, data are memorized in various ways, text remains the main and most efficient form to exchange information (Kim and Kim, 1999; Sheu *et al*., 2008).

Basically, a string matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then, it checks if the pattern occurs in the window (this specific research is called an attempt) and shifts the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text (Amintoosi *et al*., 2006).

Exact string matching means finding one or all exact occurrences of a pattern in a text. Naïve (Brute force) algorithm, as Charras and Lecroq (2004) mentioned, Boyer and Moore (1977), Morris and Pratt (Watson, 2002) and Knuth *et al*. (1977) are exact string matching algorithms.

Approximate string matching is the technique of finding approximate (may not exact) matches to a pattern in a string (Karp and Rabin, 1987). The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. The usual primitive operations are insertion, deletion and substitution (Amintoosi *et al*., 2006). So, the input of an approximate string matching algorithm is a text string T, a pattern string P and an edit cost bound k and the task of the algorithm is to answer the question: can we transform a part of T to P using at most k additions, deletions and substitutions? (Navarro and Fredriksson, 2004).

Some of the exact string matching algorithms have been presented to solve the problem of searching for a single pattern in a text, such as Boyer and Moore (1977), Morris and Pratt (Watson, 2002), Knuth *et al*. (1977) and Karp and Rabin (1987), algorithms. On the other hand, some have been presented to solve the problem of searching for multiple patterns in a text.

Although, the Knuth-Morris-Pratt algorithm has better worst-case running time than the Boyer-Moore

Corresponding Author: Rami H. Mansi, Department of Computer Science, Al-Bayt University, Mafraq, Jordan

algorithm, the latter is known to be extremely efficient in practice (Crochemore *et al*., 1994; Watson and Watson, 2003).

Since 1977, with the publication of the Boyer-Moore algorithm, there have been many papers published that deal with exact pattern matching and in particular discuss and/or introduce variants of Boyer-Moore algorithm. The pattern-matching literature has had two main categories (Danvy and Rohde, 2006; Franek *et al*., 2006):

1. Reducing the number of character comparisons required in the worst and average cases
2. Reducing the time requirement in the worst and average cases

This research is an attempt to enhance the time complexity of the naïve (Brute force) string matching algorithm in its average and worst cases.

The Brute force algorithm consists of checking, at all positions in the text between 0 and n - m, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase and a constant extra space in addition to the pattern and the text. During the searching phase, the text character comparisons can be done in any order. The time complexity of the searching phase is O (mn), where m is the length of the pattern and n is the length of the text and the expected number of text character comparisons is 2n.

In order to reduce the processing time of the naive algorithm, we present three exact single pattern matching algorithms, FC-RJ (First Character-Rami and Jehad), FLC-RJ (First and Last Characters-Rami and Jehad) and FMLC-RJ (First, Middle and Last Characters-Rami and Jehad) algorithms. The proposed algorithms improve the length of the shifts of the brute force algorithm. The extensive testing of the proposed algorithms yields to speeding up the Brute force algorithm.

FC-RJ algorithm: Most of string matching algorithms search for the pattern in the whole text and match (compare) most of the text's characters with the pattern's characters (Lecroq, 2007).

Instead, it is reasonable to assume that it will be more efficient to match the pattern with the sub-strings of the text, which start with the pattern's first character, while ignoring the rest of the characters in the text. Depending on the concept and the research of the proposed algorithm, we name it as FC-RJ algorithm. The FC-RJ algorithm finds the indices of all occurrences of the first character of the pattern in the text prior to the searching phase. These indices should be saved in a list (array) to be accessed during the searching phase, which we name it as (Occurrence_List). In the searching phase, the algorithm uses the Occurrence_List to move to the indices of the text that contain the first character of the pattern.

The main procedures of the proposed algorithm are expressed as follows:

Preprocessing phase:

1. The algorithm creates a new array called (Occurrence_List) of size (n - m + 1), where n is the size of the text and m is the size of the pattern. The length of the Occurrence_List is (n - m + 1) because it is impossible to the pattern to occur after the position (n - m) in the text
2. This array will hold the indices of the occurrences of the pattern's first character in the text using an integer variable (i) starting from (0) and incremented by one after each match
3. The algorithm scans the text in a single pass, using an integer variable (j) and compares its characters with the pattern's first character. If the current character of the text (jth character) is equal to the pattern's first character, the algorithm saves the index of the current character in the text (the value of j) in the ith index of the Occurrence_List array and increments the value of (i) by one

Searching phase:

1. If the value of (i) is greater than (0); this means the pattern's first character occurs in the text. So, go to step (2), otherwise; the pattern cannot be found in the text at all. So, go to step (6)
2. Create an integer variable (c), starts from (0) to (i - 1) and incremented by one
3. If the value of the variable (c) is less than (i); go to step (4). Otherwise, go to step (6)
4. Scan the sub-string of the text starting from the index (Occurrence_List (c) + 1) to ((Occurrence_List (c) + m - 1), which represents the size of the pattern and compare each character of this sub-string with the corresponding character in the pattern. If all characters are matched, then this is an occurrence of the pattern in the text at index (Occurrence_List (c)).
5. Increment the value of (c) by one and go to step (3)
6. Exit

In step (1) of the searching phase, if the value of (i) is equal to (0), then this means that the first character of the pattern does not occur in the text at all and there is no

need to search for the pattern. In step (2), the value of the variable (c) must not exceed the value (i-1), which is the number of the occurrences of the pattern's first character in the text.

Pseudocode of FC-RJ algorithm: The pseudocode of the preprocessing phase of FC-RJ algorithm is expressed as follows:

```
procedure PRE-FC-RJ (array T[n],array P[m])
        var j:=i:=0 as integer
             Create array:
Occurrence_List[n-m+1]
        for j from 0 to n-m do
          if T (j) == P (0) then
            Occurrence_List (i):= j
            i:= i + 1
              SEARCH-FC-RJ (T[n], P[m], i,
               Occurrence_List[n-m+1])
end procedure
```

The pseudocode of the searching phase of FC-RJ algorithm is as follows:

```
procedure SEARCH-FC-RJ (array T[n],array P[m], i,
array Occurrence_List[n-m+1])
        if i > 0 then
              if m==1 then output the
content of the Occurrence_List ()
          else
                var c:=x:=0, count:=1,
            as integer
             var value as Boolean
             while c < i do
               value:= true
                    for x from Occurrence_List (c)+1 to
Occurrence_List (c)+ m-1 do
                 if T (x)≠P (count) then
                   value:= false
                   break for loop
                 count:= count+1
               if value==true then
               output (Occurrence_List (c))
               c:=c+1
               count:=1
               else
  output ("The pattern is not
                found!")
end procedure
```

Example 1: A single pattern matching example using FC-RJ algorithm.

For simplicity, assume that we have the following text and pattern and we want to find all occurrences of the pattern in the text:

Text
AMACCOAMBACHAMABCOAMALCO

Pattern:
AMABCO

Then the algorithm creates the Occurrence_List to save the indices of the text's characters that equal the pattern's first character, which is (A) in this example. The algorithm searches for the first character of the pattern in the range of indices from (0) to (n - m = 24 - 6 = 18) of the text, because what is left is less than the length of the pattern and it is impossible to the pattern to occur after index (18) in the text. The Occurrence_List will be as follows:

Occurrence_list						
0	2	6	9	12	14	18

In the searching phase, the algorithm will make (7) matching attempts to search for the pattern in the text using the elements values of the Occurrence_List as indices, as follows:

First attempt (at index 0, three character comparisons, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
AMABCO

Second attempt (at index 2, one character comparison, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
  AMABCO

Third attempt (at index 6, two character comparisons, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
      AMABCO

Fourth attempt (at index 9, two character comparisons, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
         AMABCO

Fifth attempt (at index 12, five character comparisons, Match):
AMACCOAMBAMHAMABCOAMALCO
           AMABCO

Sixth attempt (at index 14, one character comparison, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
             AMABCO

Seventh attempt (at index 18, three character comparison, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
               AMABCO

The algorithm performed (17) character comparisons in the example.

Analysis of FC-RJ algorithm: The preprocessing phase of FC-RJ algorithm is concerned in determining and saving the indices of the text segments that represent expected occurrences of the pattern. These indices are saved in the Occurrence_List array of size (i). This variable represents the number of expected occurrences of the pattern in the text, which is at most, equals to (n-m + 1), where n is the length of the text and m is the length of the pattern.

To do so, the preprocessing phase scans the first (n-m) characters of the text. Thus, it is linear in O (n-m), in the best, average and worst cases of FC-RJ algorithm.

The searching phase uses the Occurrence_List array to move to the indices of the text that represent expected occurrences of the pattern using the variable x, which starts with the value (0) and ends with (i - 1), where i is the number of expected occurrences of the pattern in the text.

The best case of the searching phase of FC-RJ algorithm arises when the variable i equals to zero. In other words, when there are no occurrences of the pattern in the text in this case, the time complexity of the searching phase of FC-RJ algorithm is O (1).

FC-RJ algorithm uses the Occurrence_List to search the text for the pattern. The number of places (indices in the text) that the algorithm starts searching at (i) represents the number of expected occurrences of the pattern in the text. At each xth index in the text, the searching phase tries to match the segment (x + 1… x + m-1) of the text with the pattern, character by character. So, the algorithm compares m-1 characters at each xth index, until it reaches the (i-1)th element of the Occurrence_List array. This means, it takes (i)×(m-1) time. Thus, the searching phase takes O ((i×m)-i) time in the worst case of FC-RJ algorithm, where i is the number of expected occurrences of the pattern in the text and m is the length of the pattern. The algorithm performs at most (im)-i text character comparisons during the searching phase.

The preprocessing phase of the FC-RJ algorithm searches the first n-m portion of the text for the expected occurrences of the pattern. Therefore, FC-RJ algorithm requires O (n-m) extra space for the Occurrence_List array in addition to the original text and pattern. If the size of the Occurrence_List array is specified dynamically; the preprocessing phase will require i additional space instead of n-m + 1, where i is the number of expected occurrences of the pattern in the text.

FLC-RJ algorithm: The concept of FLC-RJ (first and Last Characters-Rami and Jehad) algorithm follows the concept of FC-RJ algorithm. It seems more efficient to attempt matching the pattern only with the sub-strings of the text that start with the pattern's first character and also end with the pattern's last character.

This technique decreases the number of character comparisons in the text. It can be achieved by simply adding another condition (restriction) in the preprocessing phase of FC-RJ algorithm.

Because this algorithm searches for the first and last characters of the pattern in the text; it requires that the pattern to be of length more than one character to work efficiently. If the pattern consists of only one character, then this character will be considered as the first and the last character of the pattern and it will be compared twice instead of one time at each comparison operation with the text characters. To avoid occurring of this case, the algorithm behaves as FC-RJ algorithm in such case. In other words, if the pattern consists of only one character; FLC-RJ algorithm will search the text only for the first character of the pattern and it will behave exactly as FC-RJ algorithm.

Pseudocode of FLC-RJ algorithm: The pseudocode of the preprocessing phase of FLC-RJ algorithm is as follows:

```
procedure PRE-FLC-RJ (array
                    T[n],array P[m])
    var j:=i:=0 as integer
    Create array:
Occurrence_List[n-m+1]
    if m >1 then
     for j from 0 to n-m do
          if T (j)==P (0)AND
    T (j+m-1)==P (m-1) then
              Occurrence_List (i):= j
              i:= i + 1
        else
          for j from 0 to n-m do
            if T (j) == P (0) then
              Occurrence_List (i):= j
              i:= i + 1
        SEARCH-FLC-RJ (T[n], P[m], I,
        Occurrence_List[n-m+1])
end procedure
```

The searching phase should not compare the characters that are already matched during the preprocessing phase. This implies that the first and the last characters of each segment in the text will not be compared with the characters of the pattern during the searching phase, since they have been matched during the preprocessing phase and there is no need to be compared again.

The searching phase of FLC-RJ algorithm is as follows:

```
procedure SEARCH-FLC-RJ
 (array T[n],array P[m], i,
  array Occurrence_List[n-m+1])
       if i > 0 then
         if m= =1 then
    output the content of the
    Occurrence_List ()
        else
          var c:=x:=0, count:=1,
          var value as Boolean
          while c < i do
           value:= true
           for x from
Occurrence_List (c) + 1
to Occurrence_List (c)+ m -2 do
              if T (x)≠P (count) then
                value:= false
                break the for loop
              count:= count+1
            if value==true then
            Output
    (Occurrence_List (c))
            c:=c+1
            count:=1
       else output ("The pattern is not found!")
end procedure
```

Example 2: A single pattern matching example using FLC-RJ algorithm:

Assume that the same text and pattern of example 1 are used in this example utilizing FLC-RJ algorithm.

Text
AMACCOAMBACHAMABCOAMALCO

Pattern:
AMABCO

Then the preprocessing phase will determine the indices of the expected occurrences of the pattern in the text by comparing the first character of the pattern, which is (A) in this example, with the first n-m characters of the text, since the pattern cannot be occurred after the first n - m characters in the text, using a variable j. If the current jth character in the text is matched with the pattern's first character; the last character of the pattern will be compared with the (j + m-1)th character of the text, since the segment (j…j + m-1) represents the length of the pattern (m).

If the first and the last characters of a segment in the text equal the first and the last characters of the pattern respectively, then this segment will be considered as an expected occurrence of the pattern in the text and the index of this segment in the text will be saved in the Occurrence_List array. The Occurrence_List of this example will be as follows:

Occurrence_List		
0	12	18

In the searching phase, the algorithm will make (3) matching attempts to search for the pattern in the text using the elements values of the Occurrence_List as indices, as follows:

First attempt (at index 0, three character comparisons, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
AMABCO

Second attempt (at index 12, four character comparisons, Match):
AMACCOAMBAMHAMABCOAMALCO
AMABCO

Third attempt (at index 18, three character comparison, Mismatch):
AMACCOAMBAMHAMABCOAMALCO
AMABCO

The algorithm performs (10) character comparisons in the example. As shown in this example, it is clear that FLC-RJ algorithm decreases the number of character comparisons as compared to FC-RJ algorithm, because comparing the first and last characters of the pattern eliminates some of the mismatches even before the searching phase started.

Analysis of FLC-RJ algorithm: The preprocessing phase scans the first (n-m) characters of the text to determine the expected occurrences of the pattern in the text. Thus, the preprocessing phase is linear in O (n-m) in the best, average and worst cases of FLC-RJ algorithm.

The searching phase uses the Occurrence_List array to reach the indices of the text that represent expected occurrences of the pattern using the variable (x), which starts with the value (0) and ends with (i-1).

The best case of the searching phase of FLC-RJ algorithm arises when the variable (i) equals to zero. In other words, when there are no occurrences of the pattern in the text in this case, the time complexity of the searching phase of FLC-RJ algorithm is O (1).

FLC-RJ algorithm uses the Occurrence_List to search the text for the pattern. The number of places that the algorithm starts searching at is (i), which represents the number of expected occurrences of the pattern in the text. At each xth index in the text, the searching phase tries to match the segment (x + 1…x + m-2) of the text with the pattern, character by character. It does not compare the first and the last character of the pattern and the text's segments; since they already have been matched during the preprocessing phase. So, the algorithm compares m-2

characters at each xth index, until it reaches the (i-1)th element of the Occurrence_List array. This means, it takes (i)×(m-2) time. Thus, the searching phase takes O ((i×m)-2i) time in the worst case of FLC-RJ algorithm, where i is the number of expected occurrences of the pattern in the text and m is the length of the pattern. The algorithm performs at most (im)-2i text character comparisons during the searching phase. The preprocessing phase of the FLC-RJ algorithm searches the first n-m portion of the text for the expected occurrences of the pattern. Therefore, FLC-RJ algorithm requires O (n-m) extra space for the Occurrence_List array, in addition to the original text and pattern. If the size of the Occurrence_List array is specified dynamically; the preprocessing phase will require i additional space instead of n-m + 1, where i is the number of expected occurrences of the pattern in the text.

FMLC-RJ algorithm: FMLC-RJ algorithm adds another restriction to a sub-string of the text to be considered as an expected occurrence of the pattern. It seems more efficient to attempt matching the pattern only with the sub-strings of the text that start with the pattern's first character and end with the pattern's last character and at the same time, they have middle characters equal the pattern's middle character.

This technique decreases the number of character comparisons in the text during the searching phase. It can be achieved by adding another condition in the preprocessing phase of FLC-RJ algorithm.

This algorithm requires the pattern to be of length more than two characters to work efficiently. Moreover, it should cover the case when the pattern consists of only one or two characters (m <3). The preprocessing phase of FMLC-RJ algorithm behaves as the preprocessing phase of FLC-RJ algorithm if m = 2 and as the preprocessing phase of FC-RJ algorithm if m = 1. In fact, most of these patterns used in the real world applications are of lengths more than two characters.

Pseudocode of FMLC-RJ algorithm: The preprocessing phase of FMLC-RJ algorithm can be expressed as follows:

```
procedure PRE-FMLC-RJ (array T[n],
                       array P[m])
      var j:=i:=0,
        mid:= floor (m/2) as integer
      Create array: Occurrence_List[n-m+1]
      if m > 2 then
        for j from 0 to n-m do
          if T (j)==P (0)
    AND T (j+mid)==P (mid)
    AND T (j+m-1)==P (m-1)
  then
            Occurrence_List (i):= j
            i:= i + 1
        else
          if m > 1 then
            for j from 0 to n-m do
              if T (j)==P (0)
    AND T (j+m-1)==P (m-1) then
                Occurrence_List (i):=j
                i:= i + 1
        else
          for j from 0 to n-m do
            if T (j) == P (0) then
              Occurrence_List (i):= j
              i:= i + 1
      SEARCH-FMLC-RJ (T[n], P[m], i,
  Occurrence_List[n-m+1])
end procedure
```

The searching phase compares the characters of the pattern with the characters of each expected occurrence except the first, middle and last characters, which have been matched in the preprocessing phase.

```
procedure SEARCH-FMLC-RJ (array T[n],
    array P[m], i,
    array Occurrence_List[n-m+1])
If i >0 then
        if m= =1 then
   output the content of the
   Occurrence_List ()
        else
          var c:=x:=0, count:=1,
   as integer
          var value as Boolean
          while c < i do
            value:= true
            for x from
      Occurrence_List (c)+1
    to Occurrence_List (c)+ m-2
    do
              if count==mid then
    increment x and count by 1
              if T (x)≠P (count) then
                value:= false
                break the for loop
              count:= count+1
            if value==true then
              output
        (Occurrence_List (c))
            c:=c+1
            count:=1
      else
  output ("The pattern is not found!")
end procedure
```

Example 3: A single pattern matching example using FMLC-RJ algorithm:

Assume that the same text and pattern of example 1 are used in this example utilizing FMLC-RJ algorithm, as follows:

Text
AMACCOAMBACHAMABCOAMALCO

Pattern:
AMABCO

The preprocessing phase searches for the segments of the text where the first, middle and last characters of these segments equal the first, middle and last characters of the pattern, respectively.

Since this condition only achieved at index (12) in the text of this example, then the Occurrence_List will be consisting of one element, as follows:

Occurrence_List

12

This means, there is only one expected occurrence of the pattern at index (12) in the text.

In the searching phase, the algorithm will make only (1) matching attempt, instead of (7) attempts of FC-RJ algorithm and (3) attempts of FLC-RJ algorithm, to search for the same pattern in the same text using the elements values of the Occurrence_List as indices, as follows:

First attempt (at index 12, three character comparisons, Match):

AMACCOAMBAMHAMABCOAMALCO
AMABCO

The algorithm performed (3) character comparisons in the example. As shown in this example, it is clear that FMLC-RJ algorithm decreased the number of character comparisons of both, FC-RJ and FLC-RJ algorithms, since this algorithm performed only (3) character comparisons while FC-RJ performed (17) and FLC-RJ performed (10) character comparisons to search for the same pattern in the same text used in the three algorithms.

Analysis of FMLC-RJ algorithm: The preprocessing phase of FMLC-RJ algorithm is concerned in determining and saving the indices of the text segments that represent expected occurrences of the pattern. As in FC-RJ and FLC-RJ algorithms, these indices are saved in the Occurrence_List array of size (i). This variable represents the number of expected occurrences of the pattern in the text, which is at most, equals to (n-m + 1).

The preprocessing phase scans the first (n-m) characters of the text. Thus, it is linear in O (n-m) in the best, average and worst cases.

The searching phase uses the Occurrence_List array to reach the indices of the text that represent expected occurrences of the pattern using the variable (x), which starts with the value (0) and ends with (i-1). The best case of the searching phase of FMLC-RJ algorithm arises when the variable (i) equals to zero. In other words, when there is no any expected occurrence of the pattern in the text, the best case time complexity of the searching phase of FMLC-RJ algorithm is O (1).

FMLC-RJ algorithm uses the Occurrence_List to search the text for the pattern. The number of places that the algorithm starts searching at is (i), which represents the number of expected occurrences of the pattern in the text. At each xth index in the text, the searching phase tries to match the segment (x + 1… x + m-2), except the middle character, of the text's segments with the pattern, character by character. It does not compare the first, middle and last characters of the pattern with those of the text's segments; since they already have been matched during the preprocessing phase. So, the algorithm compares m-3 characters at each xth index, until it reaches the (i-1)th element of the Occurrence_List array. This means, it takes (i)×(m-2) time. Thus, the searching phase takes O ((i×m)-2i) time in the worst case of FMLC-RJ algorithm, where i is the number of expected occurrences of the pattern in the text and m is the length of the pattern. The algorithm performs at most (im)-3i text character comparisons during the searching phase.

The preprocessing phase of the FMLC-RJ algorithm searches the first n-m portion of the text for the expected occurrences of the pattern. Thus, FMLC-RJ algorithm requires O (n-m) extra space for the Occurrence_List array in addition to the original text and pattern. If the size of the Occurrence_List array is specified dynamically; the preprocessing phase will require i additional space instead of n-m + 1, where i is the number of expected occurrences of the pattern in the text.

MATERIALS AND METHODS

To compare between the performance of our algorithms with the naïve (Brute force) algorithm; we have built a string matching tool using Visual Basic 6.0 (RJ-String Matching Tool (RJ-SMT)). In this tool, FC-RJ, FLC-RJ and FMLC-RJ algorithms have been implemented, in addition to the brute force algorithm.

Figure 1 shows the interface of the RJ-SMT. The tool is available at http://www.rjstringmatching.webs.com. We have extensively tested the proposed algorithms on random test data. A simple program is developed to create random test patterns with different lengths (1-14) of characters. Both characters of patterns and strings were in the main memory, rather than a secondary storage medium. The total number of instructions that got executed and execution time in seconds were considered in the evaluation process.

For each pattern length, 300 randomly selected samples were tested and averaged, while the total string length was 10,000 of randomly generated characters.

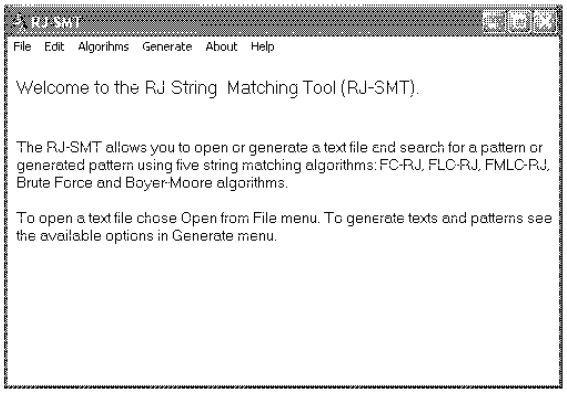

Fig. 1: Interface of RJ-SMT

RESULTS AND DISCUSSION

Table 1 shows the experimental results of the tested algorithms. The execution time in seconds is denoted by (T), while the number of executed instructions is abbreviated by (Inst) for each algorithm.

In Table 1, the execution time (T) and the number of executed instructions (Inst) of an algorithm represent the average of 300 runs of the algorithm using the same pattern length (m) and randomly selected characters of the pattern at each run.

Figure 2 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample, of each pattern length from 1-14 utilizing FC-RJ algorithm.

It is apparent that the best performance of the FC-RJ algorithms is when the length of the pattern was one character. This result is reasonable, since the algorithm only outputs the content of the Occurrence_List array if the pattern's length is only one character.

It is obvious that the execution time increases as the pattern gets longer.

Figure 3 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1-14 utilizing FLC-RJ algorithm.

The best performance of the FLC-RJ algorithms is when the length of the pattern was two characters. Since, the algorithm only outputs the content of the Occurrence_List array if the pattern's length is two characters.

Figure 4 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1-14 utilizing FMLC-RJ algorithm.

The best performance of the FMLC-RJ algorithms is when the length of the pattern was three characters. The algorithm searches for the first, middle and last characters

Table 1: Experimental results of the tested algorithms

Pattern	FC-RJ		FLC-RJ		FMLC-RJ		Brute force	
length	T	Inst	T	Inst	T	Inst	T	Inst
1	2.70	1087	2.90	50	3.00	400	3.30	30000
2	3.28	1215	2.87	34	2.90	50	3.50	30414
3	3.51	1185	2.95	54	2.50	83	4.36	30421
4	3.67	1157	2.98	68	2.71	82	4.85	30425
5	3.70	1226	3.10	69	3.10	96	4.95	30461
6	4.00	1279	3.70	90	3.50	56	4.98	30452
7	4.30	1236	3.50	78	3.30	96	5.20	30480
8	4.40	1241	4.00	47	3.80	63	5.30	30459
9	4.70	1159	4.30	76	4.10	170	5.40	30501
10	4.90	1236	4.50	84	4.30	132	5.50	30507
11	5.20	1226	4.80	125	4.50	192	5.80	30539
12	5.70	1339	5.20	55	5.00	143	6.00	30543
13	5.80	1579	5.50	98	5.20	224	6.30	30528
14	6.30	1600	5.70	80	5.40	240	6.80	30535

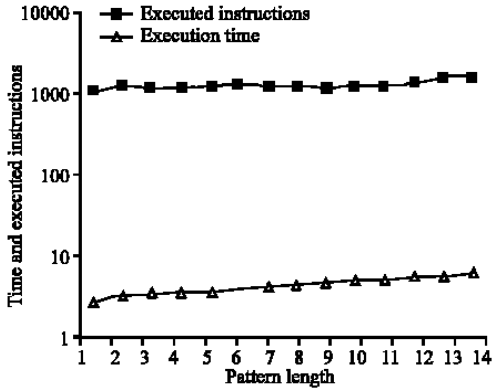

Fig. 2: Experimental results of FC-RJ algorithm

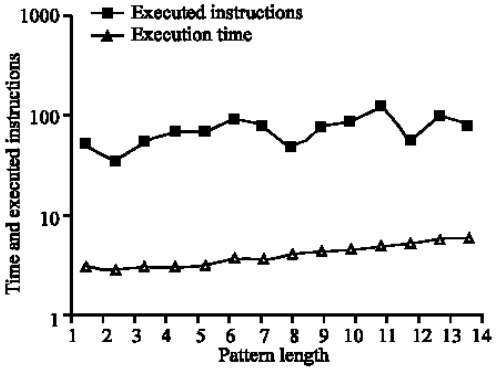

Fig. 3: Experimental results of FLC-RJ algorithm

of the pattern and then it outputs the content of the Occurrence_List array as a result. The execution time is done in the preprocessing phase in this case.

Figure 5 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1-14 when the brute force algorithm is used.

The best performance of the brute force algorithms is when the length of the pattern was relatively short. Since

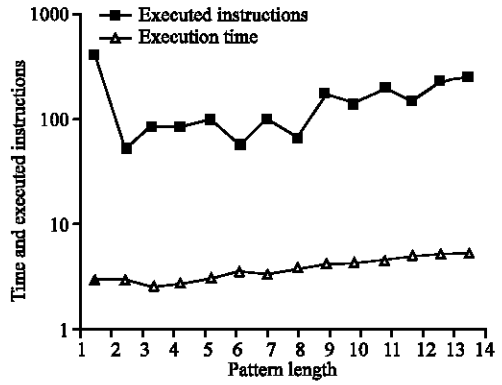

Fig. 4: Experimental results of FMLC-RJ algorithm

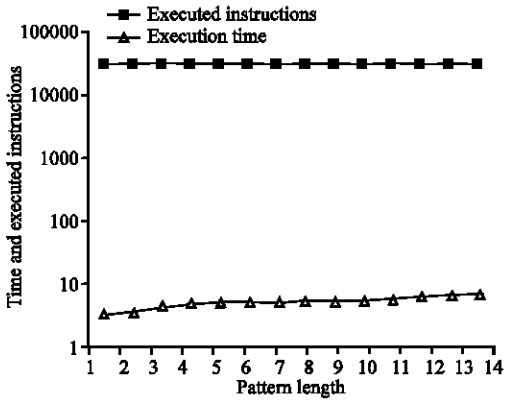

Fig. 5: Experimental results of the brute force algorithm

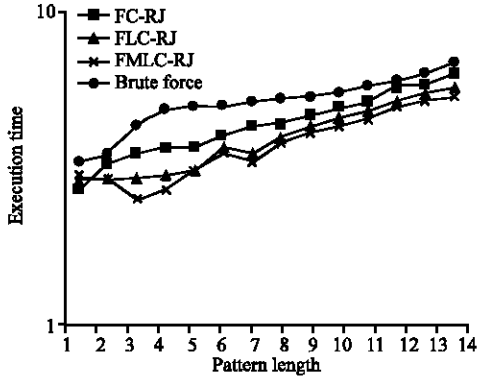

Fig. 6: Execution times of the tested algorithms

the algorithm compares almost m characters at each index of the text, the execution time increases as m gets larger.

Figure 6 shows a comparison between execution times of the FC-RJ, FLC-RJ, FMLC-RJ and Brute Force algorithms for each patterns sample of each pattern length from 1-14.

It is apparent that the FC-RJ, FLC-RJ and FMLC-RJ algorithms outperform the performance of the brute force algorithm.

Table 2: Percentages of enhancements in execution time

Algorithm	Enhancement on Brute force (%)
FC-RJ	7.4
FLC-RJ	16.2
FMLC-RJ	20.6

It is clear that our proposed algorithms enhance the execution time of string matching as compared to the brute force algorithm. This enhancement is calculated by considering the differences in execution times of the algorithms to search for 14 patterns samples as recorded in Table 1.

Table 2 presents the percentages of enhancements of the proposed algorithms as compared to the brute force algorithm.

CONCLUSION

In this study, three new single exact pattern matching algorithms are proposed. They are: FC-RJ, FLC-RJ and FMLC-RJ algorithms. Furthermore, a string matching tool (RJ-SMT) is developed to simulate and test the algorithms. The proposed algorithms, in addition to the naïve (Brute force) algorithm have been implemented and compared. The FC-RJ, FLC-RJ and FMLC-RJ algorithms enhances the execution time as compared to the brute force algorithm by 7.4, 16.2 and 20.6%, respectively. The proposed algorithms add some restrictions (conditions) in order to consider a segment in the text as an expected occurrence of the pattern and then it can be referenced during the searching phase. If the added conditions are not satisfied for a segment in the text; it will be excluded during the searching phase. The proposed algorithms were implemented, analyzed, tested and compared. The results were promising and other enhancements may be further studied.

REFERENCES

Alqadi, Z., M. Aqel and I. El-Emary, 2007. Multiple-Skip Multiple-Pattern Matching Algorithm (MSMPMA). IAENG Int. J. Comput. Sci., 34 (2): 14-20. http://www.iaeng.org/ijcs/issues_v34/issue_2/ijcs_34_2_03.pdf.

Amintoosi, M.H.Y., M. Fathy and R. Monsefi, 2006. Using pattern matching for tiling and packing problems. Eur. J. Operat. Res., 183 (3): 950-960. DOI: 10.1016/j.ejor.2006.02.029. http://www.sciencedirect.com/science?_ob=MImg&_imagekey=B6VCT-4M4TNPB-1W&_cdi=5963&_user=10&_orig=search&_coverDate=12%2F16%2F2007&_sk=998169996&view=c&wchp=dGLzVtb-zSkzV&md5=993a641b3e9decedae6415bc6080795e&ie=/sdarticle.pdf.

Boyer, R. and J. Moore, 1977. A fast string searching algorithm. Commun. ACM, 20 (10): 761-772. DOI: 359842.359859. http://portal.acm.org/citation.cfm?doid=359842.359859.

Charras, C. and T. Lecroq, 2004. Handbook of Exact String-Matching Algorithms. 1st Edn. King's College London Publications, pp: 19-24. ISBN: 978-0-7546-6498-7. www-igm.univ-mlv.fr/~lecroq/string/string.pdf.

Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski and W. Rytter, 1994. Speeding up two string matching algorithms. Algorithmica, 12 (4-5): 247-267. DOI: 10.1007/BF01185427. http://www.springerlink.com/content/p783wl7323874635.

Crochemore, M.C.H. and T. Lecroq, 2003. A unifying look at the apostolico-giancarlo string-matching algorithm. J. Disc. Alg., 1: 37-52. DOI: 10.1016/S1570-8667(03)00005-4. http://www-igm.univ-mlv.fr/~lecroq/articles/jda1.pdf.

Danvy, O. and H. Rohde, 2006. On obtaining the boyer-moore string-matching algorithm by partial evaluation. J. Inf. Process. Lett., 99: 158-162. DOI: 10.1016/j.ipl.2006.04.001.http://www.brics.dk/RS/05/14/BRICS-RS-05-14.pdf.

Franek, F., C. Jennings and W.F. Smyth, 2006. A simple fast hybrid pattern-matching algorithm. J. Disc. Alg., 5: 682-695.DOI:10.1016/j.jda.2006.11.004.http://www.sfu.ca/~cjenning/papers/franek_jennings_smyth_jda06.pdf.

Karp, R. and M. Rabin, 1987. Efficient randomized pattern-matching algorithms. IBM J. Res. Develop., 31 (2): 249-260. DOI: 10.1145/359842.359859. http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf.

Kim, S. and Y. Kim, 1999. A fast multiple string-pattern matching algorithm. In Proceedings of the 17th AoM/IAoM Int. Conf. Comput. Sci., San Diego, CA, 17 (1): 44-49. Aug 6-8. DOI: 10.1.1.36.8337. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.8337&rep=rep1&type=pdf.

Knuth, D., J. Morris and V. Pratt, 1977. Fast pattern matching in strings. SIAM J. Comput., 6 (2): 323-350. DOI: 10.1137/0206024. http://locus.siam.org/fulltext/SICOMP/volume-06/0206024.pdf.

Lecroq, T., 2007. Fast exact string matching algorithms. J. Inf. Process. Lett., 102: 229-235. DOI: 10.1016/j.ipl.2007.01.002. http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V0F-4MX56KX-1&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=f69c8886c01360138bc0881b88c2ed00.

Navarro, G. and K. Fredriksson, 2004. Average complexity of exact and approximate multiple string matching. J. Theor. Comput. Sci., 321: 283-290. DOI: 10.1016/j.tcs.2004.03.058. http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V1G-4C477DB-7&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=5249793e461f0dd0fdd94e0c82177f2a.

Sheu, T.F., N.F. Huang and H.P Lee, 2008. Hierarchical multi-pattern matching algorithm for network content inspection. J. Inform. Sci., 178: 2880-2898. DOI: 10.1016/j.ins.2008.03.006. http://portal.acm.org/citation.cfm?id=1374858.1375197.

Watson, B., 2002. A new regular grammar pattern matching algorithm. J. Theor. Comput. Sci., 299: 509-521. DOI: 10.1016/S0304-3975(02)00532-7. http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V1G-47P1XCF-2&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=47696416d68adc0ca515f5872e6d132c.

Watson, B. and R. Watson, 2003. A Boyer-Moore-style algorithm for regular expression pattern matching. J. Sci. Comput. Prog., 48: 99-117. DOI: 10.1016/S0167-6423(03)00013-3. http://portal.acm.org/citation.cfm?id=947951.

Wu, Y.C., J.C. Yang and Y.S. Lee, 2007. A weighted string pattern matching-based passage ranking algorithm for video question answering. J. Expert Syst. Applic., 34: 2588-2600. DOI: 10.1016/j.eswa.2007.04.008. http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V03-4NGRRSN-3&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=877e1320c56a662a53087745cf0fb289.