New Enhanced Exact String Searching Algorithm

Mahmoud Moh'd Mhashi ❊ *& Mohammed Alwakeel* ❊❊
College of Computers and Information Technology
University of Tabuk
Kingdom of Saudi Arabia
Tabuk, P. O. Box 1458, Tabuk, 71431

Abstract
Exact string Searching is one of the most important problems that had been investigated by many studies ranging from finding the shortest common super string in DNA sequencing to searching for occurrences of a pattern occurs in text editors. In this paper, a new Enhanced Checking and Skipping Algorithm (ECSA) is introduced. The new algorithm enhance the classical string searching algorithms by converting the character-comparison into character-access, by using the condition type character-access rather than the number-comparison, and by starting the comparison at the latest mismatch in the previous checking, which in turn increases the probability of finding the mismatch faster if there is any. A computer program is developed to compare the performance of the introduced algorithm against the conventional Naïve (brute force) and Boyer-Moore-Horsepool (BMH) algorithms. The results of the experiment show that the performance of the enhanced algorithm is outperform the performance of the introduced algorithms.

Keywords:
String-searching, pattern matching, checking and skipping, Condition type, and multiple references.

1. Introduction

The string searching algorithm is so fundamental that most computer programs use it in one form or another, In fact, exact string Searching is one of the most important problems that had been investigated by many studies ranging from finding the shortest common super string in DNA sequencing [1] to searching for occurrences of a pattern occurs in text editors. In general, string searching algorithms deal with searching the occurrences of a string (the pattern) of size m in a string (the text) of size n (where $n \geq m$) [2-4]. The string searching algorithm, in general, may be represented as follow:

```
Algorithm String-Searching
        // Find all occurrences of Pat[0,m-1] in Text[0,n-1]
{
        Preprocessing phase;
        Search phase;
        Align Pat[0] with Text[0];
        While (end of Text is not reached)
        {
            Checking step;
            Candidate Checking phase;
            Detailed comparison phase;
            If an occurrence of Pat has been found then
                Report occurrence;
            Endif
            Skipping step;
                Move forward;
        }
}
```

In literature, many exact string searching and pattern matching algorithms were introduced and their performance were investigate against classical exact string searching algorithm such as Naïve (brute force) algorithm and Boyer-Moore-Horsepool (BMH) algorithm, some of these algorithms preprocess both the text and the pattern [5] while others need only to preprocess the pattern [6-9]. In all cases, as shown in the general representation of searching algorithm above, the exact string searching problem consists of two major steps: checking and skipping, the checking step itself consists of two phases:

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

Some characters of the candidate string must be selected carefully in order to avoid the problem of repeated examination of each character of text when patterns are partially matched, intuitively, the fewer the number of character comparisons in the checking step the better the algorithm is. There are different algorithms that check in different ways if the characters in the text match with the corresponding characters in the pattern [10-17]. After the checking step, the skipping step shifts the pattern to the right to determine the next position in the text where the substring text can possibly match with the pattern. The reference character is a character in the text chosen as the basis for the shift according to the shift table. In literature, the searching algorithms may use one reference character or use two reference characters, where the references might be static or dynamic [18-20]. In addition, some of the algorithms focus on the performance of the checking operation while others focus on the performance of the skipping operation [21]. The main goal of this paper is to develop a new enhanced checking and skipping algorithm (ECSA) that can be used for string searching and patter matching and outperform the algorithms introduced in literature. The rest of this paper is organized as follow: In section 2, the classical Naïve exact string searching algorithm is explained. Boyer-Moore-Horsepool (BMH) exact string searching algorithm is introduced in section3. In section 4 the developed enhanced checking and skipping algorithm (ECSA) is introduced. Numerical results and illustrations of the performance of the developed ECSA are presented in Section 5. Finally, some concluding remarks are made in Section 6.

2. The Naïve or (Brute Force) Algorithm

To illustrate this straightforward algorithm, let us assume that the target sequence is an array *Text*[*n*] of *n* characters and the pattern sequence is the array *Pat*[*m*] of *m* characters, then a Naïve approach to the problem would be:

```
  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*), where O(nm) is the number of comparisons performed by the algorithm to find all the occurrences of Pat with size m characters in the Text with size n characters. The worst case in this algorithm occurs when we get a match on each of the n Text characters and at each position we may need to perform m comparisons.

The following example illustrates this algorithm:

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

C	F	X	*Pat*[j]														
≠																	
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘C’) ≠ (*Text*[i] = ‘A’).

Skipping right one position produces Loop 2:

	C	F	X	*Pat*[j]													
	=	≠															
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘C’) == (*Text* [i] = ‘C’); (*Pat*[j + 1] = ‘F’) ≠ (*Text* [i + 1] = ‘C’).

Loop 3 is similar to loop 2 in the sense that it needs two character comparisons to find the mismatch.

Loop 4:

			C	F	X	*Pat*[j]											
			≠														
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = 'C') ≠ (*Text* [i] = 'D').

Loop 5 and loop 6 are similar to loop 4 in the sense that each one of them needs one character comparison to find the mismatch.

Loop 7:

						C	F	X	*Pat*[j]								
						=	=	=									
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

Each character in *Pat* matches the corresponding character in *Text*, hence, there is an occurrence at location 6 to 8.
Each one of the next four loops (loop 8 to loop 11) needs one character comparison to find the mismatch.

Loop 12:

											C	F	X	*Pat*[j]			
											=	=	≠				
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*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 (loop 13 and loop 14) needs only one character comparison to find the mismatch.

Loop 15:

														C	F	X	*Pat*[j]
														=	=	≠	
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*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, hence, 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–Horsepool (BMH) Algorithm

Horspool is one of many authors extended the BM algorithm [22]. Horspool proposed the Boyer–Moore–Horspool (BMH) [23] algorithm that 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 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 in *Text* (the character that corresponds 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:	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:

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	H	…	X	Y	Z
3	3	2	3	3	1	3	3	…	3	3	3

Any character that is not in the pattern will produce a shift distance equals to m, where m = 3. Any character in the pattern *Pat*[j] produces a shift distance equals to m - j, where j = 0, 1, & 2. Searching process:

Loop 1:

Comparison starts from right to left

C	F	X	*Pat*[j]														
		≠															
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘C’). Looking up in the skip table for character ‘C’ gives value 2.

Skipping right 2 positions produces Loop 2:

		C	F	X	*Pat*[j]												
				≠													
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘E’). Lookup in the skip table for character ‘E’ gives value 3.

Skipping right 3 positions produces Loop 3:

					C	F	X	*Pat*[j]									
							≠										
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘F’). A look up in the skip table for character ‘F’ gives value 1.

Skipping right one position produces Loop 4:

						C	F	X	*Pat*[j]								
						=	=	=									
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 6 to 8. Looking up in the skip table for character ‘X’ gives value 3.

Skipping right 3 positions produces Loop 5:

									C	F	X	*Pat*[j]					
											≠						
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘C’). A look up in the skip table for character ‘C’ gives value 2.

Skipping right 2 positions produces Loop 6:

											C	F	X	*Pat*[j]			
													≠				
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘B’). Looking up in the skip table for character ‘B’ gives value 3.

Skipping right 3 positions produces Loop 7:

														C	F	X	*Pat*[j]
																≠	
A	C	C	D	E	F	C	F	X	G	H	C	F	B	C	F	B	*Text*[i]

(*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘B’). Lookup in the skip table for character ‘B’ gives value 3. Skipping right 3 positions ends the searching process.

In this example, for a pattern of 3 characters and a *Text* of 17 characters, only 9 character comparisons have been performed, in addition to 25 number comparisons, to find all the occurrences of *Pat* in *Text* (i.e., total number of comparisons is 34).

```
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<<"\nOccurence at location "<<TextIx<<"
to location "<< TextIx + PatLength –1;
        TextIx += skip_table [c];
    }
}
```

4. Enhanced Checking and Skipping Algorithm (ECSA)

As we mentioned earlier, a string matching algorithm is a succession of checking and skipping, where the aim of a good algorithm is to minimize the work done during each checking and to maximize the length distance during the skipping. Most of the string matching algorithms preprocess the pattern before the search phase to help the algorithm to maximize the length of the skips, the preprocessing phase in this new ECSA algorithm helps in both increases the performance of the checking step by converting some of the character-comparison into character-access and maximizes the length of the skips. At each attempt during the checking steps the ECSA algorithm compares the character at *last_mismatch* (the character that causes the mismatch in the previous checking step) with the corresponding character in *Text*; if they match then it compares the first character of *Pat* with the corresponding character in *Text*, finally if they match the ECSA compares the other characters from right to left including the character at *last_mismatch* (because rather the cost is high) and excluding the first character of *Pat*. The enhanced performance of this algorithm during the checking step is due to the fact that the mismatch is detected quickly by starting the comparison, after each shift, at the location *last_mismatch* (see line 17 in ECSA algorithm), if there is a match then the comparison goes from right to left, including the compared character at *last_mismatch*. The idea here is that the mismatched character must be given a high priority in the next checking operation. After a number of checking steps, this leads to start the comparison at the least frequent character without counting the frequency of each character in the text. Another enhancement is due to several improvements in the process and in the programming technique itself, including converting a number-comparison and a character-comparison into a character-access (such as converting condition of type if(*j* < *n*) into a condition of type if(*j*), These improvements may be summarized in the following points:

1) The following style of for-statement:
   for (int j = 0; j < n; j++) {
   is changed into the following style:
   for (int j = n; j ; j--) {
   In another words, the number comparison of condition type "if(j < n)" is changed into a character access of condition type "if(j)".
2) Converting the character-comparison into character-access: This conversion is explained using the following example; assume that we have the following *Pat* and *Text*.

	0	1	2	3	4
Text:	**A**	**C**	**F**	**X**	**G**
Pat:	**C**	**F**	**X**		

To compare the character 'C' in *Pat* with the character 'A' in *Text* at location zero, most programmers normally write the statement **if(*Text*[*i*] == *Pat*[*j*])**, where *i* = *j* = 0. To convert this character-comparison into a character-access, a new array must be declared with alphabet size and initialized by zero, as shown in line 12 in ECSA (**int *infix*[ASIZE] = {0};**). Performing line 14 in ECSA *infix*[*Pat*[0]] = *infix*['C'] = 1 sets the location 'C' in the array infix by one. So if *TextIx* = 0 and *Text*[0] = 'A', then executing the character-access at line 19, if(*infix*[*Text*[*TextIx*]]), which is equivalent to the condition if(*infix*['A']) = 0, produces false result. However, if the character at location zero in *Text* is the character 'C', then line 19 if(*infix*[*Text*[*TextIx*]]) = if(*infix*[*Text*[0]]) = if(*infix*['C']) = 1, produces true result. Hence, the condition if(*Text*[*i*] == *Pat*[*j*]) of type character-comparison is replaced by the condition if(*infix*[*Text*[*TextIx*]]) of type character-access. The condition at line 19 serves two goals, the first is converting the character-comparison to character-access at *Pat*[0], and the second is Checking the character at location *Pat*[0] in advance before entering the for-statement at line 21. This occurs because the value of index *PatIx* becomes zero at the end of the loop at line 21 and the control will exit the loop without checking the character at location *Pat*[0].

For the skipping step, ECSA 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* and the reference character *ref* always points to the character that corresponds to the last character in *Pat* (i.e. *ref* = *TextIx* – 1). Now let *ref1* = *TextIx*, then the reference character *ref2* can be calculated from *ref* or *ref*1, where *ref*2 can be found as "*ref2* = *TextIx* + *m* - 1" or "*ref2* = *TextIx* + *m*" depending on the existence of *ref* or *ref1* in *Pat*, where *ref2* = *TextIx* + *m* - 1 during the checking step if the character at *ref* doesn't exist in *Pat*, or *ref2* = *TextIx* + *m* after the checking step if the character at *ref1* doesn't exist in *Pat*. The above formulas may be illustrated in the following example:

Example 1:

Text:	A	B	C	D	E	F	G	H
Pat:	E	F	G					

In the above *Text* and *Pat* assume that the pointers *TextIx* and *ref1* point to the character 'D', the pointer *ref* points to the character 'C', and *m* = 3, in this case since the character 'C' at *ref* doesn't exist in *Pat*, then *ref2* = *TextIx* + *m* - 1 = 3 + 3 - 1 = 5. So, start counting from zero, *ref2* points to the character 'F' in *Text*. 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 this case the checking step will be continued to find out the occurrence of *Pat* in *Text*, then the occurrence of character 'D' at *ref1* must be examined whether *Pat* occurs in *Text* or not, and since 'D' doesn't occur in *Pat* then *ref2* = *TextIx* + *m* = 3 + 3 = 6, hence, *ref2* points to the character 'G' in *Text*.

In addition to that, ECSA pre-processes the pattern to produce two different arrays, namely *skip* and *pos*, each array has a length equals to the alphabet size. The *skip* array is used when the reference character *ref1* exists in *Pat*, it expresses how much the pattern is to be shifted forward after the checking step. While the *pos* array defines where each one of the different reference characters *ref1*, *ref2*, *ref_ref1, or ref_ref2* is located in *Pat*, if any one of them exists in *Pat*, where the two dynamic pointers *ref_ref1* and *ref_ref2* can be calculated from the two static pointers *ref1* and *ref2* respectively. In particular, the dynamic pointer *ref_ref1* which is calculated at the skipping step if *ref1* occurs in *Pat* can be found as *ref_ref1* = *ref1* + *m* - *pt*, where m is the *Pat* length and *pt* is the location of *ref1* in *Pat* (i.e., *pt* = *pos*[*Text*[*ref1*]]), and the dynamic pointer *ref_ref2* which is calculated and used only during checking step if *ref* doesn't occur in *Pat* or after the checking step if *ref1* doesn't occur in *Pat* can be found as *ref_ref2* = *ref2* + *m* - *pt1*, where *pt1* determines where *ref2* is located in *Pat*. The above formulas can be illustrated in the following examples:

Example 2 : Calculating *ref_ref1*

	0	1	2	3	4	5	6	7
Text:	A	B	G	E	E	F	G	H
Pat:	E	F	G					

Assuming the checking step is performed on the above *Text* and *Pat*, then the reference character will be character 'E' at location 3 in *Text*, and since 'E' at *ref1* occurs at location 1 in *Pat*, then *pt* = *pos*[*Text*[*ref1*]] = *pos*['E'] = 1 (note that for the case of counting position the counting starts from 1 not zero), and *ref_ref1* may be found as *ref_ref1* = *ref1* + *m* - *pt* = 3 + 3 - 1 = 5 (i.e. *ref_ref1* points to the character 'F' in *Text* at location 5).

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

	0	1	2	3	4	5	6	7	8	9	10
Text:	A	B	C	D	E	F	G	H	F	E	G
Pat:	F	E	G								

Since *ref* doesn't occur in *Pat*, then *ref2=5* (calculated above in Example 1). Consequently, *ref_ref2* = *ref2* + *m* - *pt1*, where *m* = 3 and *pt1* = *pos*[*Text*[*ref2*]]) = *pos*[*Text*['F']]) = 1. Hence, *ref_ref2* may be found as *ref_ref2* = 5 + 3 - 1 = 7. In such a case, the alignment will be with 'H' at location 7 in *Text*, however, the letter 'H" doesn't occur in *Pat*, so, *TextIx* will move forward 7 locations to point to the character 'G' at position 10.

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

Let *Text* and *Pat* as shown below:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Text:	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**E**	**D**	**C**	**M**
Pat:	**E**	**D**	**C**											

Since the character at *ref* occurs in *Pat* and there is a mismatch then *ref1* will be considered as a basis for calculating *ref2*, and since *ref1* doesn't occur in *Pat* then *ref2 = TextIx + m* = 6, and *pt1 = pos*[*Text*[*ref2*]]) = *pos*[*Text*['I']]) = 0. Hence, *ref_ref2* may be found as *ref_ref2 = ref2 + m - pt1* = 6 + 3 - 0 = 9. In addition, the *TextIx* pointer will be shifted forward 10 positions to align with the letter *Text*[*ref_ref2*] = *Text*['L'] = 3*m*+1 positions, which is the maximum shift distance that this algorithm can skip with only two–character-access. As a result, the pointer *TextIx* will point to the letter 'M' at position *Text*[13], and the result will be as follows:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Text:	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**E**	**D**	**C**	**M**
Pat:											**E**	**D**	**C**	

Based on the above discussion and examples we may conclude that the ECSA algorithm is designed to scan the characters of both the text and the pattern from right to left. At each attempt it first compares the character at *last_mismatch* with the corresponding character in *Text*; if they match then it compares the first character of *Pat* with the corresponding characters in *Text*, and then if they match ECSA compares the other characters from right to left including the character at *last_mismatch* and excluding the first character of *Pat*. Whether there is an occurrence of *Pat* in *Text* or not, the existence of the character at *ref* in *Pat* will be checked first, so 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, hence, there are two cases:
   1.1) The character at *ref1* doesn't exist in *Pat*. Then *ref2* and *ref_ref2* will be calculated. Next, the pointer *TextIx* will be moved forward to align with the character at *ref_ref2*.
   1.2) The character at *ref1* exists in *Pat*. Then *ref_ref1* will be calculated. Then, the pointer *TextIx* will be moved forward to align with the character at *ref_ref1*.
2) **The character at *ref* doesn't exist in *Pat*:**
   In this case *ref2* and *ref_ref2* will be calculated according to the pointer *ref*, then the pointer *TextIx* will be moved forward to align with the character at *ref_ref2*.

Based on that, The ECSA algorithm that reflects the above ideas is as follows:

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

9) void ECSA(char *Pat, int PatLength, char *Text, int
TextLength, int *pos, int *skip)
{
10)      int TextIx, PatIx, last_mismatch, z;
11)      int pt, pt1, ref, ref1, ref_ref1,  ref2, ref_ref2;
12)      int infix[ASIZE] = {0};
13)      /* Update infix table according to the first
character in Pat */
14)      infix[Pat[0]] = 1;       last_mismatch =0;
TextIx = PatLength;
15)      // Start Searching operation
```

```
16)     while(TextIx<=TextLength+1)
      {           // Checking step: Check first the occurrence
of the character at the previous mismatch.
17)         if(Text[TextIx - PatLength + last_mismatch] ==
Pat[last_mismatch])
18)             // Check now the character in Text that
corresponds the first charcter in Pat;
19)             if(infix[Text[TextIx - PatLength]])
20)              {  // Check the occurrence of Pat in Text
from right to left excluding first character
21)               for( z = 0, PatIx = PatLength - 1; PatIx;
PatIx--  )
22)                   if(Text[TextIx - ++z] != Pat[PatIx])
                      {
23)                             last_mismatch = PatIx;
24)                             goto next;
                      }
25)               cout<<"\nAn occurrence at location
"<<TextIx-PatLength <<" to "<<TextIx - 1<<endl;
              }
26)          // Start the skipping part
27)           next:
28)          ref = TextIx - 1;             ref1 = TextIx;
29)         if ( !pos[Text[ref]] )
         {
30)         ref2 = ref + PatLength;      pt1 =
pos[Text[ref2]];      ref_ref2 = ref2 + PatLength - pt1;
31)         TextIx  +=  3 * PatLength - pt1 -
pos[Text[ref_ref2]];
          }
32)          else
          {
33)           pt = pos[Text[ref1]];
34)             if( !pt)
                {
35)               ref2 = TextIx + PatLength;    pt1 =
pos[Text[ref2]];      ref_ref2 = ref2 + PatLength - pt1;
36)              TextIx  +=  3 * PatLength + 1 - pt1 -
pos[Text[ref_ref2]];
                 }
 37)                else
                 {
38)                 ref_ref1 = ref1 + PatLength - pt;

39)                 TextIx += skip[Text[ref_ref1]] - pt + 1;
                 }
 40)           }  // This is the end of else of main if statement
       }
41)      return;
}
```

5. Illustration and Discussion

The three algorithms Naïve, BMH, and our new algorithm ECSA were implemented and compared on English text with a size of more than two mega characters and contains 85 different characters. The algorithms executed using Intel(R) Pentium(R) 4 PC with CPU speed 2.40GHz, 246MB RAM, and Windows XP professional operating system, and a program was designed in C++ to select randomly 3000 patterns with ranges from 4 to 94 characters, and the average number of occurrences ranges from 1 to 1177. The cost of the searching process to find all the occurrences of the different patterns in each group in *Text* is measured by finding the search clock time, where the total clock time includes the preprocessing clock time of patterns and the searching clock time.

Table 1 : The clock time (seconds) required by Naïve, BMH, and our new algorithm ECSA to find the occurrences of each group of patterns

Group No.	**Pattern Length (in characters)**	**Clock Time in seconds**			
		Naïve	**BMH**	**ECSA**	**Improvements of ECSA vs. Naïve BMH**
1	**4**	**8.562**	**3.922**	**1.341**	**84.34% 65.81%**
2	**14**	**8.516**	**1.578**	**0.812**	**90.47% 48.54%**
3	**24**	**8.468**	**1.140**	**0.625**	**92.62% 45.18%**
4	**34**	**8.516**	**0.969**	**0.547**	**93.58% 43.55%**

5	**44**	**8.531**	**0.844**	**0.485**	**94.32%** **42.54%**
6	**54**	**8.422**	**0.797**	**0.468**	**94.44%** **41.28%**
7	**64**	**8.516**	**0.750**	**0.485**	**94.31%** **35.33%**
8	**74**	**8.297**	**0.703**	**0.437**	**94.73%** **37.84%**
9	**84**	**8.593**	**0.703**	**0.438**	**94.90%** **37.70%**
10	**94**	**8.454**	**0.687**	**0.437**	**94.83%** **36.39%**
Total time		**84.875**	**12.093**	**6.075**	**92.84%** **49.76%**

Table (1) shows the clock time required to find each group of patterns, and as we can see from the table, ECSA reduces the clock time required by Naïve and BMH algorithms by 84.34% to 94.9% and 35.33% to 65.81% respectively.
In Fig. 1, the clock time required by the three algorithms: Naïve, BMH, and ECSA to find all the occurrences of *Pat* in *Text* are shown, and from the figure, it is clear that the new developed algorithm outperform Naïve and BMH algorithm.

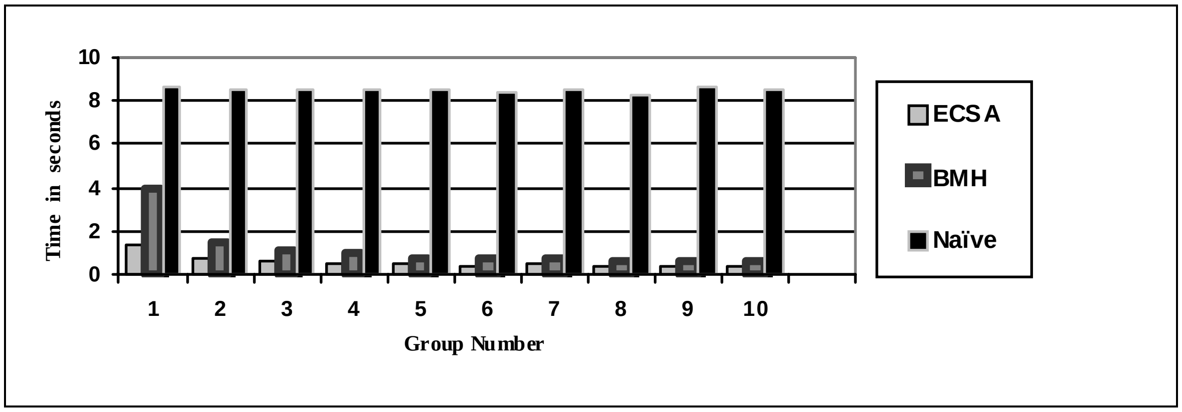

Fig.1: The clock time required by Naïve, BMH, and ECSA algorithms to find all the occurrences of patterns in each group

Finally, the analysis of the three algorithms shows that the enhanced performance of ECSA is due to several factors, including:

1) Converting character-comparison into character-access: The ECSA converts the first condition of *Pat* from character-comparison (*Text*[*TextIx*] == *Pat*[*PatIx*]) into character access (if(*infix*[*Text*[*TextIx*]]) with a reasonable overhead cost.
2) Character-access vs. number-comparison: The ECSA uses the condition type character-access (if(*i*)) (this type of condition needs 40% less time to be executed than the time needed by any other type of conditions) in the main loops rather than using the number-comparison (if (*TextIx* < *TextLen*)).
3) The starting point of checking: The ECSA algorithm starts the comparison at the latest mismatch in the previous checking. This increases the probability of finding the mismatch faster if there is a mismatch.

6. Conclusions

A new exact string searching algorithm ECSA was developed, and its performance was compared with two classical algorithms, namely, Naïve, and BMH. The developed algorithm increases the performance of both checking phase and skipping phases needed for the string searching process. In the checking phase, ECSA algorithm uses both the character-access and the character-comparison tests at the checking step while most of the current existing algorithms use only the character-comparison. On the other hand, in the skipping phase ECSA focuses on increasing the shift distance. The search clock time criteria was used in an experiment to compare the performance of ECSA against Naïve, and BMH. From the results we can see that:

1) Using ECSA improved the clock time required by Naïve and BMH by 84.34% to 94.90% and 35.33% to 65.81% respectively.

2) Decreasing the pattern length decreases the system performance.

3) In Naïve algorithm, the shift distance is only one position, and in BMH algorithm it ranges from 1 to *m* positions, while in ECSA it ranges from 1 to (3*m* + 1) positions. Obviously, increasing the shift distance has a major effect on reducing the different types of comparisons and in turn increasing the system performance.

4) In Naïve algorithm, there is no reference character, and in BMH there is only one reference character, while ECSA has five reference characters three of them are static and the other two are dynamic. Increasing the number of reference characters resulted into an increase in the shifted distance.

References

[1] Ukkonen E., "A linear–time algorithm for finding approximate shortest common superstring", Algorithmica, 1990; 5:313–323.

[2] Stephen G., "String Searching Algorithms", World Scientific, Singapore, 1994.

[3] Apostolico, "A, Galil Z. Pattern Matching Algorithms", Oxford University Press, 1997.

[4] Gusfield D., "Algorithms on strings, trees and sequences", Cambridge University Press, Cambridge, 1997.

[5] Fenwick P., "Fast string matching for multiple searches", Software–Practice and Experience, 2001, 31(9):815–833.

[6] Liu Z, Du X,and Ishii N., "An improved adaptive string searching algorithm", Software Practice and Experience, 1988, 28(2):191–198.

[7] Sunday D., "A very fast substring search algorithm", Communications of the ACM, 1990, 33(8):132–142.

[8] Raita T., "Tuning the Boyer-Moore-Horspool String Searching Algorithm", Software Practice and Experience, 1992, 22 (10):879-844.

[9] Bruce W., Watson, E., "A Boyer-Moore-style Algorithm for Regular Expression Pattern Matching", Science of Computer Programming, 2003, 48: 99-117.

[10] Ager M. S., Danvy O., and Rohde H. K., "Fast partial evaluation of pattern matching in strings", ACM/SIGPLAN Workshop Partial Evaluation and Semantic-Based Program Manipulation, San Diego, California, USA, pp. 3 – 9, 2003.W.-K. Chen, Linear Networks and Systems (Book style)., Belmont, CA: Wadsworth, 1993, pp. 123–135.

[11] Fredriksson and Grabowski S., "Practical and Optimal String Matching", Proceedings of SPIRE'2005, Lecture Notes in Computer Science 3772, 2005, pp. 374-385, Springer Verlag.

[12] Hernandez, and Rosenblueth D., "Disjunctive partial deduction of a right-to-left string-matching algorithm", Information Processing Letters, 2003, 87: 235–241.

[13] Apostolico A., and R.Giancarlo R., "The Boyer-Moore-Galil string searching strategies revisited", SIAM J. Comput., 1986, 15(1): 98-105.

[14] Crochemore, "Transducers and repetitions", Theoret. Comput. Sci., 1986; 45: 63-86.

[15] Crochemore M., and Perrin D., "Two-way string-matching", J. ACM, 1991, 38: 651-675.

[16] Galil Z., and Giancarlo R., "On the exact complexity of string matching: upper bounds", SIAM J. Comput., 1992, 21: 407-437.

[17] Smith P. D., "Experiments with a very fast substring search algorithm", SP&E, 1991, 21(10): 1065-1074.

[18] Smith P., "On Tuning the Boyer-Moore-Horspool String Searching Algorithm", Short Communication, Software Practice and Experience, 1994, 24(4):435-436.

[19] Mhashi M., "A Fast String Matching Algorithm using Double-Length Skip Distances", Dirasat Journal, University of Jordan, Jordan, 2003, 30(1):84-92.

[20] Mhashi M., "The Effect of Multiple Reference Characters on Detecting Matches in String Searching Algorithms", Software Practice and Experience, 2005, 35(13): 1299 - 1315.

[21] Mhashi, M., "The Performance of the Character-Access On the Checking Phase in String Searching Algorithms", Transactions on Enformatica, Systems Sciences and Engineering, 2005, 9: 38 –43.

[22] Boyer RS., and Moore JS., "A fast string searching algorithm", Communications of the ACM, 1977, 20(10):762–772.

[23] Horspool RN., "Practical Fast Searching in Strings", Software Practice and Experience, 1980, 10(6):501–506.

Mahmoud M. Mhashi Received his M.S. degree in computer Science from University of Colorado at Boulder, in 1988, and the Ph.D. degree in computer Science from University of Liverpool University of Liverpool, in 1991. He is currently a Professor at University of Tabuk. His current research interests include string searching algorithms.

Mohammed M. Alwakeel was born in Tabouk, Saudi Arabia, in 1970. He received the B.S. degree in computer engineering and the M.S. degree in electrical engineering, both from King Saud University, Riyadh, Saudi Arabia, in 1993 and 1998, respectively, and the Ph.D. degree in electrical engineering from Florida Atlantic University, Boca Raton, in 2005. From 1994 to 1998, he was a Communications Network Manager at The National Information Center, Saudi Arabia. From 1999 to 2001, he was with King Abdulaziz University, Saudi Arabia, as a Lecturer and was the Vice Dean of Tabouk Community College. He is currently the Dean of computers and Information Technology college at University of Tabuk. His current research interests include teletraffic analysis, cellular systems, image recognition, and string searching algorithms.