International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:9, 2007

The Performance of the Character-Access on the Checking Phase in String Searching Algorithms

Mahmoud M. Mhashi

***Abstract*—**A new algorithm called Character-Comparison to Character-Access (CCCA) is developed to test the effect of both: 1) converting character-comparison and number-comparison into character-access and 2) the starting point of checking on the performance of the checking operation in string searching. An experiment is performed; the results are compared with five algorithms, namely, Naive, BM, Inf_Suf_Pref, Raita, and Circle. With the CCCA algorithm, the results suggest that the evaluation criteria of the average number of comparisons are improved up to 74.0%. Furthermore, the results suggest that the clock time required by the other algorithms is improved in range from 28% to 68% by the new CCCA algorithm

***Keywords*—**Pattern matching, string searching, character-comparison, character-access, and checking.

I. Introduction

THE problem of exact-match string searching is addressed. The problem is to search all occurrences of the pattern $P[0\ldots m\text{-}1]$ from the text $T[0\ldots n\text{-}1]$, where m is the pattern length and n is the text length. The pattern and the text are both strings built on the same alphabet.

The checking step consists of two phases:

1) A search along the text for a reasonable candidate string, and
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. After the checking step, whether there is a mismatch or a complete match of the whole pattern, the algorithm shifts to the next position. There are different algorithms that check in different ways if the characters in *Text* match with the corresponding characters in *Pat*. Some of these algorithms scan the characters of the text:

1) From left to right [1]
2) From right to left [2-3] and by using, the smallest suffix automation of the reverse pattern [4-6]
3) From the two directions [7-9]
4) By using a static and dynamic statistics to get a good comparison order [10-11]
5) By using a good comparison order without using any statistics [12]

Most previous work focused on the improvement of jumping distance in the skipping step [13-17]. In this paper, the focus is on increasing the performance of the checking step. This can be done by reducing the number of character-comparison and by converting the character-comparison and number-comparison into character-access.

Manuscript received September 21, 2005. This work was supported in part by Mu'tah University.

Mahmoud M. Mhashi is with Mu'tah University, Mu'tah, 61710, Jordan (phone: +962795116066; e-mail: mhashi@mutah.edu.jo).

II. Checking Component in String Searching Algorithms

A. Forward Checking

Let's say the target sequence is an array *Text*[*n*] of *n* characters (i.e., *n* is the text length) and the pattern sequence is the array *Pat*[*m*] of *m* characters (i.e., *m* is the pattern length). A naive approach to the problem would be:

```
void Naive((char *Pat, long int PatLen, char *Text, long int
TextLen) {
   long int TextIx, PatIx;
   for (TextIx = 0; TextIx <= TextLen – PatLen + 1;
TextIx++) {
       PatIx = 0;
      while (Text[TextIx + PatIx] == Pat[PatIx++]) {
            if (PatIx == PatLen - 1) {
     cout <<"\n Occurence at location "<<TextIx<<"to
     location "<< TextIx + PatLen - 1 << endl;
               break;
            }
        }
    }
   return;
}
```

In the outer loop, *Text* is searched for occurrences of the first character in *Pat*. In the inner loop, a detailed comparison of the candidate string is made against *Pat* to verify the potential match. The algorithm has a worst case time of $O(nm)$, because in the worst case we may get a match on each of the *n Text* characters and at each position we may proceed to completion *m* comparisons. Assume that the next following *Text* and *Pat* are given. Then, the Comparison (loop 0) starts from left to right (*Pat*[j] = 'C') ≠ (*Text*[i] = 'A'). Skipping right one position produces Loop 1. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. Executing loop 2, we get (*Pat*[j]

= ‘C’) ≠ (*Text*[i] = ‘F’). Moving one position ends the searching process. Thus, to find all the occurrences of *Pat* in *Text*, 5 character-comparisons are needed, in addition to 4 number-comparisons.

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

B. Reverse Checking: Boyer-Moore Algorithm

The Boyer-Moore algorithm is one example of the reverse string-searching algorithm. The algorithm scans the characters of the pattern from right to left beginning with the most right character. Searching phase needs $O(mn)$ time complexity; $3n$ text character comparisons in the worst case when searching for a non-periodic pattern; $O(n / m)$ best performance.

```
void  BM(char *Pat, long int PatLen, char *Text, long int
TextLen) {
   long int TextIx, PatIx;
   for (TextIx = 0; TextIx <=TextLen – PatLen +1; TextIx++)
{
        PatIx = PatLen - 1;
        while (Text[TextIx + PatIx] == Pat[PatIx--]) {
           if (PatIx < 0) {
cout << "\nOccurence at location "<<TextIx<< " to   location "
<<TextIx+PatLen – 1  << endl;
            break;
        }
        }
   }
   return;
}
```

Example:

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

Searching process: Loop 0:

Comparison starts from right to left (*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘F’). Skipping right one position performs Loop 1. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. Executing loop 2, we get (*Pat*[j] = ‘X’) ≠ (*Text*[i] = ‘G’). Moving one position ends the searching process. Therefore, to find all the occurrences of *Pat* in *Text*, 5 character-comparisons are needed, in addition to 4 number-comparisons.

C. Infix-Suffix-Prefix Checking

Many words have the same prefix, such as “computer”, “computation”, and “computerized”. Also, many words have the same suffix, such as “absorbability”, “acceptability”, and “possibility” [Baalbaki, 1992]. Additionally, sentences might have the same prefix, such as “Computer systems support collaborative work”, and “Computer systems support discussion systems”. Also, sentences might have the same suffix, such as “Case studies for string searching algorithms”, and “fast string searching algorithms”.

It can be noticed from the above examples that there is a strong dependency between the prefixes and suffixes of the words or sentences. Such a dependency is the weakest at the middle. This suggests that it is not profitable to compare the pattern symbols strictly from left to right or from right to left. Thus it might be profitable to compare the pattern symbols from the middle to the boundaries of the pattern. This is because the probability of finding the mismatch at the middle is higher than it is at the boundaries. Thus, in the Infix-Suffix-Prefix algorithm, the comparison will start at the middle part, then the suffix part followed by the prefix part.

```
void Inf  Suf  Pref(char *Pat, long int PatLen, char *Text, long
int TextLen) {
   long int TextIx, PatIx, Pref,  Pref = PatLen /3;
for (TextIx = 0; TextIx <= TextLen – PatLen+1; TextIx++) {
      for( PatIx = Pref; PatIx <PatLen; PatIx++)
         if(Text[TextIx + PatIx] != Pat[PatIx])  goto next;
      if (PatIx == PatLen ) {
         for(PatIx = 0; PatIx < Pref; PatIx++)
            if (Text[TextIx + PatIx] != Pat[PatIx])  goto next;
   cout<<"Occurence at"<<TextIx<<"to"<<TextIx + PatLen -
1 << endl;
      }
      next: continue;
   }
   return;
}
```

Example:

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

Comparison (Loop 0) starts from the middle (infix part) to the boundaries (suffix followed by prefix) (*Pat*[j] = ‘F’) ≠ (*Text*[i] = ‘C’). Skipping right one position executes Loop 1. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. When loop 2 is executed, we get (*Pat*[j] = ‘F’) ≠ (*Text*[i] = ‘X’). Moving one position ends the searching process. So, to find all the occurrences of *Pat* in *Text*, 5 character-comparisons are needed, in addition to 4 number-comparisons.

D. Selected Characters: Raita’s Algorithm

Raita designed an algorithm so that at each attempt it first compares the last character of the pattern *Pat* with the rightmost character in *Text*: if they match, then it compares the first character of *Pat* with the leftmost character of *Text*; if they match, then it compares the middle character of *Pat* with the middle character in *Text*. Finally if they match, it compares the other characters from left to right excluding the first and the last characters in the pattern. It possibly compares again the middle character.

```
void Raita(char *Pat, long int PatLen, char *Text, long int TextLen) {
   long int TextIx, PatIx, mid, mid = PatLen/2;
   for (TextIx = 0;TextIx<TextLen - PatLen+1;TextIx++ ) {
       if(Text[TextIx + PatLen-1] == Pat[PatLen-1])
// Check last character first
       if(Text[TextIx] == Pat[0]) // Check the first character
   if(Text[TextIx + mid] == Pat[mid]) {
   // Check the middle character next
                for(PatIx = 1; PatIx<PatLen-1;  PatIx++  )
                if(Text[TextIx + PatIx] != Pat[PatIx])  goto next;
  cout<<"\nAn occurrence at location "<<TextIx <<"       to "<<TextIx+PatLen-1<<endl;
            }
        next: continue;
   }
   return;
}
```

Example:

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

Searching process (Loop 0) starts with the last character in *Pat* at (*Pat*[j] = 'X') ≠ (*Text*[i] = 'F'). Skipping right one position produces Loop 1. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. Four character comparisons are required to find this occurrence. Going to loop 2, we get (*Pat*[j] = 'X') ≠ (*Text*[i] = 'G'). Moving one position ends the searching process. Thus, to find all the occurrences of *Pat* in *Text*, 6 character comparisons are needed, in addition to 4 number-comparisons.

E. No Statistics Checking: Circle Algorithm

The *Cycle* algorithm is based on the idea that mismatched characters should be given a high priority in the next *checking* operation. In the *checking* step, there is no fixed comparison order. The *Cycle* algorithm treats the pattern as a cycle logically. At the beginning of search process, the algorithm applies the Naive principle (i.e. left to right). In each *checking* step, it always starts comparing the mismatched character in the last step. When the comparison successfully turns around in one *checking* step, a complete match is found. The following C code represents the *checking* step (More details in [12]).

```
void Circle((char *Pat, long int PatLen, char *Text, long int TextLen) {
   long int  joffset, TextI =joffset=PatLen,  PatIx=0, i, k = 0;
while(TextIx<TextLen+1) {
       i = TextIx – joffset ;
       if( Pat[PatIx] == Text[i] )
          for(k=2; k<=PatLen; k++) {
             if(++i==TextIx) {
                i=TextIx – PatLen ;     PatIx=0;
             }
             else PatIx++;
              if(Pat[PatIx] != Text[i])  break;
          }
       if (k > PatLen ) {
   cout << "\n Occurence at location " <<TextIx – PatLen  <<" to "<<TextIx – 1 <<;       k = 0;
     }
       joffset=TextIx – i ;       TextIx++;
  }  // End while
   return;
}
```

The variable *joffset* is used to compute the distance between *TextIx* from *i* before entering the checking step. The variable *i* is used to indicate the current substring. After the pattern is shifted to the next position and by using the *joffset*, the *TextIx* should be adjusted to align the *PatIx* since the pair of characters pointed by the *PatIx* and *TextIx* will be first compared (i.e., the mismatched character in the previous step).

Example:

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

Searching process: Loop0:

The naïve algorithm is applied first. Comparison starts with the first character in *Pat* at (*Pat*[*j*] = 'C') ≠ (*Text*[*i*] = 'A'). Skipping right one position executes Loop1. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. For this loop only, three character-comparisons and three number-comparisons are needed. Going to loop 2, Pat[0] is checked first because the previous mismatched occurred at that location. We get (*Pat*[j] = 'C') ≠ (*Text*[i] = 'F'). Moving one position ends the searching process. Therefore, to find all the occurrences of *Pat* in *Text*, 5 character-comparisons are needed, in addition to 6 number comparisons.

III. Character-Comparison to Character-Access (CCCA)

Let *Text*[0...*n*-1] and *Pat*[0...*m*-1] be arrays of characters. The array *Text* is the text and the array *Pat* is the pattern. The problem is to find all the exact occurrences of *Pat* in *Text*. The text and the pattern are both words built on the same characters. A string-matching algorithm is a succession of checking and skipping. 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 strings matching algorithms preprocess the pattern before the search phase. The work done during the preprocessing phase helps the algorithm to maximize the length of the skips. The preprocessing phase in this new CCCA algorithm helps in increasing the performance of the checking step by converting some of the character-comparison into character-access. The performance of this algorithm comes from two directions:

1) By detecting mismatch quickly, and
2) By converting a number-comparison and a character-comparison into a character-access (such as converting condition of type if(*index* < *n*) into a condition of type if(*index*)).

Regarding the first direction, at the beginning of the search, the first character will be compared first. If any mismatch is found, then that location will be stored in a variable called *last_mismatch* (see line 13 in CCCA algorithm). After the pattern is shifted to align a new substring, the comparison will start at location *last_mismatch* (see line 9 in CCCA algorithm). If there is a match, then the comparison go from left to right 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 rare character or at least frequency character without counting the frequency of each character in the text.

Regarding the second direction, the following improvements are made:

1) Programmers, normally, write the for-statement at line (8) in CCCA with the following style:

for(TextIx=0;TextIx<TextLen–PatLen+1; TextIx++) {

This for-statement is changed into the following style:

for (TextIx = TextLen - PatLen; TextIx; TextIx--) {

In other words, the number comparison of condition type " if(TextIx < TextLen – PatLen + 1)" is changed into a character access of condition type "if(TextIx)".

2) Again, the programmers write the for-statement at line (11) in CCCA with the following style:

for(PatIx = 0; PatIx < PatLen ; PatIx++)

This for-statement is changed into the following style:

for(PatIx = PatLen – 1; PatIx; PatIx--)

In the same way at line 8, the number comparison "if(TextIx < TextLen)" is changed into a character access "if(TextIx)".

3) Looking at lines (14) and (18) in CCCA algorithm, the statements "goto next" and "next: continue" are found. Programmers, normally, use the following style:

```
(13)                    last_mismatch = PatIx;
(14)                    break;
(15)                }
(16) if(PatIx == PatLen) cout<<"\nAn occurrence at location "<<TextIx <<" to "<<TextIx+PatLen–1;
(17)                }
```

In other words, programmers use break instead of "goto next", but they have to add a condition to test whether there is an occurrence or not (see line 16 above). Thus, using the new style reduces the number of conditions.

4) Converting the character-comparison into character-access: This conversion can be explained by the following example. Assume that we have the following *Pat* and *Text*.

	0	1	2	3	4
Text[i]	A	C	F	X	G
Pat[j]	C	F	X		

To compare the character 'C' in *Pat* with the character 'A' in *Text* at location zero, 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, such as line 4 in CCCA:

int infix[ALPHABET_SIZE] = {0};

Performing line 6 in CCCA **infix[Pat[0]] = infix['C'] = 1** sets the location 'C' in the array infix by one. Executing the character-access at line 10, **if(*infix*[*Text*[*TextIx*]]),** where **TextIx = 0** and **Text[0] = 'A'**. This condition is equivalent to the condition **if(*infix*['A']) = 0,** that produces false result (i.e., there is a mismatch). Assuming that the character at location zero in *Text* is the character 'C', then line 10 **if(*infix*[*Text*[*TextIx*]])=if(*infix*[*Text*[0]])= if(*infix*['C']) = 1,** produces true result (i.e., there is a match between the corresponding characters). So, 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 10 serves two things: 1) converting the character-comparison to character-access at *Pat*[0], and 2) Checking the character at location *Pat*[0] in advance before entering the for-statement at line 11. This occurs because the value of index *PatIx* becomes zero at the end of the loop at line 11 and the control will exit the loop without checking the character at location *Pat*[0].

```
(1)  void  CCCA(char *Pat, long int PatLen, char *Text, long int TextLen)
(2)  {
(3)   long int TextIx, PatIx, last_mismatch;
(4)   long int infix[ALPHABET_SIZE] = {0};
(5)   /* Update infix table according to the first character in Pat */
(6)   infix[Pat[0]] = 1;
(7)   last_mismatch =0;
(8)   for (TextIx = TextLen - PatLen; TextIx; TextIx-- ) {
(9)    if(Text[TextIx+last_mismatch]== Pat[last_mismatch])
(10)          if(infix[Text[TextIx]]) {
(11)             for(PatIx = PatLen – 1; PatIx; PatIx-- )
(12)                 if(Text[TextIx + PatIx] != Pat[PatIx]) {
(13)                     last_mismatch = PatIx;
(14)                     goto next;
(15)                 }
(16) cout<<"\nAn occurrence at location "<<TextIx <<" to "<<TextIx+PatLen–1<<endl;
(17)          }
(18)       next: continue;
(19)   }
(20)   return;
(21) }
```

Example:

	0	1	2	3	4	
Text[i]	A	C	F	X	G	Loop
Pat[j]	C	F	X			0
		C	F	X		1
			C	F	X	2

Searching process: Loop 0:

The Naïve algorithm is applied first. Comparison starts with the first character in *Pat* at (*Pat*[j] = 'C') ≠ (*Text*[i] = 'A'). Skipping right one position produces Loop 1. Because the mismatch occurred at location zero in the previous check, comparison starts with the first character in *Pat* at (*Pat*[j] = 'C') == (*Text*[i] = 'C'). There is a match between the two corresponding characters. The character 'C' in *Text* will be

compared again with the corresponding character 'C' in *Pat* through the character-access test **if(*infix*[*Text*[*TextIx*]]) = if(*infix*[*Text*[1]]) = if(*infix*['C']) = 1,** produces true result (i.e., there is a match). The character at location zero in *Pat* will be checked only twice, if the mismatch occurred at *Pat*[0] in the previous check and there is a match at the current check. Otherwise it will be checked once. Each character in *Pat* matches the corresponding character in *Text*. There is an occurrence at location 1 to 3. For this loop only, three character-comparisons, one character-access, and one number-comparison are needed. Going to loop 2, Pat[0] is checked first because the previous mismatched occurred at that location. We get (*Pat*[j] = 'C') ≠ (*Text*[i] = 'F'). Moving one position ends the searching process. Thus, to find all the occurrences of *Pat* in *Text*, 5 character-comparisons are needed, in addition to one character-access and 4 number comparisons.

IV. Experimental Results and Discussion

In this experiment, the seven algorithms Naive, BM, Inf_suf_Pref, Raita, Circle, and the new algorithm CCCA were implemented and compared on English text with a size more than two mega characters (exactly 2,006,655 characters). A program was designed in C++ to select randomly 3000 patterns. The pattern length ranges from 3 to 93 characters. The average number of occurrences ranges from 1 to 1158. The cost of the searching process to find all the occurrences of the different patterns in each group in *Text* is measured by finding:

1) The average number of first checking,
2) The average number of second checking, and
3) The search clock time.

The results of the experiment are presented in Table I and in Table II. The average number of checks is presented in Table I. The average number of 1st checks ranges from 5,599,507,298 (algorithm no. 3) to 5,893,025,734 (algorithm no. 5)**.** Intuitively, the higher the average number of checks in the first check at the checking step, the better the algorithm is. One can notice that the average number of checks by using the new CCCA algorithm is higher than the average by each one of the other algorithms, except the Circle algorithm (number 5). Furthermore, the average number of second checks by the Circle algorithm is smaller than it is by using CCCA. However, looking at Table II, the time required to find all the occurrences of *Pat* in *Text* by using Circle and CCCA is 47.985 sec and 30.531 sec, respectively. In other words, by using CCCA, the time required by Circle is reduced by 57.17%. This result is expected because the Circle algorithm needs more character comparisons than CCCA to find all the occurrences of *Pat* in *Text*. At each check, the circle algorithm needs one number-comparison at each time the index *TextIx* and *PatIx* adjusted to point to the next pair of characters to examine whether the *PatIx* reaches the end of the pattern. If the check is true, the *PatIx* will be turned back to the first character in the pattern.

Table I also presents the average number of the second checks. It ranges from (108,203,599) to (383,040,878). One can notice that the average number of checks by using CCCA is (See section 2.5) smaller than the average number of checks by the other algorithms, except the Circle algorithm. Intuitively, the smaller the average number of the second checks, the better the algorithm is. In other words, the number of comparisons required by an algorithm to find all the occurrences of *Pat* in *Text* in the second check equals the average number of second checks multiplied by two. Thus increasing the average number of first checks leads to decreasing the average number of second checks. Of course, this leads to decreasing the average number of comparisons and in turn reduces the time required to find the occurrences of *Pat* in *Text*.

Table II presents the clock time required to find all the occurrences of all patterns in *Text*. The clock time includes the time required for reading and pre-processing the patterns. The time ranges from 30.531 seconds (CCCA algorithm 6) to 51.187 seconds (BM algorithm 2). By using the new algorithm CCCA, the clock time required by the other algorithms are reduced by 28.1% (Raita's algorithm) to 67.66% (algorithm BM).

V. Conclusions

A new algorithm Character-Comparison to Character-Access (CCCA) is developed and compared with six algorithms, namely, Naive, BM, Inf_Suf_Pref, Raita, and Circle. The CCCA algorithm uses both the character-access and the character-comparison tests at the checking step while the rest of algorithms use only the character-comparison. An experiment was performed to evaluate the new algorithm CCCA. There are many different criteria used to compare between the different algorithms, including:

1) The average number of comparisons for the first check,
2) The average number of comparisons for the second check,
3) The running time.

In comparison between CCCA and the rest of algorithms and according to the experiment, we have the following results:

1) The average number of first check and the average number of second check required by Naive, BM, Inf_Suf_Pref, Raita, and Circle are improved by CCCA in the following ranges from –2.22% (Circle algorithm) to 2.87% (Inf_suf_Pref) and from –54.36% to 61.56% (see Table I)
2) The clock time required by the algorithms Naive, BM, Inf_Suf_Pref, Raita, and Circle are improved by CCCA in the range of percentage from 28.1% (Raita) to 67.66% (BM) (see Table II).

From these results, one can notice that the CCCA algorithm gains its performance from more than one direction, including:

1) Converting character-comparison into character-access: The CCCA 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 (see section 3).

TABLE I
A COMPARISON IS PRESENTED BETWEEN CCCA AND THE REST OF ALGORITHMS INCLUDING, NAÏVE, BM, INF_SUF_PREF, RAITA, AND CIRCLE, IN TERMS OF THE AVERAGE NUMBER OF FIRST AND SECOND CHECKING AND THE PERCENTAGE OF IMPROVEMENTS

Algorithm No.	Algorithm name	Average number of 1st check	Average number of 2nd chek	Improvement of CCCA vs. other algorithms in 1st check	Improvement of CCCA vs. other algorithms in 2nd check
1	Naive	5,606,801,852	370,011,660	2.75%	56.06%
2	BM	5,605,389,051	371,458,278	2.78%	56.67%
3	Inf_suf_Pref	5,599,507,298	377,673,491	2.87%	59.29%
4	Raita	5,605,392,051	383,040,878	2.77%	61.56%
5	Circle	5,893,025,734	108,203,599	- 2.22%	- 54.36%
6	CCCA	5,765,222,841	237,096,395	0.00%	0.00%

TABLE II
A COMPARISON BETWEEN CCCA AND THE REST OF ALGORITHMS INCLUDING, NAÏVE, BM, INF_SUF_PREF, RAITA, AND CIRCLE IN TERMS OF THE CLOCK TIME REQUIRED TO FIND THE OCCURRENCES OF 3000 PATTERNS IN TWO MEGA BYTES OF TEXT AND THE PERCENTAGE OF IMPROVEMENTS

Algorithm No.	Algorithm name	Clock time in Seconds (Sec)	Improvement of CCCA vs. other algorithms
1	Naïve	43.984 Sec.	44.06%
2	BM	51.187 Sec.	67.66%
3	Inf_suf_Pref	49.860 Sec.	63.31%
4	Raita	39.110 Sec.	28.10%
5	Circle	47.985 Sec.	57.17%
6	CCCA	30.531 Sec.	0.00%

2) Character-access vs. number-comparison: The CCCA uses the condition type character-access (if(*i*)) (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 CCCA 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. Finding the mismatch faster decreases the number of comparisons required to find the *Pat* in *Text*.

As a result, during the checking operation, converting the conditions of type character-comparison and number-comparison into character-access affects on the time required to find the occurrences of *Pat* in *Text*. Furthermore, starting the checking at the latest mismatch in the previous step reduces the number of comparisons.

The algorithm CCCA in this paper concentrates on the performance of the checking operation. The Algorithm Multiple Reference Characters Algorithm (MRCA) concentrates on the performance of the skipping operation. One might look for an algorithm that concentrates on the performance of both operations checking and skipping (i.e., all in one). Such work needs to be investigated in further studies.

REFERENCES

[1] D. E. Knuth, J. H. Morris, and V. R Pratt., Fast pattern matching in strings, SIAM J. Comput. Vol. 6, no. 2, pp. 323-350, 1977.

[2] RS. Boyer, and JS. Moore, A fast string searching algorithm. *Communications of the ACM* Vol. 20, no. 10, pp. 762-772, 1977.

[3] A. Apostolico, and R. R.Giancarlo, "The Boyer-Moore-Galil string searching strategies revisited", SIAM J. Comput. Vol. 15, no. 1, pp. 98-105, 1986.

[4] G. De, V. Smit, A comparison of three string matching algorithms, *Software-Practice and Experience* Vol. 12, pp. 57-66, 1982.

[5] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci., Vol. 40, pp. 31-55, 1985.

[6] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci., Vol. 45, pp. 63-86, 1986.

[7] M. Crochemore, D. Perrin, Two-way string-matching, J. ACM, Vol. 38, pp. 651-675, 1991.

[8] L. Colussi, Correctness and efficiency of the pattern matching algorithms, Information and Computation, Vol. 95, pp. 225-251, 1991.

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

[10] P. D. Smith, Experiments with a very fast substring search algorithm, *Software-Practice and Experience* Vol. 21, no. 10, pp. 1065-1074, 1991.

[11] D. M. Sunday, A very fast substring search algorithm, *Communications of the ACM* Vol. 33, no. 8, pp. 132-142, 1990.

[12] Z. Liu, X. Du, N. Ishii, An improved adaptive string searching algorithm, *Software–Practice and Experience* Vol. 28, no. 2, pp. 191-198, 1998.

[13] P. Fenwick, Fast string matching for multiple searches, *Software–Practice and Experience* Vol. 31, no. 9, pp. 815-833, 2001.

[14] M. Mhashi, A Fast String Matching Algorithm using Double-Length Skip Distances. *Dirasat Journal*, University of Jordan, Jordan Vol. 30, no. 1, pp. 84-92, 2003.

[15] P. Fenwick, Some perils of performance prediction: a case study on pattern matching. *Software–Practice and Experience* Vol. 31, no. 9, pp. 835-843, 2001.

[16] A. Al-jaber, M. Mhashi, A modified double skip algorithm in string searching, *AMSE*(Association for the advancement of modelling & Simulation Techniques in Enterprises) Periodicals Vol.8, no. 4, pp. 1-16, 2003.

[17] M. Mhashi, The effect of multiple reference characters on detecting matches in string searching algorithms, to appear in Software–Practice and Experience 2005.