Enhanced Two Sliding Windows Algorithm For Pattern Matching (ETSW)

Mariam Itriq[1], Amjad Hudaib[2], Aseel Al-Anani[2], Rola Al-Khalid[2], Dima Suleiman[1]

[1.] Department of Business Information Systems, King Abdullah II School for Information Technology, The University of Jordan, Amman 11942 Jordan
[2.] Department of Computer Information Systems, King Abdullah II School for Information Technology, The University of Jordan, Amman 11942 Jordan
r.khalid@ju.edu.jo

Abstract: In this paper, we propose a string matching algorithm - Enhanced Two Sliding Windows (ETSW), which made an improvement on the Two Sliding Windows algorithm (TSW). The TSW algorithm scans the text from both sides simultaneously using two sliding windows. The ETSW algorithm enhances the TSW's process by utilizing the idea of the two sliding windows and focusing on making comparisons with the pattern from both sides simultaneously. The comparisons done between the text and the pattern are done from both sides in parallel. The experimental results show that the ETSW algorithm has enhanced the process of pattern matching by reducing the number of comparisons performed. The best time case is calculated and found to be $O(\lceil m/2 \rceil)$ while the average case time complexity $O(n/(2m))$, where *m* is the pattern length and *n* in the text length.

[Mariam Itriq, Amjad Hudaib, Aseel Al-Anani, Rola Al-Khalid, Dima Suleiman. **Enhanced Two Sliding Windows Algorithm For Pattern Matching (ETSW).** J Am Sci 2012;8(5):607-616]. (ISSN: 1545-1003). http://www.americanscience.org. 64

Keywords: Pattern matching; Two Sliding Windows algorithm; string matching; Berry-Ravindran algorithm.

1. Introduction

Pattern matching is a fundamental theme in various applications such as text processing, searching, computational biology and disease analysis. Pattern matching concentrates on finding all the occurrences of a pattern of length *m* in a text of length *n*. Many researchers have introduced and developed pattern matching algorithms to improve the search process of finding the pattern by decreasing the number of character comparisons (Horspool, 1980; Sheik et al., 2004; Ping and Jiang, 2011; Tarhio, 1993; Claude et al., 2012). Extensive analysis and comparisons on the performance of the algorithms have been conducted.

In this paper, we propose a pattern algorithm: the Enhanced Two Sliding Windows (ETSW). The ETSW algorithm made an over the TSW algorithm, since the TSW algorithm focused on scanning the text from both sides simultaneously while the pattern is scanned only one side. On the other hand, the ETSW algorithm concentrates on both the pattern and the text to be scanned from both sides simultaneously. The algorithm uses two sliding windows, to search the from both sides in parallel. Comparisons done with the pattern are also done from both sides simultaneously. The length of each window is *m* which is the same length as the pattern. The text is divided into left and right parts and the pattern is divided into left and right parts. Each part of the of length $\lceil n/2 \rceil$ while each part of the pattern is of length $\lceil m/2 \rceil$. The ETSW algorithm finds either first occurrence of the pattern in the text through left window or the last occurrence of the pattern through the right window. The experiments showed that the ETSW algorithm reduced the of comparisons needed to search for a pattern in a Comparing the number of comparisons made by ETSW with other algorithms such as TSW, KMP, BoyerMore, BruteForce and Berry-Ravindran that our new algorithm's results were the

2. Related Works

Many researchers have introduced various algorithms to find the exact pattern matching by making use of windowing technique whose length equal to the pattern length. Each algorithms (Kim Kim, 1999; Lecroq, 2007; Franek el al., 2007; Crochemore el al., 1994; Ahmed el al., 2003; He el 2005; Sheu el al., 2006)[14-20] aim to improve the performance and the efficiency by minimizing the number of comparisons between the characters of text and that of the pattern. Al-Emary and Japer, proposed an algorithm to improve the search (El emery and Jaber, 2008). In the preprocessing phase, they split the unchangeable text into *n* equal parts depending on the length of the text and then construct *n* tables. Each table consists of two for each part of the text, the first one is the words' length and the second one is the start position of word in the text classified by the same length. The algorithm searches for the words that consist of the same length in each table. The overall complexity the preprocessing phase is O($n*n\log n$) while the whole complexity for the searching phase where ∑ is the number of character comparison done in each row, the worst case.

Devaki-Paul algorithm (DP), results in better performance and efficiency (Devaki and Paul, 2010). Before starting the search, the algorithm requires a preprocessing of the pattern which prepares a table of occurrences of the first and the last characters of the pattern in the given input text. The search phase uses the table to find the probability of having an occurrence of a pattern in the given input text and find if the probability will lead to successful or unsuccessful search. The time complexity of the preprocessing phase of the DP algorithm is O(m) while the time complexity of the search phase is directly proportional to the total number of occurrences of the first and the last characters of the pattern in the given input text.

Boyer-Moor's string matching algorithm (BM) uses two shift functions: the bad-character shift and the good-suffix shift (Hudaib et al., 2008)(Boyer and Moore, 1977). In BM, the pattern is scanned from right to left, in case of a mismatch the pattern is shifted with the maximum value taken between the two shift functions. The worst case time complexity and the best performance are O(mn) and O(nm^{-1}) respectively. An alternative way to compute the shift table in Boyer-Moor's string matching algorithm, Yang Wang proposed a new method to obtain the shift through array *suffixLength* (Wang, 2009). The new method is more straightforward and preserves the high performance of BM. For a pattern of length m, Yang's method has a O(m) complexity in both space and time. Knuth, Morris and Pratt (KMP) algorithm compares the text with the pattern from left to right (Knuth, Morris, 1977). The complexity of the pre-processing phase is O(m) while the running time is O($n+m$).

TSW scans the text from both sides simultaneously (Hudaib et al., 2008). It uses two sliding windows; each window has a length that is equal to the pattern length. The first window is aligned with the left end of the text while, the second window is aligned with the right end of the text. Both windows slide in parallel over the text until the first occurrence of the pattern is found or until both windows reach the middle of the text. To get better shift values during the searching phase, TSW utilizes the idea of the Berry-Ravindran bad character shift function (Berry and Ravindran, 1999). In TSW, the best time complexity is O(m) and the worst case time complexity is O(((n/2-m+1))(m)). The pre-process time complexity is O(2(m-1)).

3. The Enhanced Two Sliding Windows (ETSW) algorithm

The Enhanced Two Sliding Windows algorithm (ETSW) scans the text as well as the pattern from both sides simultaneously in order to improve the search process. The ETSW algorithm uses two sliding windows to search the text from both sides in parallel. Comparisons done with the pattern is also done from both sides simultaneously. The length of each window is m which is the same length as the pattern. The text is divided into left and right parts, and the pattern is also divided into left and right parts. Each part of the text is of length $\lceil n/2 \rceil$ while each part of the pattern is of length $\lceil m/2 \rceil$. There are two windows: the first window starts scanning the text from the left so we name it the left window, and the second window starts scanning the text from the right; so we name it the right window. Both windows slide in parallel. In each side of the text, the pattern is compared with the text from both the left and the right sides of the pattern simultaneously.

ETSW algorithm stops when one of the two sliding windows finds the pattern or the pattern is not found within the text string at all. The ETSW algorithm finds either the first occurrence of the pattern in the text through the left window or the last occurrence of the pattern through the right window.

Both the TSW (Hudaib et al., 2008) and the ETSW algorithms utilize the idea of BR bad character shift function (Tarhio and Ukkonen, 1993) to get better shift values during the searching phase. BR algorithm provides a maximum shift value in most cases without losing any characters. Therefore, the number of comparisons to determine the amount of shift in both algorithms is the same.

The main difference between the TSW algorithm (Hudaib et al., 2008) and the Enhanced TSW algorithm is that the comparisons made to find if there is a match between the pattern and the text in the TSW are done only from the left side of the pattern while in the new Enhanced TSW algorithm the comparisons are done from the left and right sides of the pattern in the same time. This addition to the old algorithm decreases the search time and the number of comparisons done.

3.1. Pre-processing phase

The pre-processing phase as in TSW algorithm generates two arrays *nextl* and *nextr*, array is a one-dimensional array. The shift values the *nextl* array are calculated according to Berry-Ravindran bad character algorithm (BR) (Boyer Moore, 1977) as in equation (1). The shift values needed to search the text from the left side. The values of the *nextr* array that are needed to search text from the right side are calculated according to TSW shift function as in equation (2). During the searching process, the *nextl* and the *nextr* arrays be invariable.

$$\text{Bad Char } shiftl[a,b] = \min \begin{cases} 1 & \text{if } p[m-1] = a \\ m-i & \text{if } p[i]p[i+1] = ab \\ m+1 & \text{if } p[0] = b \\ m+2 & \text{Otherwise} \end{cases} \quad (1)$$

$$\text{BadChar } shiftr[a,b] = \min \begin{cases} m+1 & \text{if } p[m-1] = a \\ m-((m-2)-i) & \text{if } p[i]p[i+1] = ab \\ 1 & \text{if } p[0] = b \\ m+2 & \text{Otherwise} \end{cases} \quad (2)$$

The pre-processing phase is the same in both TSW and ETSW algorithms while the searching phase is being enhanced in ETSW.

3.2. Searching phase:

In the ETSW algorithm, the text string is scanned from two directions from left to right and from right to left. In mismatch cases, during the searching process from the left, the left window is shifted to the right, while during the searching process from the right, the right window is shifted to the left. Both windows are shifted until the pattern is found or the windows reach the middle of the text. The algorithm used for searching uses four pointers, two for each window. The left window uses the *L* and *temp_newlindex* pointers while the right window uses the *R* and *temp_newrindex* pointers, (Figure 1).

At the beginning of the algorithm, in each window, the first character of the pattern is compared with the corresponding character of the text while at the same time the last character of the same pattern is also compared with the corresponding character of the text. This primary step will reduce the number of comparisons done later in the left and the right windows. We will discuss searching by the left and right windows.

3.2.1. Left_window search process

While searching the left window, the *L* and *temp_newlindex* pointers are used to compare the and the pattern, the *L* pointer points at the last character of the pattern and the *temp_newlindex* at the first character of the pattern, the characters of both the text and the pattern are compared. If a mismatch occurs in one of the a shift occurs according to the Berry-Ravindran character algorithm (BR). In case of a match the pointers will move. The *L* pointer will move to the and the *temp_newlindex* will move to the right time a match occurs the pointers move until they the middle of the pattern or the *L* pointer is less or equal the *temp_newlindex* ,in either case the is found.

3.2.2. Right_window search process

While searching the right window, the *R* and *temp_newrindex* pointers are used to compare the text and the pattern, the *R* pointer points at the first character of the pattern and the *temp_newrindex* points at the last character of the pattern, the corresponding characters of both the text and the pattern are compared. If a mismatch occurs in one of the pointers a shift occurs according to the Berry-Ravindran bad character algorithm (BR). In case of a match the two pointers will move. The *R* pointer will move to the right and the *temp_newrindex* will move to the left .Each time a match occurs the pointers move until they reach the middle of the pattern or the *R* pointer is greater than or equal to the *temp_newrindex* ,in either case the pattern is found. This algorithm searches the left and the right windows in parallel. (Figure 1)

3.3. Working Example

In this section we will present an example to clarify the ETSW algorithm.

Given:

*Pattern(P)="*GAATCCAT*", m=8*

Text(T)="GAATAGCTTCATAACGATAATTTGAGAG AGAGAATCCATCGATTAT",n=47

Pre-processing phase

Initially, shiftl = shiftr = m+2 = 10.

The shift values are stored in two arrays *nextl* and *nextr* as shown in Figure 3(a) and Figure 3(b) respectively.

Shift Values from the left

Index	0	1	2	3	4	5	6	
	8	7	6	5	4	3	2	*nextl*

(a)

Shift Values from the right

Index	0	1	2	3	4	5	6	
	2	3	4	5	6	7	8	*nextr*

(b)

Figure 3. The *nextl* and *nextr* arrays

```
L=m-1; //text index used from left
R=n-(m-1)-1; //text index used from right
Tindex=0;//text index used to control the scanning process

While (Tindex<=⌈n/2⌉)
Begin
  foundleft = false;
  foundright = false;
  l=m-2 ;  // pattern index used at left side from the end of the pattern
  r=0;  // pattern index used at right side from the beginning of the pattern
  temp-lindex=temp-rindex=0;//keep record of the text index where the pattern match the text during comparison
  temp_newlindex=0; // pattern index used at left side from the beginning of the pattern
  temp_newrindex= (m-1); // pattern index used at right side from the end of the pattern

 if (P[m-1]=T[L]  and  p[0]=T[L-m+1])
    begin
       temp-lindex=L;
      L=L-1;
      temp_newlindex++;
     while ((l>=0 and P[l]=T[L])  and (P[temp_newlindex]=T[L-l+ temp_newlindex] ))
         { L=L-1, l=l-1;  temp_newlindex++;
           if  ((L-l+ temp_newlindex) >=L)
           {foundleft = true; exit from while loop;}
          }    //search from left
     end

 if (P[0]=T[R] and  p[temp_newrindex]=T[L+m-1])
   begin
     temp-rindex=R;
    R=R+1;
    temp_newrindex--;
    while( (r<m and P[r]=T[R])  and P[temp_newrindex]=T[R+ temp_newrindex-r]  )
       { R=R+1, r=r+1; temp_newrindex --;
           if  (R+ temp_newrindex-r<=R)
            {foundright = true; exit from while loop;}
             }    //while

       }   //search from right
   end

    if (foundright) {display “match at right:    "+ temp-rindex) ; exit from outer loop;}
    if (foundleft) {display “match at left:    "+ temp-lindex  –m +1); exit from outer loop;}
    //exit in case if we search for one occurrence the first or last one
   R= temp-rindex; //to avoid skipping characters after partial matching at right
   L=temp-lindex; // to avoid skipping characters after partial matching at left
   if(not foundleft and not foundright){ display (“not found”); exit from outer loop;}
   L=L+get(shiftl);//from pre-processing step
   R=R-get(shiftr);//from pre-processing step
   Tindex= Tindex+1;
   End;
```

Figure 1. ETSW Algorithm

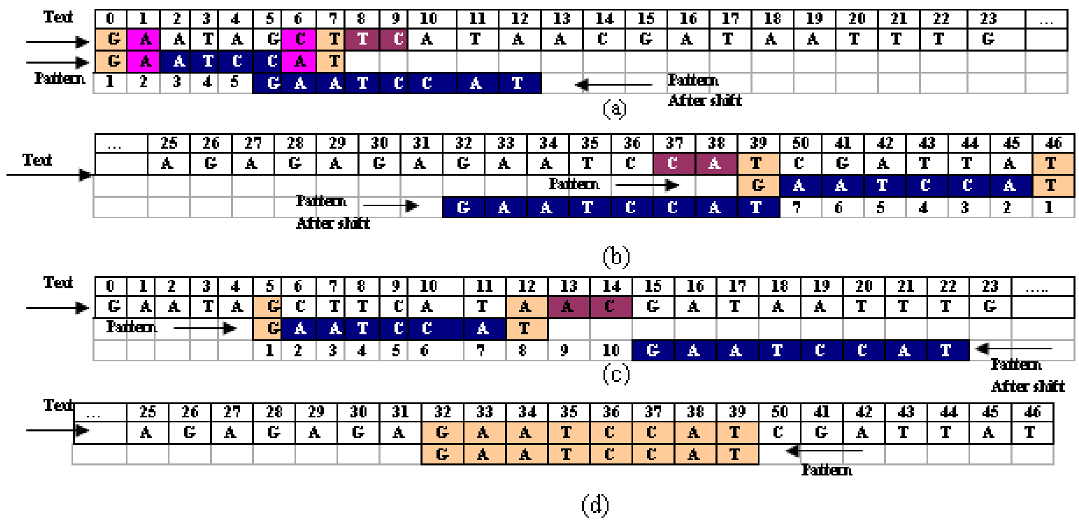

Figure 2. Working Example

To build the two next arrays (*nextl* and *nextr*), we take each two consecutive characters of the pattern and give it an index starting from 0. For example for the pattern structure GAATCCAT, the consecutive characters GA,AA,AT,TC,CC,CA and AT are given the indexes 0,1,2,3,4,5 and 6 respectively.

The shift values for the *nextl* array are calculated according to Equation (1) while the shift values for the *nextr* array are calculated according to Equation (2).

Searching phase

The searching process for the pattern *P* is illustrated through the working example as shown in Figure 2.

First attempt:

In the first attempt (see Figure 2(a)), we align the first sliding window with the text from the left. In this case, comparisons are made between the text character located at index 0 (character G) with the leftmost character in the pattern (character G). At the same time, comparisons are made between the text character at index 7 (character T) with the rightmost character in the pattern (character T). As a result, a match occurs so we continue by comparing the text character at index 1 (character A) with the second leftmost character in the pattern (character A). At the same time, we compare the text character at index 6 (character C) with the pattern character at index 6 (character A). Since there is a mismatch at index 6, (where a match must occur in both comparisons); the pattern should be shifted. Therefore to determine the amount of shift (*shiftl*) we will do the following:

a) Take the two consecutive characters from the text at index 8 and 9 which are (T and C) respectively.
b) We find the index of TC in the pattern which is 3.
c) Since we search the text from the left side we use *nextl* array, and *shiftl*= *nextl*[3] = 5

Therefore the window is shifted to the right 5 steps.

As explained in the example the number of comparisons needed to determine if there is a match or not is two; this is because two character comparisons between the text and the pattern are performed at the same time as seen in the if statement in Figure:

Using TSW algorithm we need 5 comparisons.

Second attempt:

In the second attempt (see Figure 2 (b)), we align the second sliding window with the text from the right. In this case, a match occurs between the text character at index 46 (character T) and the rightmost character in the pattern (character T) while there is a mismatch between the text character at index 39 (T) and the leftmost character in the pattern (character G);therefore we take the two consecutive characters from the text at index 37 and 38 which are (C and A) respectively. To determine the amount of shift (*shiftr*), we have to do the following:

a) We find the index of CA in the pattern which is 5.
b) Since we search the text from the right side we use *nextr* array, and *shiftr*= *nextr*[5]=7.

Therefore the window is shifted to the left 7 steps.

As explained in the example the number comparisons needed to determine if there is a or not is one; while by using TSW algorithm we 3 comparisons.

Third attempt:

In the third attempt (see Figure 2(c)), a mismatch occurs from the left between the text character at index 12 (character A) and the rightmost character in the pattern (character T) while there is a match between the text character at index 5 (character G) and the leftmost character in the pattern (character G)); therefore we take the two consecutive characters from the text at index 13 and 14 which are (A and C) respectively, since AC is not found in the pattern, so the window is shifted to the right 10 steps.

Fourth attempt:

We align the leftmost character of the pattern P[0]with T[32]. A comparison between the pattern and the text characters leads to a complete match at index 32. In this case, the occurrence of the pattern is found using the right window. The number of comparisons needed to determine if there is an exact match is 4; while by using TSW algorithm we need 8 comparisons.

4. Analysis

Preposition 1: The space complexity is O(2(m-1)) where m is the pattern length.

Preposition 2: The pre-process time complexity is O(2(m-1)).

Lemma 1: The worst case time complexity is $O(((n/2-\lceil m/2\rceil+1))(\lceil m/2\rceil))$

Proof: The worst case occurs when at each attempt, all the compared characters of both pattern sides matched the corresponding text characters except the pattern character indexed $\lceil m \rceil$, and at the same time the shift value is equal to 1.

Lemma 2: The best case time complexity is $O(\lceil m/2\rceil)$.

Proof: The best case occurs when the pattern is found at the first index or at the last index (n-m),in this case the number of comparisons made to compare m pattern characters are $\lceil m/2\rceil$.

Lemma 3: The Average case time complexity is $O(n/(2m))$.

Proof: The Average case occurs when the two consecutive characters of the text directly following the sliding window is not found in the pattern. In this case, the shift value will be (m+2) and hence the time complexity is $O(n/(2m))$.

5. Results and Discussions

To demonstrate the working process of the ETSW algorithm, several experiments have been done using Book1 from the Calgary corpus to be the text. Book1 consists of 141,274 words (752,149 characters).

The experiments compare the ETSW with the TSW algorithm taking into account many variables depending on the pattern length and the pattern position in the text. The searching process is performed from the left and the right sides of Book1. Comparisons done with the pattern is also done from both sides simultaneously. The pattern is found wither it is located at the beginning of the text, at the middle of the text or even at the end of the text. Figure 4 and Table1 show the results of comparing ETSW and TSW algorithms. In Table 1 the first column displays the pattern length while the second column displays the number of words for each pattern length.

For example, 1988 words of length 7 were taken. It can be noticed that the number of attempts and comparisons made by TSW is 9341 and 10263 respectively. While the number of attempts and comparisons made by ETSW for the same pattern length is 9341 and 9118 respectively. This is a considerable reduction in the number of comparisons made by ETSW.

This can be explained since the pattern in the ETSW algorithm is being compared from the left and right sides at the same time. Both TSW and ETSW uses two sliding windows which explain why the number of attempts made by the two algorithms are the same.

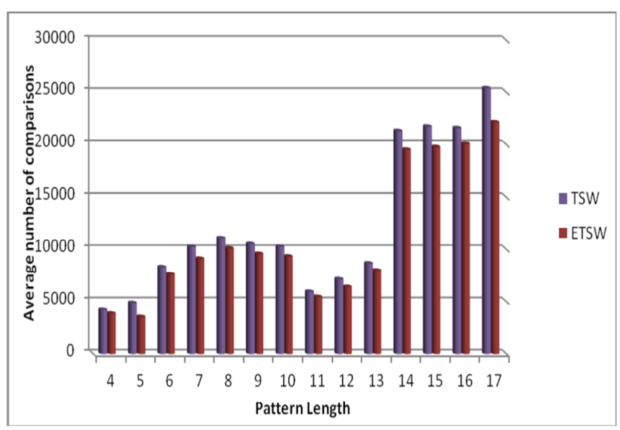

Figure 4. The average number of comparisons for patterns with different lengths

Table 1. The average number of attempts and comparisons for patterns with different lengths

Pattern length	Number Of Words	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	8103	3904	4213	3904	3875
5	4535	4456	4896	4456	3549
6	2896	7596	8311	7596	7633
7	1988	9341	10263	9341	9118
8	1167	10056	11087	10056	10115
9	681	9538	10538	9538	9590
10	382	9283	10272	9283	9339
11	191	5451	5967	5451	5482
12	69	6384	7168	6384	6433
13	55	7947	8673	7947	7986
14	139	19437	21319	19437	19535
15	32	19682	21739	19682	19782
16	10	20029	21596	20029	20092
17	3	21897	25404	21897	22147

Table 2. The number of attempts and comparisons performed to search for the first appearance of selected pattern from the beginning of the text

Pattern length	Index	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	67	25	29	25	26
5	33	11	15	11	12
6	82	23	28	23	25
7	39	11	17	11	13
8	99	21	28	21	24
9	260	51	65	51	54
10	590	105	120	105	109
11	189	35	47	35	39
12	2401	363	402	363	368

Tables 2-4 display the pattern length, the index, number of comparisons and attempts made by TSW and ETSW. These results are needed to search for the first appearance of the pattern at the beginning, middle and at the end of Book1. Table 2 shows patterns of different lengths located at the beginning of the text in Book1. For example, it took the TSW 28 comparisons to find a pattern of length 8 located at index 99. On the other hand, it took the ETSW 24 comparisons to locate the same pattern. Table 3 shows patterns of different lengths located at the middle of the text of Book1. For example, 100526 comparisons are made by TSW to locate a pattern of length 7 located at index 380422. Noticeably, less number of comparisons (90905) are made by ETSW to locate the same pattern. Table 4 shows patterns of different lengths located at the end of the text. For example, the pattern of length 7 located at index 689847 was located by TSW after 20962 comparisons and located by ETSW after 18897 comparisons.

Table 3. The number of attempts and comparisons performed to search for the first appearance of a selected pattern from the middle of the text

Pattern length	Index	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	380375	134763	150571	134763	136198
5	380438	115397	129100	115397	116800
6	380416	100903	112695	100903	102153
7	380422	89959	100526	89959	90905
8	380409	80905	90538	80905	81888
9	380471	73553	82269	73553	74371
10	380537	67377	75237	67377	68116
11	380548	62139	69407	62139	62806
12	380568	57793	64663	57793	58453

Table 4. The number of attempts and comparisons performed to search for the first appearance of a selected pattern from the end of the text

Pattern length	index	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	689749	28062	31392	28062	28393
5	689788	24020	26957	24020	24349
6	689795	21044	23605	21044	21323
7	689847	18706	20962	18706	18897
8	689928	16768	18885	16768	16989
9	689942	15256	17123	15256	15463
10	689974	13922	15574	13922	14090
11	690033	12910	14486	12910	13047
12	690041	11982	13498	11982	12145

Table 5 and table 6 show the average number of comparisons and attempts needed to search for the first and the middle appearance of 100 words selected from Book1. The results of taking 100 words are similar to that of taking a single word with different lengths.

The ETSW algorithm finds the pattern with minimum effort. In case of a complete mismatch, as in Table 7, the average number of comparisons and attempts of the ETSW algorithm is the minimum.

Table 8 and Figure 5 show the average number of attempts and comparisons for patterns with different lengths, performed by ETSW algorithm and other algorithms. ETSW algorithm has the minimum average number of comparisons and attempts among all other algorithms. The results are expected because ETSW has the following advantages over the other algorithms: It searches the text from both sides at the same time; it also concentrates on comparing the pattern form both its sides simultaneously. In case of a mismatch the pattern is shifted by a value that ranges from 1 up to m+2 positions based on the BR shift function.

Table 5. The average number of attempts and comparisons performed to search for (100) selected from the beginning of the text

Pattern length	Number of words	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	100	143	157	143	145
5	100	185	206	185	187
6	100	227	255	227	230
7	100	347	388	347	351
8	100	504	568	504	510
9	100	670	750	670	677
10	100	1160	1290	1160	1170
11	100	622	705	622	628
12	100	865	972	865	878

Table 6. The average number of attempts and comparisons performed to search for (100) selected from the middle of the text

Pattern length	Number of words	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	100	2726	2959	2726	2737
5	100	13965	15140	13965	11618
6	100	16682	18317	16682	16771
7	100	27267	30095	27267	26242
8	100	27830	30915	27830	28015
9	100	33929	37200	33929	34069
10	100	29676	32817	29676	29845
11	100	23195	24646	23195	23242
12	100	26806	30222	26806	27009

Table 7. The number of attempts and comparisons performed to search for a set of patterns that do exist in the text

Pattern length	TSW Attempts	TSW Comparisons	ETSW Attempts	ETSW Comparisons
4	129670	132754	129670	129696
5	113866	122233	113866	114063
6	99610	106441	99610	99783
7	88628	94812	88628	88818
8	77846	79928	77846	77881
9	72504	77837	72504	72668
10	66400	70297	66400	66497
11	60880	63549	60880	60961
12	57088	61118	57088	57196

These advantages have a considerable effect on the number of comparisons and attempts in most cases. On the other hand, the largest number of attempt and comparisons are performed by BF(Brute Force algorithm) because in case of a mismatch, it shifts the pattern one position to the right.

TSW searching results are better than that of KMP, BF, BM and BR. This is because TSW searches the text from both sides while all other algorithms search the text from one side. TSW searching results is close to ESTW. The number of comparisons of ETSW is less than that of TSW because of the additional feature of comparing the pattern from both sides.

ETSW best performance compared to TSW is seen when we search for the first appearance of a selected pattern from the end of the text. The number of comparisons in TSW is m where the number of comparisons in ETSW is $\lceil m/2 \rceil$.

Table 9 and Figure 6 show the average number of comparisons and attempts performed to search for a set of patterns that do not exist in the text that is there is a complete mismatch. ETSW algorithm is the minimum.

6. Conclusion

In this paper, we presented a new pattern matching algorithm the Enhanced Two Sliding Windows (ETSW) algorithm. This algorithm enhances the performance of the previous Two Sliding Windows (TSW) algorithm. Both ETSW and TSW algorithms employs the main idea of BR by maximizing the shift value and using two sliding windows rather than using one sliding window working in parallel, to scan all text characters. In both algorithms, two arrays are used to store the calculated shift values for the two sliding windows . Each array is a one dimensional array of length (m-1). The main difference between TSW and ETSW which added value to ETSW is that comparisons made to find if there is a match between the pattern and the text in the TSW are done only from the left side of the pattern while in the new Enhanced TSW algorithm the comparisons are done from the left and right sides of the pattern at the same time. This enhancement decreases the number of comparisons and the search time. The performance of ETSW is evaluated by using a text string and various set of patterns. Searching the text from both sides, and comparing the pattern from both sides simultaneously gives ETSW algorithm a preference over the TSW and other well known algorithms.

In future researches, we intend to the idea of the enhanced two sliding windows algorithm on other algorithms such as KMP and

Table 8. The average number of attempts and comparisons for patterns with different lengths

Pattern length	Number of words	TSW		ETSW		BR		KMP		BM		BF	
		Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	8103	3904	4213	3904	3875	6409	7039	35946	36972	9549	10055	36029	37056
5	4535	4456	4896	4456	3549	9577	10645	61500	63460	13435	14246	61685	63645
6	2896	7596	8311	7596	7633	10898	12173	79064	81663	14793	15749	79353	81952
7	1988	9341	10263	9341	9118	11953	13345	97291	100722	15797	16817	97667	101100
8	1167	10056	11087	10056	10115	13256	14807	117903	122341	17190	18314	118360	122799
9	681	9538	10538	9538	9590	14149	15892	136829	142234	18145	19403	137387	142793
10	382	9283	10272	9283	9339	14127	15799	148359	154279	18048	19254	148997	154917
11	191	5451	5967	5451	5482	12808	14243	144335	149852	16449	17477	145007	150525
12	69	6384	7168	6384	6433	9598	10923	114781	120531	12074	13001	115338	121088
13	55	7947	8673	7947	7986	10334	11370	133469	140255	13422	14176	133952	140739
14	139	19437	21319	19437	19535	19548	21673	265189	275981	25075	26603	266460	277257
15	32	19682	21739	19682	19782	19817	22384	277260	288103	24791	26609	278900	289750
16	10	20029	21596	20029	20092	26086	28644	391604	403333	33423	35146	393580	405313
17	3	21897	25404	21897	22147	22554	28148	334855	347547	26266	30016	336367	349060

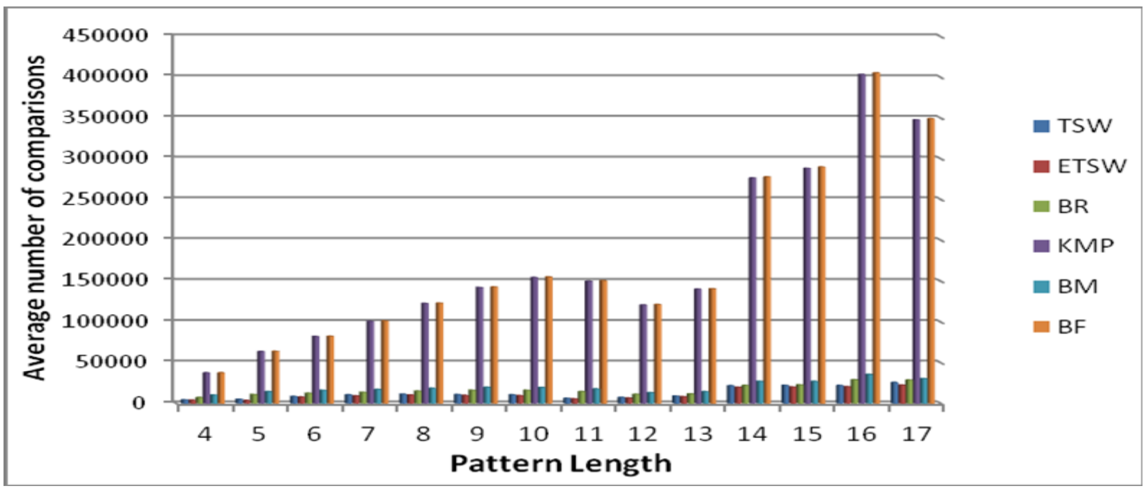

Figure 5. The average number of comparisons for patterns with different lengths

Table 9. The number of attempts and comparisons performed to search for a set of patterns that do not exist text

Pattern length	TSW		ETSW		BR		KMP		BM		BF	
	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	129670	132754	129670	129696	130003	133090	768769	777901	192750	194978	768769	777941
5	113866	122233	113866	114063	115997	130327	768768	777900	155232	165498	768768	777940
6	99610	106441	99610	99783	101610	113474	768767	777899	130572	138714	768767	777939
7	88628	94812	88628	88818	90409	100959	768766	777898	111651	118515	768766	777938
8	77846	79928	77846	77881	78016	80319	768765	777897	101486	103024	768765	777937
9	72504	77837	72504	72668	74049	83339	768764	777896	86940	92798	768764	777936
10	66400	70297	66400	66497	69760	76152	768763	777895	82517	86563	768763	777935
11	60880	63549	60880	60961	66286	70597	768762	777894	77424	79823	768762	777934
12	57088	61118	57088	57196	62142	69466	768761	777893	70058	74553	768761	777933

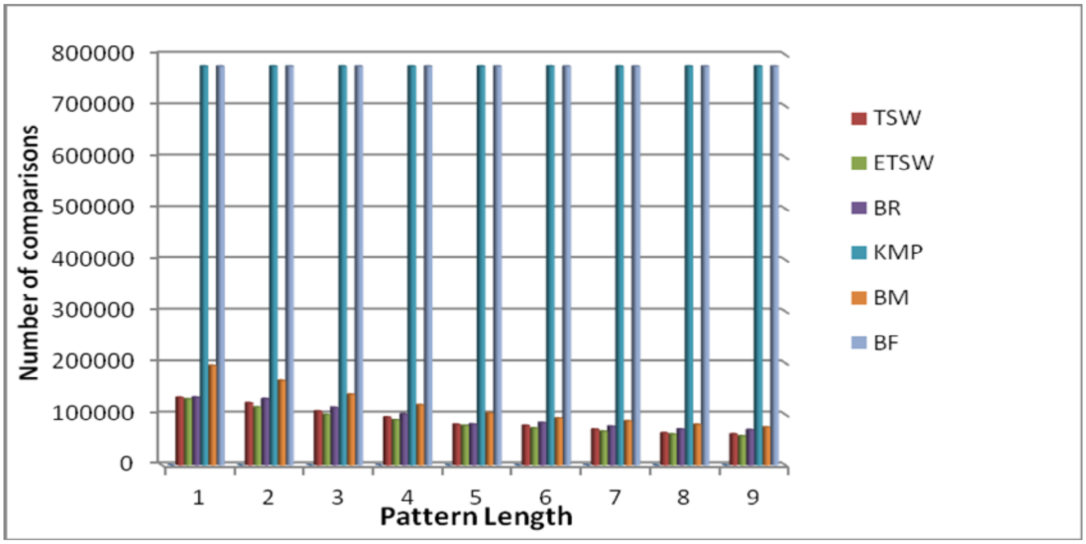

Figure 6. The number of comparisons Performed to search for a set of patterns that do not exist in the text

References

1. El Emary I. and Jaber M. A New Approach for Solving String Matching Problem through Splitting the Unchangeable Text. World Applied Sciences Journal 2008; 4(5): 626-633.
2. Devaki Pendlimarri and Paul Bharath Bhushan Petlu. Novel Pattern Matching algorithm for Single Pattern Matching. International Journal on Computer Science and Engineering (IJCSE) 2010; 2(8): 2698-2704.
3. Hudaib A., Al-Khalid R., Suleiman D., Itriq M. and Al-Anani A. A Fast Pattern Matching Algorithm with Two Sliding Windows (TSW). Journal of Computer Science 2008; 4 (5): 393-401.
4. R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of the ACM 1977; 20(10):762-772.
5. Yang Wang. On the shift-table in Boyer-Moore's String Matching Algorithm. JDCTA 2009; 3(4): 10-20, doi: 10.4156/jdcta,.
6. Knuth, D.E., J.H. Morris and V.R. Pratt. Fast pattern matching in strings. SIAM J. Comput. 1977; 6(2):323-350.
7. Berry, T. and S. Ravindran. A fast string matching algorithm and experimental results. Proceedings of the Prague Stringology Club Workshop ’99, Liverpool John Moores University 1999; pp: 16-28.
8. Tarhio, J., Ukkonen, E. Approximate Boyer-Moore String Matching. SIAM J. Comput. 1993;22(2):243-260.
9. Horspool, R. N. Practical fast searching in strings. Software-Practice and Experience 1980;10(6):501–506.
10. Sheik, S.S., Aggarwal, Sumit K., Poddar, Anindya, Balakrishnan, N. and Sekar, K. *A FAST Pattern Matching Algorithm.*. Journal of Chemical Information and Computer Sciences 2004; 44 (4):1251-1256.
11. Ping Zhang, Jiang Hui Liu. An Improved Pattern Matching Algorithm in the Intrusion Detection System. Applied Mechanics and Materials 2011; 48-49:203-207.
12. Tarhio J.. A Boyer-Moore Approach for Two-Dimensional Matching. Report UCB/UCD 93/784, Computer Science Division, University of California, Berkeley 1993.
13. Claude, F., Navarro, G., Peltola, H., Salmela, L. and Tarhio, J. String matching with alphabet sampling. *Journal of Discrete Algorithms* 2012; 11: 37-50.
14. Kim S., Kim Y. A fast multiple string-pattern matching algorithm. in: Proc. 17th AoM/IAoM Conference on Computer Science 1999; San Diego, CA, 17: 44-49.
15. Lecroq, T..Fast exact string matching algorithms. Information Processing Letters 2007; 102(6): 229-235.
16. Franek, F., Jennings, C.G., Smyth, W.F. A simple fast hybrid pattern-matching algorithm. J. Discrete Algorithms 2007; 5(4): 682–695.
17. Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, L., Lecroq, T., Plandowski,, W., Rytter, W. Speeding up two string matching algorithms. Algorithmica 1994; 12(4): 247-267.
18. Ahmed M., Kaykobad M., and Chowdhury R.A.. A New String Matching Algorithm. presented at Int. J. Comput. Math. 2003; 80(7): 825-834.
19. He, L., Fang, B. and Sui, J. The wide window string matching algorithm. Theoretical Computer Science 2005; 332(1–3):391–404.
20. Sheu T.F., Huang N.F. and Lee H.P. A Time and Memory Efficient String Matching Algorithm for Intrusion Detection Systems. IEEE Proceedings of Global Telecommunications Conference (GLOBECOM'06), San Francisco, USA 2006; pp. 1-5.

4/27/2012