Journal of Software Engineering and Applications, 2015, 8, 154-165
Published Online March 2015 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2015.83016

Scientific Research Publishing

Four Sliding Windows Pattern Matching Algorithm (FSW)

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

[1]Department of Computer Information Systems, King Abdullah II School for Information Technology, The University of Jordan, Amman, Jordan
[2]Department of Business Information Technology, King Abdullah II School for Information Technology, The University of Jordan, Amman, Jordan
Email: dima.suleiman@ju.edu.jo, m.itriq@ju.edu.jo, a.anani@ju.edu.jo, r.khalid@ju.edu.jo, ahudaib@ju.edu.jo

Received 25 February 2015; accepted 18 March 2015; published 20 March 2015

Abstract

This paper presents an efficient pattern matching algorithm (FSW). FSW improves the searching process for a pattern in a text. It scans the text with the help of four sliding windows. The windows are equal to the length of the pattern, allowing multiple alignments in the searching process. The text is divided into two parts; each part is scanned from both sides simultaneously using two sliding windows. The four windows slide in parallel in both parts of the text. The comparisons done between the text and the pattern are done from both of the pattern sides in parallel. The conducted experiments show that FSW achieves the best overall results in the number of attempts and the number of character comparisons compared to the pattern matching algorithms: Two Sliding Windows (TSW), Enhanced Two Sliding Windows algorithm (ETSW) and Berry-Ravindran algorithm (BR). The best time case is calculated and found to be $O\left(\frac{m}{4}\right)$ while the average case time complexity is $O\left(\frac{n}{4m}\right)$.

Keywords

Pattern Matching, FWS, Enhanced Two Sliding Windows Algorithm, RS-A Fast Pattern Matching Algorithm

1. Introduction

String matching is a challenging subject in computer science. Many researchers proposed and designed different

How to cite this paper: Hudaib, A., Al-Khalid, R., Al-Anani, A., Itriq, M. and Suleiman, D. (2015) Four Sliding Windows Pattern Matching Algorithm (FSW). *Journal of Software Engineering and Applications*, **8**, 154-165.
http://dx.doi.org/10.4236/jsea.2015.83016

techniques and algorithms to find all possible occurrences of a pattern *P* of size *m* from the text string *T* of size *n* [1]-[3]. The researchers focus on reducing the number of character comparisons and processing time. String matching algorithms are used in various applications such as matching DNA sequences [4] [5], voice recognition, image processing, text processing [6]-[8], network security, real-time problem, web applications and information retrieval from databases [9] [10].

In this paper, we combine the searching strategy of the ETSW algorithm and the shifting process of the BR algorithm [11] [12]. This paper ends up with a new algorithm (FSW) that uses four sliding windows which are equal to the length of the pattern. FSW divides the text into two parts; each part is scanned from both sides simultaneously using two sliding windows. The four windows slide in parallel in both halves of the text. The comparisons done between the text and the pattern are done from both of the pattern sides in parallel. The FSW algorithm finds the first occurrence of the pattern from either the left windows or the right windows from both parts of the text. In all the cases we tested and the comparisons we performed with other matching algorithms such as BR, TSW and ETSW, the proposed algorithm FSW proved to be the best in reducing the number of comparisons and attempts needed to find the pattern [13] [14]. This paper is organized as follows: Section 2 provides an overview of the related works; Section 3 explains the FSW algorithm; Section 4 includes the performance analysis and Section 5 concludes the paper.

2. Related Works

Recently, several new pattern matching algorithms have been proposed to minimize the number of comparisons done to locate a pattern in a text [15]-[17]. Enhancements are made on both the searching process by using several sliding windows that scan the text in parallel and on the preprocessing phase by determining the shift value that the pattern should move in the process of searching the text for the pattern [18].

The Berry-Ravindran algorithm (BR) uses the bad character shift function to calculate the shift value for the two consecutive characters in the text immediately to the right of the pattern window. In BR the searching time complexity is calculated to be $O(nm)$ and the pre-processing time complexity is $O(\sigma 2)$ [14]. The Two-Sliding Window algorithm (TSW) determines the shift value by using the idea of Berry-Ravindran bad character shift function. The pre-processing time complexity is found to be $2(m-1)$.

In the searching phase, TSW uses two sliding windows to scan the text from both sides in parallel. The search process continues until the first occurrence of the pattern is found or until both windows reach the middle of the text. The size of each sliding window is equal to the length of the pattern. In TSW, the best time complexity is $O(m)$ and the worst case time complexity is $O\left(\left(\frac{n}{2}-m+1\right)(m)\right)$ [12].

The Enhanced Two Sliding Windows algorithm (ETSW) utilizes the idea of Berry-Ravindran bad character shift function to get better shift values during the searching phase. In the searching phase, ETSW scans both of the text and the pattern from both sides in parallel. Both the text and the pattern are divided into left and right parts. So, the text is searched from both parts simultaneously and the comparisons with the pattern are done from both its parts at the same time. ETSW algorithm stops when the pattern is not found. In ETSW, the best time case is $O\left(\frac{m}{2}\right)$ while the average case time complexity is $O\left(\frac{n}{2m}\right)$ [11]. The Enhanced RS-A algorithm (ERS-A) [19] utilizes the idea of RS-A algorithm to get better shift values. ERS-A algorithm uses four consecutive characters in the text immediately to the right of the pattern window.

The ERS-A algorithm uses two sliding windows to search for a pattern in a text. The two windows slide from both sides of the text simultaneously. The searching process continues until a match is found. It stops immediately if the pattern is not found in the text. In ERS-A, the best case complexity is $O(m)$ while the average case time complexity is $O\left(\frac{n}{2*(m+4)}\right)$ [19].

In this paper enhancements are made on the ETSW algorithm, the preprocessing phase is the same while the searching process is made better using four sliding windows to scan the text simultaneously. The comparisons with the pattern are also done from both of the pattern sides in parallel.

3. The FSW Algorithm

The FSW algorithm scans the text as well as the pattern from both sides simultaneously in order to improve the search process. The proposed algorithm (FSW) scans the text using four sliding windows, allowing multiple alignments in the searching process. Each window size is equal to the length of the pattern. In the searching phase, the text is divided in into two parts; each part is scanned from both sides simultaneously using two sliding windows. The four windows slide in parallel in both halves of the text. The comparisons done between the text and the pattern are done from both sides of the pattern in parallel.

Two of the siding windows are aligned with the left and the right sides of the first part of the text and at the same time the other two sliding windows are aligned with the left and the right sides of the second part of the text resulting in four sliding windows that scam the text simultaneously. FSW algorithm stops when a sliding window finds the pattern or the pattern is not found within the text string at all.

FSW algorithm enhances the searching process in the ETSW algorithm. Both the FSW and ETSW algorithms utilize the idea of BR bad character shift function to get better shift values during the searching phase.

The main difference between the FSW and the ETSW algorithms lies in the searching process. During the search the comparisons between the pattern and the text in the FSW are made using four sliding windows while in the Enhanced TSW algorithm two sliding windows are used. Using two additional windows during the search process decreases the number of comparisons and attempts done.

3.1. Pre-Processing Phase

The pre-processing phase of the FSW algorithm is the same as in ETSW algorithm. Two arrays *nextl* and *nextr* are generated. Each array is a one-dimensional array. The shift values are calculated according to Berry-Ravindran bad character algorithm (BR). The shift values needed to search the text from the left side are stored in the *nextl* array. On the other hand *nextr* array contains the shift values needed to search the text from the right side.

To build the two 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 abcd, the consecutive characters ab, bc and cd are given the indexes 0, 1 and 2 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). In Equation (1), we compare between the last character in the pattern $m-1$ with a if there is a match the window is shifted 1 character to the right. If there is a mismatch the shift is the minimum of $m-i$ in case of $p[i]\,p[i+1]=\text{ab}$, $m+1$ in case of $p[0]=\text{b}$ and $m+2$ otherwise. In Equation (2), if the first character of the pattern matches b then the window is shifted 1 character to the left otherwise we take the minimum of $m-((m-2)-i)$ in case of $p[i]\,p[i+1]=\text{ab}$, $m+1$ in case of $p[m-1]=\text{a}$, $m+2$ otherwise.

$$\text{Bad Char } shiftl\,[\text{a},\text{b}] = \min\begin{Bmatrix} 1 & \text{if } p[m-1]=\text{a} \\ m-i & \text{if } p[i]\,p[i+1]=\text{ab} \\ m+1 & \text{if } p[0]=\text{b} \\ m+2 & \text{otherwise} \end{Bmatrix} \tag{1}$$

$$\text{Bad Char } shiftr\,[\text{a},\text{b}] = \min\begin{Bmatrix} m+1 & \text{if } p[m-1]=\text{a} \\ m-((m-2)-i) & \text{if } p[i]\,p[i+1]=\text{ab} \\ 1 & \text{if } p[0]=\text{b} \\ m+2 & \text{otherwise} \end{Bmatrix} \tag{2}$$

3.2. Searching Phase

In the four sliding windows algorithm, the text is divided into two parts. The left part is named part 1 while the right part is named part 2. Four windows are created for the whole test. Two windows are created for each part of the text, to search for the pattern in parallel. The left and right windows of part 1 are named p_{1L} and p_{1R} respectively. The left and right windows of part 2 are named p_{2L} and p_{2R} respectively. At the beginning of the search, p_{1L} and p_{2R} windows are aligned with the left most and rightmost sides of the text. p_{2L} window is aligned

with the text at index $n/2+m-1$ while p_{1R} window is aligned with the text at index $n/2-1$ where n is the text length and m is the pattern length. The alignments of p_{2L} and p_{1R} are calculated taking into consideration the case where some characters of the pattern may appear in part1 of the text and the rest may appear in the second part of the text.

Figure 1 explains the algorithm of the FSW algorithm.

```
L1 = m − 1; //text index used from left in part 1
R1 = n/2 − 1; //text index used from right in part 1
L2 = n/2 + m − 1; //text index used from left in part 2
R2 = n − m; //text index used from right in part 2
T index = 0; //text index used to control the scanning process
While (T index <= ⌈ n/4⌉ )
  begin
    found Part 1 Left = false.
    found Part 1 Right = false.
    found Part 2 Left = false.
    found Part 2 Right = false.
     l1 = m − 1; // pattern index used at left side of part 1
     r1 = 0; // pattern index used at right side of part 1
     l2 = m − 1; // pattern index used at left side of part 2
     r2 = 0; // pattern index used at right side of part 2
     //keep record of the text index where the pattern match the text during comparison
     temp-l index 1 = temp-r index 1 = 0, temp-l index 2 = temp-r index 2 = 0;
     templ = 0; tempr = m − 1;
     if (P[m − 1] = T[L1] and P[0] = T[L1 − m + 1]) //search from left of part 1
     begin
                temp-l index 1 = L1
                L1 = L1 − 1
                templ++
                while ((l1 >= 0 and P[l1] = T[L1]) and (P[templ] = T[L1 − l1 + templ]))
                {L1 = L1 − 1, l1 = l1 − 1; templ++;
           if ((L1 − l1 + templ) >= L1)
                {foundPart1Left = true; exit from while loop;}
                }
     end
     if (P[0] = T[R1] and P[tempr] = T[R1 + m − 1]) //search from right of part 1

     begin
                temp-r index 1 = R1
                R1 = R1 + 1
                tempr--;
                while ((r1 < m and P[r1] = T[R1]) and P[tempr] = T[R1 + tempr − r1])
                {R1 = R1 + 1, r1 = r1 + 1; tempr--;
if (R1 + tempr − r1 <= R1)
                {foundPart1Right = true; exit from while loop;}
                }
     end
     templ = 0; tempr = m − 1;
     if (P[m − 1] = T[L2] and p[0] = T[L2 − m + 1]) //search from left of part 2

     begin
                temp-l index 2 = L2
                L2 = L2 − 1
                templ++
                while ((l2 >= 0 and P[l2] = T[L2]) and (P[templ] = T[L2 − l2 + templ]))
                {L2 = L2 − 1, l2 = l2 − 1; templ++;
           if ((L2 − l2+ templ) >= L2)
                {found Part 2 Left = true; exit from while loop;}
                }
     end
     if (P[0] = T[R2] and P[tempr] = T[R2 + m − 1]) //search from right of part 2

     begin
                temp-r index 2 = R2
                R2 = R2 + 1
                tempr--;
                while((r2 < m and P[r2] = T[R2]) and P[tempr] = T[R2 + tempr − r2])
                {R2 = R2 + 1, r2 = r2 + 1; tempr--;
if (R2 + tempr − r2 <= R2)
                {found Part 2 Right = true; exit from while loop;}
                }
     end

     if (found Part 1 Left) {display "match at left of part 1:" +L1 + 1); exit from outer loop;}
     if (found Part 1 Right) {display "match at right of part 1:" +R1 − m); exit from outer loop;}
     if found Part 2 Left) {display "match at left of part 2:" +L2 + 1); exit from outer loop;}
     if found Part 2 Right) {display "match at right of part 2:" +R2 − m); exit from outer loop;}
     //To avoid skipping characters after partial matching
     L1 = temp-l index 1; R1 = temp-r index 1;
     L2 = temp-l index 2; R2 = temp-r index 2;
     if (L1 > R1){ display ("not found"); exit from outer loop;}
     //from pre-processing step
     L1 = L1 + get (shiftl); R1 = R1 − get (shiftr);
     L2 = L2 + get (shiftl); R2 = R2 − get (shiftr);
     T index = Tindex + 1;

  End
```

Figure 1. FSW pattern matching algorithm.

3.3. Working Example

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

Given:

Pattern (P) = "abcd", $m = 4$,

Text (T) = "abaccbacdacdbadcbaadcbbcacbbcaaddcaabcbaaacbddababcdddabdaabaabccdabccdbacbdcbcdaccdbcbddaadddbcabdb", $n = 100$.

3.3.1. Pre-Processing Phase

Initially, $shiftl1 = shiftr1 = shiftl2 = shiftr2 = m + 2 = 6$.

The shift values are calculated using equations 1 and 2. The values are then stored in two arrays *nextl* and *nextr* as shown in **Figure 2(a)** and **Figure 2(b)** respectively.

3.3.2. Searching Phase

The searching process for the pattern *P* is illustrated through the working example as shown in **Figure 3**.

***First attempt*:** (see **Figure 3(b)**)

We align p_{1L} with the text from the left of part 1. In this case, comparisons are made between the text character located at index 0 (character a) with the leftmost character in the pattern (character a). At the same time, comparisons are made between the text character at index 3 (character c) with the rightmost character in the pattern (character d). As a result, a mismatch occurs between text character c and pattern character d; therefore we take the two consecutive characters from the text at index 4 and 5 which are c and b respectively. To determine the amount of shift (*shiftl*) we have to do the following two steps:

a) We look for the index of cb in the pattern.

b) Since cb is not found in the pattern, so the window is shifted to the right 6 steps (see Equation (1)).

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

***Second attempt*:** (see **Figure 3(c)**)

We align p_{1R} with the text from the right of part 1. In this case, a match occurs between the text character at index 52 (d) and the rightmost character in the pattern d while there is a mismatch between the text character at index 49 (b) and the leftmost character in the pattern a; therefore we take the two consecutive characters from the text at index 47 and 48 which are b and a respectively. To determine the amount of shift (*shiftr*), we have to do the following two steps:

a) We look for the index of ba in the pattern.

b) Since ba is not found in the pattern, but the $p[0]$ which is a matches the text character at index 48 then according to the pre-processing phase the sliding window will be shifted 1 step to the left.

***Third attempt*:** (see **Figure 3(d)**)

We align p_{2L} with the text from the left of part 2. In this case, a mismatch occurs between the text character at index 50 (c) and the leftmost character in the pattern a while there is a match between the text character at index 53 (d) and the rightmost character in the pattern d; therefore we take the two consecutive characters from the text at index 54 and 55 which are a and b respectively. To determine the amount of shift (*shiftl*) we have to do the following two steps:

a) We look for the index of ab in the pattern, which is found 0.

Index	**0**	**1**	**2**
nextl	**4**	**3**	**2**

(a) Shift values from the left

Index	**0**	**1**	**2**
nextr	**2**	**3**	**4**

(b) Shift values from the right

Figure 2. The *nextl* and *nextr* arrays.

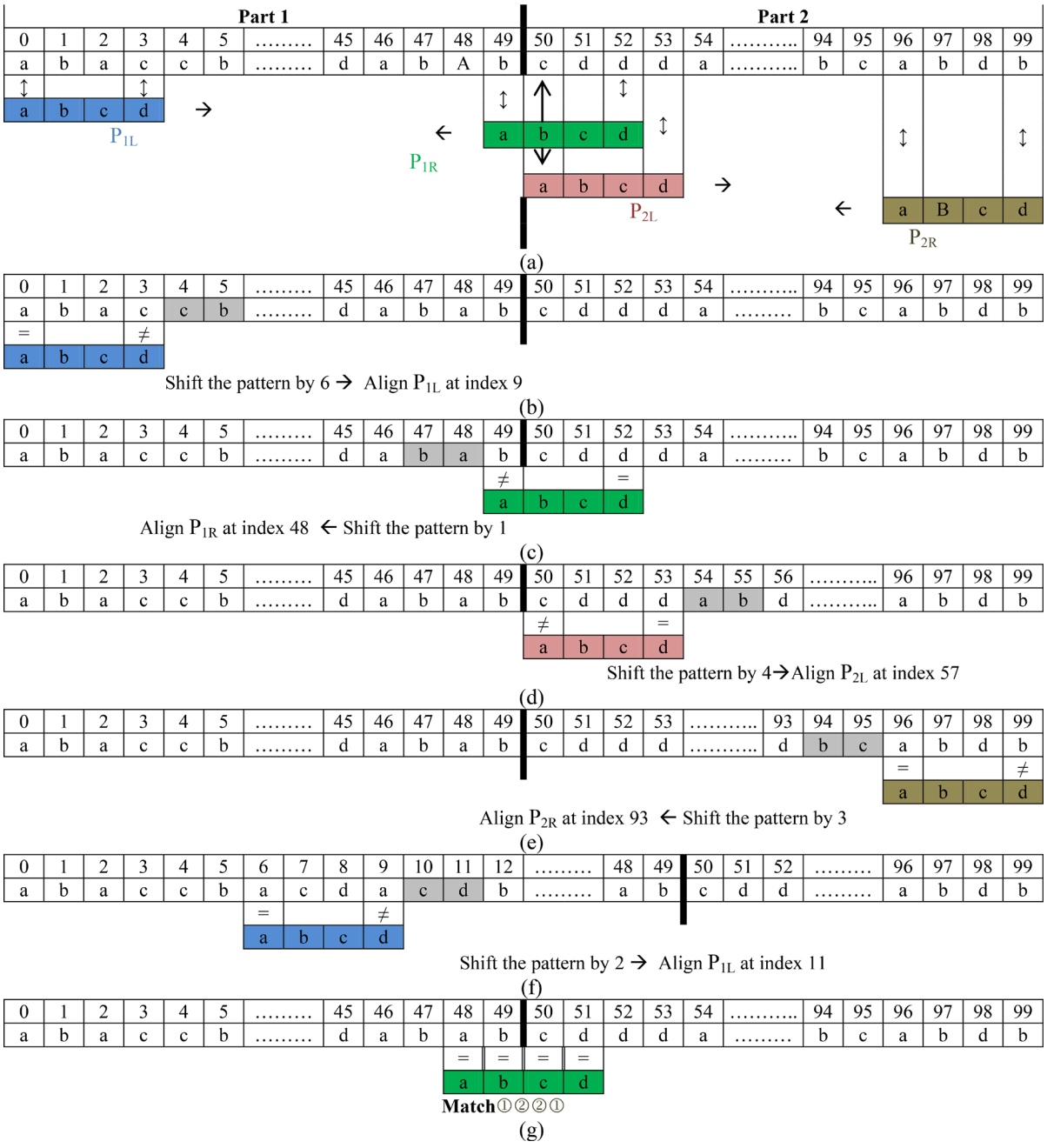

Figure 3. Working example.

b) Since we search from the left side we use *nextl* array, and *shiftl* = *nextl* [0] = 4.

Therefore the window will be shifted to the right 4 steps.

***Fourth attempt*:**

In the fourth attempt (see **Figure 3(e)**), we align the fourth sliding window with the text from the right of part 2. In this case, a mismatch occurs between the text character at index 99 (b) and the rightmost character in the pattern (character d) while there is a match between the text character at index 96 (character a) and the leftmost character in the pattern (character a); therefore we take the two consecutive characters from the text at index 94 and 95 which are b and c respectively. To determine the amount of shift (*shiftr*), we have to do the following two steps:

a) We look for the index of bc in the pattern.

b) Since we search from the right side, we use *nextr* array, and *shiftr* = *nextr* [1] = 3.
Therefore the window is shifted to the left 3 steps.

***Fifth attempt*:**

In the fifth attempt (see **Figure 3(f)**), we align the first sliding window with the text from the left of part 1. In this case, a match occurs between the text character at index 6 (a) and the leftmost character in the pattern (character a) while there is a mismatch between the text character at index 9 (character a) and the rightmost character in the pattern (character d); therefore we take the two consecutive characters from the text at index 10 and 11 which are c and d respectively,

a) We look for the index of cd in the pattern.
b) Since we search from the left side we use *nextl* array, and *shiftl* = *nextl* [2] = 2.
Therefore the window is shifted to the right 2 steps.

***Sixth attempt*:**

In the sixth attempt (see **Figure 3(g)**), we align the second sliding window with the text from the right of part 1. A comparison between the pattern and the text characters leads to a complete match at index 48. In this case, the occurrence of the pattern is found using the right window of part 1.

4. Analysis

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

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

Lemma 1: The worst case time complexity is $O\left(\left(\frac{n}{2}-\frac{m}{4}+1\right)\left(\frac{m}{4}\right)\right)$.

Proof: The worst case occurs when at each attempt, all the compared characters of both pattern sides at 4 windows that slide simultaneously are matched the corresponding text characters except the pattern character indexed (*m*), and at the same time the shift value is equal to 1.

Lemma 2: The best case time complexity is $O\left(\frac{m}{4}\right)$.

Proof: The best case occurs when the pattern is found at the first index $\frac{n}{2}-1$, $\frac{n}{2}+m-1$, or at the last index ($n - m$), in this case the number of comparisons made to compare *m* pattern characters are $\frac{m}{4}$.

Lemma 3: The average case time complexity is $\left(\frac{n}{4m}\right)$.

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) for each window from 4 available windows.

5. Results

In order to ensure that the FSW algorithm gives extraordinary results in the searching process, several experiments were performed. The FSW algorithm searches the text using four sliding windows. All the windows slide in parallel. Comparisons done with the pattern is also done from both sides simultaneously. **Tables 1-5** as well as **Figures 4-7** show the results of comparing FSW with ETSW, TSW and BR algorithms.

Table 1, **Figure 4** and **Figure 5** show the average number of attempts and comparisons for patterns with different lengths. It is noticeable that the number of comparisons and attempts in FSW is much better than the others. This is because in FSW four windows are used while in both ETSW and TSW algorithms two sliding windows are used. On the other hand, BR algorithm uses only one sliding window. For example, if the text has 1167 words, each of length 8, then the average number of comparisons and attempts made by FSW is 3577 and 3502 respectively. The number of comparisons and attempts made by ETSW is 10115 and 10056 respectively. Looking at **Table 1**, the number of comparisons and attempts of TSW and BR are also greater than FSW. This makes FSW algorithm better than the other algorithms in terms of the average number of comparisons and attempts.

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

Pattern length	Number of words	FSW		ETSW		TSW		BR	
		Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	8103	2525	2488	3904	3875	3904	4213	6409	7039
5	4535	3133	2856	4456	3549	4456	4896	9577	10645
6	2896	3232	3252	7596	7633	7596	8311	10898	12173
7	1988	3723	3697	9341	9118	9341	10263	11953	13345
8	1167	3502	3577	10056	10115	10056	11087	13256	14807
9	681	3330	3350	9538	9590	9538	10538	14149	15892
10	382	3708	3822	9283	9339	9283	10272	14127	15799
11	191	3341	3363	5451	5482	5451	5967	12808	14243
12	69	3232	3255	6384	6433	6384	7168	9598	10923
13	55	4781	4807	7947	7986	7947	8673	10334	11370

Table 2. The average number of attempts and comparisons performed to search for (100) patterns selected from the middle of the text.

Pattern length	FSW		ETSW		TSW		BR	
	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	875	879	2726	2737	2726	2959	3645	4070
5	4706	4730	11582	11618	13965	15140	11558	12793
6	5152	5175	16682	16771	16682	18317	12878	14337
7	1895	1907	26104	26242	26104	30095	19547	22006
8	3511	3530	27830	28015	27830	30915	20831	23336
9	2021	2029	33929	34069	33929	37200	23284	25852
10	4152	4176	29676	29845	29676	32817	20546	22989
11	1333	1341	23195	23242	23195	24646	20264	22005
12	2413	2435	26806	27009	26806	30222	21113	24235

Table 3. The number of attempts and comparisons performed to search for a set of patterns that do not exist in the text.

Pattern length	FSW		ETSW		TSW	
	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	135592	136887	136188	137630	136188	152137
5	116040	1173911	116644	118058	116644	130485
6	101652	102826	102076	103338	102076	113994
7	90480	91359	90854	91798	90854	101469
8	81472	82381	81700	82691	81700	91419
9	74080	74886	74326	75157	74326	83085
10	68012	68698	67984	68722	67984	75863
11	62648	63220	62738	63405	62738	70012
12	58220	58816	58412	59073	58412	65315

Table 4. The average number of attempts and comparisons performed to search for (100) patterns selected from the beginning of the text.

Pattern length	FSW		ETSW		TSW		BR	
	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	278	280	143	145	143	157	76	85
5	359	361	185	187	185	206	100	115
6	443	448	227	230	227	255	121	142
7	686	691	347	351	347	388	195	226
8	967	975	504	510	504	568	270	310
9	1340	1349	670	677	670	750	363	417
10	2269	2285	1160	1170	1160	1290	640	727
11	1243	1251	622	628	622	705	331	396
12	1729	1747	865	878	865	972	478	557

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

Pattern length	FSW		ETSW		TSW		BR	
	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons	Attempts	Comparisons
4	251	253	133	135	133	148	6899	7719
5	508	510	268	270	268	297	12930	14404
6	716	720	364	368	364	402	21315	23957
7	792	797	402	405	402	447	22237	24731
8	1056	1063	536	541	536	592	21495	23841
9	1489	1498	776	783	776	859	24919	28257
10	3047	3067	1579	1593	1579	1756	31603	35360
11	1238	1244	619	624	619	669	32797	36438
12	3269	3296	1667	1685	1667	1872	30928	35069

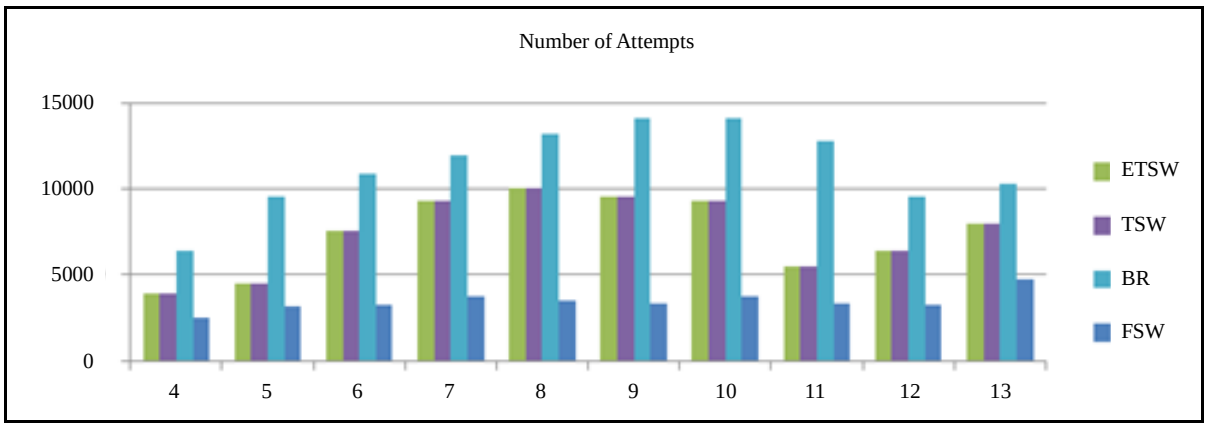

Figure 4. The average number of attempts for patterns with different lengths.

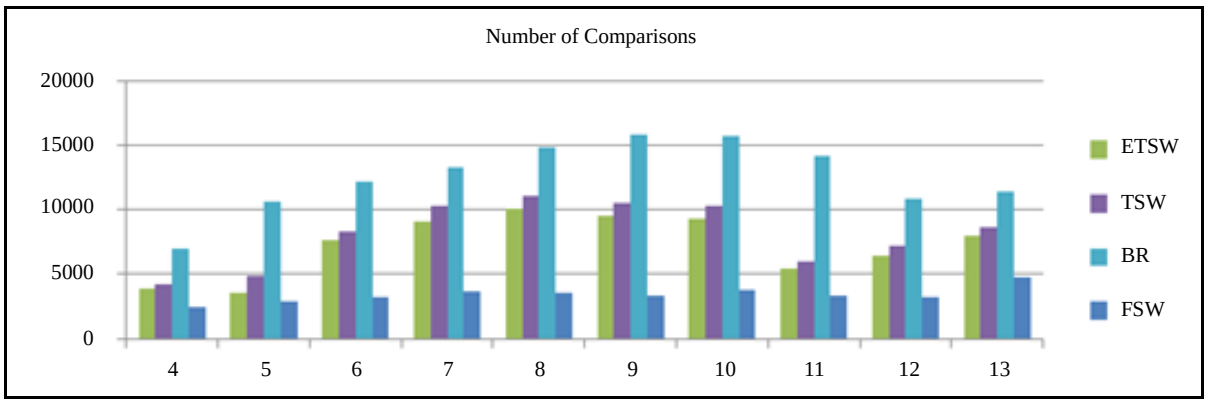

Figure 5. The average number of comparisons for patterns with different lengths.

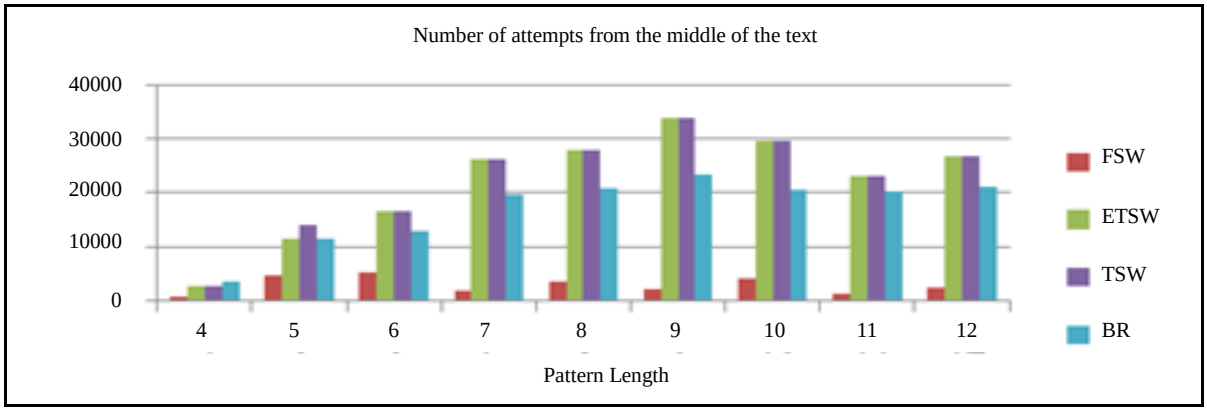

Figure 6. The average number of attempts performed to search for (100) patterns from the middle of the text.

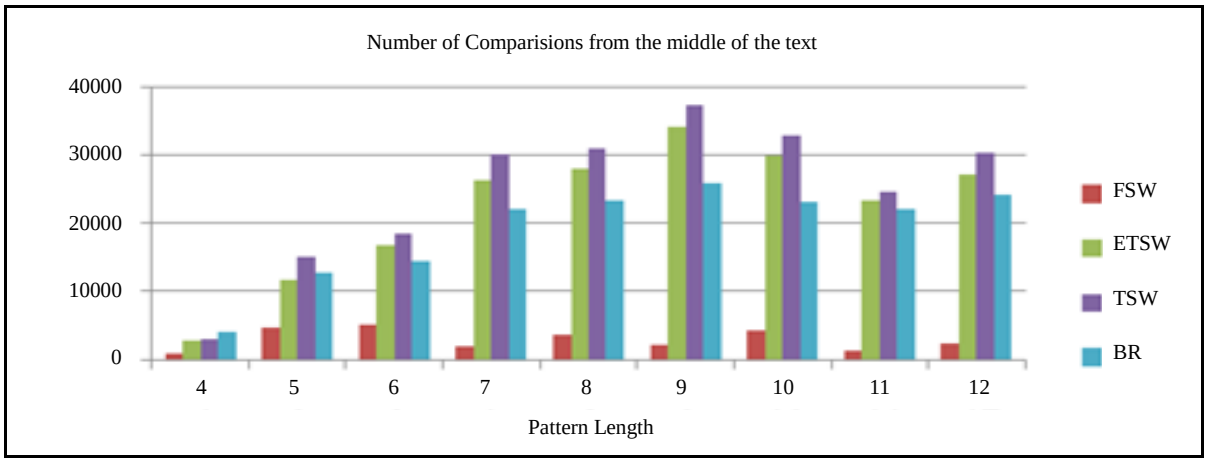

Figure 7. The average number of comparisons performed to search for (100) patterns from the middle of the text.

Table 2, **Figure 6** and **Figure 7** show the average number of attempts and comparisons performed to search for 100 patterns selected from the middle of the text. FSW algorithm shows the best results in both number of comparisons and attempts.

This is expected since FSW search the text using four windows, two of them starts from the middle of the text. On the other hand, ETSW and TSW uses two windows aligned at the rightmost and the leftmost sides of the text.

BR algorithm uses only one sliding window starting from the left of the text.

For example, to search for a pattern of length 4, the average number of comparisons and attempts made by FSW is 879 and 875 respectively. Compared to ETSW, TSW and BR, the results are far better in FSW than in the other algorithms.

FSW algorithm has the minimum number of comparisons and attempts performed to search for patterns of different lengths that are not found in the text as shown in **Table 3**.

Table 4 show the average number of comparisons and attempts performed to search for 100 patterns selected from the beginning of the text. BR algorithm has the minimum number since it searches the text using only one window from the left, *i.e.* from the beginning of the text. On the other hand, ETSW and TSW use two sliding windows that slide from the left side and the right side of the text which increases the number compared to BR algorithm. FSW algorithm's results show that there is an increase in the number of comparisons and attempts performed especially if the pattern is found in the middle of the text. This is expected since four sliding widows are used.

Table 5 show the average number of comparisons and attempts performed to search for 100 patterns selected from the end of the text. The results of FSW are reasonable since the pattern is found at the end of the text. BR on the other hand performed a large number of comparisons and attempts since it searches the text starting from the left side of the text.

6. Conclusions

In this paper, we presented a new pattern—matching algorithm (FSW) which finds all occurrences of a given pattern *p* in a given text *t* using four sliding windows. The new algorithm enhances the ETSW algorithm which uses only two sliding windows. Extensive experiments have been conducted. The results show that FSW best performance appears when the pattern is found in the middle of the text. If the pattern is in the beginning or the end of the text, the number of comparisons and attempts in FSW increases compared to other algorithms.

Using four sliding windows that search the text in parallel as well as comparing the pattern from both sides simultaneously makes the FSW performance better and decreases the searching time. In the future we intend to apply the FSW algorithm to additional applications such as computational biology and search engines. Also we intend to use threads to implement the FSW algorithm.

References

[1] Simone, F. and Thierry, L. (2013) The Exact Online String Matching Problem: A Review of the Most Recent Results. *ACM Computing Surveys*, **45**, 13.

[2] Yang, Z., Yu, J. and Kitsuregawa, M. (2010) Fast Algorithms for Top-k Approximate String Matching. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, Atlanta, 11-15 July 2010.

[3] Pendlimarri, D. and Petlu, P.B.B. (2010) Novel Pattern Matching Algorithm for Single Pattern Matching. *International Journal on Computer Science and Engineering*, **2**, 2698-2704.

[4] Bhukya, R. and Somayajulu, D. (2011) Article: Exact Multiple Pattern Matching Algorithm Using DNA Sequence and Pattern Pair. *International Journal of Computer Applications*, **17**, 32-38. http://dx.doi.org/10.5120/2239-2862

[5] Alsmadi, I. and Nuser, M. (2012) String Matching Evaluation Methods for DNA Comparison. *International Journal of Advanced Science and Technology*, **47**, 13-32.

[6] Bhandaru, J. and Kumar, A. (2014) A Survey of Fast Hybrid String Matching Algorithms. *International Journal of Emerging Sciences*, **4**, 24-37.

[7] Linhai, C. (2014) An Innovative Approach for Regular Expression Matching Based on NoC Architecture. *International Journal of Smart Home*, **8**, 45-52. http://dx.doi.org/10.14257/ijsh.2014.8.1.06

[8] Diwate, R. and Alaspurkar, S. (2013) Study of Different Algorithms for Pattern Matching. *International Journal of Advanced Research in Computer science and Software Engineering*, **3**, 615-620.

[9] Singla, N. and Garg, D. (2012) String Matching Algorithms and Their Applicability in Various Applications. *International Journal of Soft Computing and Engineering* (*IJSCE*), **I**, 218-222.

[10] Guo, L., Du, S., Ren, M., Liu, Y., Li, J., He, J., Tian, N. and Li, K. (2013) Parallel Algorithm for Approximate String Matching with K Differences. *IEEE Eighth International Conference on Networking, Architecture and Storage*, 17-19 July 2013, 257-261. http://dx.doi.org/10.1109/NAS.2013.40

[11] Itriq, M., Hudaib, A., Al-Anani, A., Al-Khalid, R. and Suleiman, D (2012) Enhanced Two Sliding Windows Algorithm for Pattern Matching (ETSW). *Journal of American Science*, **8**, 607- 616.

[12] Hudaib, A., Al-Khalid, R., Suleiman, D., Itriq, M. and Al-Anani, A, (2008) A Fast Pattern Matching Algorithm with Two Sliding Windows (TSW). *Journal of Computer Science*, **4**, 393-401. http://dx.doi.org/10.3844/jcssp.2008.393.401

[13] Suleiman, D. (2014) Enhanced Berry Ravindran Pattern Matching Algorithm (EBR). *Life Science Journal*, **11**, 395-402.

[14] Berry, T. and Ravindran, S. (2001) A Fast String Matching Algorithm and Experimental Results. In: Holub, J. and Simanek, M., Eds., *Proceedings of the Prague Stringology Club Workshop*'99, Collaborative Report DC-99-05, Czech Technical University, Prague, 16-26.

[15] Khan, Z. and Pateriya, R.K. (2012) Multiple Pattern String Matching Methodologies: A Comparative Analysis. *International Journal of Scientific and Research Publications*, **2**, 2250-3153.

[16] Claude, F., Navarro, G., Peltola, H., Salmela, L. and Tarhio, J. (2012) String Matching with Alphabet Sampling. *Journal of Discrete Algorithms*, **11**, 37-50. http://dx.doi.org/10.1016/j.jda.2010.09.004

[17] Zhang, P. and Liu, J. (2011) An Improved Pattern Matching Algorithm in the Intrusion Detection System. *Applied Mechanics and Materials*, **48-49**, 203-207. http://dx.doi.org/10.4028/www.scientific.net/AMM.48-49.203

[18] Faro, S. and Lecroq, T. (2012) A Multiple Sliding Windows Approach to Speed Up String Matching Algorithms. SEA, 172-183.

[19] Suleiman, D., Hudaib, A., Al-Anani, A., Al-Khalid, R. and Itriq, M. (2013) ERS-A Algorithm for Pattern Matching. *Middle East Journal of Scientific Research*, **15**, 1067-1075.