# New and Efficient Recursive-based String Matching Algorithm (RSMA-FLFC)

Jehad Q. Odeh
Sabbatical Leave from Al al-Bayt University
Computer Science Department
Faculty of Information Technology,
Al al-Bayt University, Jordan

## ABSTRACT
The need for simple and efficient string matching algorithms is essential for many applications, and especially for database query. In this paper, two major algorithms are proposed, namely first least frequency character algorithm (FLFC) and recursive-based string matching algorithm (RSMA). FLFC is considered as an enhanced version of scan for lowest frequency character SLFC proposed by Horspool [12]. FLFC algorithm extracts first least frequency character in the pattern and identifies the occurrences of such character in the whole text in a preprocessing phase, while the recursive algorithm (RSMA) recursively partitioning the pattern and the targeted substring in the text and compares them at mid-point (q) each time. The FLFC search accelerates the searching process, while RSMA enhances the speed of performance of the matching phase. The extensive testing and comparisons with Naïve (Brute force), Boyer-Moore (BM), and the FLFC without deploying recursive matching show that the proposed algorithms enhance the speed of performance dramatically.

## General Terms
Exact String Matching

## Keywords
Recursive string matching, brute force, Boyer-Moore, least frequency characters

## 1. INTRODUCTION
String matching is crucial to many applications including database query, DNA and protein sequence analysis. The efficiency of string matching has a great impact on the performance of these applications [1]. String matching algorithms are classified into either exact string matching or in-exact (approximate) string matching. In [2, 3], they define exact string matching problem as identifying one or more of the occurrences of a pattern *P* of length *m* in a text *T* of length *n*. Tremendous number of techniques and algorithms has been proposed to tackle this problem. These algorithms have extensive use in information retrieval, bibliographic search, and molecular biology. Among the most cited papers on approximate string matching are the articles [4, 5] by Esko Ukkonen as mentioned in [6], they define approximate string matching problem as if we have a pattern P[1...m] of m characters drawn from an alphabet Σ of size σ, a text T[1...n] of n characters over the same alphabet, and an integer k. We need to find all such positions i of the text that the distance between the pattern and a substring of the text ending at that position is at most k. In the k-difference problem the distance between two strings is the standard edit distance where substitutions, deletions, and insertions are allowed.

It is essential in any information retrieval and text-editing applications to be able to locate efficiently the recurrences of a user-specified pattern of words and phrases in a text [7]. Efficiency is crucial to any string matching technique, since the problem of searching a huge block of text to allocate the first occurrence of the pattern or even all occurrences can be overwhelming. Naïve string matching techniques requires a worst running of *O(mn)*, where *m is* the length of pattern and *n* is length of text.

This paper is organized as follows: section 2 presents the literature review, section 3 introduces the proposed algorithm, section 4 shows experimental results, and section 4 draws the conclusion.

## 2. LITERATURE REVIEW
There are many algorithms classified as exact string matching algorithms. Naïve (brute force) algorithm, Boyer and Moore (1977), Morris and Pratt (Watson, 2002) and Knuth-Morris-Pratt (1977), have been presented as exact string matching algorithms to solve the problem of searching for a single pattern in a text. The brute force algorithm checks all positions in the text between 0 and n – m without any consideration to pattern's occurrence position. Then, after each attempt, it shifts the pattern by exactly one position to the right. The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. The time complexity of the searching phase is O(mn), where m is the length of the pattern and n is the length of the text [8]. Beginning with the rightmost character of the pattern Boyer-Moore algorithm scans the characters of the pattern from right to left [9]. If a complete match of the whole pattern is occurred or a mismatch it uses two pre-computed functions to shift the window to the right. These two shift functions are called the good-suffix shift and bad-character shift. Assume that a mismatch occurs between the character x[i] = a of the pattern and the character y[i+j] = b for the text during an attempt at position j. Then, x[i + 1 .. m – 1] = y[i + j + 1 .. j + m – 1] = u and x[i] ≠ y[i + j]. The good-suffix shift consists in aligning the segment y[i + j + 1 .. j + m – 1] = x[i + 1 .. m – 1] with its rightmost occurrence in x that is preceded by a character different from x[i]. If there exists no such segment, the shift consists in aligning the longest suffix v of y[i + j + 1 .. j + m – 1] with a matching prefix of x. The bad-character shift aligning the text character y[i + j] with its rightmost occurrence in x[0 .. m – 2]. Knuth-Morris-Pratt algorithm has better worst-case running time than the Boyer-Moore algorithm in spite of that the latter is known to be extremely efficient in practice [1, 14]. As mentioned in [10, 11], the extensive pattern-matching literature has had two main categories: decreases the number of character comparisons required and reducing the time requirement in the worst and average cases. In [12], Horspool presented SFC (Scan for First Character) and

SLFC (Scan for Lowest Frequency Character) approach in which searching is based on the occurrences of the first character of the pattern in the text. He shows that a reasonable reduction in number of characters that are skipped before finding the lowest frequency characters in the pattern is achieved. In this research paper a new and efficient exact string matching algorithms are proposed. The efficiency came through modifying the original SLFC algorithm presented by [12] to utilize the extracted information in the scanning phase as an input to the newly proposed recursive string matching algorithm RSMA. The major problem of the original SLFC algorithm presented by Horspool is that the matching process is naïve and time consuming process, especially for long patterns with similar characters (i.e. DNA strings).

The proposed algorithms were implemented, analyzed, and tested. RSMA-FLFC is compared with Brute force algorithm, Boyer-Moore and the FLFC without deploying recursive matching. The results of extensive testing showed significant enhancement in performance. Moreover, the new approach can be adopted by any well known algorithm in string matching.

## 3. RSMA-FLFC ALGORITHMS

Since most of string matching algorithms search for the pattern in the whole text, and match most of the text's characters with the pattern's characters, it is reasonable to assume that it will be more efficient to match the pattern with the sub-strings of the text in a specific locations in the text, these locations identified in a precise way to be the only candidate locations in which matching may occur, while ignoring the rest of the characters in the text. To accelerate the searching and matching process it is beneficial to make use of the well-known frequency of characters in English, utilizing least frequency character (LFC) efficiently reduces candidate targeted substrings in the text as proven by [12].

The proposed algorithms handle the problem of exact string matching in two phases. Pre-processing phase in which the identification of all positions of the first least frequency character in the pattern is achieved and saved for later use in the processing phase. In [13], they did a comprehensive analysis to the letters occurring in the words listed in the main entries of the Concise Oxford Dictionary (11th edition revised, 2004). The results of letters analysis is shown in table 1. In this research the same table has been sorted for the purpose of string comparison to identify the LFC in the pattern.

**Table 1. English character frequency**

| E | A | R | I | O | T |
|---|---|---|---|---|---|
| 11.16% | 8.49 % | 7.58 % | 7.54 % | 7.16 % | 6.95% |
| **N** | **S** | **L** | **C** | **U** | **D** |
| 6.65% | 5.73% | 5.48 % | 4.53 % | 3.63 % | 3.38% |
| **P** | **M** | **H** | **G** | **B** | **F** |
| 3.16% | 3.01% | 3.00 % | 2.47 % | 2.07 % | 1.81% |
| **Y** | **W** | **K** | **V** | **X** | **Z** |
| 1.77% | 1.28% | 1.10 % | 1.00 % | 0.29 % | 0.27% |
| **J** | **Q** | | | | |
| 0.1965 % | 0.1962 % | | | | |

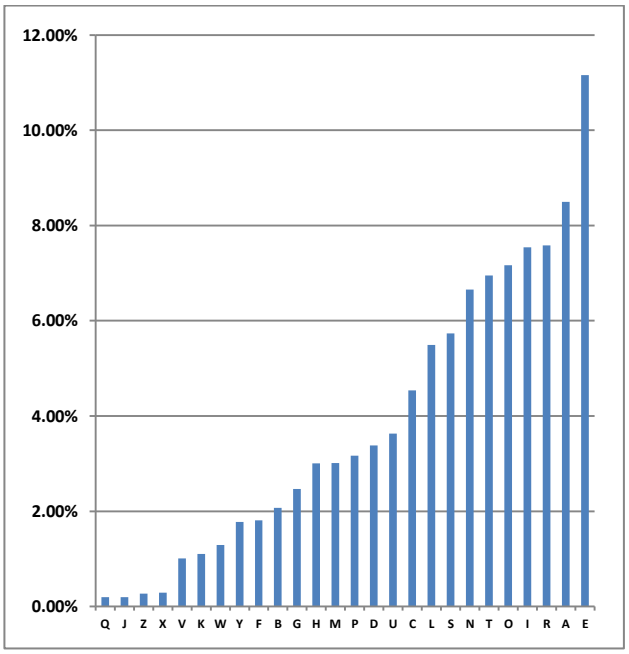


**Fig 1: Sorted English character frequency**

In this research, sorted version of table 1 shown in figure 1 is utilized to identify the first least frequency character in the pattern and scan the text to specify the candidate positions for later use in the processing phase. The proposed processing phase algorithm (RSMA) relies on the output of the pre-processing, then it applies a simple shifting technique to specify the position of the first character in the text's substring opposite to the first character in the pattern. After that, it identifies the position of the mid-point (q) and compares the substring with the pattern at that specific position.

If the two characters are similar then the algorithm continues recursively till comparing all characters in the pattern with the candidate substring and in case of similarity or dissimilarity it proceeds to the second candidate position and so on.

## 3.1 Preprocessing phase

The pre-processing phase is used to identify all recurrences of the first LFC of the pattern in the whole text T[n]. Figure 2, presents the proposed FLFC algorithm.

**Assume:** T [1…n]: text of size n, P [1…m]: pattern of size m.
**FLFC-Algorithm**

1. **if** m > n **then return 0**
2. **else**
3. Find LFC (least frequency character) in the pattern and identify its index j.
4. i = j – 1
5. Do:
   - **Search** T [i + 1 ... n - m +1] to identify indices of LFC in Text.
   - **If** LFC was not found **then return 0.**
   - **Else create** an array $R[k]=\{v_1,v_2,...,v_k\}$, where $v_i, i=1,2,..,k$, represents the occurrences of the first LFC in T[n].
6. Repeat while true.
7. return R[k]

**Fig 2: FLFC algorithm**

## 3.2 Processing phase

The following figure 3 shows the proposed (RSMA) algorithm.

**Assume:** T [1…*n*]: text of size *n*, P [1…*m*]: pattern of size *m*.

**1.** For each $v \in V$, identify the $(k^{th})$ position in T[n].
   - *Align P[m] with T[n] substring based on first LFC.*
   - *Identify* $(j^{th})$ *position in P[m] opposite to* $(k^{th})$ *position in T[n].*
   - *Identify* $(i^{th})$ *position in T[n] by shifting to the left of* $(k^{th})$ *by* $(j^{th})$ *-1.*

**If** $(i^{th}) \geq 1$ **then go to step 2.**
***else*** *next for.*

**2. RSMA(T[n],P[m], i)**

a) If $n \leq 2$ then compare the two substrings directly
**else**

- Let z, $P\left[\left\lfloor \frac{1+m}{2} \right\rfloor\right] = \beta$

b) If $T[q] == \beta$ then

- **RSMA** $\left(T\left[i \ \ldots \ \left\lfloor \frac{2i+m-1}{2} \right\rfloor\right], P\left[1 \ \ldots \ \left\lfloor \frac{1+m}{2} \right\rfloor\right], \ i\right)$ // left recursion
- **RSMA** $\left(T\left[\left\lfloor \frac{2i+m-1}{2} \right\rfloor \ \ldots \ i+m-1\right], P\left[\left\lfloor \frac{1+m}{2} \right\rfloor \ \ldots \ m\right], \ \left\lfloor \frac{2i+m-1}{2} \right\rfloor\right)$ // right recursion
  - **PRINT** ("The pattern is found at $(i^{th})$ position ")

**else**
return to 1.

**Fig 3: RSMA algorithm**

Figure 4, shows how to specify $(k^{th})$ position based on the recurrences of character shown in Table 1. Specifying the $(k^{th})$ position is done by FLFC-Algorithm**.**

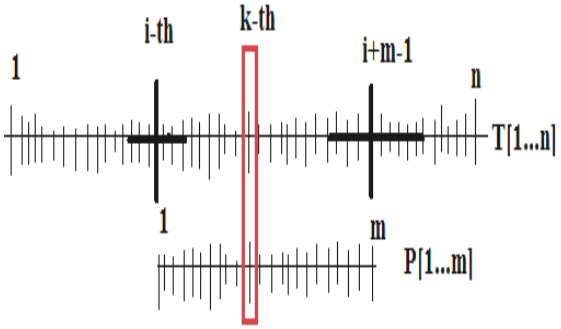


**Fig 4: Specifying the first LFC**

Figure 5, visualize the way in which RSMA partitioning the pattern by identifying the *q* value.

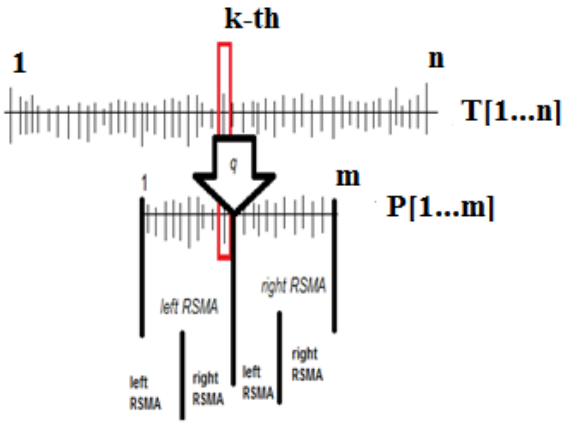


**Fig 5**: **Partitioning the pattern by identifying mid-point (q)**

## 3.3 Analysis of proposed algorithms

**FLFC**: this algorithm concerns about determining and saving the occurrences of the LFC in T(n). Step (3) in the proposed algorithm is done through comparing the sorted version of Table 1 with the pattern not the opposite since this will reduce the running time dramatically. The characters in Table 1 are sorted in ascending order and saved as an off-line process for later use, so the character (Q) is the first character to be

compared with the pattern. In this case, the worst running time **O**(26m) is occurred when all characters in the pattern are (E's), where m is the length of the pattern. Accordingly, running time complexity **O**(1) is registered when the first character in the pattern is (Q.

In step (5), the algorithm searches the text to identify the candidate positions. The worst case when (i) is close to (0) which means there is no satisfying reduction or exclusion of characters in the text. In such situation, the time complexity will be **O**(n-m), where n is the length of the text. While the best case when (i) is close to (n-m) in which the reduction in the text's searching characters is maximized, so the best case running time complexity is **O**(1). As a result, the overall worst case running time complexity of FLFC is **O**(n-m) and the best case running time complexity is **O**(1).

**RSMA:** this algorithm depends completely on the number of elements in (V) which is passed by FLFC algorithm. The loop in step (1) executes V times, where V is the set of all indices represent the occurrences of the first LFC in T(n). Moreover, step (2) executes V times, so the worst running scenario is when $T[q] == \beta$ is true for all elements in the candidate substrings of the text except the first or the last element. In such case, the worst running time complexity is O(V(m-1)). While the best case running time complexity **O**(V) is occurred when each time $T[q] \neq \beta$. Since V represents the occurrences of the pattern's first LFC in T(n) it is obvious that V will be dramatically less than n, so the efficiency of the proposed algorithm is better than most string matching algorithms close to **O**(nm) worst case complexity.

## 4. EXPERIMENTAL RESULTS AND DISCUSSION

In order to evaluate the efficiency of the proposed string matching algorithms (RSMA-FLFC) two major experiments have been conducted with different algorithms.

### 4.1 First Experiment

The tested algorithms are brute force (BF), Boyer-Moore (BM), and RSMA-FLFC (RSMA). The different algorithms have been coded in C in a consistent way and compiled with gcc with full optimization option. The machine used for testing purpose has an Intel(R) Core(TM) i5 processor at 3.00 GHZ running window7.

**Data set 1:** The first set of test data is a natural language file (world192.txt) of the Large Canterbury Corpus, available at: http://corpus.canterbury.ac.nz/descriptions/large/world.html. Its size is 2,473,400 bytes.

**Data set 2:** The second set of test data is the King James of the English Bible (bible.txt), the file was downloaded from Large Canterbury Corpus, available at: http://corpus.canterbury.ac.nz/descriptions/large/world.html. Its size is 4,047,392 bytes. In this experiment, selected short length patterns (5, 10, 15, 20, and 25) and long patterns (32, 64, 128, 256, and 512) have been considered. For each length 200 patterns selected randomly from the text and the results are averaged. The time is shown in milliseconds (*ms*).

#### *4.1.1 Test results for data set 1*

Table 2 shows the average execution time in milliseconds for each pattern sample utilizing RSMA, BM, and BF.

**Table 2: Results of tested algorithms using data set 1**

| Pattern Length | RSMA | BM | BF |
|---|---|---|---|
| 5 | 680 | 1120 | 10100 |
| 10 | 505 | 590 | 10250 |
| 15 | 356 | 475 | 10300 |
| 20 | 253 | 388 | 10325 |
| 25 | 243 | 320 | 10540 |
| 32 | 220 | 260 | 10500 |
| 64 | 213 | 253 | 11040 |
| 128 | 225 | 236 | 10567 |
| 256 | 220 | 257 | 10790 |
| 512 | 200 | 225 | 10990 |

#### *4.1.2 Test results for data set 2*

Table 3 shows the average execution time in milliseconds for each pattern sample utilizing RSMA, BM, and BF.

**Table 3: Results of tested algorithms using data set 2**

| Pattern Length | RSMA | BM | BF |
|---|---|---|---|
| 5 | 978 | 1792 | 12200 |
| 10 | 645 | 826 | 12355 |
| 15 | 577 | 712 | 12340 |
| 20 | 458 | 620 | 12365 |
| 25 | 405 | 415 | 13640 |
| 32 | 390 | 425 | 17500 |
| 64 | 411 | 413 | 18040 |
| 128 | 349 | 436 | 19567 |
| 256 | 325 | 389 | 18790 |
| 512 | 298 | 370 | 16990 |

Figure 6 compares between execution times of RSMA, and Boyer-Moore (BM) using data set 1 and data set 2. RSMA(1) denote to RSMA using data set 1, and RSMA(2) denote to RSMA using data set 2 and BM(1) denote to BM using data set 1, while BM(2) denote to BM using data set 2.

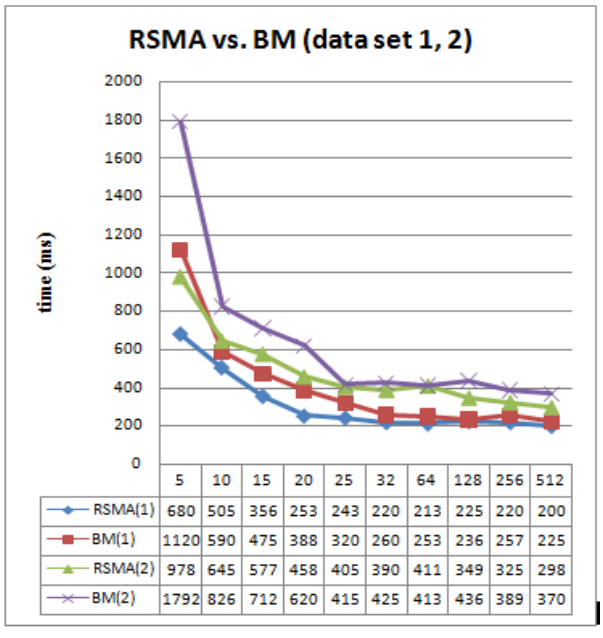


**Fig 6: RSMA vs. BM using data set 1 and data 2**

It is apparent that the RSMA algorithm performs well on the two set of data as compared to Boyer-Moore algorithm. Both algorithms show faster execution time with the long patterns as compared to the short patterns.

The size of the data set 2 is almost double the size of data set 1, which obviously affects the speed of performance of both algorithms as compared to data set 1. As shown in Figure 7, RSMA outperforms brute force dramatically on the different data sets. The size of data set 2 affects the speed of performance for both algorithms. Brute force with long patterns starting from 25, it shows slower execution time, especially with regards to data set 2.

Figure 7 compares between execution times of RSMA, and brute force (BM) algorithm using data set 1 and data set 2.

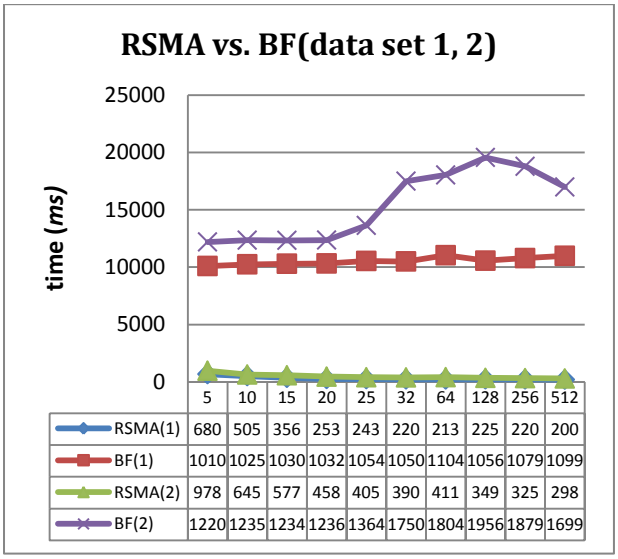


**Fig 7: RSMA vs. BF using data set 1 and data set 2**

## 4.2 Second Experiment

In this experiment the effect of the recursive part of the proposed algorithm was tested. Applying the LFC with and without the recursive part gives us a clear understanding of the importance of the proposed recursive technique. Figure 8, shows a modified version of RSMA called SMA$^{without}$ , its matching the string based on least frequency character LFC by comparing the whole pattern with the substring of the text excluding the recursive nature of RSMA.

**Assume:** T [1…*n*]: text of size *n*, P [1…*m*]: pattern of size *m*.

**1.** For each $v \in V$ , identify the $(k^{th})$ position in T[n].

- *Align P[m] with T[n] substring based on first LFC.*
- *Identify* $(j^{th})$ *position in P[m] opposite to* $(k^{th})$ *position in T[n].*
- *Identify* $(i^{th})$ *position in T[n] by shifting to the left of* $(k^{th})$ *by* $(j^{th})$ *-1.*

**If** $(i^{th}) \geq$ **1 then go to step 2.**

***else*** *next for.*

**2. SMA$^{without}$** *(T[n],P[m], i)*
*Compare the two substrings directly if they are similar*
*PRINT ("The pattern is found at* $(i^{th})$ *position ")*
*return to 1*
*else*
*return to 1*

**Fig 8: SMA$^{without}$ algorithm**

**Data set 3:** The data set is the genome (E.coli), the file was downloaded from Large Canterbury Corpus, available at http://corpus.canterbury.ac.nz/descriptions/large/world.html. Its size is 4,638,690 base pairs of *Escherichia coli*. In this experiment, selected short length patterns (5, 10, 15, 20, and 25) and long patterns (32, 64, 128, 256, and 512) have been considered. For each length 300 patterns selected randomly chosen from the text and the results are averaged. The time is shown in milliseconds (*ms*).

Table 4 shows the average execution time in milliseconds for each pattern sample utilizing RSMA, and SMA$^{without}$ algorithms. RSMA with genome behaves differently as compared to the first experiment, that due to the nature of data set 3, since genome consist of only four different characters (ACGT) and the size of data set is huge and close to the size of data set 2. So, the speed of performance of RSMA slightly decreases as the pattern length increases. Since, we have only limited number of different characters in the text the character repetition is potentially high. Consequently, many substrings of the text and the pattern will be dissimilar in just a limited number of positions that will increases the number of comparisons.

It is apparent that there is a dramatic enhancement - almost 50% - when recursive string matching is deployed as compared to the naïve way of matching even if the LFC technique is utilized.

**Table 4**: **Results of tested algorithms using data set 3**

| Pattern Length | RSMA | SMA$^{without}$ |
|---|---|---|
| 5 | 989 | 1800 |
| 10 | 1010 | 1820 |
| 15 | 1105 | 1905 |
| 20 | 1223 | 2100 |
| 25 | 1250 | 2106 |
| 32 | 1305 | 2210 |
| 64 | 1340 | 2320 |
| 128 | 1360 | 2370 |
| 256 | 1367 | 2413 |
| 512 | 1405 | 2470 |

## 5. CONCLUSION

This paper presented new simple and efficient single exact pattern matching algorithms. Namely, FLFC and RSMA. The proposed algorithms were implemented, analyzed, tested and compared with the naïve (brute force) algorithm and Boyer-Moore using different well-known data sets with different sizes. The different algorithms were tested using the same machine and hundreds of samples representing short and long patterns chosen randomly. The results were averaged, analyzed, and compared. The RSMA-FLFC algorithm enhances the execution time as compared to brute force and Boyer-Moore. Moreover, testing to measure the effectiveness of the proposed recursive string matching as compared to the

FLFC without deploying the recursive technique proves that applying FLFC is more beneficial if it merges with the recursive matching technique and the percentage of enhancement is close to 50%. The results were promising and the recursive matching approach can be utilized further in the future.

## 6. REFERENCES

[1] Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski and Rytter W. 1994. Speeding Up Two String Matching Algorithms. Algorithmica, 12(4-5): 247-267.

[2] Lecroq, T. 2007. Fast exact string matching algorithms. Journal of Information Processing Letter, 102: 229-235.

[3] Wu, Y.-C., J.-C. Yang and Y.-S Lee. 2007. A Weighted String Pattern Matching-Based Passage Ranking Algorithm for Video Question Answering. Journal of Expert Systems with Applications, 34: 2588-2600.

[4] Ukkonen, E. 1985. Algorithms for approximate string matching. Information and Control, 64(1-3) 100–118

[5] Ukkonen, E. 1985. Finding approximate patterns in strings. Journal of Algorithms 6(1), 132–137

[6] Salmela , L and Tarhio , J. 2010. Approximate string matching with reduced alphabet. In T Elomaa , H Mannila & P Orponen (eds) , Algorithms and applications, Lecture Notes in Computer Science 6060 , Heidelberg, Berlin, Springer Verlag.

[7] Alqadi, Z., M. Aqel and I. El Emary, 2007. Multiple-Skip Multiple-Pattern Matching Algorithm (MSMPMA). IAENG International Journal of Computer Science, 34(2): 14-20.

[8] Charras, C.; Lecroq, T. 2004. Handbook of Exact String-Matching Algorithms, King's College Publications. Available from: http://www-igm.univ-mlv.fr/~lecroq/string/string.ps.

[9] Boyer, R.; Moore, J. 1977. A fast string searching algorithm. Communications of the ACM, 20(10), 62-72.

[10] Danvy, O. and H. Rohde, 2006. On Obtaining the Boyer–Moore String-Matching Algorithm by Partial Evaluation. Journal of Information Processing Letter, 99: 158-162.

[11] Franek, F., C. Jennings and W.F. Smyth, 2006. A simple fast hybrid pattern-matching algorithm. Journal of Discrete Algorithms, 5: 682-695.

[12] R. Nigel Horspool, 1980. Practical Fast Searching in Strings. Journal of Software Practice and Experience, vol. 10, pp: 501-506.

[13] Oxford Dictionary. Oxford University Press, What is the frequency of the letters of the alphabet in English?". http://www.oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english, Last visited: 5th of December, 2013.

[14] Watson, B. and R. Watson. 2003. A Boyer–Moore-Style Algorithm for Regular Expression Pattern Matching. Journal of Science of Computer Programming, 48: 99-117.