International Journal of Agricultural Sciences
Volume **11** | Issue 2 | June, 2015 | 330-336 ■ e ISSN–0976–5670
☞ **DOI:10.15740/HAS/IJAS/11.2/330-336**
Visit us : *www.researchjournal.co.in*

A CASE STUDY

New algorithm for multiple pattern matching using least count of pattern

NITIPRASAD NAMDEORAO JAMBHULKAR* **AND** LOTAN KUMAR BOSE[1]
Division of Social Sciences, Central Rice Research Institute, CUTTACK (ODISHA) INDIA
(Email : nitiprasad1@gmail.com)

Abstract : Rapid advances in genome research in the past have resulted in generation of large set of data for DNA and protein sequences from different prokaryotic and eukaryotic genomes. DNA is made up of four nitrogenous bases known as adenine *(A)*, guanine *(G)*, thymine *(T)* and cytosine *(C)*. Pattern matching technique generally divides into two categories *i.e.* single pattern matching and multiple pattern matching. When it is required to find all occurrences of the pattern in the given string, it is known as single pattern matching. When more than one pattern are matched against the given string simultaneously, it is known as multiple pattern matching. The study of pattern matching is one of the applications in the field of bioinformatics. Many algorithms are available in literature for pattern matching. The purpose of pattern matching algorithm is to reduce the number of character comparisons. Hence, in this paper an algorithm Multiple Pattern Matching using Least Count of Pattern (MPMLCP) has been proposed for multiple pattern matching. The proposed algorithm has been compared with the available algorithm and it has been shown that the number of comparisons reduces over the available algorithms. SAS 9.3 software package has been used for calculating the number of comparisons.

Key Words : Pattern matching, Least count, Pre-processing, Algorithm, SAS software

View Point Article : Jambhulkar, Nitiprasad Namdeorao and Bose, Lotan Kumar (2015). New algorithm for multiple pattern matching using least count of pattern. *Internat. J. agric. Sci.*, **11** (2) : 330-336.

Article History : Received : 01.04.2015**; Accepted :** 16.05.2015

INTRODUCTION

Rapid advances in genome research in the past have resulted in generation of large set of data for DNA and protein sequences from different prokaryotic and eukaryotic genomes. Massive information is being generated in terms of genome sequences in different organisms which will help in understanding basic and applied research in biology. DNA is made up of four nitrogenous bases known as adenine *(A)*, guanine *(G)*, thymine *(T)* and cytosine *(C)*. Pattern matching is an important and active area of research with its varied large number of applications.

The field of bioinformatics has many applications in the modern world including agriculture and comparative biology. As the size of the data grows it becomes more difficult for users to retrieve necessary information from the sequences. Hence, more efficient and robust methods are needed for fast pattern matching techniques. Let sequence $S = \{s_1, s_2,....,s_n\}$ be the set of string and sequence $P = \{p_1, p_2,....,p_m\}$ be the set of

*** Author for correspondence**
[1]Crop Improvement Division, Central Rice Research Institute, CUTTACK (ODISHA) INDIA

pattern. Both the sequences are comprised of set of four fixed alphabet character called $\Sigma=\{A, C, G, T\}$. Now our aim is to find out all occurrences of pattern *P* in string *S*.

Pattern matching technique generally divides into two categories *i.e.*, single pattern matching and multiple pattern matching. When it is required to find all occurrences of the pattern in the given string, it is known as single pattern matching. When more than one pattern are matched against the given string simultaneously, it is known as multiple pattern matching. Pattern matching techniques sometimes divided into exact string matching algorithm and inexact/approximate string matching algorithm. Exact pattern matching algorithm will find that whether the probability lead to either successful or unsuccessful search. It will find all the occurrences of patter *P* of length *m* in string *S* of length *n*. the examples of exact string matching algorithms are Boyer and Moore algorithm (1977) and Kunth-Morris-Pratt algorithm (1977). Inexact/approximate pattern matching is sometimes referred to as approximate string matching or matching with *k* mismatches/differences. The examples of inexact pattern matching are Needleman and Wunch algorithm (1970) and Smith and Waterman algorithm (1981).

String matching deals with the problem of finding all occurrences of a character of patterns in a string. Some of the exact string matching algorithms are Boyer and Moore algorithm (1977), Knuth-Morris-Pratt algorithm (1977). In Brute-force algorithm the first character of the pattern *P* is compared with the first character of the string *S*. If it matches, then pattern *P* and string *S* are matched character by character until a mismatch is found or the end of the pattern *P* is detected. If mismatch is found, the pattern *P* is shifted one character to the right and the process continues. The complexity of the algorithm is $O(mn)$. The Boyer and Moore algorithm (1977) applies larger shift increment for each mismatch detection. The main difference the naïve algorithm had is the matching of pattern *P* in string *S* is done from right to left *i.e.*, after aligning P and string *S* the last character of *P* will matches to the first of *T*. If a mismatch is detected, say *C* in *S* is not in *P* then *P* is shifted right so that *C* is aligning with their right most occurrence of *C* in *P*. The worst case complexity of this algorithm is $O(m+n)$ and the average case complexity is $O(n/m)$. The Kunth-Morris-Pratt algorithm (1977) is based on the finite state machine automation. The pattern *P* is pre-processed to create a finite state machine *M* that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case is $O(m+n)$. The Aho-Corasick algorithm (Aho and Corasick, 1975) is an extension of the Kunth-Morris-Pratt algorithm (1977). Horspool (1980) used the bad character shift with right most character. The time complexity of the algorithm is $O(mn)$. Ukkonen (1985) proposed automation method for finding approximate patterns in strings. Zhu and Takaoka (1987) proposed using a combination of the Kunth-Morris-Pratt algorithm (1977) and RK methods in an algorithm developed for two dimensional cases. Sunday (1990) designed an algorithm quick search which scans the character of the window in any order and computes its shift with the occurrence shift of the character *T* immediately after the right end of the window. Wu and Manber (1992) proposed the algorithm for fast text searching allowing errors. Raita (1992) designed an algorithm in which the order of character comparisons has been changed to attain maximum efficiency. First the right most character of the pattern and the window are compared and on a match the left most character of the pattern and the window are compared. If they match it compares the middle character of both the pattern and the window. Second if they match it compares the characters from the second to the penultimate (n-1) position of the pattern and the window. The Naïve string matching algorithm finds the entire valid shifts by comparing each character of the pattern to text character, if the whole pattern is match to the text than it has a valid shift otherwise in case of mismatch it shift the pattern by one character and again compare the each character of the pattern to the text in this manner it finds entire valid shift of the pattern in the text. Berry and Ravindran (1999) calculates the shift value based on the bad character shift for two consecutive text characters in the text immediately to the right of the window. The time complexity of the algorithm is $O(nm)$. Kurtz (1996) proposed another way to reduce the space requirements of almost $O(mn)$. Ziad *et al.* (2007) proposed Multiple Skip Multiple Pattern Matching Algorithm (MSMPMA). This algorithm scans the input file to find all occurrences of the pattern based upon the skip technique. By using this index as the starting point of matching, it compares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges

from 1 to *m*-1). Rami and Jehad (2009) proposed an algorithm which searches the whole text string for the first character of the pattern and maintains an occurrence list by storing the index of the corresponding character. It uses an array equal to size of the string S for maintaining occurrence list. Time and space complexity of preprocessing is *O(n)*. An Index based Forward Backward Multiple Pattern Matching Algorithm (IFBMPMA) was proposed by Bhukya and Somayajulu (2010a). Here, the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches. In Index Based Forward backward Multiple Pattern Matching Algorithm (IFBMPMA), the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches. (Bhukya and Somayajulu, 2010a and b). The Deviki-Paul algorithm for multiple pattern matching requires a pre-processing of the given input text to prepare a table of the occurrences of the 256 member ASCII character set (Paul, 2011). Bhukya and Somayajulu (2011) proposed an Even Odd Multiple Pattern Matching (EOMPM) algorithm. In this method, the elements in the pattern are matched one by one first at the even places and then at the odd places until a mismatch occurs or a complete pattern matches. Some other work on multi-pattern string matching are also available (Aho and Corasick 1975; Commentz-Walter 1979 and Wu and Manber, 1994).

In many cases most of the algorithms operate in two stages. Pre-processing stage and searching phase. The pre-processing phase collects the full information and is used to optimize the number of comparisons. Whereas searching phase finds the pattern by the information collected in pre-processing phase. The purpose of pattern matching algorithm is to reduce the number of character comparisons. Hence, in this paper a new algorithm has been proposed for multiple pattern matching method based on the least count of pattern.

MATERIAL AND METHODS

A new algorithm known as multiple pattern matching using least count of pattern (MPMLCP) has been proposed. The algorithm is as follows :

Let *S* be the DNA sequence and *P* be the pattern composed of characters *A, C, G* and *T*. The string *S* be of length *n* and pattern *P* having length *m* where $m < n$. First we calculate the position and number of counts of the characters *A, C, G* and *T* in the string and pattern. Let s_a, s_c, s_g and s_t be the row vector of the positions of characters *A, C, G* and *T*, respectively of string *S*. $S_{a_n}, S_{c_n}, S_{g_n}$ and S_{t_n} be the count value of the characters A, C, G and T, respectively. p_a, p_c, p_g and P_t be the row vector of the position of characters *A, C, G* and T, respectively of pattern P. $p_{a_n}, p_{c_n}, p_{g_n}$ and p_{t_n} be the count value of the characters *A, C, G* and *T,* respectively of pattern *P*. Now compare $S_{a_n}, S_{c_n}, S_{g_n}$ and S_{t_n} for calculating minimum value. The character corresponding to minimum value is used to align the sequence. Suppose S_{g_n} is minimum then the corresponding character *G* is used for aligning purpose. Align the string *S* with first character *G* in the sequence with first character *G* in the pattern *P*.

Next compare the values of pa_n, pc_n, pg_n and pt_n and arrange them in ascending order. The order in which it assigns be the matching pattern of the characters *A, C, G* and *T*. Suppose $pc_n < pa_n < pg_n < pt_n$ then after aligning the string and pattern, first compare character *C* of pattern with the corresponding character of string. If all the characters *C* in the pattern matches then compare character *A* of the pattern with that of string. If it matches then compare character *G* and then *T*. Suppose at any time, the respective character of string and pattern does not matches, then move to the next position of *G* in the string. Again start matching the characters *C, A, G, T* of the pattern and string. Repeat this process till it reaches the last position of *G* of the string. For all the analysis SAS 9.3 software has been used and the program is written in SAS iml. This method has been illustrated with an example given below :

Example :

Let S=ATCGTCTACAATCTGTCTATCGCCTAACT be the string of 29 characters and P=CTGTCTATCG be the pattern of 10 characters. The method has been divided into two phases *i.e.* pre-processing phase and searching phase. In the pre-processing phase we will locate and count the position of different characters *A, C, G* and *T* and in the searching phase the pattern will be searched.

Pre-processing phase :

The position of *A, C, G* and *T* has been counted starting from first position to n^{th} (29^{th}) position for string and m^{th} (10^{th}) position for pattern. The position of different

characters *A*, *C*, *G* and *T* of string *S* and pattern *P* has been given in Table 1 and 2, respectively. The character *A*, *C*, *G* and *T* appears in 7, 9, 3 and 10 times, respectively in string *S*. Here the minimum value is 3 so we will select the character *G* for sequence alignment. The character *A*, *C*, *G* and *T* appears in the pattern in 1, 3, 2 and 4 times, respectively.

Searching phase :

The least count of string is *G*, so pattern will be aligned by position of *G*. The first position of *G* is at fourth position; hence the position of *G* in pattern will be aligned with the position of *G* in the string as given below:

ATCGTCTACAATCTGTCTATCGCCTAACT
CTGTCTATCG

The least count of pattern is A, so the matching will start with character *A*.

ATCGTCT**A**CAATCTGTCTATCGCCTAACT
CTGTCT**A**TCG

Character *A* of pattern matches with the character of string. Then the next least value of pattern is *G*. Hence, matching will be done with character *G* of the pattern.

ATC**G**TCTACAATCTGTCTATCGCCTAACT
CT**G**TCTATCG

Here, the second character is also matching. Now, we match with next character *G* of pattern.

ATCGTCTACA**A**TCTGTCTATCGCCTAACT
CTGTCTATC**G**

Here, the characters of string and pattern are not matching. Hence, we move to next character *G* which is at position 15 of the string and align it with the character *G* of the pattern.

ATCGTCTACAATCTGTCT**A**TCGCCTAACT
CTGTCT**A**TCG

Least count of pattern is *A* so it is matching in string and pattern. Next least value of pattern is *G*.

ATCGTCTACAATCT**G**TCTATCGCCTAACT
CT**G**TCTATCG

The *G* is matching in both string and pattern, then move to next *G* character of the pattern.

ATCGTCTACAATCTGTCTATC**G**CCTAACT
CTGTCTATC**G**

Third character *G* is matching. Next least value of the pattern is 3 of *C*, hence, start matching with first *C* in the pattern.

ATCGTCTACAAT**C**TGTCTATCGCCTAACT
CTGTCTATCG

The character *C* of string and pattern is matching, then move to next character *C*.

ATCGTCTACAATCTGT**C**TATCGCCTAACT
CTGT**C**TATCG

The character *C* is matching, then move to next character.

ATCGTCTACAATCTGTCTAT**C**GCCTAACT
CTGTCTAT**C**G

The character *C* of pattern matching with character of string. The next character is *T*. Start matching with first character *T* of the pattern with that of string.

ATCGTCTACAATC**T**GTCTATCGCCTAACT
C**T**GTCTATCG

The first character *T* of pattern and string is matches. Then move to next *T* character of the pattern.

ATCGTCTACAATCTG**T**CTATCGCCTAACT
CTG**T**CTATCG

The second *T* character matches, then move to next *T* character.

ATCGTCTACAATCTGTC**T**ATCGCCTAACT
CTGTC**T**ATCG

The character *T* matches, then move to next *T* character.

ATCGTCTACAATCTGTCTA**T**CGCCTAACT
CTGTCTA**T**CG

The character *T* of pattern matches with the character of string and there are no more characters in the pattern. Hence, the pattern *P* is matched at position 15th of the string. Now, move to next position of *G* which is at position 22nd in the string and align it with the position of *G* in the pattern.

ATCGTCTACAATCTGTCTATCGCCTAACT
CTGTCTATCG

Again start with least count of pattern *i.e. A*.

ATCGTCTACAATCTGTCTATCGCCT**A**ACT
CTGTCT**A**TCG

The character *A* in the pattern matches with the character in the string. The next least value of the pattern is 2 of *G*. Then match the first character of *G* in the pattern.

ATCGTCTACAATCTGTCTATC**G**CCTAACT
CT**G**TCTATCG

The first *G* of pattern matches with the character of string, then move to next *G* of the pattern.

ATCGTCTACAATCTGTCTATCGCCTAACT
CTGTCTATC**G**

The characters in the pattern and string are not matching, hence we move to the next character *G* in the string. But, there are no more character *G* in the string so we stop here. Only one occurrence of pattern is found in the string.

RESULTS AND DISCUSSION

In this section the DNA sequence has been taken from Ziad *et al.* (2007). The DNA sequence *S* be of size $n = 1024$. Let *S* be the following DNA sequence:

AGAACGCAGAGACAAGGTTCTCATTGTG
TCTCGCAATAGTGTTACCAACTCGGGTGCCTATT
GGCCTCCAAAAAAGGCTGTTCAACGCTCCAAG
CTCGTGACCTCGTCACTACGACGGCGAGTAAGAAC
GCCGAGAAGGTAAGGGAACTAATGACGCGTGG
TGAATCCTATGGGTTAGGATCGTGTCTAC
CCCAAATTCTTAATAAAAAACCTAGGACC
CCCTTCGACCTAGACTATCGTATTATGGACAAGC
TTTAACTGTCGTACTGTGGAGGCTTCAAAAC
GGAGGGACCAAAAAAATTTGCTTCTAGC
GTCAATGAAAAGAAGTCGGGTGTATG
CCCCAATTCCTTGCTGCCCGGACGGCC
AGGCTTATGTACAATCCACGCGGTACTACATCTTGTC
TCTTATGTAGGGTTCAGTTCTTCGCGCAATCA
TAGCGGTACTTCATAATGGGACACAACGAA
TCGCGGCCGGATATCACATCTGCTCCTGTG
ATGGAATTGCTGAATGCGCAGGTGTGAATACTG
CGGCTCCATTCGTTTTGCCGTGTTGA
TCGGGAATGCACCTCGGGGACTGTTCGATAC
GACCTGGGATTTGGCTATACTCC
ATTCCTCGCGAGTTTTCGATTGCTCATTAG

Table 1 : Positions of *A*, *C*, *G* and *T* of the string *S*

Characters	Position in the string	Count value
A	1, 8, 10, 11, 19, 26, 27	7
C	3, 6, 9, 13, 17, 21, 23, 24, 28	9
G	4, 15, 22	3
T	2, 5, 7, 12, 14, 16, 18, 20, 25, 29	10

Table 2 : Positions of *A*, *C*, *G* and *T* of the pattern *P*

Characters	Position in the pattern	Count value
A	7	1
C	1, 5, 9	3
G	3, 10	2
T	2, 4, 6, 8	4

Table 3 : Number of occurrences, number of comparison and CPC of proposed MPMLCP algorithm

Sr. No.	Pattern (P's)	Number of character	Number of occurrence	Number of comparison	CPC
1.	A	1	259	259	0.25
2.	AG	2	53	494	0.48
3.	CAT	3	11	542	0.53
4.	AACG	4	5	574	0.56
5.	AAGAA	5	2	581	0.57
6.	AAAAAA	6	3	647	0.63
7.	AGAACGC	7	2	345	0.34
8.	AAAAAAGG	8	1	583	0.57
9.	GCTCATTAG	9	1	354	0.35
10.	CCTTTTCCGG	10	1	324	0.32
11.	TTTTGCCGTGT	11	1	571	0.56
12.	TTCTTAATAAAA	12	1	598	0.58
13.	GGGACCAAAAAAT	13	1	414	0.40
14.	TTTTGCCGTGTTGA	14	1	368	0.36
15.	CCTCCAAAAAAGGCT	15	1	347	0.34
16.	GGCTGTTCAACGCTCC	16	1	336	0.33
17.	TTTTCGATTGCTCATTA	17	1	348	0.34
18.	GGGATTTGGCTATACTCC	18	1	337	0.33
19.	GGCCTTGTCTAAAGGTATG	19	1	576	0.56
20.	CCTGAGCGCGTCCTCCGTAC	20	1	341	0.33

GCTTTGCGGTAAGTAAGTTCTGGCCACCC
ACTTCGAGAAGTGAATGGCTGGCTCCTGAGCG
CGTCCTCCGTACAATGAAGACCGGTCTCGCGC
TAAATTTCCCCCAGCTTGTACAATAGTCCAGT
TTATTATCAAAGATGCGACAAATAAATTGATCA
GCATAATCGAAGATTGCGGAGCATAAGTTTGGA
AAACTGGGAGGTTGCCAGAAAACTCCGCG
CCTACTTTCGTCAGGATGATTAAGAGTATCGA
GGCCCCGCCGTCAATACCGATGTTCTTC
GAGCGAATAAGTACTGCTATTTTGCAG
ACCCTTTGCCAGGCCTTGTCTAAAGGT
ATGTTACTTAATATTGACAATACATGCGTATG
GCCTTTTCCGGTTAACTCCCTG.

For different pattern size which has been chosen randomly for experimental analysis from the DNA sequence the number of occurrence and the number of comparisons is given in Table 3. The first column in the Table 3 is serial number, second column refers to pattern, third column refers to number of columns in pattern, fourth column shows number of occurrence of the pattern in the string, fifth column shows the number of comparisons and sixth column shows the number of comparisons per character (CPC) which is equal to (number of comparisons/file size). CPC can be used as a measurement factor. This factor affects complexity time. Then it decreases, the complexity also decreases.

The proposed algorithm multiple pattern matching using least count of pattern (MPMLCP) has been compared with Brute-force, Naïve string, Tri-match, Multiple Skip Multiple Pattern Matching Algorithm (MSMPMA), Index based Forward Backward Multiple Pattern Matching (IFBPMP) and Even Odd Multiple Pattern Matching (EOMPM) algorithms. The comparison of the randomly selected patterns of different size has been given in Table 4. The table shows that the proposed algorithm multiple pattern matching using least count of pattern (MPMLCP) reduces the number of comparisons over other methods. The number of occurrences for different pattern size is same for all methods but the number of comparisons and CPC ratio are different for different algorithms.

Table 4 : Comparison of different algorithms

Pattern	Number of character	MPMLCP		Brute-force		Naïve String		Tri-Match		MSMPMA		IFBMPM		EOMPM	
		Number of comparison	CPC	Number of comparison	CPC	Number of comparison	CPC	Number of comparison	CPC	Number of comparison	CPC	Number of comparison	CPC	Number of comparison	CPC
A	1	259	0.25	1024	1	1024	1	1025	1	1024	1	518	0.51	259	0.25
AG	2	494	0.48	1282	1.2	1281	1.2	1284	1.2	1230	1.2	624	0.61	518	0.51
CAT	3	542	0.53	1318	1.2	1310	1.2	1321	1.2	1298	1.2	567	0.55	566	0.55
AACG	4	574	0.56	1376	1.3	1376	1.3	1389	1.3	1359	1.3	614	0.60	578	0.56
AAGAA	5	581	0.57	1388	1.3	1387	1.3	1393	1.3	1379	1.3	616	0.60	606	0.59
AAAAAAGG	8	583	0.57	1409	1.3	1407	1.3	1417	1.3	1394	1.3	634	0.62	616	0.60
TTCTTAATAAAA	12	598	0.58	1390	1.3	1399	1.3	1402	1.3	1390	1.3	651	0.64	646	0.63
GGCTGTTCAACGC	16	336	0.33	1349	1.3	1349	1.3	1365	1.3	1349	1.3	598	0.58	580	0.57
TCC															

Conclusion :

In this paper an algorithm for multiple pattern matching known as Multiple Pattern Matching using Least Count of Pattern (PMPLCP) has been proposed. In this algorithm, the aligning of the string *S* and pattern *P* has been done using least count of string and the matching has been done using least count of the pattern. The proposed algorithm reduces the number of

comparisons and CPC ratio as compared to the other algorithms (Table 4). SAS 9.3 software package has been used and programme has been written in SAS iml package for different stages of the method and analysis of the results.

REFERENCES

Aho, A.V. and Corasick, M.J. (1975). Efficient string matching: an aid to bibliographic Search. *Communications of the ACM*, **18** (6) : 333 340.

Berry, T. and Ravindran, S. (1999). *A fast string matching algorithm and experimental results*. In: Proceedings of the Prague Stringology Club Workshop '99, Liverpool John Moores University, 16-28pp.

Bhukya, R. and Somayajulu, D.V.L.N. (2010a). An index based forward backward multiple pattern matching algorithm. *World Acad. Sci. Engg. & Technol.*, **4** (6) : 17-25.

Bhukya, R. and Somayajulu, D.V.L.N. (2010b). An Index Based Forward Backward Multiple Pattern Matching Algorithm. *World Academy of Science and Technology*, June 2010, pp. 347-355.

Bhukya, R. and Somayajulu, D.V.L.N. (2011). An even odd multiple pattern matching algorithm. *Internat. J. Engg. Sci. & Technol.*, **3** (3) : 2118-2126.

Boyer, R.S. and Moore, J.S. (1977). A fast string searching algorithm. *Communi. ACM*, **20** (10) : 762-772.

Commentz-Walter, B. (1979). A string matching algorithm fast on the average. *Proceeding of ICALP*, 118-132pp.

Horspool, R.N. (1980). Practical fast searching in strings. *Software Practice Exp.*, **10** (6) : 501-506.

Kunth, D., Morris, J. and Pratt, V. (1977). Fast pattern matching in strings. *SIAM J. Comput.*, **6** (1) : 323-350.

Kurtz, S. (1996). Approximate string searching under weighted edit distance. In proceedings of the 3rd South American workshop on string processing. Carleton Univ. Press, 156-170pp.

Needleman, S.B. and Wunch, C.D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biol.*, **48** (3) : 443-453.

Pendlimarri, Devaki, Petlu, Paul, Bharath Bhushan and Satrasala, Ramesh Babu (2011). Novel devaki-Paul algorithm for multiple pattern matching. *Internat. J. Computer Applications* (0975-8887), **13** (3) : 37-42.

Raita, T. (1992). Tuning the Boyer-Moore-Horspool string-searching algorithm. *Software-Practice Exp.*, **22**(10) : 879- 884.

Rami, H.M. and Jehad, Q.Q. (2009). On improving the naive string matching algorithm. *Asian J. Informa. Technol.*, **8** (1): 14-23.

SAS Institute (2012). SAS/IML Version 9.3 SAS Institute, Cary, North Carolina, USA.

Smith, T.E. and Waterman, M.S. (1981). Identification of common molecular subsequences. *J. Molecular Biol.*, **147** (1) : 195-197.

Sunday, D.M. (1990). A very fast substring search algorithm. *Communications of the ACM*, **33** (8) : 132–142.

Ukkonen, E. (1985). Finding approximate patterns in strings. *J. Algor.*, **6** (1) : 132-137.

Wu, S. and Manber, U. (1992). Fast text searching allowing errors. *Communications of the ACM*, **35** (1) : 83-91.

Wu, S. and Manber, U. (1994). A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona.

Zhu, R.F. and Takaoka, T. (1987). On improving the average case of the boyer-moore string matching algorithm. *J. Information Process.*, **10** (3) : 173-177.

Ziad, A.A. Alqadi, Musbah Aqel and Ibrahiem M.M. EI Emary (2007). Multiple skip multiple pattern matching algorithms. *IAENG Internat. J. Computer Sci.*, **34** : 2