A Two-Hashing Table Multiple String Pattern Matching Algorithm

Chouvalit Khancome
Department of Computer Science,
Faculty of Science and Technology
Rajanagarindra Rajabhat University
Chachoengsao, Thailand
e-mail: chouvalit@hotmail.com,
chouvalit.kha@csit.rru.ac.th

Pisit Chanvarasuth
School of Management Technology
Sirindhorn International Institute of Technology,
Thammasat University
Pathumthani, Thailand
e-mail: pisit@siit.tu.ac.th

Veera Boonjing
Department of Computer Science,
King Monkut's Institute of Technology at Ladkrabang
Ladkrabang, Bangkok,Thailand
e-mail: kbveera@kmitl.ac.th

Abstract*—Multiple string pattern matching is one of many approaches to simultaneously search occurrences of a large number of patterns in a given text. In this paper, a new solution to this problem is presented by using two-hashing tables to minimize attempting times. This solution, called Two-Hashing Table Multiple String Patterns Matching Algorithm (Two-HT-MSPMA), is suitable for very long length of minimum pattern length. Its time complexity is more efficient than classic algorithms. Empirical results showed that its attempting times were less than of traditional algorithms.

Keywords- Hashing Data Structure, Static Dictionary Matching, Pattern Detection Algorithm, Exact String Matching.

I. INTRODUCTION

Multiple string pattern matching is a classic problem in computer science to simultaneously search for occurrences of known patterns in a text. It plays important roles in various fields (e.g., [18], [21], [22], [25]) including the operating system commands (Unix grep command using Commentz-Walter [8] and agrep using Wu-Manber[27]), intrusion detection systems (e.g., SNORT using Aho-Corasick[1], Commentz-Walter [8], and Wu-Manber[27]), and so on. Its good solution relies on excellent data structures provided for several searching aspects. Formally, this matching simultaneously searches for all occurrences of patterns $P=\{p^1, p^2, p^3,..,p^r\}$ which appeared in a given text $T=\{t_1,t_2,t_3...t_n\}$ over a finite alphabet Σ. Therefore, a more efficient data structure, provided for accommodating the target patterns, is needed. Moreover, a faster searching algorithm is always required.

A powerful search is required in a minimal attempting time. Such a search can be obtained via a fast accessing data structure. A hashing table is the most widely known as the fastest accessing data structure; because it can be accessed in a constant time. Unfortunately, when implementing the hashing table to a single string pattern matching highly took a consuming time. Moreover, if any multiple string pattern matching algorithms inherited only one hashing table to create their data structure for searching, then the time complexity of those algorithms was more time consuming. Thus, this way seems to get to the dead end of inherited hashing algorithms and is the major barrier of hashing algorithm developments. Fortunately, the two-hashing multiple string pattern matching algorithm (called Two-HT-MSPMA) breaks through the barriers of traditional hashing algorithms.

The new approach uses two related hashing table for storing the target patterns. Including, the shift table is used to recognize the shift value for a new window search as well as the algorithm in [4]. In the searching phase, the matching method hashes one time or twice in the first hashing table if there is no pattern overlapping or twice in both hashing tables if there is only one or more pattern overlapping.

Two-HT-MSPMA is very high efficient when the minimum pattern length is very long. It takes $O(|P|)$ time and space preprocessing where $|P|$ is the sum of all pattern lengths. The search takes $O(|t||P|)$ in the worst case, $O(|t|)$ in an average case, and $O(|t|/lmin)$ in the best case scenario where $|t|$ is the length of the given text, and *lmin* is the minimum length of patterns. Furthermore, the empirical results showed that the attempting time to match were less than the well known algorithms such as Aho-Corasick [1], SetHorspool [8], and *q-grams* of [1] and [8]; especially when using the large number of pattern overlapping and very long patterns.

The remaining sections are organized as follows. Section II shows the previous research on multiple string pattern matching. Section III gives basic definitions, which are shown by examples and algorithm details. Section IV explains how to create the hashing tables and their algorithms. Section V illustrates the proposed search algorithm. Section VI analyzes complexities of the preprocessing phase and the searching phase. Section VII shows empirical results and gives a discussion. Finally, section VIII gives a conclusion.

978-0-7695-4967-5/13 $26.00 © 2013 IEEE
DOI 10.1109/ITNG.2013.108

II. PREVIOUS RESEARCH

In general, multiple string pattern matching has become increasingly popular in research subject, e.g. [8]. There are two phases in string pattern matching: preprocessing and searching. The preprocessing generates the patterns to a designated data structure providing to a powerful search algorithm. Trie, Bit-parallel, and Hashing table are well-known data structures for this purpose. Trie is a prefix-tree data structure used in the first famous algorithm [1]. This algorithm is the first linear time solution which is inherited from [5]. Then, the Commentz-Walter [2] and the SetHorspool [8] are the sub-linear time solutions which extended from [23].

There are a number of algorithms that apply Trie; for example, SBOM [3], SDBM (shown in [8]), [10], MultiBDM [13], [16], and [17]. All of them improved the searching time (mentioned in [8]). However, implementing Trie to applications consumes tremendous memory in practice.

Alternatively, the Bit-parallelism, like Trie for its popularity, uses data of bits for the pattern representation. Navarov [8] showed how to apply the Bit-parallel in the single string Shift-Or and the single Shift-And to the Multiple Shift-And [7], the Multiple-BNDM [7], and algorithm in [9]. Unfortunately, Bit-parallel algorithms are restricted by the computer word and need to deal with the complexity of the bit-conversion methods. Moreover, it is difficult to update the pattern when implement the data structure.

Employing a hashing table for dictionary was an important choice; Karp and Rabin [11] were the first to find a solution with this structure. This algorithm took more time in a worst case scenario: O(mn) time where m is the single pattern length. The disadvantage of this principle is the lengthy processing time when directly extended to the multiple string patterns matching. Nevertheless, an efficient algorithm was presented by Wu and Manber [27], which created the shift table and implemented the hashing table to store the block of patterns. A more efficient solution [29] improved Wu and Manber's method in [27] and saved searching time in an average case However, the worst case scenario was not improved.

In addition, there are some other techniques which combine algorithms with several structures in order to improve the time complexity such as the q-gram structure and the partitioning technique. However, the worst case time is again not improved. More details on the development of the new standard can be found in [8], [14], [15], [24], and [31].

Recently, solutions [5], [20], and [28] improved the Trie structure to accommodate the patterns especially in [5] which has minimal space of solution. For other solutions, there exists a wealth of literature devoted to employ those classic data structures (e.g., Trie, Bit-parallel, and Hashing). These can be found in [12], [26], and [30].

III. BASIC DEFINITIONS

As mentioned in the introduction section, our new algorithm employs two hashing tables for accommodating the patterns; as well as, the searching phase hashes into these hashing tables for matching inspection. Then, this section introduces the definitions and illustrates their data structures.

Definition 1 shows a main shifting table which is applied from [4], definition 2 describes the first level of hashing table of the minimum prefix patterns, and definition 3 explains the second-level of the hashing table to store the patterns which are overlapped and related with the first hashing table in definition 2.

Firstly, let *lmin* be the minimum length of patterns, and then, the following explanations indicate the meaning of referring notations to be used.

Definition 1. The hashing table, which consists of two columns: $\sum$ and the shifting values, is called the shifting table and denoted by *ST*; where $\sum$ is the single alphabet which appear in *P*. The shifting value is an integer number which is used for setting the new window search.

Example 1. The *ST* of *P={aab**a**, aab**a**b, aab**a**bc, aab**a**bcd, aab**a**bcde, abc**b**, , zmn**d**, qop**e**, jmq**f**m }* into *ST* where * is the other characters that are not appeared in $\sum$ (the shifting values calculated by algorithm 1 in the next section). That algorithm is inherited from [21].

TABLE I. SHIFTING VALUE TABLE(ST)

$\sum$ of p^i*[lmin]*	Shifting Values
a	2
b	1
c	1
d	1
e	1
f	1
m	2
n	4
o	3
p	1
q	3
j	3
z	3
*	4

Definition 2. The hashing table, which stores the prefix of all patterns that the number of characters equal the minimum length of patterns and the lists(*L*) of patterns overlapping lengths, is called the hashing table level 1 (denoted by *HL1*); where *Min_keys* is the column which is used to store the prefix of all patterns and *L* is the list of patterns overlapping lengths.

Example 2. The *HL1* of example 1 shown as *P={aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm }*.

TABLE II. HASHING TABLE LEVEL 1

Min_keys ***($p^i[1...lmin]$)***	***L***
aaba	*4,5,6,7,8*
abcb	-
zmnd	-
qope	-
jmqf	*5*

Definition 3. The hashing table, which stores the patterns that are overlapping in *HL1*, is called the overlapping patterns table (denoted by *HL2*); where *Ovp_Keys* is the column, stored the patterns which are overlapped from *HL1* and their lengths are longer than patterns in *HL1*.

Example 3. The *HL2* of example 1 shown as *P={aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm }*.

TABLE III. HASHING TABLE LEVEL 2

Ovp_Keys
aabab
aababc
aababcde
aababcde
jmqfm

IV. PREPROCESSING PHASE

This phase creates two related hashing tables and one shift table. The following algorithms show how to create *ST*, *HL1*, *HL2* respectively; where *lmin* is the minimum length of patterns and *ShiftingValue* is the shifting value of the individual character in ∑ which is initiated at *lmin*.

ST is the hashing table consisted of the keys of characters which are appeared in all patterns at the positions of *lmin*. Each character is checked for overlapping when the search window is slid, and the shifting value can be shifted to the farthest distance with no pattern overlapping. The algorithm (inherited from [4]) is shown below.

Algorithm 1 : Create *ST*
Input: $P=\{p^1, p^2, p^3,...,p^r\}$,*lmin*
Output: Shifting Table (*ST*)
1. Initiate the empty *ST*
2. For *i=1* to *r* Do
3. *ShiftingValue=lmin*
4. For *j= lmin -1* down to *1* Do
5. If $p^i[j]$ exists in *ST* Then
6. *ShiftingValue* = the shifting value at $p^i[j]$ in *ST*
7. If $p^i[j] = p^i[lmin]$ Then
8. *ShiftingValue = ShiftingValue*-1
9. If *ShiftingValue* at $p^i[j] >$*ShiftingValue* Then *ShiftingValue* at $ST[p^i[j]]=$ *ShiftingValue*
10. End If
11. End If
12. Else
13. $ST \leftarrow p^i[j]$ at column of ∑
14. Add *lmin* to column of shifting Value
15. End For
16. End If
17. End For
18. Return *ST*

HL1 is created for storing the patterns of *P* with the number of characters equal *lmin* characters into *Min_Keys*. Meanwhile, the values to be inserted into the column *L* are the list of pattern lengths that are overlapped to *HL2*. The algorithm is shown below.

Algorithm 2 : Create *HL1*
Input: $P=\{p^1, p^2, p^3,...,p^r\}$, *lmin*
Output: *HL1*
1. Initiate the empty *PPT*
2. For *i=1* to r Do
3. If $p^i[1...lmin]$ does not exist in *HL1* Then
4. *HL1 column keys* $\leftarrow p^i[1...lmin]$
5. If length of p^i = *lmin* Then
6. Add *lmin* to *L* of *HL1* in p^i column
7. End of If
8. End If
9. End For
10. Return *HL1*

HL2 is created for the overlapping patterns and the pattern that the lengths are longer than the minimum pattern lengths which are stored in *HL1*. The inputs are *P* and *lmin* in *HL1*. Meanwhile, the overlapping patterns and the patterns that the lengths are longer than *lmin* are stored in the *HL2*. The details are shown in Algorithm 3.

Algorithm 3 : Create *HL2*
Input: $P=\{p^1, p^2, p^3,...,p^r\}$,*lmin*
Output: *HL2*
1. Initiate the empty *HL2*
2. For *i=1* to r Do
3. If length of $p^i >$ *lmin* Then
4. Add length of p^i to *L* of *HL1* in p^i column
5. *HL2 column keys* $\leftarrow p^i$
6. End For
7. Return *HL2*

After the preprocessing phase, all tables have already been created for searching algorithm. In addition, they will be referred for all search windows. For the our example, if *P={aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm }*, then *ST*, *HL1*, and *HL2* are initiated as Table I, Table II, and Table III respectively.

.

V. SEARCHING PHASE

The following details show the searching steps and the details on algorithms. The searching method is driven by initiating the window search at the beginning of the given text with the minimum length. Then, the substring of that window is hashed into HL1. The matching position is then checked when the first item of L equal lmin. If there are many items in L, then that substring including the next character is hashed into HL2 until the last item in L is taken. Next, the character that appears at the lmin in each window is used to hash into ST for initiating the next window search.

The following algorithms show the idea above:

Algorithm 4: Two-HT-MSPMA
Input: *T, ST, HL1,HL2, P, lmin*
Output : all matched positions are reported
1. *Rear, Cur=lmin*
2. While *Cur* <=*n* Do
3. If *Hash[T[Cur-lmin]...T[lmin]]* into *HL1 = true* Then
4. Report the matched position at *Cur* if *L[1] = lmin* or *L[1]=* -
5. For (*i=1* if *L[1]>lmin* or *i=2* if *L[1]=lmin*) to size of *L* and *Cur*<=*n* Do
6. *Rear= L[i]*
7. If *Hash[T[Cur-lmin]...T[Rear]]* into *HL2 = true* Then
8. Report the matched position at *Rear*
9. End For
10. End If
11. End If
12. *NewCur= ShiftingVvalue* at *Hash[ST[T[Cur]]*
13. *Cur = Cur+NewCur*
14. *Rear=Cur*
15. End While

An example of searching is shown as example 4.

Example 4. Searching *P={aaba, aabab, aababc, aababcd, aababcde, abcb, zmnd, qope, jmqfm}* in the given text *T= aababcdezmndjmqfmaababcd.*

Searching window :1

a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*aaba*")->*HL1 =true*, *L=4,5,6,7,8*, match: *OK*

a a b a b c d e z m n d j m q f m a a b a b c d

Process *L[2]=5*, Hash("*aabab*")->*HL2 =true*, match: *OK*

a a b a b c d e z m n d j m q f m a a b a b c d

Process *L[3]=6*, Hash("*aababc*")->*HL2 =true*, match: *OK*

a a b a b c d e z m n d j m q f m a a b a b c d

Process *L[4]=7*, Hash("*aababcd*")->*HL2 =true*, match: *OK*

a a b a b c d e z m n d j m q f m a a b a b c d

Process *L[5]=8*, Hash("*aababcde*")->*HL2 =true*, match: *OK*

Searching Window: 2, *ShiftingValue* '*a*'=2
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*babc*")->*HL1* = -, *L*= - match: -

Searching Window: 3, *ShiftingValue* '*c*'=1
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*abcd*")->*HL1* = -, *L*= -, match: -

Searching Window: 4, ShiftingValue '*d*'=3
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*bcde*")->*HL1* = -, *L*= -, match: -

Searching Window: 5, *ShiftingValue* '*e*'=1
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*cdez*")->*HL1* = -, *L*= -, match: -

Searching Window: 6, *ShiftingValue* '*z*'=3
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*zmnd*")->HL1 =true, L= - match: OK

Searching Window: 7, *ShiftingValue* '*d*'=1
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*mndj*")->*HL1* = -, *L*= -, match: -

Searching Window: 8, *ShiftingValue* '*j*'=3
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*jmqf*")->*HL1 =true*, *L=5*, match: -

Process *L=5*
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*jmqfm*")->*HL2 =true*, *L*= -, match: *OK*

Searching Window: 9, *ShiftingValue* '*f*'=1
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*mqfm*")->*HL1* = -, *L*= -, match: -

Searching Window: 10, *ShiftingValue* '*m*'=2
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*fmaa*")->*HL1* = -, *L*= -, match: -

Searching Window: 11, *ShiftingValue* '*a*'=2
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*aaba*")->*HL1 =true*, *L=4, 5,6,7,8*, match: *OK*

Process *L[2]=5*
a a b a b c d e z m n d j m q f m a a b a b c d
Hash("*aabab*")->*HL2 =true*, match: *OK*

Process *L[3]=6*

a a b a b c d e z m n d j m q f m [*a a b a*] [*b*] [*c*] *d*

Hash("*aababc*")->*HL2 =true*, match: *OK*

Process *L[4]=7*

a a b a b c d e z m n d j m q f m [*a a b a*] [*b*] [*c*] [*d*]

Hash("*aababcd*")->*HL2 =true*, match: *OK*

Searching Window 12: *Cur>n*, then the search is finished.

With this search, we can save times to scan the given text when the minimum length is too long. Unfortunately, if there are no characters which can be skipped, then the comparing time will be equal to the length of the given text.

VI. Complexity Analysis

A. Preprocessing Time and Space Complexity

Algorithm 1 is run to equal the number of pattern (*r*) which takes *r* time or $|P|$. Meanwhile, the space complexity equals $\sum$. Like the algorithm 1, the algorithm 2 (creating *HL1*) is run in $|P|$ time. The space complexity is $|lmin*r|$. Creating the hashing table of *HL2* (Algorithm 3) uses $O(|P|)$ time and takes the maximum space $O(|P|)$ when every pattern is overlapped or uses $O(1)$ if there is no pattern overlapping (only *HL1*).

Therefore, overall time complexity is $O(|P|+|P|+|P|)$ for the preprocessing phase, shown as $O(3|P|)$ or $O(|P|)$ time. The space complexity equals the space of $O(\sum +|lmin*r| +|P|)$ which is shown as $O(|P|)$.

B. Searching Time Complexity

The algorithm 4 needs to capture the string in each searching window at once. Therefore, the maximum of time takes |t| time if the shifting values in ST are equal, which takes O(|t|) (shown in the average case). Meanwhile, if there is no overlapping or the shifting value equals lmin, then it takes O(|t|/lmin) (shown in the best case scenario).

In the average case scenario, the comparisons need to capture all character in the searching window in each comparison. Then, the shifting values are differenced, which take O(|t|) time in maximum when the shifting value is activated more than one position.

In the worst case scenario, the basic comparison is equivalent to the average case when the shifting value is activated only one position, which finally lead to O(|t||P|) time. However, the substring method of each comparison is taken only once. This case may be occurred when there are several overlapping patterns.

VII. Empirical Results and Discussion

In our empirical results, the given text of 1,000,000 characters and the minimum length of all patterns to be matched 100 characters were assumed. The matching times were set to 10,000 times with no overlapping and the shifting value equals the minimum length. On the other hand, when the mismatches occurred at the first character, the last character of each search window will also be assumed.

The classic Aho-Corasick [1], the simple algorithm of Commentz-Walter[23] called as SetHorspool [shown in 10], the current excellent *q-gram* technique of [19] (developed from classic Aho-Corasick [1] and SetHorspool) were chosen for the empirical comparisons. As well as, the *5-q grams*, which were larger than the original experiments shown in [19], were set for the comparison as well.

Table IV shows the empirical results of the attempting time for matching where *Two-HT-MSPMA* is the two-hashing table multiple string pattern matching algorithm.

TABLE IV. Emperical Results of attempting time for matching (times) when there is no patterns overlapping

Algorithms	***Complete matching***	***Mismatch at p^i [1]***	***Mismatch at p^i [lmin]***
Aho-Corasick[1]	1,000,000	1,000,000	1,000,000
SetHorspool[9]	1,000,000	1,000,000	10,000
5-q grams of [1]	200,000	200,000	10,000
5-q grams of [9]	200,000	1,000,000	200,000
Two-HT-MSPMA	10,000	10,000	10,000

TABLE V. % of attempting times for matching in table IV

Algorithms	***Complete matching***	***Mismatch at p^i [1]***	***Mismatch at p^i [lmin]***
Aho-Corasick[1]	100.00	100.00	100.00
SetHorspool[9]	100.00	100.00	1.00
5-q grams of [1]	20.00	20.00	1.00
5-q grams of [9]	20.00	100.00	20.00
Two-HT-MSPMA	1.00	1.00	1.00

The empirical results showed that the attempting times of new algorithms were less than the traditional algorithms especially in the cases of long length and no overlapping examples.

For algorithm implementation, the preprocessing phase is relatively easy to create especially when using a high level of a hashing data structure such as java.util.HashSet. This structure provides the string methods for accessing by hashing properties such as .add, .remove, .contain, and so on.

VIII. Conclusion

The multiple-string pattern matching employing two-hashing tables was presented. This approach takes $O(|P|)$ time and space preprocessing where $|P|$ is the sum of lengths in *P*. The search takes $O(|t||P|)$ in the worst case, $O(|t|)$ in an average case, and $O(|t|/lmin)$ in the best case scenario; where $|t|$ is the length of text *T* and *lmin* is the minimum length of patterns. As shown in our empirical results, the attempting times were less than of the traditional algorithms especially in the case of a very long minimum pattern length.

REFERENCES

[1] A. V. Aho, and M. J. Corasick, "Efficient string matching: An aid to bibliographic search", Comm. ACM, 1975, pp.333-340.

[2] B. Commentz-Walter, "A string matching algorithm fast on the average", In Proceedings of the Sixth International Collogium on Automata Languagees and Programming, 1979, pp.118-132.

[3] C. Allauzen, and M. Raffinot, "Factor oracle of a set of words", Technical report 99-11, Institute Gaspard Monge, Universitĕ de Marne-la-Vallĕe, 1999.

[4] C. Khancome and V. Boonjing, "New Hashing-Based Multiple String Pattern Matching Algorithms", 2012 Ninth International Conference on Information Technology- New Generations (ITNG 2012), LasVegas, USA, 2-4 April 2012, pp.195-200.

[5] D. Belazzougui, "Worst Case Efficient Single and Multiple String Matching in the RAM Model", 21st International Workshop on Combinatorial Algorithms (IWOCA 2010), LNCS 6460, 2011, pp. 90–102.

[6] D.E. Knuth, J.H. Morris, V.R Pratt, "Fast pattern matching in strings", SIAM Journal on Computing 6(1), 1997, pp.323-350.

[7] G. Navarro, "Improved approximate pattern matching on hypertext", Theoretical Computer Science, 2000, 237:pp.455-463.

[8] G. Navarro, and M. Raffinot, "Flexible Pattern Matching in Strings", The press Syndicate of The University of Cambridge. 2002.

[9] H. HYYRO, K. F. SSON, and G. Navarro, "Increased Bit-Parallelism for Approximate and Multiple String Matching", ACM Journal of Experimental Algorithms, Vol.10, Article No. 2.6, 2005, pp.1-27.

[10] J. J. Fan, and K. Y. Su, "An efficient algorithm for match multiple patterns", IEEE Trans. On Knowledge and Data Engineering, 1993, Vol.5, No.2, pp.339-351.

[11] K. M. Karp, and M.O. Rabin, "Efficient randomized pattern-matching algorithms" IBM Journal of Research and Development, 31(2), 1987, pp.249-260.

[12] L. Dai, and Y. Xia, "A Lightweight Multiple String Matching Algorithm", International Conference on Computer Science and Information Technology 2008 (ICCSIT'08), Singapore, Aug. 29-Sept. 2, 2008, pp. 611–615.

[13] L. Gongshen, L. Jianhua, and L. Shenghong, "New multi-pattern matching algorithm", Journal of Systems Engineering and Electronics, Vol. 17, No. 2, 2006, pp.437-442.

[14] L. Ping, T. Jian-Long, and L. Yan-Bing, "A partition-based efficient algorithm for large scale multiple-string matching", In Proceeding of 12th Symposium on String Processing and Information Retrieval (SPIRE'05). Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, Berlin, 2005.

[15] L. Salmela, J. Tarhio, and J. Kytöjoki, "Multipattern string matching with q-grams", ACM Journal of Experimental Algorithmics (JEA), Vol. 11, Article No. 1.1, 2006, pp.1-19.

[16] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching", Report 93-3, Institute Gaspard Monge, Universitĕ de Marne-la-Vallĕe, 1993.

[17] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching", Information Processing Letters, 71(3/4), 1999, pp.107-113.

[18] M. Makula, and L. Benuskova "Interactive visualisation of oligomer frequency in DNA", Computing and Informatics, Vol. 28, No. 5, 2009, pp. 695–710.

[19] M. Raffinot, "On the multi backward dawg matching algorithm (MultiBDM)", In R. Baeza-Yates, editor, Proceedings of the 4th South American Workshop on String Processing, Valparaìso, Chile. Carleton University Press, 1997, pp.149-165.

[20] N. Askitis, and J. Zobel, "Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache", ACM Journal of Experimental Algorithmics, Vol. 15, No. 1, Article 1.7, 2011, pp. 1–61.

[21] P. C. Bosnjak, and S. M. Cisar, "EWMA Based Threshold Algorithm for Intrusion Detection", Computing and Informatics, Vol. 29 No. 6+, 2010, pp. 1089–1101.

[22] P. Lu, Y. Che, and Z. WangK, "UMDA/S: An Effective Iterative Compilation Algorithm for Parameter Search", Computing and Informatics, Vol. 29, No. 6+, 2010, pp. 1159–1179.

[23] R.S. Boyer, and J.S. Moore, "A fast string searching algorithm", Communications of the ACM, 20(10), 1977, pp.762-772.

[24] S. Klein, T. R. Shalom, and Y. Kaufman, "Searching for a set of correlated patterns", Journal of Discrete Algorithm, Elsevier, 2006, pp.1-13.

[25] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel "Iterative Dictionary Construction for Compression of Large DNA Data Sets", IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 9, No. 1, 2012, pp. 137–149.

[26] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel "Iterative Dictionary Construction for Compression of Large DNA Data Sets", IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 9, No. 1, 2012, pp. 137–149.

[27] S. Wu, and U. Manber, "A fast algorithm for multi-pattern searching", Report tr-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[28] T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen, "Online Dictionary Matching with Variable-Length Gaps", 10th International Symposium on Experimental Algorithms (SEA 2011), LNCS 6630, 2011, pp. 76–87.

[29] Y. Hong, D. X. Ke, and C. Yong, "An improved Wu-Manber multiple patterns matching algorithm", Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International 10-12, 2006, pp.675-680.

[30] Y. Hu, P. F. Wang, and H. Kai, "A Fast Algorithm for Multi-String Matching Based on Automata Optimization", C2010 2nd International Conference on Future Computer and Communication, Vol. 2, 2010, pp. 379–383.

[31] Z. A.A. Alqadi, M. Aqel, and I. M.M. El Emary, "Multiple skip Multiple pattern matching algorithm (MSMPMA)", IAENG International Journal of Computer Science, 34:2, IJCS_34_2_03, 2007.