A New Multiple-Pattern Matching Algorithm for the Network Intrusion Detection System

Nguyen Le Dang, Dac-Nhuong Le, and Vinh Trong Le

***Abstract*—String matching algorithms are essential for network application devices that filter packets and flows based on their payload. Applications like intrusion detection/ prevention, web filtering, anti-virus, and anti-spam all raise the demand for efficient algorithms dealing with string matching. In this paper, we present a new algorithm for multiple-pattern exact matching. Our approach reduces character comparisons and memory space based on graph transition structure and search technique using dynamic linked list. Theoretical analysis and experimental results, when compared with previously known pattern-matching algorithms, show that our algorithm is highly efficient in both space and time.**

***Index Terms*—Pattern matching, multi-pattern matching, network intrusion detection system.**

I. Introduction

String matching algorithms in software applications like virus scanners (anti-virus) or intrusion detection systems are commonly used for improving data security over the internet [1]. String-matching techniques are used for sequence analysis, gene finding, evolutionary biology studies and analysis of protein expression. Other fields, such as music technology, computational linguistics, artificial intelligence, artificial vision, have been using string matching algorithms as their integral part of theoretical and practical tools. There are various problems in string matching appeared as a result of such continuous, exhaustive use, which in turn were promptly solved by the computer scientists.

There are many good solutions have been presented for exact string matching of multiple patterns, such as: Aho-Corasick, Commentz-Walte, Navarro and Raffinot, Rabin Karp, Muth and Manber algorithms with their variations [2]. However, almost of the earlier algorithms have been designed for pattern sets of moderate size, i.e. a few dozens, and they unfortunately do not scale very well to larger pattern sets. The multi-pattern matching problem has many applications. It is used in data filtering (data mining) to find selected patterns, for example, anti-virus scanning, intrusion detection, content scanning and filtering, and specific data mining problems.

A. Multiple Pattern Matching Problem

String matching is a technique to find out a pattern from given text. Let $P = \{p_1, p_2, ..., p_m\}$ be a set of patterns, which are strings of characters from a fixed alphabet. Let $T =\{t_1, t_2, ..., t_n\}$ be a large text, again consisting of characters from the above alphabet. The problem is to find all occurrences of all the patterns of P in T. Given a pattern set P and a text T, report all occurrences of all the patterns in the text. The text T is a string of n characters drawn from the alphabet Σ (of size σ). The pattern set P is a set of m patterns each of which is a string of characters over the alphabet Σ. For simplicity we assume that all patterns have the same length m. We are especially interested in searching for large pattern sets. For example, the UNIX fgrep and egrep programs support multi-pattern matching through the -f option [3]-[6].

Pattern matching algorithms have two main objectives: reduce the number of character comparisons and reduce the time requirement in the worst and average case analysis. Most of the algorithms operate in two stages. The first stage is a preprocessing of the set of patterns. Applications that use a fixed set of patterns for many searches may benefit from saving the preprocessing results in a file (or even in memory). This step is quite efficient and in most cases it can be done on the fly. The second stage is searching phase to find the pattern by the information collected in the pre-processing stage.

B. Single and Multiple Pattern Matching

In a standard problem, we are required to find all occurrences of a pattern in a given input text, known as single pattern matching. Suppose, if more than one pattern are matched against the given input text simultaneously, then it is known as, multiple pattern matching. Whereas single pattern matching is widely used in network security environments. Multiple pattern matching algorithms can search multiple patterns in a text at the same time. They have a high performance and good practicability, and are more useful than the single pattern matching algorithms.

C. Exact and Inexact Pattern Matching

Exact pattern matching algorithms will lead to either successful or unsuccessful search. The problem can be stated as: Given a pattern P of length m and a string/text T of length n ($m \leq n$). Find all the occurrences of P in T. The matching is to be exact, which means that the exact word or pattern needs to be found. Some exact matching algorithms are Naïve Brute-force, Boyer-Moore, KMP [1]. Approximate (Inexact) pattern matching is sometimes referred as approximate pattern matching or matches with k mismatches/ differences. This problem in general can be stated as: Given a pattern P of length m and a string/text T of length n ($m \leq n$). Find all the occurrences of sub string X in T that are similar to P, allowing a limited number, say k different characters in similar matches. The edit/transformation operations are insertion,

Manuscript received August 18, 2014; revised October 30, 2014.

Dac-Nhuong Le and Nguyen Le Dang are with Haiphong University, Haiphong, Vietnam (e-mail: Nhuongld@hus.edu.vn)

Vinh Trong Le is with the Hanoi University of Science, Vietnam National University, Hanoi, Vietnam.

DOI: 10.7763/IJET.2016.V8.865

deletion and substitution. Approximate string matching algorithms are classified into: Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. Inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing, etc.

D. Pattern Matching for NIDS

Matching patterns in a NIDS (Network Intrusion Detection System) is a problem more specialized than the general patterns matching problem. In the context of signature matching in a NIDS the signature database corresponds to the pattern set and the network packets, which the system scans, correspond to the text input for a pattern matching algorithm. Pattern-matching problems in NIDS have several different forms as follows:

1) Searching for Large Sets of Patterns: the number of known intrusions is growing and is almost surely to continue to do so. This growth was observed in the past in the rapid expansion of the size of the signature database for the Snort NIDS [16].
2) Searching With a Large Alphabet Size: NIDSs' input and signatures have no restrictions on the alphabet. In short, any byte of input can contain any of the 256 possible values, and hence we are dealing with an alphabet of size 256. With respect to most string matching literature this is a large alphabet. Typical alphabet sizes considered in string matching literature are: 4, for DNA/RNA sequences; 52, for the English dictionary; or 128 for ASCII. However, it may be used to search for binary patterns in network packets resulting in requiring them to work on a larger alphabet of size 256.
3) Searching With a Wide Range of Pattern Lengths: The lengths of individual keywords within a keyword set can have great consequences on the performance and memory requirements of an algorithm used for matching. A requirement of a NIDS signature matching is that the algorithm must be capable of handling patterns of various lengths.

In this paper, we present a new algorithm for multiple-pattern exact matching. The paper is organized as follows. Section II surveys on the most significant algorithms for multiple pattern matching algorithms like Aho-Corasick, Commentz Walter, and Wu Manber. Section III presents our proposed algorithm. In Section IV, a comparative study of various algorithms is described. Finally, Section V is for conclusion and our further works.

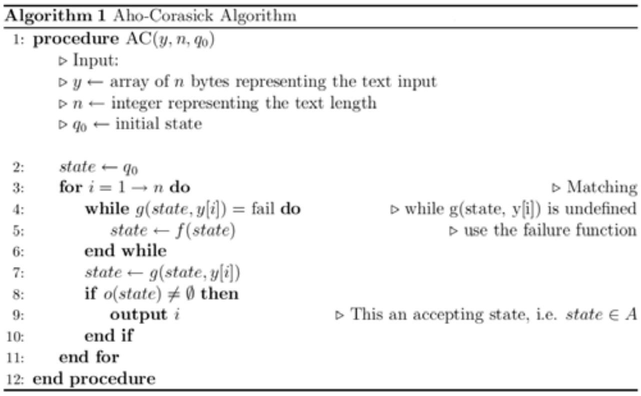
II. Related Works

A. Aho-Corasick Algorithm (AC)

The Aho-Corasick [2] algorithm was proposed in 1975 at Bell Labs by Alfred Aho and Corasick is an extension of the KMP algorithm and remains, to this day, one of the most effective pattern matching algorithms when matching pattern sets. The idea of AC algorithm is that a finite automaton is constructed using a set of keywords during the pre-computation phase of the algorithm and the matching involves the automaton scanning the input text string, reading every character in input string exactly once and taking constant time for each read of a character.

Initially, the AC algorithm combines all the patterns in a set into a syntax tree which is then converted into a non-deterministic automaton (NFA) and, finally, into a deterministic automaton (DFA). The resulting finite state machine is then used to process the text one character at a time, performing one state transition for every text character. A pattern in patterns set *P* has matched whenever the finite state machine reaches designated "final" states. The pseudo-code for the matching phase of the AC algorithm is given by Algorithm 1.

Building the AC automaton takes running time linear in the sum of the lengths of all keywords. This involves constructing a keyword tree for the set of keywords and then converting the tree to an automaton by defining the functions g and f and labeling states in A with the keyword(s) matched. The space or memory requirements of the AC algorithm can be taken directly from the automaton built during the pre-computation because it is the only structure used in the matching. Unfortunately the space can be quite large depending on the alphabet and keyword set. In the worst case it would be O ($M|\Sigma|$) where $|\Sigma|$ is the size of the alphabet Σ.



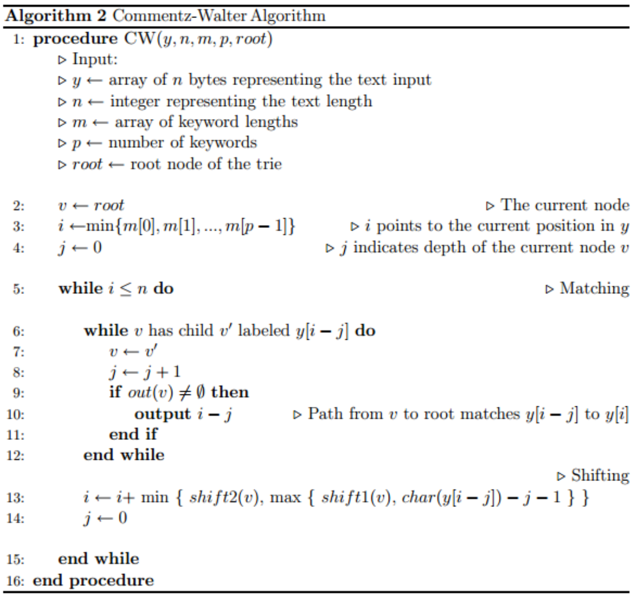
Algorithm 1 Aho-Corasick Algorithm

```
procedure AC($y, n, q_0$)
        ▷ Input:
        ▷ $y \leftarrow$ array of $n$ bytes representing the text input
        ▷ $n \leftarrow$ integer representing the text length
        ▷ $q_0 \leftarrow$ initial state

    $state \leftarrow q_0$
    for $i = 1 \rightarrow n$ do                              ▷ Matching
        while $g(state, y[i])$ = fail do          ▷ while g(state, y[i]) is undefined
            $state \leftarrow f(state)$                 ▷ use the failure function
        end while
        $state \leftarrow g(state, y[i])$
        if $o(state) \neq \emptyset$ then
            output $i$                 ▷ This an accepting state, i.e. $state \in A$
        end if
    end for
end procedure
```

Once the automaton is built, the matching is straightforward and simply involves stepping through the input characters one at a time and changing the state of the automaton- which happens in constant time. At every step we check if there's a match by observing if the current state is an accepting state. Using this simple functionality the AC matcher always operates in $O(n)$ running time, where n is the length of the text, regardless of the number of patterns or their length. The AC algorithm has a significant advantage that every text character is examined only once. A major disadvantage of the AC algorithm is the high memory cost required to store the transition rules of the underlying DFA.

B. Commentz Walter Algorithm (CW)

The popular GNU fgrep utility uses the CW [3] algorithm for multiple string search. CW algorithm combines the Boyer-Moore technique with the AC algorithm. In preprocessing stage, differing from AC algorithm, CW algorithm constructs a converse state machine from the patterns to be matched. Each pattern to be matched adds states to the machine, starting from the right side and going to the first character of the pattern, and combining the same node. In searching stage, CW algorithm uses the idea of Boyer-Moore algorithm. The length of matching window is the minimum pattern length. In matching window, CW scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses a recomputed shift table to shift the window to the right. A multiple string matching algorithm is used to compare from the end of the pattern, like Boyer-Moore, using a finite state machine, like AC. In computer science, the CW algorithm is a string searching algorithm invented by Beate CW. Like the AC string matching algorithm, it can search for multiple patterns at once. The pseudo-code for the CW algorithm is given below:



Algorithm 2 Commentz-Walter Algorithm

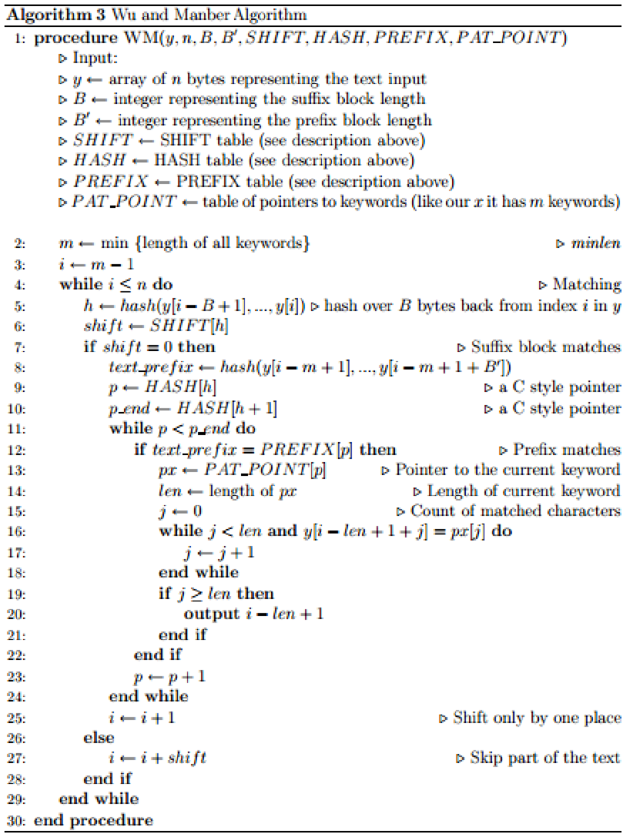
```
procedure CW($y, n, m, p, root$)
        ▷ Input:
        ▷ $y \leftarrow$ array of $n$ bytes representing the text input
        ▷ $n \leftarrow$ integer representing the text length
        ▷ $m \leftarrow$ array of keyword lengths
        ▷ $p \leftarrow$ number of keywords
        ▷ $root \leftarrow$ root node of the trie

    $v \leftarrow root$                                        ▷ The current node
    $i \leftarrow \min\{m[0], m[1], ..., m[p-1]\}$      ▷ $i$ points to the current position in $y$
    $j \leftarrow 0$                              ▷ $j$ indicates depth of the current node $v$

    while $i \leq n$ do                                          ▷ Matching

        while $v$ has child $v'$ labeled $y[i-j]$ do
            $v \leftarrow v'$
            $j \leftarrow j+1$
            if $out(v) \neq \emptyset$ then
                output $i-j$            ▷ Path from $v$ to root matches $y[i-j]$ to $y[i]$
            end if
        end while
                                                                  ▷ Shifting
        $i \leftarrow i+ \min\{\ shift2(v), \max\{\ shift1(v), char(y[i-j]) - j - 1\ \}\ \}$
        $j \leftarrow 0$

    end while
end procedure
```

CW also noted that the quadratic ($O(n*m)$) worst-case running time behavior of the Boyer-Moore algorithm could be improved upon to be linear in n. As such, CW derived two different algorithms called B and B1 which have quadratic ($O(n\times\max\{m[0],\ m[1],...,\ m[p-1]\})$) and linear ($O(n)$) worst-case running times respectively. Algorithm B is the main work of the simpler of CW's literature and has a simpler pre-computation phase than $B1$. Furthermore, $B1$ takes more memory during the pre-computation and search than B by remembering the input text bytes that were already scanned. Both algorithms maintain a pre-computation phase that is linear in the total length of all keywords or $O(M)$, and both achieve slightly sub linear (in n) matching-phase running times on average which could be as good as O ($n/\min\{m[0],\ m[1],\ ...,\ m[p-1]\}$) in the best case. Algorithm B will be described first starting with the functions created during the pre-computation phase [3], [4]. In CW's algorithm B [3] some substrings of the input text y are scanned over and over in the worst case which leads to the quadratic behavior of the running time during the matching phase. In algorithm $B1$ [4] we have the exact same tire as for algorithm B; however, in order to reduce the worst case matching phase running time to linear in n, we use a stack that remembers the characters of the input that have just been scanned. The size of the stack could, in theory, grow as large as n, the length of y, but fortunately only the last *wmax* (where *wmax* is the length of the longest pattern in x) entries of the stack are needed. This means the memory or space requirement during matching is still proportional to the pattern set or M in particular.

C. Wu-Manber Algorithm (WM)

Wu and Manber created the UNIX tool agrep [5] to search for many patterns in files. Wu-Manber algorithm extended BM to concurrently search multiple strings. Instead of using bad character heuristic to compute the shift value, WM uses a character block including 2 or 3 characters. WMstores the shift values of these blocks in SHIFT table and builds HASH table to link the blocks and the related patterns. The SHIFT table and the HASH table are both hash tables which enable efficient search. Moreover, in order to further speed up the algorithm, WM also builds another hash table, the PREFIX table, with the two-byte prefixes of the patterns. This algorithm has excellent average time performance in practical usage. But, its performance is limited by minimum pattern length m since the maximum shift value in SHIFT table equals to m-1 [6]. However, when the pattern set is comparatively large, the average shift value in WM algorithm will decrease and thus the searching performance will be compromised.

The pseudo-code for the matching phase of the WM algorithm is given below:



Algorithm 3 Wu and Manber Algorithm

```
procedure WM($y, n, B, B', SHIFT, HASH, PREFIX, PAT\_POINT$)
        ▷ Input:
        ▷ $y \leftarrow$ array of $n$ bytes representing the text input
        ▷ $B \leftarrow$ integer representing the suffix block length
        ▷ $B' \leftarrow$ integer representing the prefix block length
        ▷ $SHIFT \leftarrow$ SHIFT table (see description above)
        ▷ $HASH \leftarrow$ HASH table (see description above)
        ▷ $PREFIX \leftarrow$ PREFIX table (see description above)
        ▷ $PAT\_POINT \leftarrow$ table of pointers to keywords (like our $x$ it has $m$ keywords)

    $m \leftarrow \min$ {length of all keywords}                        ▷ $minlen$
    $i \leftarrow m - 1$
    while $i \leq n$ do                                               ▷ Matching
        $h \leftarrow hash(y[i-B+1], ..., y[i])$ ▷ hash over $B$ bytes back from index $i$ in $y$
        $shift \leftarrow SHIFT[h]$
        if $shift = 0$ then                                   ▷ Suffix block matches
            $text\_prefix \leftarrow hash(y[i-m+1], ..., y[i-m+1+B'])$
            $p \leftarrow HASH[h]$                                ▷ a C style pointer
            $p\_end \leftarrow HASH[h+1]$                          ▷ a C style pointer
            while $p < p\_end$ do
                if $text\_prefix = PREFIX[p]$ then                 ▷ Prefix matches
                    $px \leftarrow PAT\_POINT[p]$          ▷ Pointer to the current keyword
                    $len \leftarrow$ length of $px$               ▷ Length of current keyword
                    $j \leftarrow 0$                           ▷ Count of matched characters
                    while $j < len$ and $y[i-len+1+j] = px[j]$ do
                        $j \leftarrow j+1$
                    end while
                    if $j \geq len$ then
                        output $i - len + 1$
                    end if
                end if
                $p \leftarrow p+1$
            end while
            $i \leftarrow i+1$                                 ▷ Shift only by one place
        else
            $i \leftarrow i + shift$                              ▷ Skip part of the text
        end if
    end while
end procedure
```

This algorithm uses three tables built during the pre-computation phase: a SHIFT table, a HASH table, and a PREFIX table. The SHIFT table is similar to the Boyer-Moore bad character skip table, and the other two tables are only used when the SHIFT table indicates not to shift-with a shift value of zero because there's a potential match at the current position under examination in the input. As with the Boyer-Moore shifting, the size of the shift is limited to the length of the pattern and in this case, the length of the minimum length pattern (call it *minlen*). Therefore, short patterns in the keyword set inherently make this algorithm less efficient [6]. The analysis of the expected running-time complexity of the main matching phase is shown by WM to be slightly less than linear in *n*, the length of the input text. This analysis assumes both an input text and pattern that are random byte strings with uniform distribution.

D. Other Algorithms

In [7], Michael O. Rabin and Richard M. Karp proposed the Rabin–Karp algorithm is a string searching algorithm in 1987 that uses hashing to find any one of a set of pattern strings in a text. The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each *M*-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next *M*-character sequence. If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match. In [8], J. Kytojoki, L. Salmela, and J. Tarhioin also presented a *q*-Grams based BoyerMoore-Horspool algorithm (BMH). This algorithm cuts a pattern into several *q*-length blocks and builds *q*-Grams tables to calculate the shift value of the text window. This algorithm shows excellent performance on moderate size of pattern set. However, when coming into large-scale scope, it is not good enough both in searching time and memory requirement. Here are also some other popular Backward algorithms which combine the BM heuristic idea and AC automaton idea. In [9], C. Coit, S. Staniford, and J. McAlerney proposed AC_BM algorithm. This algorithm constructs a prefix tree of all patterns in preprocessing stage, and then takes both BM bad character and good suffix heuristics in shift value computation. A similar algorithm called Setwise Boyer Moore Horspool (SBMH) [10] is proposed by M. Fisk and G. Varghese. It utilizes a tire structure according to suffixes of all patterns and compute shift value only using the bad character heuristic. However, these two algorithms are also limited by the memory consumption when the pattern set is large. In [11], C. Allauzen and M. Raffinot introduced Set Backward Oracle Matching Algorithm (SBOM). Its basic idea is to construct a more lightweight data structure called factor oracle, which is built only on all reverse suffixes of minimum pattern length m window in every pattern. It consumes reasonable memory when pattern set is comparatively large. In [12], B. Xu and J. Li proposed the Recursive Shift Indexing (RSI) algorithm for this problem. RSI engages a heuristic with a combination of the two neighboring suffix character blocks in the window. It also uses bitmaps and recursive tables to enhance matching efficiency. These ideas are enlightening for large-scale string matching algorithms. In [13], Zhou proposed MDH algorithm which optimized WM algorithm with multi-phase hash and dynamic-cut heuristics strategies. According to Zhou's experiments, the performance of MDH is superior to WM and some other algorithms. In 2010, Baeza-Yates and Gonnet introduced the Bit-parallelism technique [14]. In which takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing cutting down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word. Bit parallelism is particularly suitable for the efficient simulation of nondeterministic (suffix) automata. In 2013, Zhenlong Yuan et al propose a multi-pattern matching algorithm named TFD for large-scale and high-speed URL filtering [15]. TFD employs Two-phase hash, Finite state machine and Double-array storage to eliminate the performance bottleneck of blacklist filter.

III. Our Proposed Algorithm

Our work is different from these previous efforts as it focuses on building a graph transition structure and dynamic linked list search technique for multipattern matching that can handle a large number of patterns, and can easily be combined with any existing multi-pattern matching application.

To illustrate the process of the algorithm, we consider the following example:

Patterns set *P* = {"search", "ear", "arch", "chart"}
T= "strcmatecadnsearchof".

A. Preprocessing stage

According to AC algorithm approach, we will build an automaton as follows Fig. 1.

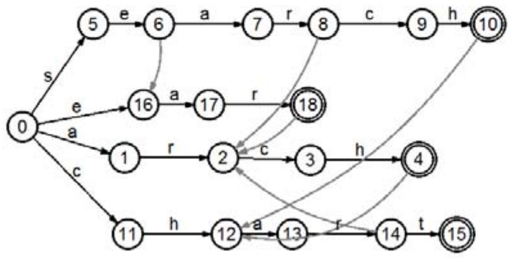
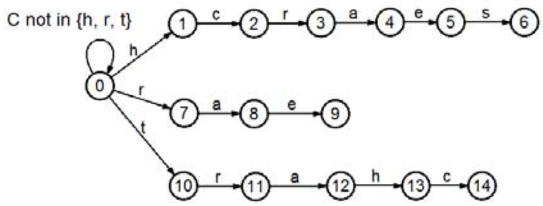


Fig. 1. Preprocessing stage of AC algorithm.



State machine and goto function.

nodes	4	6	9	14
out	{arch}	{search}	{ear}	{chart}

Output function.

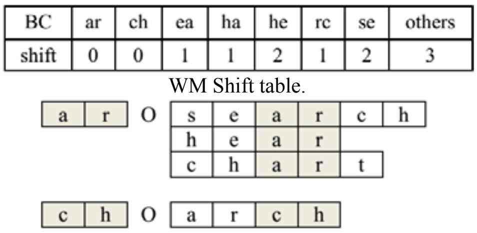
	a	c	e	h	r	others
0	1	1	2	3	1	3
7	1	1	1	2	1	3
8	1	1	1	1	1	2
9	2	2	2	2	2	2
others	3	3	3	3	3	3

CW Shift table.

Fig. 2. Preprocessing stage of CW algorithm.

The CW algorithm creates a basic tire data structure using the reversed keywords. Each node *v,* except the root node, is labeled with a character (byte) from a pattern. A basic CW style tire for pattern set *P* shown in Fig. 2. The output receives a tire node *v* and returns whether or not the path to the root from node *v* represents a keyword. If so, out returns the keyword. Otherwise it returns nothing (the empty set is denoted ∅), and the path from *v* to the root is simply a proper suffix of one or more pattern in *P*.

In preprocessing stage, the WM algorithm builds three tables, a SHIFT table, a HASH table, and a PREFIX table. The HASH and PREFIX tables are used when the shift value is 0. Fig. 3 shows the SHIFT table and HASH table for *B*=2.



BC	ar	ch	ea	ha	he	rc	se	others
shift	0	0	1	1	2	1	2	3

WM Shift table.

a r O s e a r c h
h e a r
c h a r t

c h O a r c h

WM Hash table.

Fig. 3. Preprocessing stage of WM algorithm.

In preprocessing stage, we create a graph transition structure representing pattern set *P*. Our graph *G* has *n* levels (*n* is the maximum length of patterns in *P*), at every level we just have to keep the different characters in each patterns. Our graph structure for patterns set *P* = {"search", "ear", "arch", "chart"} shows in Fig. 4 bellows:

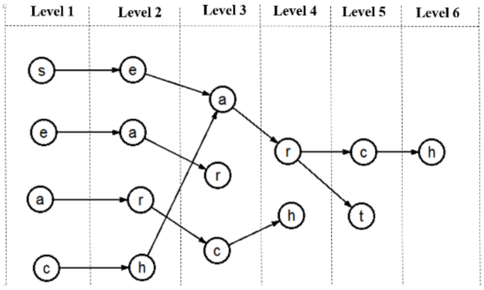
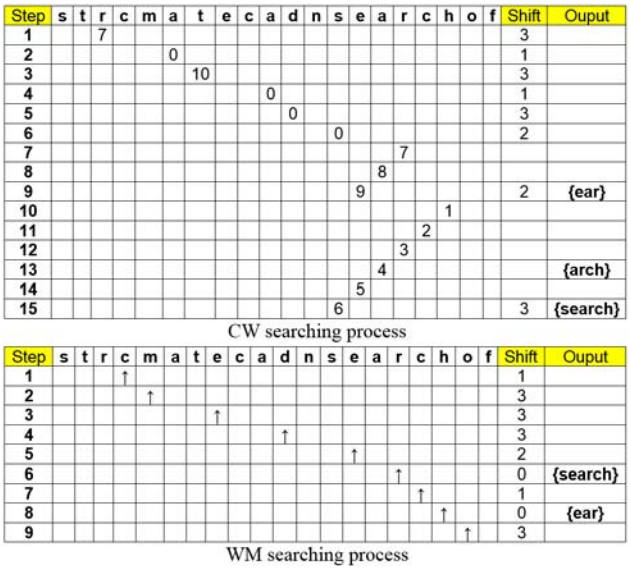


Fig. 4. Preprocessing stage of our algorithm.



Step	s	t	r	c	m	a	t	e	c	a	d	n	s	e	a	r	c	h	o	f	Shift	Ouput
1			7																		3	
2						0															1	
3							10														3	
4										0											1	
5											0										3	
6													0								2	
7																7						
8															8							
9														9							2	{ear}
10																		1				
11																	2					
12																3						
13															4							{arch}
14														5								
15													6								3	{search}

CW searching process

Step	s	t	r	c	m	a	t	e	c	a	d	n	s	e	a	r	c	h	o	f	Shift	Ouput
1				↑																	1	
2					↑																3	
3								↑													3	
4											↑										3	
5														↑							2	
6																↑					0	{search}
7																	↑				1	
8																		↑			0	{ear}
9																			↑		3	

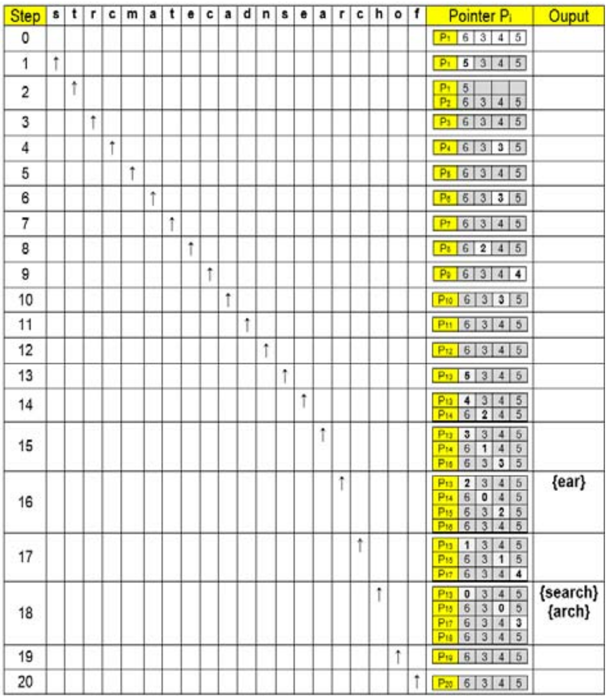
WM searching process

Fig. 5. Searching stage of CW and WM algorithms.

First, following our approach, the storage space is reduced by just storing the different characters at each level. This memory requirements equal to or less than the storage space of DFA matching and state machine and goto function of the AC, WM, CW algorithm. Second, as our algorithm does not use the SHIFT and HASH table, it should reduce the construction time tables and table storage space.

B. Searching Stage

To analyze the search process, we assume the input string is *T*= "strcmatecadnsearchof". The searching stage of Aho Corasick is to walk through the automata for any transition; if so, the transition takes place, otherwise check the failure function. The CW algorithm use 15 steps to detect three patterns output= {ear, arch, search}, as seen in Fig. 5. While, the CW algorithm use 9 steps to detect two patterns output= {ear, search}.



Step	s	t	r	c	m	a	t	e	c	a	d	n	s	e	a	r	c	h	o	f	Pointer P_i	Ouput
0																					P_1 6 3 4 5	
1	↑																				P_1 5 3 4 5	
2		↑																			P_1 5 P_2 6 3 4 5	
3			↑																		P_3 6 3 4 5	
4				↑																	P_4 6 3 3 5	
5					↑																P_5 6 3 4 5	
6						↑															P_6 6 3 3 5	
7							↑														P_7 6 3 4 5	
8								↑													P_8 6 2 4 5	
9									↑												P_9 6 3 4 4	
10										↑											P_{10} 6 3 3 5	
11											↑										P_{11} 6 3 4 5	
12												↑									P_{12} 6 3 4 5	
13													↑								P_{13} 5 3 4 5	
14														↑							P_{13} 4 3 4 5 P_{14} 6 2 4 5	
15															↑						P_{13} 3 3 4 5 P_{14} 6 1 4 5 P_{15} 6 3 3 5	
16																↑					P_{13} 2 3 4 5 P_{14} 6 0 4 5 P_{15} 6 3 2 5 P_{16} 6 3 4 5	{ear}
17																	↑				P_{13} 1 3 4 5 P_{15} 6 3 1 5 P_{17} 6 3 4 4	
18																		↑			P_{13} 0 3 4 5 P_{15} 6 3 0 5 P_{17} 6 3 4 3 P_{18} 6 3 4 5	{search} {arch}
19																			↑		P_{19} 6 3 4 5	
20																				↑	P_{20} 6 3 4 5	

Fig. 6. Searching stage of our algorithms.

In the searching stage, we use a list of pointers for searching to minimize the memory space. The maximum number of element in the pointer is equal the number of patterns. We initialize the pointer value by the length of each pattern. The structure of pointer is shown below:

P_i	6	3	4	5

While browsing on the input string, at every step we initialize pointer P_i (*i* corresponds to the current character position). If the current character matches the character of any pattern P_i then the pointer value will be reduced by 1, otherwise the value will be removed. If the current character does not match in graph *G* then remove P_i, we continue to maintain the operation of the P_i. The operation of searching stage of our algorithms is illustrated in Fig. 6.

The number of steps in our algorithm is the length of the input string *T*. For a text of length *n,* maximum length of the pattern is *L*, and *m* is number of patterns. The worst-case running time is *O* (*n*×*m*×*L*), though the average case is often much better as we do not always maintain all *m* pointers P_i at the same time.

C. Our Proposed Algorithm

The pseudo-code for our algorithm is given below:

Algorithm 4. Our Algorithm

```
Procedure DNL (T, n, m, p, G)
Input:
            ▷ T ← array of n byte representing the text input
            ▷ n ← integer represent the text length
            ▷ P[j] ← array of patterns
            ▷ $P_0$ ← array of keyword lengths $P_0[j]$ (j=1…m)
            ▷ m ← number of patterns
            ▷ G ← graph of pattern P
            ▷ S ← Set of $P_i$
S = Ø;
for i=1 → n do
    Init pointer $P_i = P_0$;
    S = S U {$P_i$}
    If (T[i] in G) and ($P_j$ in S) then
     $P_j$[position of T[i] in P[j]] = $P_j$[position of T[i] in P[j]] -1;
      If ($P_j[k] = 0$) then
          Output P[k] detected;
          Remove $P_j$;
      endif
      If ($P_j$ in S) and ($P_j$ not change)  then Remove $P_j$
       endif
    endif
end for
End procedure
```

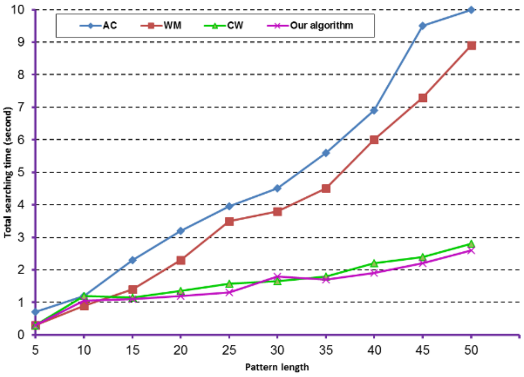


Fig. 7. Time with fixed pattern number.

IV. Experiment and Result

Experiments are designed to verify the performance of our proposed algorithm both on searching time and space occupation, and to compare it with AC, CW and WM algorithms. We have implemented in pure C source code of AC, CW and WM excerpted from Snort version 2.8.3.1 [16]. All the experimental results reported were obtained on PC with Intel Pentium 3 GHz CPU Dual Core and 2 GB memory. Comparisons are done from two aspects that complement each other: one with fixed number of patterns and varying pattern lengths; the other with fixed pattern length and varying pattern numbers. All patterns are generated randomly with equal length, but the text is self-correlated, which means part of the text is generated randomly but the rest is generated according to that part. For instance, if we need a text of the length 10,000, we generate the first 500 characters randomly, and the rest of the text is generated like this: each time we pick out several characters from the first 500 characters and append them to the end of the text until the length arrives at 10,000; the position and the length of each pick are both random. The aim to derive such text is to guarantee that there exist a number of matches, as we know random patterns and text will result in few matches. Thus we can better simulate the patterns and traffic in real network. Though these are only meaningless characters, they provide a meaningful reference of the performance. Fig. 7 shows the total searching time for a text with 50,000 characters and 500 patterns with a varying length.

The time in Fig. 8 is the accumulative time of 1,000 repeated times of lookup.

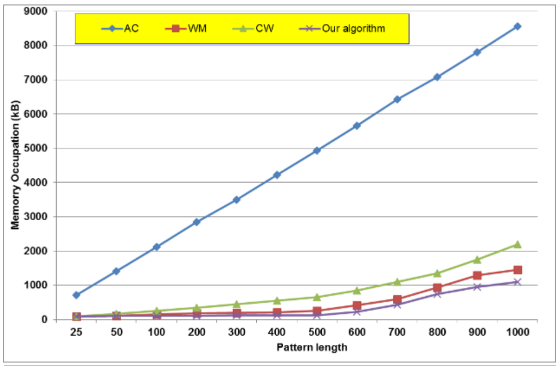


Fig. 8. Space with fixed pattern number.

V. Conclusion

In this paper, we have presented a new algorithm for multiple pattern exact matching. Our approach reduces character comparisons and memory space based on graph transition structure and search technique using dynamic linked list. Theoretical analysis and experimental results, when compared with previously known pattern-matching algorithms, shows that our algorithm is highly efficient in both space and time.

Acknowledgment

This research is partly supported by the QG.12.21 project of Vietnam National University, Hanoi

References

[1] A. Apostolico and Z. Galil, *Pattern Matching Algorithms*, Oxford University Press, New York, USA, 1997.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[3] B. C. Walter, "A string matching algorithm fast on the average," in *Proc. the 6th Colloquium on Automata, Languages and Programming*, pp. 118–132, London, UK, Springer-Verlag, 1979.

[4] B. C. Walter, "A string matching algorithm fast on the average," Technical Report, IBM Heidelberg Scientific Center, 1979.

[5] S. Wu and U. Manber, "Agrep – A fast approximate pattern-matching tool," in *Proc. USENIX Winter 1992 Technical Conference*, 1992, pp. 153–162, San Francisco, CA.

[6] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Technical Report TR-94-17, Department of Computer Science, Chung-Cheng University, 1994.

[7] H. C. Thoma, E. L. Charles, L. R. Ronald, and S. Clifford, *The Rabin–Karp Algorithm*, *Introduction to Algorithms*, Cambridge, Massachusetts: MIT Press, pp. 911–916, 2001.

[8] J. Kytojoki, L. Salmela, and J. Tarhio, *Tuningstring Matching for Huge Pattern Set?* vol. 2676, pp. 211–224, 2003.

[9] C. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort," *DARPA Information Survivability Conference and Exposition*, pp. 367–373, 2001.

[10] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," Technical Report CS2001-0607 (updated version), University of California-San Diego, 2002.

[11] C. Allauzen and M. Raffinot, “Factor oracle of a set of words,” Technical report 99-11, Institut Gaspard-Monge, Universite De Marne-la-Vallee, 1999.

[12] B. Xu, X. Zhou, and J. Li, “Recursive shift indexing: a fast multi-pattern string matching Algorithm,” *Applied Cryptography and Network Security*, vol. 3989, 2006.

[13] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li, “Mdh: A high speed multiphase dynamic hash string matching algorithm for large-scale pattern set,” *Information and Communications Security*, pp. 201–215, 2007

[14] M. O. Kulekci, “BLIM: a new bit-parallel pattern matching algorithm overcoming computer word size limitation,” *Mathematics in Computer Science*, vol. 3, no. 4, pp. 407-420, 2010.

[15] Z. L. Yuan, B. H. Yang, X. Q. Ren, and Y. B. Xue, “A multi-pattern matching algorithm for large-scale URL filtering,” in *Proc. the 2013 International Conference on omputing, Networking and Communications, Communications and Information Security Symposium*, pp. 359-363, 2013.

[16] M. R. Snort, “Lightweight intrusion detection for networks,” in *Proc. of the 1999 USENIX LISA Systems Administration Conference*, pp. 229-238, 1999.

Dac-Nhuong Le received the BSc degree in computer science and the MSc degree in information technology from College of Technology, Vietnam National University, Vietnam, in 2005 and 2009, respectively. He is currently a lecture at the Faculty of information technology in Haiphong University, Vietnam. His research interests include algorithm theory, computer network and networks security.

Nguyen Dang Le received the BSc degree in computer science and the MSc degree in information technology from College of technology, Vietnam National University in Hanoi, Vietnam, in 1996 and 2005, respectively. He currently works in Haiphong University, Vietnam. His research interests include algorithm theory, network and wireless security.

Vinh Trong Le received the MSc degree in information technology from Faculty of Mathematics, Mechanics and Informatics, Hanoi University of Science, Vietnam National University in 1997, PhD degree in computer science from Japan Advanced Institute of Science and Technology in 2006, respectively. He is currently on associate professor at the Faculty of Mathematics, Mechanics and Informatics, Hanoi University of Science, Vietnam National University. His research interests include algorithm theory, network and wireless security.