ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 2, Issue 4, pp: (254-264), Month: October - December 2014, Available at: www.researchpublish.com

# New Inverted Lists-Multiple String Patterns Matching Algorithm

Chouvalit Khancome[1], Veera Boonjing[2]

[1]Department of Computer Science, Ramkhamhaeng University, Bangkok, Thailand.
[2]International College, King Monkut's Institute of Technology at Ladkrabang, Ladkrabang, Bangkok, 10520, Thailand.

---

*Abstract:* **Multiple string pattern matching is one of the most important fundamental in solving string processing. This principle simultaneously searches for all patterns appeared in a large given text. A new algorithm to this problem called *"IVL-MSPM"* is presented. The new solution adapted the "*inverted lists*"(shown in [39]) for accommodating the collection of patterns. The experimental results showed that the proposed data structures were constructed faster and more economic on space than the well known data structures: Trie, Reverted-Trie, and Suffix tree. The searching results were faster than the traditional algorithms especially small number of patterns and small text sizes.**

*Keywords:* **Multiple Pattern String Matching, String Patterns Matching, Inverted Lists, String Processing, Static Dictionary Matching.**

---

## I. INTRODUCTION

Multiple string pattern matching principle, which is often derived from single string pattern matching, simultaneously searches for all occurrences of patterns $P=\{p^1,p^2,..,p^r\}$ appeared in a given text $T=\{t_1,t_2,t_3...t_n\}$ over a finite alphabet $\sum$ . Several fields in computer science employed this principle to solve their problems. For instances, the operating system commands used the classic algorithms to embed in their command sets e.g., Unix *grep* command using Commentz-Walter [3] and *agrep* using Wu-Manber[23]. Including the intrusion detection systems used the famous algorithms to implement such as SNORT system using Aho-Corasick[1], Commentz-Walter [3], and Wu-Manber[23]), SetHorspool [9], and so on.

Even though this principle is viewed as the classic fundamental, the current issues are still interested in new aspects of solutions. For example, the new solutions, which are shown in [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], and [38], are the related multiple string matching algorithms that are always provided the new ideas and the new ways to improve the previous algorithms. Traditionally, Trie, Reverted-Trie, and Suffix tree are the data structures adopted for algorithms. However, these data structures take a large space and hard to implement. Then, the new data structure, which is easier to construct and more economic on space, is required.

This research article provides the new data structure that takes less space, easy to create, and faster search. The main contribution of this article is a new algorithm of multiple string pattern matching using the inverted lists (Inverted Lists-Multiple String Patterns Matching: *IVL-MSPM*). This solution takes $O(|P|)$ time and $O(|\sum|+|P|)$ space for preprocessing phase where $|\sum|$ is the alphabet size. As well as, the searching phase takes $O(n+locc)$ time where $n$ is the length of the given text, and $locc$ is the number of the matched characters that includes the mismatched time.

Experiments showed that the processing time were compared with well known structures: Trie, Reverted-Trie, and Suffix tree. The results illustrated that the inverted lists structure was constructed faster and was more economic on space than the structures to be compared. Furthermore, the searching time were significantly faster than Aho-Corasick[1] and SetHorspool[9] in the cases of a small number of patterns.

The rest of this article is organized as follows. Section 2 indicates the related researches. Section 3 shows the basic definitions and the details of inverted lists construction. Section 4 presents the proposed search algorithm. Section 5 shows the experimental results to compare the processing times, the memory usages, and the searching results. Section 6 is the conclusion.

## II. RELATED RESEARCHES

Traditionally, many data structures for multiple pattern string matching algorithms were directly derived from the principle of single string matching (employing Trie, Bit-parallel, Hashing table, and the combined data structures).

The first Trie-based algorithm, which scans the given text from the left to the right, is the Aho-Corasick [1]. This algorithm was derived from the KMP [7] to create the AC Trie for storing the patterns. Obviously, it takes a linear time for searching in a given text. The second algorithm introduced by Commentz-Walter [3] and it was inherited from the single matching of BM [4] to create the reverted Trie for accommodating patterns $P$. This algorithm scans the text $T$ by the suffix approach (i.e., scanning the text from the right to the left of searching window) in sub-linear time. The third one is the SetHorspool algorithm [9] called the easy version of Commentz-Walter [3]. The algorithm employed the reverted Trie and the shift table to store the patterns. Other solutions [18], [19], MultiBDM[20], SBOM[5], SDBM[9], [15], and [11] improved the Trie for decreasing the searching time, but they are more complex in worst case scenario and inefficient in searching (shown in [9]).

Based on Bit-parallel, the single Shift-Or and the single Shift-And were applied to the Multiple Shift-And [8] and the Multiple BNDM[9], and [10] (shown by Navarov [9]). These are restricted by the computer word architectures.

The first hashing idea was presented by Karp and Rabin [14] in the single string matching, but it took the worst case as the simplest way of searching. Then Wu and Manber [23] presented the algorithm by creating the block of pattern and implementing the hashing table to store the patterns. The solution of [25] improves the Wu and Manber[23] for saving the searching time, but the worst case scenario is not improved.

The other ideas (e.g., [16], [17] and [24]) combine several structures to improve the time complexity such as q-gram [16] and the partitioning technique [17], but the exhaustive worst case still remains. With the evident, the algorithms, which based on Trie, are more efficient than the others. There are the valuable literature reviews provided in [9], [24], and [16].

In a part few years, solutions [28]-[30] improved Trie structure to accommodate the patterns especially [29] shown minimal space of solution. Other solutions, which employed those classic data structures (e.g., Trie, Bit-parallel and Hashing), can be found in [27], [32], [33].

## III. INVERTED LISTS STRUCTURE

The new algorithm of dynamic dictionary matching using the inverted lists structure [39] accommodated the dynamic patterns in a linear-time. This article derived that structure to store the multiple string patterns. Therefore, some definitions and some algorithms are similar to [39] in this section.

Let $p^i$ be the pattern in $P$, and $p^i$ contains the string $\{c_1,c_2,...,c_m\}$ where $1 \le i \le r$ . Let $\sum$ be a finite alphabet cover $P$ and $T$, and let $\lambda$ be any characters which $\lambda \subseteq \sum$ . The following sub-sections show how to create the inverted lists structure, which are divided to the basic definitions and the pre-processing phase.

### A. Basic Definitions:

The posting lists are the pairs of indices between all characters in $\sum$ and their positions in $P$. The individual posting lists are grouped to the new form called the inverted lists. The following definitions examine their details.

**Definition 1** Let $P=\{p^1, p^2, p^3,..., p^r\}$ be a set of patterns where $p^i$ is the individual pattern $i^{th}$ of $m$ character $\{c_1c_2c_3...c_m\}$ and $1 \le i \le r$ .

**Definition 2** Every character $c_k$ in $P$ can be represented as an individual of posting list as follows.

1. If $k<m$ a character $c_k$ is $c_k$:<$k$:0:$i$> and denoted by $\varphi_0^{k_i}$ , or

2. if $k=m$ a character $c_k$ is $c_k$:<$k$:1:$i$> and denoted denoted by $\varphi_1^{k_i}$ , where $1 \le k \le m$ .

**Definition 3** Let $l_{max}$ be the maximum length of patterns in $P$, and $\varepsilon$ be the position of the same character $\lambda$ appeared in the various patterns in $P$ where $1 \le \varepsilon \le l_{\max}$ and $\lambda \subseteq \sum$ . The posting lists of $\lambda$ are { $\varphi_0^{\varepsilon_i}$ , $\varphi_0^{\varepsilon_l}$ ,…, $\varphi_0^{\varepsilon_p}$ , $\varphi_0^{\varepsilon_q}$ } or { $\varphi_1^{\varepsilon_i}$ , $\varphi_1^{\varepsilon_l}$ ,…, $\varphi_1^{\varepsilon_p}$ , $\varphi_1^{\varepsilon_q}$ } where $1 \le \{i, l, \ldots, p, q\} \le r$ . $\lambda_{\varepsilon,0}$ represents { $\varphi_0^{\varepsilon_i}$ , $\varphi_0^{\varepsilon_l}$ ,…, $\varphi_0^{\varepsilon_p}$ , $\varphi_0^{\varepsilon_q}$ }, and $\lambda_{\varepsilon,1}$ represents { $\varphi_1^{\varepsilon_i}$ , $\varphi_1^{\varepsilon_l}$ ,…, $\varphi_1^{\varepsilon_p}$ , $\varphi_1^{\varepsilon_q}$ }.

**Definition 4** The inverted list (i.e., IVL) of alphabet $\lambda$ occurring $\lambda_{\varepsilon,0}$ is defined as $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ occurring $\lambda_{\varepsilon,1}$ .

**Definition 5** The hashing table provided to store $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$ is called the inverted lists table and denoted as $\tau$ .

**Definition 6** The temporary space provided for any inverted lists $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$ is called the *SET*.

### B. *Inverted Lists for Multiple String Patterns:*

The inverted lists construction is begun by each character in $P$ to be read and generated to the data structure. Before generating, the empty table $\tau$ will be created for all alphabets of $\sum$ . For putting the inverted lists to $\tau$ , if the pattern has already filled in the table, only the number of pattern is added to the corresponding inverted lists. Otherwise, the new inverted list will be created and added into the table. The algorithm proceeds as described below.

**Algorithm 1** Pre-processing phase

**Input**: $P=\{p^1, p^2, \ldots, p^r\}$

**Output**: table $\tau$ of $P$

```
1. Create empty table $\tau$ and add all characters $\lambda \subseteq \sum$
2. For $i=1$ To $r$ Do
3.   For $j=1$ to $m$ of $p^i$ Do
4.     If $\tau$ does not exist $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ Then
5.       add $\varphi_0^{j_i}$ if $j<m$ or $\varphi_1^{j_i}$ if $j=m$ into the first Level of $\tau$
6.     Else
7.       add $i$ into the second level of $\tau$ ($I_{char(j)_{j,0}}$ if $j<m$ or $I_{char(j)_{j,1}}$ if $j=m$)
8.     End of If
9.   End of For
10. End of For
11. Return $\tau$
```

Example 1 Adding the inverted lists of $P=\{aab, aabc, aade\}$ to $\tau$ .

The table $\tau$ is created from line 1, and each pattern is read one by one from line 2 to line 10. In this case, line 2 will be processed to read the pattern $p^1$ to $p^3$. Each round of line 3 will be repeated to equal the length of each pattern $p^i$. From example 1, the inverted list *a:<1:0:{1}>*, *<2:0:{1}>*, and *b:<3:1:{1}>* are built from $p^1=aab$, and every inverted list is put into $\tau$ . In the next loop of line 2, we consider *a:<1:0{2}>*, *<2:0:{2}>* of $p^2=aabc$ and put only the number of pattern by line 4 and line 5, respectively. The results are *a:<1:0:{1,2}>* and *<2:0:{1,2}>*. The inverted lists of '*b*' and '*c*' are new inverted lists which are generated into the table $\tau$ as a first round. In the last round, the characters '*a*' of $p^3=aade$ are processed as the second round, but the inverted list of '*d*' and '*e*' are generated as the new inverted list. All inverted lists are scattered to the table (e.g., shown in table 1).

Example 2 Implementation of individual inverted lists from $P=\{aab, aabc, aade\}$ to $\tau$ .

Firstly, all individual posting lists are grouped by the form of *character : position :{set of number of patterns}*. Therefore, all individual posting lists above can be grouped to the new form below.

*a: <1:0:1>,<1:0:2>,*

*<2:0:1>,<2:0:2>,*

*<3:0:1>,<3:0:2>,*

*b: <1:1:3>,<2:0:3>,*

*c: <2:1:4>,*

*d: <3:0:3>,*

*e: <3:1:4>.*

A general outlook of inverted lists in such a hashing table is demonstrated in figure 1.

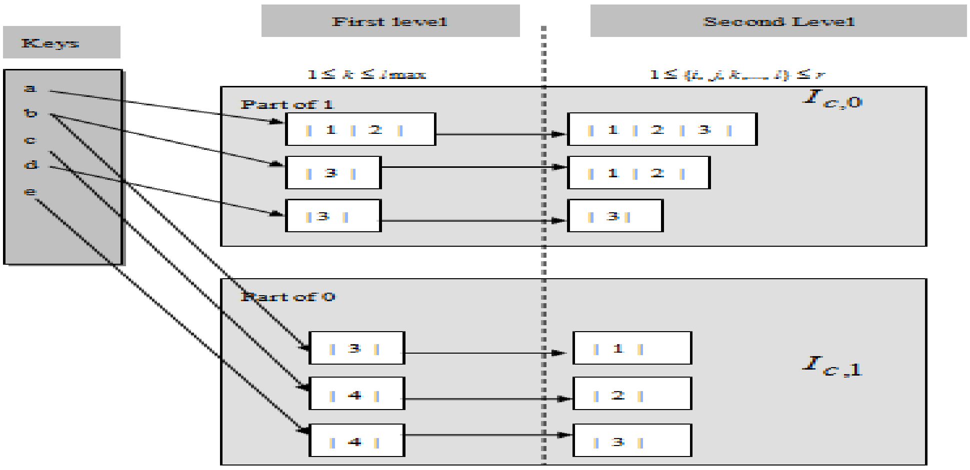

**Fig. 1 implementing of the perfect hashing table $\tau$**

Before complexity analysis, algorithm 1 is referred for proof of how to access the inverted lists table, and Lemma 1 and Theorem 1 show these proofs.

**Lemma 1** To access $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ takes O(*1*) time where $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$ are the inverted lists of $\lambda$ in $\tau$ .

**Proof** Each alphabet $\lambda$ is a unique character in $\sum$ . Thus, it is a unique character, and it has only one key to access the inverted list groups in $\tau$ . The table $\tau$ is implemented by the hash table; hence, each operation to retrieve $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ takes one time. By the hashing properties, each operation takes O(*1*) time. Therefore, to access $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ takes only O(*1*) time. ☐

**Theorem 1** The time complexity for generating *P* to the inverted lists takes O(|*P*|) time where |*P*| is the sum of all patterns length in *P*.

**Proof** Given |*P*| be the sum of lengths of $\{p^1, p2, p3, \ldots, p^r\}$. All length of patterns are denoted by $|p^1|$, $|p^2|$, $|p^3|,\ldots, |p^r|$. For the initial step, the table $\tau$ is built in O(*1*) time. As soon as the table $\tau$ is built completely, every pattern is scanned by the loop of line 2 in *r* rounds. Each round of line 2 stimulates line 3 into operation. Each execution of line 3 equals the length of each pattern. This step takes the processing time as $|p^1|+|p^2|+|p^3|+\ldots+|p^r|=|P|$, and it reaches to the hypothesis step by the last character of $p^r$. Therefore the inverted lists construction takes |*P*| time. This leads to O(|*P*|) time complexity; meanwhile, line 4, line 5, and line 7 access the table $\tau$ in O(*1*) by Lemma 1. Hence, the pre-processing time is proved by O(|*P*|) time. ☐

The table $\tau$ contains the alphabet $\lambda$ over a finite alphabet $\sum$ . The inverted lists of $\lambda$ are $I_{\lambda_{\varepsilon,0}}$ and $I_{\lambda_{\varepsilon,1}}$ which are stored in the second column. The space depends on the number of $\lambda$ and the posting lists in $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ , which takes $O(|\sum|+|P|)$ space for accommodating the inverted lists of $P$.

**Theorem 2** The table $\tau$ requires $O(|\sum|+|P|)$ space for accommodating the whole inverted lists of $P$ where $|P|$ is the sum lengths of patterns in $P$, and $\tau$ is the inverted lists table.

**Proof** All patterns in $P$ contain the various strings over $\lambda$ with the size $|\lambda|$ where $\lambda \subseteq \sum$ . The hypothesis is that all characters are generated to inverted lists and added into the table $\tau$ with $|P|$ space. Given the length of $P$ be $|p^1|$, $|p^2|$, $|p^3|$, …, $|p^r|$, each $p^i$ contains the string of characters $\{c_1c_2c_3...c_m\}$ where $1 \le i \le r$ . The length of this string is denoted by $|p^i|$. For the initial step, the first column of table $\tau$ is created with $|\lambda|$ size. If $\lambda$ equals $\sum$ , then the maximum space also equals $|\sum|$ space. As the initial step, both cases lead to $O(|\lambda|)$ or $O(|\sum|)$ space. Each inverted list is created by the preprocessing phase for all patterns such that each inverted list of string $\{c_1c_2c_3...c_m\}$ in each $p^i$ takes one space per one posting list. Thus the sum of space equals $|p^1|+|p^2|+|p^3|+\ldots+|p^r|=|P|$. Therefore, the overall space is $O(|\sum|+|P|)$ space. ☐

## IV. PROPOSED SEARCHING ALGORITHM

The search employs the variables '*N*', '*SHIFT*', 'pos' and '*n*' to propel the searching window where 'N' is the target position in the text, '*SHIFT*' is the initial position of the next searching window, '*pos*' is the required position of inverted lists to be matched, and 'n' is the length of the text *T*. In addition, '*SET1*' and '*SET2*' are the temporary variables used to operate the continuity and the matching during the searching execution.

In initial searching, the variables *N* and *SHIFT* are initiated to enforce the searching window, and the variable 'pos' is used to control the required position in the text *T*. Afterwards, the text is scanned and searched from the left to the right. While scanning, we look for the inverted lists in $\tau$ , and the positions are equal 'pos' at the row of $\lambda$ by *text[N]* storing to *SET1* or *SET2*. Algorithm 2 illustrates this methodology.

```
Algorithm 2: IVL-MSPM
Input : $P = \{p^1, p^2 p^3, . . . , p^r\}(\tau)$ and $T = t_1t_2t_3 . . . t_n$
Output : all occurrences are reported, and T is scanned.
$N=1, SHIFT=2, pos=1, SET1=SET2=\{\}, RESULTS=\{\}$
While $N <= n$ and $SHIFT <= n$ Do
If $pos = 1$ Then
$SET1 \leftarrow \tau(text[N], 1)$
Else
    $SET2 \leftarrow \tau(text[N], pos)$
End of If
$SET1 \leftarrow SET1 \cap SET2$
Store the matched position to RESULTS if SET1 contains $\varphi(N^{pos}),1$
If $SET1 <> \{\}$ Then
$N{+}{+}$ and $pos{+}{+}$
Else
    $N = SHIFT, SHIFT{+}{+}$ and $pos = 1$
End of If
End of While
Return RESULTS
```

With careful attention to Algorithm 3, the special function is *INTERSECTION*(∩). This function considers the position in *SET1* and *SET2* and searches for the matched positions and returns the continuity of inverted lists into *SET1* for the next comparison. The intersection function can be efficiently implemented through a simple procedure indicated as follows.

**Algorithm 3** *INTERSECTION (SET1, SET2, N, pos)*

```
1. Report the successful matching at N if IVL in SET2 containing $\varphi_1^{(pos)}$
2. Add every IVL in SET2 that are continued (i.e., $\varphi_0^{(pos)}$ ) from SET1 to TEMP
3. Return TEMP
```

Every comparison takes the inverted lists from the table $\tau$ to the sub-hash variable *SET1* or *SET2*. Whenever the inverted lists are taken, the *INTERSECTION* is invoked to operate the continuity and occurrence of patterns. For instance, if *SET1={<1:0:{1,2}>}* and *SET2={<***2***:0: {1,3}>}* operate, then the intersection is ordered by the positions *1* to *2* between *SET1* and *SET2*. In this case, the first consideration is by the sequence of inverted lists in *SET1* which are described by *SET2*. Thus, the pattern number *{1}* in *SET1* is described by position *{1}* in *SET2*, while the required position is '*2*' in *{<2:0:{1,3}>}*. If the inverted lists are considered, the indicated number '*0*' and '*1*' are also considered, and the occurrence is reported if the indicated number is '*1*'. Consequently, the indicated number of *SET2* is *{<2:***0***:{1,3}>}*, which is not the last character of the pattern: it does not match at this position. Therefore, the result is *SET1={<2:0:{1}>}.*

**Lemma 2** The time complexity to take $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$ from *SET* uses O(*1*).

**Proof** Because the *SET* contains only one row of inverted lists; hence, the inverted lists in the *SET* can be taken only once which implies O(*1*) time. □

As shown above, the intersection between *SET1* and *SET2* finds a set of numbers in *SET2* that continue from *SET1*. Importantly, this method reports the matched position whenever the terminate status equals *1*. The continuity is concentrated on the posting lists in *SET1* described by *SET2*. If the numbers of positing lists in *SET2* are superior to *SET1* one position, they are kept to *SET1* for the next operation.

**Lemma 3** The intersection between *SET1* and *SET2* takes O(*1*) time.

**Proof** Let *SET1* and *SET2* be the instances of *SET*. *SET1* contains the inverted list groups $I_{\lambda_{\varepsilon 1,0}}$ and/or $I_{\lambda_{\varepsilon 1,1}}$, and *SET2* contains the inverted list $I_{\lambda_{\varepsilon 2,0}}$ and /or $I_{\lambda_{\varepsilon 2,1}}$. Then every operation of *SET* can be solved by Lemma 2 in O(*1*) time. Hence, every operation to access $I_{\lambda_{\varepsilon 1,0}}, I_{\lambda_{\varepsilon 1,1}}, I_{\lambda_{\varepsilon 2,0}}$ and $I_{\lambda_{\varepsilon 2,1}}$ takes O(*1*) time by Lemma 1. □

**Theorem 3.** Searching for all occurrences of patterns in $P=\{p^1, p^2, p^3,..,p^r\}$ which occur in the text $T=\{t_1t_2t_3...t_n\}$ takes *O(n+locc)* time where *n* is the length of *T*, and *locc* is the numbers of matched characters, which includes the mismatched time.

***Proof.*** The hypothesis is that all characters of $t_1t_2t_3...t_n$ are scanned, and all matched patterns are reported. The initial step takes O(*1*) time by line 1. The time complexity is dominated by the variables *SHIFT*, *N*, *SET1*, and *SET2*, and these following cases give an explanation of the time complexity.

In the first case, loop *while* is run from $t_1$ to $t_n$. All operations are dominated by the variable *N* and *SHIFT*, and the variable *SHIFT* orders to inspect all characters in the text *T*. It can be said that line 3 takes O(*n*) time because this step is processed from the initial step to *n* times.

In the second case, the variable *N* drives line 4 and line 6 to operate in *locc* time at most. Each domination of *N* drives line 4 and line 6 to take O(*1*) time by Lemma 1. This stimulates line 10 to equal *locc* time as well. However, each operation of line 8 takes O(*1*) time by Lemma 3. The variable *N* orders the loop and returns to line 3, and at most equals the number of the characters to be matched with the inverted lists in the table $\tau$ . This takes *locc* time. Line 3 and line 10 take a constant time to control the other steps. Thus, the hypothesis is reached by line 13 and line 6, and the searching time is computed in O(*n*+*locc*) time.

## V. EXPERIMENTAL RESULTS

### *A. Hardware:*

The experiments were performed on a Dell Vostro 3400 notebook with Intel(R) CORE(TM) i5 CPU, M 560 @2.67 GHz, 4 GB of RAM, and running on Windows 7 Professional (32-bits) as an application machine. All programs were implemented in Java with JavaTM 2 SDK, Standard Edition Version 1.6.22 built in the Netbeans 6.9.1.

### *B. Data:*

The |∑ | was 52 letters of English alphabets; ‘A’ to ‘Z’ and ‘a’ to ‘z’. Each pattern was randomized with the various lengths of 3 to 20 characters (on average 12 characters). The programs randomized the patterns of 10, 50, 100, 500, 1,000, 5,000, 10,000, 50, 000, 100,000, and 300,000 for testing the data structure construction, and the patterns of 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 500, 1,000, 10,000, 50, 000, 100,000, and 300,000 were also randomized for the searching tests. The texts were randomized from the ∑ by the size of 1 KB, 10 KB, 100 KB, 1 MB, 5 MB, and 10 MB. Each experiment was performed 10 times and the average was given.

### *C. Preprocessing Results:*

In preprocessing phase, the inverted lists structure was constructed faster and used smaller space than the earlier Aho-Corasick [1] and SetHorspool in [9] for all cases. In case of pattern numbers more than 1,000; the suffix tree could not create the structure (represented by ‘-’) because the computer was out of heap memory in java while generating the structure. As well as, AC-Trie and RT-Trie were similar to suffix tree when using the pattern number 300,000. These results are illustrated by Table 1 and Table 2, respectively.

**TABLE I: PROCESSING TIMES FOR CREATING THE DATA STRUCTURES (SECONDS)**

| #patterns | AC-Trie | RT-Trie | Suffix Tree | IVL |
|---|---|---|---|---|
| 10 | 0.149 | 0.110 | 0.144 | **0.029** |
| 50 | 0.152 | 0.224 | 0.245 | **0.123** |
| 100 | 0.398 | 0.423 | 0.546 | **0.257** |
| 500 | 0.592 | 0.732 | 20.273 | **0.362** |
| 1,000 | 1.152 | 1.786 | 807.374 | **0.843** |
| 5,000 | 9.543 | 7.992 | - | **5.437** |
| 10,000 | 44.321 | 19.842 | - | **6.549** |
| 50,000 | 501.432 | 109.421 | - | **41.732** |
| 100,000 | 3,491.732 | 5,648.945 | - | **840.153** |
| 300,000 | - | - | - | **1,076.432** |

**TABLE II:MEMORY USAGES OF DATA STRUCTURES(KB)**

| #patterns | AC-Trie | RT-Trie | Suffix Tree | IVL |
|---|---|---|---|---|
| 10 | 4.52 | 4.78 | 25.18 | **4.37** |
| 50 | 4.79 | 4.88 | 47.79 | **4.58** |
| 100 | 4.87 | 4.95 | 899.55 | **4.85** |
| 500 | 5.48 | 5.70 | 2,721.48 | **5.07** |
| 1,000 | 5.92 | 6.15 | 5,872.87 | **5.29** |
| 5,000 | 10.91 | 11.05 | - | **7.37** |
| 10,000 | 14.76 | 15.82 | - | **8.98** |
| 50,000 | 55.82 | 56.94 | - | **22.66** |
| 100,000 | 154.10 | 149.12 | - | **46.54** |
| 300,000 | - | - | - | **158.91** |

### *D. Searching Results:*

The searching times were more efficient than the Aho-Corasick and the SetHorspool in the cases of 10, 20, 30, 40, 50, and 60 patterns for all comparable cases. For the number of pattern 70 to 90; the proposed algorithm took less time than the SetHorspool but longer than the Aho-Corasick. Table 3 illustrates the searching times in the second unit.Table 4 shows the larger scale of the experimental results when using the pattern number from 100 to 300,000 patterns. These results showed the new algorithm was faster than the SetHorspool algorithm but slower than the Aho-Corasick algorithm.

**TABLE III: Searching times (Seconds) when the pattern numbers 10-90 and the text sizes 1 KB to 5 MB**

| **Text sizes** | **#patterns** | **Aho-Corasick** | **SetHorspool** | **IVL-MSPM** |
|---|---|---|---|---|
| 1 KB | 10 | 0.0028 | 0.0197 | **0.0018** |
| | 20 | 0.0033 | 0.0457 | **0.0023** |
| | 30 | 0.0041 | 0.0570 | **0.0030** |
| | 40 | 0.0041 | 0.1787 | **0.0030** |
| | 50 | 0.0058 | 0.0900 | **0.0049** |
| | 60 | 0.0054 | 0.0966 | 0.0056 |
| | 70 | 0.0058 | 0.1091 | 0.0064 |
| | 80 | 0.0059 | 0.1097 | 0.0073 |
| | 90 | 0.0060 | 0.1962 | 0.0092 |
| 10 KB | 10 | 0.1517 | 0.2711 | **0.0138** |
| | 20 | 0.0333 | 0.7285 | **0.0211** |
| | 30 | 0.0426 | 0.9014 | **0.0283** |
| | 40 | 0.0453 | 1.3333 | **0.0376** |
| | 50 | 0.0585 | 1.4188 | **0.0493** |
| | 60 | 0.0567 | 1.5479 | 0.0608 |
| | 70 | 0.0569 | 1.5830 | 0.0647 |
| | 80 | 0.0587 | 1.5174 | 0.0728 |
| | 90 | 0.0623 | 1.6631 | 0.0650 |
| 100 KB | 10 | 0.3042 | 2.5075 | **0.0693** |
| | 20 | 0.3434 | 6.3085 | **0.2692** |
| | 30 | 0.4558 | 7.0720 | **0.3144** |
| | 40 | 0.4742 | 13.4095 | **0.3890** |
| | 50 | 0.6446 | 13.7770 | **0.5822** |
| | 60 | 0.6699 | 13.1947 | **0.6049** |
| | 70 | 0.6503 | 16.1672 | **0.6349** |
| | 80 | 0.6654 | 9.9408 | 0.8115 |
| | 90 | 0.7230 | 11.6783 | 0.9528 |
| 1 MB | 10 | 3.2589 | 24.3629 | **1.4462** |
| | 20 | 3.6699 | 69.2808 | **2.2862** |
| | 30 | 4.5725 | 81.2985 | **3.4074** |
| | 40 | 4.7033 | 128.1007 | **4.2012** |
| | 50 | 6.1650 | 116.8468 | **5.3145** |
| | 60 | 6.3105 | 135.9463 | **5.6377** |
| | 70 | 6.2101 | 142.4764 | **5.9012** |
| | 80 | 6.7162 | 140.8990 | 7.0458 |
| | 90 | 7.3199 | 154.3418 | 9.9342 |
| 5 MB | 10 | 15.2375 | 123.5881 | **7.2263** |
| | 20 | 17.7920 | 324.1327 | **12.5403** |
| | 30 | 22.0295 | 412.2096 | **12.8643** |
| | 40 | 22.1210 | 648.2799 | **20.0707** |
| | 50 | 27.2012 | 583.1193 | **26.1849** |
| | 60 | 30.1121 | 611.7581 | **27.0252** |
| | 70 | 31.3953 | 641.1439 | 33.5463 |
| | 80 | 32.2099 | 634.0454 | 34.2042 |
| | 90 | 31.9742 | 694.5383 | 45.5356 |

TABLE IV: SEARCHING TIMES (SECONDS) WHEN THE PATTERN NUMBERS 100-300,000 AND THE TEXT SIZES 1 KB TO 10 MB

| Text sizes | #patterns | Aho-Corasick | SetHorspool | IVL-MSPM |
|---|---|---|---|---|
| 1 KB | 100 | 0.019 | 0.122 | 0.060 |
| | 500 | 0.020 | 0.195 | 0.183 |
| | 1,000 | 0.037 | 0.207 | 0.203 |
| | 5,000 | 0.040 | 0.177 | 0.878 |
| | 10,000 | 0.030 | 0.162 | 0.943 |
| | 50,000 | 0.035 | 0.798 | 5.576 |
| | 100,000 | 0.037 | 0.802 | 7.683 |
| | 300,000 | - | - | 8.872 |
| 10 KB | 100 | 0.101 | 1.478 | 0.301 |
| | 500 | 0.117 | 1.532 | 0.732 |
| | 1,000 | 0.125 | 1.511 | 1.271 |
| | 5,000 | 0.173 | 1.572 | 4.575 |
| | 10,000 | 0.202 | 1.771 | 7.986 |
| | 50,000 | 0.549 | 2.011 | 10.942 |
| | 100,000 | 1.553 | 2.431 | 21.981 |
| | 300,000 | - | - | 29.762 |
| 100 KB | 100 | 0.492 | 12.842 | 3.671 |
| | 500 | 0.579 | 16.211 | 6.912 |
| | 1,000 | 0.599 | 17.192 | 8.841 |
| | 5,000 | 1.376 | 18.444 | 42.118 |
| | 10,000 | 1.432 | 18.976 | 62.127 |
| | 50,000 | 2.902 | 19.981 | 79.125 |
| | 100,000 | 4.177 | 25.211 | 221.421 |
| | 300,000 | - | - | 244.772 |
| 1 MB | 100 | 7.812 | 130.042 | 36.912 |
| | 500 | 8.942 | 149.721 | 45.414 |
| | 1,000 | 10.332 | 176.421 | 81.464 |
| | 5,000 | 20.842 | 181.123 | 127.166 |
| | 10,000 | 45.762 | 188.284 | 245.593 |
| | 50,000 | 78.421 | 217.125 | 788.541 |
| | 100,000 | 102.184 | 307.190 | 842.428 |
| | 300,000 | - | - | 942.112 |
| 5 MB | 100 | 29.199 | 759.178 | 566.315 |
| | 500 | 37.241 | 799.351 | 644.124 |
| | 1,000 | 43.442 | 858.324 | 755.311 |
| | 5,000 | 140.104 | 897.148 | 899.712 |
| | 10,000 | 167.123 | 907.518 | 902.814 |
| | 50,000 | 198.452 | 1,535.523 | 1,200.314 |
| | 100,000 | 397.671 | 1,976.434 | 1,488.619 |
| | 300,000 | - | - | 1,684.971 |
| 10 MB | 100 | 71.197 | 1,578.189 | 655.175 |
| | 500 | 88.812 | 1,768.557 | 751.324 |
| | 1,000 | 95.148 | 2,487.166 | 855.152 |
| | 5,000 | 299.182 | 2,555.190 | 991.412 |
| | 10,000 | 355.275 | 2,575.101 | 1,010.318 |
| | 50,000 | 430.723 | 2,642.342 | 1,401.714 |
| | 100,000 | 689.333 | 2,740.784 | 1,800.878 |
| | 300,000 | - | - | 2,113.998 |

## VI. CONCLUSION

This research article presented a multiple string patterns matching algorithm using inverted lists represented by hashing principle. This algorithm searches by the prefix approach taking (1) O($|P|$) time and O($|\sum|$+$|P|$) space in preprocessing phase where $|P|$ is the sum of all pattern lengths in $P$, and (2) O($n$+$locc$) time for searching where $n$ is the length of input text and $locc$ is the number of the matched characters which includes the mismatched time. The experimental results showed that the inverted lists structure is faster to construct and smaller space than the popular structure Trie. The searching time results were faster in the cases of small pattern number.

## REFERENCES

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

[2] Moffat, and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval," ACM Transactions on Information Systems, Vol. 14, No. 4, pp. 349-379, 1996.

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

[4] R.S. Boyer. and J.S. Moore, "A fast string searching algorithm," Communications of the ACM. vol. 20, no.10, pp. 762-772, 1977.

[5] Allauzen and M. Raffinot, "Factor oracle of a set of words," Technical report 99-11, Institute Gaspard Monge, Universitěde Marne-la-Valle, 1999.

[6] Monz and M. de Rijke, Inverted Index Construction http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf, February 2002.

[7] D.E. Knuth, J.H. Morris, V.R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, vol. 6, no.1, pp. 323-350, 1997.

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

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

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

[11] J. J. Fan and K. Y. Su, "An efficient algorithm for match multiple patterns," IEEE Transaction On Knowledge and Data Engineering, vol.5, no. 2, pp.339-351, 1993.

[12] J. Zobel and A. Moffat, "Inverted Files Versus Signature Files for Text Indexing," ACM Transaction on Database Systems, Vol. 23, No. 4, pp. 453-490, 1998.

[13] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," ACM Computing Surveys, vol. 38, no. 2, pp. 1-56, 2006.

[14] K. M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms," IBM Journal of Research and Development, vol. 31, no. 2, pp. 249-260, 1987.

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

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

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

[18] M.Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching," Rapport 93-3, Institute Gaspard Monge, Universityěde Marne-la-Valle, 1993.

[19] M.Crochemore, A. Czumaj, L. Gsieniec, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching," Information Processing Letters, vol. 71, no.3/4, pp. 107-113, 1999.

[20] 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, pp. 149-165, 1997.

[21] O. R. Zaïane, CMPUT 391: Inverted Index for Information Retrieval, University of Alberta. 2001.

[22] R. B. Yates and B. R. Neto, Mordern Information Retrieval, The ACM press.A Division of the Association for Computing Machinery,Inc. 1999, pp. 191-227.

[23] 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.

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

[25] Y. D. hong, 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, pp. 675-680, 2006.

[26] 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.

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

[28] 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, pp. 1–61, 2011.

[29] Belazzougui, "Worst Case Efficient Single and Multiple String Matching in the RAM Model," IWOCA 2010, LNCS 6460, pp. 90–102, 2011.

[30] T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen, "Online Dictionary Matching with Variable-Length Gaps". SEA 2011, LNCS 6630, pp. 76–87, 2011.

[31] 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, pp. 137–149, 2012.

[32] H. J. Kim, H.-S. Kim, and S. Kang, "A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems," IEEE Transaction on Parallel and Distributed Systems, vol. 22 no. 11, pp. 1904–1911, 2011.

[33] L. Dai, and Y. Xia, "A Lightweight Multiple String Matching Algorithm," International Conference on Computer Science and Information Technology 2008, pp. 611–615, 2008.

[34] Khancome and V. Boonjing. Data Structure for Dynamic Pattern. International MultiConference of Engineers and Computer Scientists 2010, HK, 17-19 March 2010, Pp. 399-404.

[35] Daoudi, S.E. Oautik, A. El Kharraz, K. Idrissi, and D. Aboutajdine, "Vector Approximation based Indexing for High-Dimensional Multimedia Database," Engineering Letter, 16:2, EL_16_2_05, 2008.

[36] Y. Lu, and D. Lou, "An Algorithm to Find the Optimal Matching in Halin Graphs," IAENG International Journal of Computer Science, 34:2, IJCS_34_2_09, 2007.

[37] H. Mryajima, M. Fujisai, and N. Shigei, "Quantum Search Algorithms in Analog and Digital Models," IAENG International Journal of Computer Science, 39:2, IJCS_39_2_05, 2012.

[38] O. Guth, and B. Melichar, "Finite Automata Approach to Computing All Seeds of String with the Smallest Hamming Distance," IAENG International Journal of Computer Science, 36:2, IJCS_36_2_05, 2009.

[39] Khancome and V. Boonjing, "A new linear-time dynamic dictionary matching algorithm", Computing and Informatics, Vol. 32, 2013, 1001-1027, V 2013-Sep-30.