Computing and Informatics, Vol. 32, 2013, 897–923

A NEW LINEAR-TIME DYNAMIC DICTIONARY MATCHING ALGORITHM

Chouvalit Khancome

Software Systems Engineering Laboratory
Department of Mathematics and Computer Science
Faculty of Science, King Monkut's Institute of Technology at Ladkrabang (KMITL)
Ladkrabang, Bangkok 10520, Thailand
e-mail: `chouvalit@hotmail.com, chouvalit.kha@csit.rru.ac.th`

V. Boonjing

Software Systems Engineering Laboratory
Department of Mathematics and Computer Science
Faculty of Science, King Monkut's Institute of Technology at Ladkrabang (KMITL)
Ladkrabang, Bangkok 10520, Thailand
&
National Centre of Excellence in Mathematics, PERDO
Bangkok, Thailand 10400
e-mail: `kbveera@kmitl.ac.th`

Communicated by Jacek Kitowski

Abstract. This research presents *inverted lists* as a new data structure for the dynamic dictionary matching algorithm. The inverted lists structure, which derives from the inverted index, is implemented by the perfect hashing table. The dictionary is constructed in optimal time and the individual patterns can be updated in minimal time. The searching phase scans the given text in a single pass, even in a worst case scenario. In experimental results, the inverted lists used less time and space than the traditional structures; the searches were processed and showed an efficient linear time.

Keywords: Dynamic dictionary matching, static dictionary matching, multiple pattern string matching, inverted index, inverted lists, trie, bit-parallel, hashing table

1 INTRODUCTION

Dictionary matching is one of the main priciples in classical string processing. This principle deals with a large number of string patterns $P = \{p^1, p^2, \ldots, p^r\}$ that can be searched simultaneously in a given text $T = \{t_1 t_2 t_3 \ldots t_n\}$. There are plenty of new applications in computer science that apply this principle to solve their problems (e.g., [32, 33, 34, 39]) including the operating system commands (Unix grep command using Commentz-Walter [9] and agrep using Wu-Manber[25]), intrusion detection systems (e.g., SNORT using Aho-Corasick [1], Commentz-Walter [9], and Wu-Manber [25]), and so on. Traditionally, the target patterns are generated to a suitable data structure (called dictionary) in the pre-processing phase. Then the searching phase scans and compares the given text with data in the dictionary for finding all pattern occurrences. Typically, if the dictionary can not support the updating of individual patterns, it is called static dictionary. In contrast, if the dictionary has the ability to delete or insert individual patterns over time, it is called dynamic dictionary.

Since dictionary matching is a fundamental problem, trie and bit-parallel data structures have been used to accommodate the static dictionary. Trie-based algorithms (Aho-Corasick [1], Commentz-Walter [9], and SetHorspool (mentioned in [18])) take the linear time or sub-linear time to solve problems. On the other hand, the suffix tree has also traditionally been used for creating the dynamic dictionary. Naturally, suffix tree-based algorithms work in logarithmic time ($O(n \log |P|)$) where $|P|$ is the sum of pattern lengths from p^1 to p^r and n is the length of T.

Although suffix tree-based algorithms are able to handle the mechanism of dynamic dictionary matching, all of them need some flexible structures such as McCreight [14], DS-List [22], or Weiner [23]. Unfortunately, these structures are also embedded with a logarithmic time ($\log |P|$) which often drives the searching phase to $\mathrm{O}(n \log |P|)$. It can therefore be said that all suffix tree-based algorithms are limited by the $\log |P|$ time trap. The key to solving the dynamic dictionary problem can be found in reducing the space and resolving the $\log |P|$ time complexity.

Considered by efficiency, the algorithms using static dictionary are more efficient than the algorithms implementing the dynamic dictionary, but the dynamic dictionary can be updated by the individual patterns in an optimal time while the static dictionary takes an exhaustive time (regenerates all patterns). Although there were several static dictionary matching solutions [35, 36, 37, 38, 40, 41] shown recently, they still tried to improve the classic data structures for accommodating the dictionary. For example, solutions [36, 37, 38] improved trie structure, solution [35] used tree, solution [40] employed bit-parallel; as well as the solution [41] used the hashing of Wu-Manber[25] and a quick sort principle. Therefore, a new superior data structure and a new faster algorithm will always need to incorporate both efficiency and flexibility.

Until now, the inverted index and the perfect hashing table have been popular data structures used for solving a variety of problems. The inverted index has been viewed as an excellent data structure in solving information retrieval problems such as [16, 28, 29, 15]. The principle of the inverted index found in [17, 27, 26] focused on the keywords and their positions. This principle can be applied to several data structures for offering a faster search. Also, the perfect hashing table is used to accommodate the data in minimal space $O(n)$ and to provide the search method in minimal time as $O(1)$. It must be asked then if there is a way to combine both these excellent data structures to create a new dynamic dictionary matching algorithm.

The first solution [12] presented a new method of dynamic dictionary matching using the inverted lists. This solution combined an inverted index idea and a normal hashing table to create the inverted lists structure. This algorithm takes $O(n + locc)$ time in an average case where *locc* is the number of characters to be matched while comparing characters in the given text. Unfortunately, this solution leads to backtracking which takes an exhausive time in $O(n|P|)$. The next solution [30] showed a new static dictionary matching algorithm. This solution concentrated on the perfect hashing table to implement the inverted lists structure. Importantly, it takes the linear time ($O(n)$) to search the given text even in a worst case scenario. Surprisingly, when the inverted lists were adapted to accommdate the dynamic dictionary; the searching algorithm still works in a linear time. Furthermore, deriving the inverted lists structure to the other field of computer science, the solution [31] illustrated two algorithms of single string matching that were more efficient in the case of small alphabet sizes, especially when searching on binary digits.

This paper proposes to adapt a linear time static dictionary matching [30] to create a new solution of linear time dynamic dictionary matching, which avoids the backtracking of text scaning of [12]. This new approach concentrates on the inverted lists structure which is implemented by the perfect hashing table and explores different ways to utilize flexible updating and efficient linear-time usage. In theoretical results, the inverted lists structure is constructed in $O(|P|)$ time and space, and the insertion or the deletion of the individual pattern take $O(|p|)$ time where $|p|$ is the length of pattern to be inserted or deleted. The searching phase takes only $O(n)$ time even in a worst case scenario. In experimental results, the inverted lists structure consumes less time and space than the traditional data stuctures. The new algorithm takes a linear-time to process the searching phase. Compared to previously well known static dictionary algorithms – Aho-Corasick[1] and SetHorspool (mentioned in [18]), the inverted lists algorithm searches faster than SetHorspool but slower than Aho-Corasick.

Section 2 summarizes the related algorithms and shows the derivations from the inverted index and the perfect hashing principle. Section 3 explains the details of the inverted lists structure and the dynamic mechanism. Section 4 illustrates the searching algorithm and its example. Section 5 details the implementations, and the experimental results are reported. Section 6 is the discussion and the suggestions for improving the algorithm and the data structure. Section 7 is the conclusion and planned future works.

2 RELATED WORKS AND DERIVING PRINCIPLES

This section presents a history on related works of static and dynamic dictionary matching algorithms. Furthermore, the ideas that derived the inverted index and the perfect hashing table are described.

2.1 Related Works

2.1.1 Static Dictionary Matching Algorithms

Basically, trie, bit-parallel and hashing table have been employed for storing the dictionary. The static dictionary algorithms always work best in linear or sub-linear time, but in a worst case scenario they often take an exhausive time in $O(n|P|)$. An overview of this principle is described below.

Trie has been used for accommodating patterns for a long time. The first linear time (Aho-Corasick [1] – extended from [13] using $O(n + nocc)$), the sub-linear time (Commentz-Walter [9]) and SetHorspool(mentioned in [18] taking $O(n|P|)$ in a worst case scenario) are the solutions using trie where *nocc* is the number of pattern occurences. Although the existing solution [10] tries to improve the static dictionary, the patterns still need to regenerate when the dictionary is updated. The main disadvantage is that when implementing trie to applications a large amount of memory is cosumed.

Alternatively, bit-parallel is also popular in accommodating the static dictionary. Bit-parallel-based algorithms employ the sequences of bits to store the patterns. Navarro and Raffinot [18] showed how to apply the single string Shift-Or and Shift-And to Multiple Shift-And[3] and Multiple-BNDM[18]. This principle is restricted by the word length of computer architecture; moreover, it requires special methods which are more complex in converting the patterns to the bit form.

On the other hand, the first hashing algorithm was presented by Karp and Rabin [22] in single string matching. This algorithm takes the worst case scenario in $O(mn)$ time where m is the pattern length. Unfortunately, the dictionary matching algorithms which directly extend from [22] take $O(n|P|)$ time (comparable to the exhaustive solution). A more efficient algorithm presented by Wu and Manber [25] creates the shift table and implements the hashing table for storing the block of patterns to solve the problem. The new solution [11] improves Wu and Manber [25] and provides a faster solution to this principle.

Recently developed solutions [36, 37, 38] improved trie structure to accommodate the patterns; especially [37] showed minimal space of solution. Other solutions, which employed those classic data structures (e.g., trie, bit-parallel, and hashing), can be found in [35, 40, 41].

2.1.2 Dynamic Dictionary Matching Algorithms

Dynamic dictionary matching algorithms are scalable in terms of the flexibility of their patterns, but they are disadvantaged in time and memory consumption. Suffix tree-based algorithms are able to handle the mechanism of dynamical patterns. This structure has led the dynamic dictionary research community to explore new solutions. The first adaptive suffix tree base algorithm was introduced by Amir and Farach [2]. However, it is very slow and can be categorized as an exhausive algorithm($O(n|P|)$). Subsequently, [3, 4, 5, 8] showed the logarithmic algorithms of suffix tree generalization. All of them required one of the dynamic data structures such as McCreight [14], DS-List [22], and Weiner [23] for managing the dictionary.

This principle was straight away challenged by how to escape from the factor of logarithmic time ($n \log |P|$). Although AFGGP [3] was the first algorithm with almost linear time efficiency, the $\log |P|$ factor still remained problematic. It can be said that all suffix tree approaches fall into the $\log |P|$ time trap. Furthermore, when implementing the suffix tree to applications it takes more space than trie. Nevertheless, [20] tried to improve DS-List [22] for storing patterns, but the time complexity is still affected by logarithmic time. For a clearer understanding, there are many sources [6, 8, 21] which provide good information on this principle.

2.2 Deriving the Inverted Index

The inverted index is the method for creating the index of keywords which appear in $D = \{D_1 \dots D_n\}$ where D_i is any individual document which contains various keywords over $\sum$, and $1 \leq i \leq n$. Then, the keywords are represented by $\langle documentID, word : pos \rangle$ where *documentID* is the indicated number referring to the number of documents, *word* is the keywords in the document, and *pos* is the occurrence position of *word* in the *documentID*.

For example, assume that there are the documents D_1: sun of sun, D_2: moon of moon, and D_3: *star of star.* Then, each document is analyzed for keeping keywords and their positions. Thus the keywords in the documents are *sun*, *of*, *moon* and *star.* Then, all keywords can be rewritten by the form of *word: (posting lists)* where *posting list* is *(documentID: position of words in that document).* In this case, all keywords in these documents are re-written as *sun*: $(D_1 : 1)$, $(D_1 : 3)$; *of*: $(D_1 : 2)$, $(D_2 : 2)$, $(D_3 : 2)$; *moon*: $(D_2 : 1)$, $(D_2 : 3)$, and *star*: $(D_3 : 1)$, $(D_3 : 3)$. Afterwards, the keywords and posting lists are converted into the suitable data structures such as B^+tree, suffix tree, and suffix array.

Motivated by the positions of keywords, this research focuses on the position of characters instead of the keywords. For deriving the principle, the document D is first replaced by the pattern P, and each D_i is replaced by p^i. For instance, if there are the patterns $P = \{ram, run, running\}$ then the patterns are assigned as $D_1 = ram$, $D_2 = run$, and $D_3 = running$. In the next step, they are re-written by the form of *character:* $\langle$*the occurrence position of character in pattern: the indicated status of the last character of pattern: the number of pattern in P*$\rangle$; e.g., $r : \langle 1 : 0 : 1 \rangle$,

$\langle 1:0:2\rangle$, $\langle 1:0:3\rangle$, ... Each item of this form is called the *individual posting list.* Then, the context is determined using the individual posting lists that accommodate the dictionary (shown in Section 3).

2.3 Deriving the Perfect Hashing

The perfect hashing principle is the most powerful hashing table because it is completely devoid of collision. Importantly, this priciple takes $O(1)$ time in worst-case performances (shown in [7, 19, 24]). Moreover, it takes $O(n)$ space where n is the size of data. This structure is suitable for the set of static keys such as the reserved words in the programming language. Similary, the alphabets ($\sum$), which are used in all languages, are as limited as the static keys. This is the reason why perfect hashing should be chosen for implementing the inverted lists.

Fundamentally, the perfect hashing table consists of 2 levels. The first level is the universal key U to accommodate all keys for accessing all data in the table. This level has the n keys for hashing to access the second level by the function $f(n)$. The second level contains the data items associated with the corresponding key of n. This level splits into 2 buckets which avoid collision when accessing data. By using this method, accessing data may need re-hashing 2 times.

This research assigns $\sum$ as the universal key U and $f(\lambda)$ as $f(n)$ for the first level of the perfect hashing table and represents the groups of posting lists as the data items in the second level where $\lambda \subseteq \sum$.

The first level has the hashing function $h(\mathit{key}) \rightarrow (\mathit{data\ in\ level\ 2})$. If there are collisions then they need to re-hash by $h(\mathit{key\ of\ level\ 2})$. However, the time complexity still takes $O(1)$. For implementing, $\sum$ and λ are unnecessary to store in the memory because they can be calculated using the special function $f(\mathit{character}, \mathit{pos})$ (shown in definition 3.6) when accessing the inverted lists in the second level. This method decreases the space in the first level of hashing table while the accessing of the items takes a constant time.

3 INVERTED LISTS DATA STRUCTURE

The main ideas to accommodating dictionary are highlighted in this section. Initially, all characters of each pattern are analyzed and given their positions. Then the positions are grouped to a new form and are arranged into the perfect hashing table. The following sub-sections present all basic definitions for the next sections, the pre-processing algorithm, pattern insertion, and pattern deletion.

3.1 Basic Definitions

As mentioned earlier, this paper adapts the inverted lists structure and the searching algorithm presented in [30] to improve the approach outlined in [12] and thus some definitions and notations are the same in both [30, 12]. In representing the characters by position, Definitions 3.1 and 3.2 illustrate the individual lists and their form. Definition 3.3 represents the inverted lists in new context. Definition 3.4 to 3.7 are for creating the table, keeping the inverted lists by temporary variables, the functions for accessing the table, and a theorem called intersection for analyzing the continuity of patterns.

Definition 1. Given P is a set of patterns $\{p^1, p^2, \ldots, p^r\}$ where p^i denotes a pattern i^{th} which $1 \leq i \leq r$. The length of p^i is m and p^i is formed by the character sequence $\{c_1c_2c_3 \ldots c_m\}$. A single individual posting list of a character c_k is defined as $c_k : \langle k : 0 : i\rangle$ if $k\langle m$ or $c_k : \langle k : 1 : i\rangle$, if $k = m$ where $1 \leq k\leq m$. The individual posting list of $c_k : \langle k : 0 : i\rangle$ is denoted by $\varphi_0^{k_i}$, and $c_k : \langle k : 1 : i\rangle$ is denoted by $\varphi_1^{k_i}$.

Example 1. If there is the set of $\{ram, run, running\}$ then each pattern can be assigned as $p^1 = r_1a_2m_3$, $p^2 = r_1u_2n_3$ and $p^3 = r_1u_2n_3n_4i_5n_6g_7$. All individual posting lists are represented below.

$$\begin{aligned} p^1 &= r : \langle 1 : 0 : 1\rangle, a : \langle 2 : 0 : 1\rangle, m : \langle 3 : 1 : 1\rangle, \\ p^2 &= r : \langle 1 : 0 : 2\rangle, u : \langle 2 : 0 : 2\rangle, n : \langle 3 : 1 : 2\rangle, \\ p^3 &= r : \langle 1 : 0 : 3\rangle, u : \langle 2 : 0 : 3\rangle, n : \langle 3 : 0 : 3\rangle, \\ & \quad n : \langle 4 : 0 : 3\rangle, i : \langle 5 : 0 : 3\rangle, n : \langle 6 : 0 : 3\rangle, g : \langle 7 : 1 : 3\rangle. \end{aligned}$$

The next step is that all individual posting lists are grouped to a new form as *character* : $\langle$*position* : *terminate status* : $\{$*set of patterns which occur in the same position*$\}\rangle$. Then, the groups of all characters can be shown as $r : \langle 1 : 0 : \{1, 2, 3\}\rangle$, $n : \langle 3 : 0 : \{2, 3\}\rangle$, $\langle 3 : 1 : \{2\}\rangle$, $\langle 4 : 0 : \{3\}\rangle$, $\langle 6 : 0 : \{3\}\rangle$, and so on. Definition 3.2 shows how to group the posting lists to the new form.

Definition 2. Let $l_{\max}$ be the maximum length of patterns in $\{p^1, p^2, p^3, \ldots, p^r\}$, and let ε be the position of any character λ which appears in the various patterns at the same position where $1 \leq \varepsilon \leq l_{\max}$ and $\lambda \subseteq \Sigma$. Then the posting lists are $\{\varphi_0^{\varepsilon_i}, \varphi_0^{\varepsilon_l}, \ldots, \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ or $\{\varphi_1^{\varepsilon_i}, \varphi_1^{\varepsilon_l}, \ldots, \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ where $1 \leq \{i, l, \ldots, p, q\} \leq r$. A group of posting lists of λ can be defined as follows.

1. If the posting lists of λ are $\{\varphi_0^{\varepsilon_i}, \varphi_0^{\varepsilon_l}, \ldots, \varphi_0^{\varepsilon_p}, \varphi_0^{\varepsilon_q}\}$ then a group of posting lists is defined by $\lambda_{\varepsilon,0}$.

2. If the posting lists of λ are $\{\varphi_1^{\varepsilon_i}, \varphi_1^{\varepsilon_l}, \ldots, \varphi_1^{\varepsilon_p}, \varphi_1^{\varepsilon_q}\}$ then a group of posting lists is definded by $\lambda_{\varepsilon,1}$.

Example 2. The posting lists of $P = \{ram, run, running\}$.

Posting lists		$\lambda_{\varepsilon,0}/\lambda_{\varepsilon,1}$
a :	$\langle 2:0:\{3\}\rangle$,	$a_{2,0}$,
g :	$\langle 7:1:\{3\}\rangle$,	$g_{7,1}$,
i :	$\langle 5:0:\{3\}\rangle$,	$i_{5,0}$,
m :	$\langle 3:1:\{1\}\rangle$,	$m_{3,1}$,
n :	$\langle 3:0:\{3\}\rangle, \langle 3:1:\{2\}\rangle$,	$n_{3,0}, n_{3,1}$,
	$\langle 4:0:\{3\}\rangle, \langle 6:0:\{3\}\rangle$,	$n_{4,0}, n_{6,0}$,
r :	$\langle 1:0:\{1,2,3\}\rangle$,	$r_{1,0}$,
u :	$\langle 2:0:\{2,3\}\rangle$.	$u_{2,0}$.

Definition 3. Let I be the inverted list structure of any group of the posting lists. For any inverted lists structure of alphabet λ if the posting lists group is $\lambda_{\varepsilon,0}$ then the inverted lists structure is defined as $I_{\lambda\varepsilon,0}$. Similarly, if the posting lists group is $\lambda_{\varepsilon,1}$ then the inverted lists structure is denoted as $I_{\lambda\varepsilon,1}$.

Example 3. The groups of posting lists shown in Example 2 can be re-written as $I_{a2,0}$, $I_{g7,1}$, $I_{i5,0}$, $I_{m3,1}$, $I_{n3,0}$, $I_{n3,1}$, $I_{n4,0}$, $I_{n6,0}$, $I_{r1,0}$, and $I_{u2,0}$.

Definition 4. The perfect hashing table which provides for all alphabets over $\sum$ and their corresponding inverted lists is called the inverted lists table and denoted by τ.

Example 4. The groups of posting lists shown in Example 3 can be stored in the table τ as shown in Table 1. It is unnecessary to store the first column in the real table because it can be calculated by the code of ASCII or Unicode when implemented, but the second column is stored in the memory which is split into two parts. These are described in the third and the fourth columns.

$f(\lambda)$ (first level)	second level	set of positions	set of pattern numbers
a	$I_{a2,0}$	$2:0$	$\{3\}$
g	$I_{g7,1}$	$7:1$	$\{3\}$
i	$I_{i5,0}$	$5:0$	$\{3\}$
m	$I_{m3,1}$	$3:1$	$\{1\}$
n	$I_{n3,0}$, $I_{n3,1}$	$3:0$, $3:1$	$\{3\}$, $\{2\}$,
	$I_{n4,0}$, $I_{n6,0}$	$4:0$, $6:0$	$\{3\}$, $\{3\}$
r	$I_{r1,0}$	$1:0$	$\{1,2,3\}$
u	$I_{u2,0}$	$2:0$	$\{2,3\}$

Table 1. Table of the inverted lists of P

Definition 5. Two hashing sets which are provided for storing any inverted lists $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ are called $SET1$ and $SET2$.

Definition 6. A hashing function which takes $I_{\lambda pos,0}$ and/or $I_{\lambda pos,1}$ from τ is called inverted lists hashing function, denoted by $IVL(\lambda, pos)$ where $\lambda \subseteq \sum$ and *pos* is the required position of posting lists which are stored in the second level of τ.

Definition 7. If $SET1$ and $SET2$ contain the inverted lists groups, then the continuity of patterns is operated by the intersecting function which is denoted by $SET1$ $\cap$ $SET2$.

Example 5. Supposing that $SET1 = \{\langle 1 : 0 : \{1, 2\}\rangle\}$ and $SET2 = \{\langle 2 : 0 : \{1, 3\}\rangle\}$ then $SET1 \cap SET2$ is ordered by the position 1 to 2. The first consideration is that the sequence of inverted lists in *SET1* is described by *SET2*. In this case, the pattern number $\{1\}$ of *SET1* is described by the position of $\{1\}$ in *SET2* while the required position is '2' in $\{\langle 2 : 0 : \{1, 3\}\rangle\}$ (prior to the positions in *SET1*1). Therefore, the result is $SET1 = \{\langle 2 : 0 : \{1\}\rangle\}$.

3.2 Pre-Processing Phase

This section shows the algorithm for generating the table τ. Lemma 1 shows how to get the inverted lists in constant time. Theorem 1, Theorem 2, and Theorem 3 define the correctness, time, and space of Algorithm 1, respectively.

Pre-processing represents the steps for creating the inverted lists which take $O(|P|)$ time. The first step is creating the empty table τ. The second step is reading p^1 to p^r. Whenever each pattern is read, the character is converted to the inverted lists. Then if an inverted list of the considering character exists in the table, the number of pattern is added into the part of {*set of patterns which occur in the same position*}. Otherwise, a new inverted list is created and added into the table.

Algoirthm 1: Pre-processing phase

```
Input: $P = \{p^1, p^2, \ldots, p^r\}$
Output: table $\tau$ of $P$
1.  Create table $\tau$
2.  for $i = 1$ to $r$ do
3.     for $j = 1$ to $m$ of $p^i$ do
4.        if $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ does not exist in $\tau$ then
5.           $\tau \leftarrow \varphi_0^{j_i}$ if $j \langle m$ or $\tau \leftarrow \varphi_1^{j_i}$ if $j = m$
6.        else
7.           $I_{char(j)j,0} \leftarrow i$ if $j \langle m$ or $I_{char(j)j,1} \leftarrow i$ if $j = m$
8.        end if
9.     end for
10. end for
11. return table $\tau$
```

Lemma 1. If there are the groups of inverted lists $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ in τ then accessing all inverted lists of $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ uses $O(1)$ time.

Proof. Since $\lambda \subseteq \sum$ then each alphabet is a unique character, and λ is implemented as the first level of the perfect hashing table taking $O(1)$ time. The inverted lists $\lambda_{\varepsilon,0}$ or $\lambda_{\varepsilon,1}$ are implemented as the second level of the perfect hashing table; therefore, each data item takes $O(1)$ time, and all items in the second level of table are taken in $O(1)$ as well as the individual item. □

Theorem 1. Algorithm 1 can generate all patterns $\{p^1, p^2, p^3, \ldots p^r\}$ to the inverted lists and store them into τ correctly.

Proof. The correctness is proved when p^1 to p^r are generated to the inverted lists, and all inverted lists are added to the table τ. The proofs are organized by

1. proving the initial step,
2. proving *for* of inner loop, and
3. proving *for* of outer loop.

For the initial step, line 1 needs to be true, and the table must be created for running the other steps of proof.

Regarding the inner loop, the proof is made by the induction on j for $j = 1$ to $j = m$. The invariants are still at the end of each j^{th} iteration on $1 \leq j \leq m$ and $1 \leq i \leq r$ for $j = 1$ to $j = m$. The pre-condition is that p^i does not exist in the table, and the length of each p^i is m. Also, the variable of m can be changed when each p^i is changed. The post-condition is that each p^i is formed by the sequence of $\{c_1c_2c_3 \ldots c_m\}$. All characters $c_1c_2c_3 \ldots c_m$ are generated to inverted lists and are added to the table. Since the *for* loop is executed by a fixed number, this therefore guarantees the termination of the loop. In the base case, c_1 of p^i is converted to $\varphi_0^{1_1}$ and added to the table as a new inverted list. This result is true, and the invariants remain. Assuming the proposed invariants are true after $m-1$ iteration, proof can be demonstrated using the two following cases.

In the first case, if there are no inverted lists of p^i_j, then a new inverted list $\varphi_0^{j_i}$ if $j \langle m$ or $\varphi_1^{j_i}$ if $j = m$ is generated and the table at the $I_{char(j)j,0} \leftarrow i$ or $I_{char(j)j,1} \leftarrow i$ is created. Then $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ is stored in the table, and the invariants are unchanged. In the second case, if there are the inverted lists of p^i_j, the number of $m-1$ is stored in τ. This then implies that the variable j and $1 \leq j \leq m-1 \leq m$ and $1 \leq i \leq r$ for $j = 1$ to $j = m-1$. Also by induction, the variable j and $1 \leq j \leq m-1 \leq m$ and $1 \leq i \leq r$ for $j = 1$ to $j = m-1$. Then after adding $\varphi_0^{j_i}$ or $\varphi_1^{j_i}$ to the table it implies an iteration of $j = m$ as the hypothesis induction, and the post-condition is shown when c_m is added to the inverted list. In either case the proposed invariants remain and the termination is guaranteed by a fixed number of j; therefore, the inner loop is correct.

The outer loop is proved by induction on i. The pre-condition is that there are P and τ, the post-condition is all patterns in P are generated to the inverted lists and are added to τ. The proposed invariant is $1 \leq i \leq r$. For the base case, if $i = 1$ then it is true by the inner loop and the proposed invariant $1 \leq i \leq r$ remains. The inverted lists from c_1 to c_m are followed by the inner loop, and the loop is run on the fixed number of i; this also guarantees its termination. In the induction step, the iteration of $i = r - 1$ must be proved; the pattern p^{r-1} is formed by $\{c_1c_2c_3 \dots c_m\}$, and all of the characters are sent to the inner loop which are then true after running the inner loop. The termination is guaranteed by the fixed number of m, and when all inverted lists are stored in τ. The invariant still remains while τ stores the inverted lists from pattern p^1 to p^{r-1} after running the inner loop. By induction, the hypothesis is reached, and the correctness is proved. □

Theorem 2. Generating the patterns $\{p^1, p^2, p^3, \dots p^r\}$ to the inverted lists and adding them into τ takes $O(|P|)$ time where $|P|$ is the sum of all pattern lengths.

Proof. The hypothesis is that all characters of $\{p^1, p^2, p^3, \dots p^r\}$ are generated to inverted lists, and they are added into τ. Referring to Algorithm 1, all pattern lengths are denoted by $|p^1|$, $|p^2|$, $|p^3|$, …, $|p^r|$. For the initial step, the table τ is built in $O(1)$ time. Each round processes the inner loop to execute line 3 or line 8 until they equal the length of each pattern. The summation is $|p^1| + |p^2| + |p^3| + \dots + |p^r|$ which equals $|P|$, and it reaches to the hypothesis step by the last character of p^r. Therefore the inverted lists are constructed in $|P|$ time; this is called $O(|P|)$ time complexity. Meanwhile, lines 4, 5, and 7 access the table in $O(1)$ by Lemma 1. Hence, the preprocessing time is proved in $O(|P|)$ time. □

Theorem 3. Table τ requires $O(|P|)$ space for accommodating whole inverted lists of $\{p^1, p^2, p^3, \dots p^r\}$ where $|P|$ is the sum of pattern lengths.

Proof. The space is proved when all characters of P are generated to inverted lists and are added into the table τ taking $|P|$ space. The pattern lengths in P are $|p^1|$, $|p^2|$, $|p^3|$, …, $|p^r|$, and each p^i contains the sequence string $\{c_1c_2c_3 \dots c_m\}$ which has the length m. The length m is denoted by $|p^i|$. For the initial step, the first column of table τ is created for all patterns. Each inverted list is created by the pre-processing phase for all patterns of P; therefore, each inverted list of string $\{c_1c_2c_3 \dots c_m\}$ in each p^i only takes one space per one list. Thus, the space is equal to $|p^1| + |p^2| + |p^3| + \dots + |p^r| = |P|$ for the second level of the perfect hashing table. As mentioned earlier, the perfect hashing table required $O(n)$ space to accommodate the data items; hence, the space of τ is $O(|P|)$. □

3.3 Pattern Insertion

The method of pattern insertion is similar to that of the inner loop of Algorithm 1. Let p^ϕ be a new pattern which does not appear in τ, and contains the sequence string $\{c_1c_2c_3 \dots c_m\}$. Then all inverted lists of p^ϕ are generated and added into the table as the pre-processing phase. This method is illustrated by Algorithm 2. Theorem 4 shows the correctness of Algorithm 2, and Theorem 5 proves the time of the individual pattern insertion.

Example 6. Assume there is the new pattern $p^4 = rap$ to be inserted into the table τ of $P = \{ram, run, running\}$. In this case, all characters are formed to

```
Algorithm 2: Pattern Insertion
Input: $p^{\phi} = \{c_1c_2, c_3, \ldots, c_m\}$
Output: $\tau$ after insertion the inverted lists of $c_1c_2, c_3, \ldots, c_m$
1.  for $j = 1$ to $m$ do
2.      if $\varphi_0^{j_\phi}$ or $\varphi_1^{j_\phi}$ of $p^{\phi}$ does not exist in $\tau$ then
3.          $\tau \leftarrow \varphi_0^{j_\phi}$ if $j < m$ or $\tau \leftarrow \varphi_1^{j_\phi}$ if $j = m$
4.      else
5.          $I_{char(j)j,0} \leftarrow \phi$ if $j < m$ or $I_{char(j)j,1} \leftarrow \phi$ if $j = m$
6.      end if
7.  end for
8.  return table $\tau$
```

inverted lists as $r : \langle 1 : 0 : 4\rangle$, $a : \langle 2 : 0 : 4\rangle$, and $p : \langle 3 : 1 : 4\rangle$. Then, the result is $r : \langle 1 : 0 : \{1, 2, 3, \mathbf{4}\}\rangle$. Similarly, the character $'a'$ can be generated and added in $a : \langle 2 : 0 : \{3, \mathbf{4}\}\rangle$; but then the character $'p'$ is a new character that does not exist in the table. The inverted list of $'p'$ is generated as $p : \langle 3 : 1 : \{4\}\rangle$ by line 3.

Theorem 4. Let $p^{\phi} = \{c_1c_2, c_3, \ldots, c_m\}$ be the new pattern which is not contained in the table τ. Algorithm 2 inserts the inverted lists of pattern p^{ϕ} into the table τ correctly.

Proof. Correctness of Algorithm 2 is proved when all inverted lists of p^{ϕ} are added into the existing table τ. Let τ_{old} be the table before adding the new pattern, and let τ_{new} be the table after adding the new pattern.

The pre-condition is that τ_{old} contains the inverted lists $|P|$, and the post-condition is $|P| + |p^{\phi}|(\tau_{new})$. The invariants are $1 \leq j \leq m$, and $1 \leq i \leq r$ for $j = 1$ to $j = m$ where m is the length of $\{c_1c_2, \ldots, c_m\}$. The proof is by induction on j as the inner loop of Algorithm 1.

Obviously then, the pattern p^{ϕ} is similar to the pattern p^i in P. Algorithm 2 is run as the inner loop of Algorithm 1. Therefore, the proof in the loop of Algorithm 2 is claimed as well. Also, the invariants remain because there is nothing to change them. Hence, the post-condition is shown at c_m, and τ_{new} is shown. □

Theorem 5. Inserting $p^{\phi} = \{c_1c_2, c_3, \ldots, c_m\}$, which does not appear in the table τ, takes $O(|p|)$ where $|p|$ is the length of p^{ϕ}.

Proof. Algorithm 2 is referred to for straightforward proof. The length of p^{ϕ} is m and is denoted by $|p|$. The loop *for* reads all characters and converts them to the inverted lists. Then each individual inverted list is added into τ one by one. This loop creates the inverted lists from c_1 to c_m, and it takes m operations. Thus, $O(|p|)$ time is shown and then the hypothesis is also proved. The other lines (2,3,5) access the table taking $O(1)$ by Lemma 1. Therefore, to insert all characters of p^{ϕ} into the existing dictionary takes $O(|p|)$ time. □

3.4 Pattern Deletion

Assume that p^σ is the existing pattern in τ, and p^σ is formed by the sequence string $\{c_1c_2c_3\dots c_m\}$. Then, all characters from p^σ are read one by one, and each inverted list is removed from the dictionary. For the mechanism of deletion, if the corresponding inverted list exists in only one posting list, it will be deleted immediately. Otherwise, only the inverted lists which the pattern number equals to σ are deleted. The method is described by Algorithm 3.

Algorithm 3: Pattern Deletion

```
Input: $p^\sigma = \{c_1c_2c_3\dots c_m\}$
Output: $\tau$ after deletion the inverted lists of $c_1c_2c_3\dots c_m$.
1.  for $j = 1$ to $m$ do
2.      if the posting lists in $I_{char(j)j,0}$ or $I_{char(j)j,1} > 1$ then
3.          Delete $\varphi_0^{j\sigma}$ if $j < m$ or $\varphi_1^{j\sigma}$ if $j = m$
4.      else
5.          Delete $I_{char(j)j,0}$ if $j < m$ or $I_{char(j)j,1}$ if $j = m$
6.      end if
7.  end for
8.  return table $\tau$
```

An illustrative example is shown below, followed by the proof of correctness and time complexity.

Example 7. Taking $p = ram$ off $P = \{ram, run, running\}$. In this case, the target pattern is 1, and all characters of *ram* are formed to $r : \langle 1 : 0 : \{1\}\rangle$, $a : \langle 2 : 0 : \{1\}\rangle$, and $m : \langle 3 : 1 : \{1\}\rangle$. Line 3 takes the inverted list of $r : \langle 1 : 0 : \{2\}\rangle$ from $r : \langle 1 : 0 : \{1, 2, 3\}\rangle$; then the result is $r : \langle 1 : 0 : \{\mathbf{2,\ 3}\}\rangle$; but the inverted lists of $a : \langle 2 : 0 : \{1\}\rangle$ and $m : \langle 3 : 1 : \{1\}\rangle$ are removed from the table by line 5.

Theorem 6. Deleting the existing pattern $p^\sigma = \{c_1c_2, c_3, \dots, c_m\}$ from the table τ by Algorithm 3 is correct.

Proof. τ contains the inverted lists of P with the size $|P|$. Let τ_{aft} be the inverted lists table after deleting the pattern $p^\sigma = \{c_1c_2, c_3, \dots, c_m\}$, and the size of τ_{aft} be $|P| - |p^\sigma|$ where σ is the pattern number which appears in the table. The proof needs to show all inverted lists of $\{c_1c_2, c_3, \dots, c_m\}$ that are removed from the table. The pre-condition is that the p^σ exists in table τ, and the post-condition is τ_{aft}. The proposed invariant is $1 \leq j \leq m$ for $j = 1$ to $j = m$.

The proof is by induction on j. The base case is in $j = 1$ and the character c_1 is converted to the inverted list. Then, the inverted list $\varphi_0^{j\sigma}$ is formed by line 3. The proof needs to show both conditions of *if*. In the first case, if the number of posting list c_1 in the table is more than 1 then the number of $\varphi_0^{j\sigma}$ is removed from τ. In the second case, if there is only one inverted list in τ then $I_{char(j)j,0}$ is removed by line 5.

Thus, $\varphi_0^{j_\sigma}$ is removed from the table after the first iteration, and the size of the table is decreased by 1. In both cases, the invariant $1 \leq j \leq m$ remains. According to the fixed number of loops, the termination of loop is guaranteed by the value of m.

In the inductive step, the invariant needs to be true after the iteration of $j = m - 1$. The character of c_{m-1} is created as $\varphi_0^{\sigma_{m-1}}$. If the number of inverted lists of c_{m-1} is more than 1 then the number of $\varphi_0^{\sigma_{m-1}}$ is removed from τ, and if there is only one inverted list then $I_{char(m-1)m-1,0}$ is removed. The invariant still remains. The number of inverted lists in the table equals $|P| - |p^\sigma - 1|$ while $1 \leq j \leq m - 1 \leq m$ for $j = 1$ to $j = m$, and τ_{aft} is shown. By induction, the hypothesis is implied. Therefore, Algorithm 3 is correct. $\square$

Theorem 7. Deleting the existing pattern p^i from the dictionary P takes $O(|p|)$ time where p^i is the target pattern to be deleted and $|p|$ is the length of pattern p^i.

Proof. Assume that p^i is the existing pattern to be deleted, and p^i is formed by the sequence string $\{c_1c_2c_3 \dots c_m\}$. The length of p^i is m and is denoted by $|p|$, and i is the number of the pattern i^{th} in P. The hypothesis is that all inverted lists of p^i are removed from the dictionary of P.

The process of deleting repeats to remove the inverted lists from c_1 to c_m. Each operation for accessing the inverted list uses $O(1)$ by Lemma 1. The operations remove the matched inverted lists from the table one by one. All operations take $|p|$ time while line 3 or line 5 takes the constant time by Lemma 1. Thus, to delete all characters of pattern p^i from the dictionary P takes $|p|$ which is $O(|p|)$ time. $\square$

4 SEARCHING PHASE

Before describing the searching methodology in depth, this section refers back to the basic definitions which are used for running the searching algorithm. Let N be the target position in the given text to be compared; *pos* is the required position of the inverted lists to be matched; and n is the length of the text T. In addition, *SET1* and *SET2* are the variables that are operated for continuity during the search.

Initially, the variables N *pos*, *SET1*, and *SET2* are set to enforce the searching window, and the variable *pos* is used to control the required position in the text T. This search is based on reading from left to right along the text T. While reading, the inverted lists that equal *pos* in the row of *text*[N] are taken to *SET1* or *SET2*. When considering continuity, the intersection ($\cap$) is used for checking patterns in *SET1* and *SET2*.

The intersection between *SET1* and *SET2* finds a set of numbers in *SET2* that continue from *SET1*. Importantly, it reports the matched position whenever the terminate status equals 1. The continuity is concentrated on the posting lists in *SET1* that are described by *SET2*. If the numbers of positing lists in *SET2* are superior to *SET1*, these are kept in *SET1* for the next operation. For reporting the occurrences, the indicated number '0' or '1' of $SET1$ is considered; if the indicated number in *SET1* is '1' then the matched position is reported.

In the case of the overlapping patterns, the inverted lists which are equal to $\langle 1 : 0 : \{\ldots\}\rangle$ must be attached to *SET2* when accessing the inverted lists of any positions. For instance, if the patterns are 'ram' and 'amazing', the inverted lists of 'a' are $\langle 2 : 0 : \{1\}\rangle$ and $\langle 1 : 0 : \{2\}\rangle$. In this case they are taken together when the character $'a'$ in the given text is scanned.

The algorithm, an illustrative example, the proofs of the correctness and the time complexity are shown below.

Algorithm 4: Searching Algorithm
Input: P and T
Output: all occurrences are reported, and T is scanned.

```
1.  $N = 1 pos = 1, SET1 = SET2 = null, RESULTS = \{\}$
2.  $SET1 \leftarrow (IVL(text[N]), pos), N++$
3.  while ($N \leq n$) do
4.      Store the matched position into $RESULTS$ set if $SET1$ contains $\varphi_1^{pos}$
5.    if $SET1 <> null$ then
6.       $pos++$
7.       $SET2 \leftarrow (IVL(text[N]), pos$ or 1)
8.       $SET1 \leftarrow SET1 \cap SET2$
9.    else
10.       $pos = 1$
11.       $SET1 \leftarrow (IVL(text[N]), pos)$
12.    end if
13.   $N++$
14.  end while
15.  report all occurrences in $RESULTS$
```

Example 8. Searching $P = \{ram, run, running\}$ in the given text $T =$ run as running on ram by searching algorithm.

1. Initiate the variables $N = 1$, $SET1 = SET2 = \{\}$.
2. Skip to line 2 and $SET1 = \{\langle 1 : 0 : \{1, 2, 3\}\rangle\}$, and $N = 2$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

3. Take the first loop of *while*, $pos = 2$, and $SET2 = \{\langle 2 : 0 : \{1, 2, 3\}\rangle\}$.

r	**u**	n		a	s		r	u	n	n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1 = \{\langle 2 : 0 : \{1, 2, 3\}\rangle\}$ and $N = 3$.

4. Skip to the next loop of *while*, $pos = 3$, $SET2 = \{\langle 3 : 1 : \{1\}\rangle, \langle 3 : 0 : \{2\}\rangle\}$

r	**u**	**n**		a	s		r	u	n	n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

$SET1 \leftarrow SET1 \cap SET2$ thus $SET1 = \{\langle 3 : 1 : \{2\}\rangle, \langle 3 : 0 : \{3\}\rangle\}$ and n is matched at $\langle 3:1:\{2\}\rangle$ in pattern 2. After matching report, set $N = 4$ and *SET1* $= \{\langle 3:0:\{3\}\rangle\}$.

5. Skip to the next loop of *while*, *pos* $= 4$, and *SET2* $= \{\}$
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
6. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 5$.
r u n **a** s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
7. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 6$.
r u n a **s** r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
8. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 7$.
r u n a s r u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
9. Skip to the next loop of *while*, and set *pos* $= 2$, *SET2* $= \{\langle 1:0:\{1,2,3\}\rangle\}$, and $N = 8$.
r u n a s **r** u n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
10. Skip to the next loop of *while*, and set *pos* $= 2$, *SET2* $= \{\langle 2:0:\{1,2,3\}\rangle\}$, and $N = 9$.
r u n a s **r u** n n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 2:0:\{1,2,3\}\rangle\}$ and $N = 10$.
11. Skip to the next loop of *while*, and set *pos* $= 3$, *SET2* $= \{\langle 3:1:\{2\}\rangle, \langle 3:0:\{3\}\rangle\}$, and $N = 10$.
r u n a s **r u n** n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 3:1:\{2\}\rangle, \langle 3:0:\{3\}\rangle\}$ and n is matched at $\langle 3:1:\{2\}\rangle$ in pattern 2. After matching report, set $N = 11$ and *SET1* $= \{\langle 3:0:\{3\}\rangle\}$.
12. Skip to the next loop of *while*, and *pos* $= 4$, *SET2* $= \{\langle 4:0:\{3\}\rangle\}$, and $N = 11$.
r u n a s **r u n** n i n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 4:0:\{3\}\rangle\}$ and set $N = 12$.
13. Skip to the next loop of *while*, and *pos* $= 5$, *SET2* $= \{\langle 5:0:\{3\}\rangle\}$, and $N = 12$.
r u n a s **r u n n i** n g o n r a m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 5:0:\{3\}\rangle\}$ and set $N = 13$.

14. Skip to the next loop of *while*, and *pos* $= 6$, *SET2* $= \{\langle 6 : 0 : \{3\}\rangle\}$, and $N = 13$.

r	u	n		a	s		**r**	**u**	**n**	**n**	**i**	**n**	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 6 : 0 : \{3\}\rangle\}$ and set $N = 14$.

15. Skip to the next loop of *while*, and *pos* $= 7$, *SET2* $= \{\langle 7 : 1 : \{3\}\rangle\}$, and $N = 14$.

r	u	n		a	s		**r**	**u**	**n**	**n**	**i**	**n**	**g**		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 7 : 1 : \{3\}\rangle\}$ and g is matched at $\langle 7 : 1 : \{3\}\rangle$ in pattern 3. After matching report, set $N = 15$ and *SET1* $= \{\}$.

16. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 15$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

17. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 16$.

r	u	n		a	s		r	u	n	n	i	n	g		**o**	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

18. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 17$.

r	u	n		a	s		r	u	n	n	i	n	g		o	**n**		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

19. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\}$, and $N = 18$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		r	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

20. Skip to the next loop of *while*, and it takes the condition of *else* that *pos* $= 1$, *SET1* $= \{\langle 1 : 0 : \{1, 2, 3\}\rangle\}$, and $N = 19$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		**r**	a	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

21. Take the first loop of *while*, *pos* $= 2$, and *SET2* $= \{\langle 2 : 0 : \{1\}\rangle\}$, $N = 20$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		**r**	**a**	m
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 2 : 0 : \{1\}\rangle\}$ and $N = 21$.

22. Take the first loop of *while*, *pos* $= 2$, and *SET2* $= \{\langle 3 : 1 : \{1\}\rangle\}$, $N = 22$.

r	u	n		a	s		r	u	n	n	i	n	g		o	n		**r**	**a**	**m**
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

SET1 $\leftarrow$ *SET1* $\cap$ *SET2* thus *SET1* $= \{\langle 3 : 1 : \{1\}\rangle\}$ and the matched position is reported. Therefore, $N = 22$ and $N > n$, and the searching is finished.

Lemma 2. If SET1 and SET2 contain the inverted lists; then *SET1* $\leftarrow$ *SET1* $\cap$ *SET2* is correct.

Proof. The correctness is that all inverted lists in φ_0^{pos} or φ_1^{pos} of *SET2* which continue from *SET1* are returned and put into *SET1.* The pre-conditions are $N \geq 2$, and the inverted lists stored in *SET1* and *SET2.* The post-condition is that all inverted lists of φ_0^{pos} or φ_1^{pos} in *SET2* which continue from *SET1* are returned and put into *SET1.*

The required position is *pos*, and the continuity is that all inverted lists φ_0^{pos-1} or φ_1^{pos-1} of *SET1* are described by φ_0^{pos} or φ_1^{pos} in *SET2. SET1* and *SET2* contain any $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ of the hashing set in Definition 3, and they are the second level of the perfect hashing table. It can be said that every inverted list of *SET2* must be inspected and compared with the inverted lists in *SET1* by the properties of intersection. Thus, the results are φ_0^{pos} and/or φ_1^{pos} and φ_0^1. The post-condition is then reached. □

Theorem 8. Algorithm 4 is correct for searching P in the given text T.

Proof. The correctness is proved by the induction on n for t_1 to t_n. The proposed invariants are $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$.

Assume that line 1 and line 2 are true; then in the base case the variable N is increased by 1 before getting to the *while* loop. This step needs to be proved in the case of *SET1 <> null* and *SET1 = null.* Then if *SET1 <> null* the inverted lists of $text[N]$ are taken to *SET2* and $1 \leq pos \leq l_{max}$, the correctness is proved by the intersection in Lemma 4.1. If *SET1 = null*, then the $text[N]$ is taken to *SET1* and then after the end of this iteration $1 \leq N \leq n$ and $1 \leq pos \leq l_{max}$. In both cases, the invariants remain unchanged and thus this step is true. In the inductive step, the iteration $n-1$ needs to prove when the case of *SET1 <> null.* This algorithm runs the variable n by the fixed number and then the iteration $n-1$ is reached and all iterations from t_3 to t_{n-1} are true (by reporting the matched patterns in line 4 which prove both the correctness and the pattern continuity). It can then be claimed the iteration t_n is true by induction and the algorithm is correct. □

Lemma 3. The time complexity for taking any $I_{\lambda\varepsilon,0}$ and/or $I_{\lambda\varepsilon,1}$ from SET is $O(1)$.

Proof. From Definition 5, *SET1* or *SET2* contains only one row of inverted lists; also, both of them are the perfect hashing set. Thus, it implies $O(1)$ time by their hashing properties. □

Lemma 4. If SET1 and SET2 contain the inverted lists then *SET1* $\cap$ *SET2* takes $O(1)$ time.

Proof. Let *SET1* contain the inverted list groups $I_{\lambda\varepsilon 1,0}$ and/or $I_{\lambda\varepsilon 1,1}$. Let *SET2* contain the inverted list groups $I_{\lambda\varepsilon 2,0}$ and/or $I_{\lambda\varepsilon 2,1}$. *SET1* and *SET2* are the perfect hashing set; then every operation can be solved in $O(1)$ time using Lemma 3. Therefore, every operation to access $I_{\lambda\varepsilon 1,0}$, $I_{\lambda\varepsilon 1,1}$,$I_{\lambda\varepsilon 2,0}$, and $I_{\lambda\varepsilon 2,1}$ also takes $O(1)$ time by Lemma 1. □

Theorem 9. Searching the occurrences of $P = \{p^1, p^2, \ldots, p^r\}$ which appear in the given text $T = \{t_1 t_2 t_3 \ldots t_n\}$ takes $O(n)$ time where n is the length of T.

Proof. The proof is that all characters of $t_1t_2t_3 \dots t_n$ are scanned, and all occurrences are reported in n time. Referring back to the searching algorithm, the time complexity is dominated by the variables N, *SET1*, and *SET2*. The *while* loop (line 3) is repeated to inspect the inverted lists of t_2 to t_n. Each iteration of the loop definitely reports all occurrences in line 4 with $O(1)$ by Lemma 4. It can be said that the loops of line 3 take $O(n)$ time because this step is processed from the initial step to n time, and lines 5, 6, and 8 take a constant time by Lemma 3 and Lemma 4. Therefore, the time complexity takes only $O(n)$ time. Also, this algorithm is able to perform in both an average case and a worst case scenario. $\square$

5 EXPERIMENTAL METHODS AND THEIR RESULTS

The sub-sections below begin to explain the implementation details, then the first set of experiments shows the time and the space requirements of the inverted lists structure. Furthermore, the searching times in several patterns and several given text sizes are also presented.

5.1 Implementation

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.

Implementing data structures for pre-processing phase, Aho-Corasick Trie [1] (named AC-Trie), Reverted Trie of SetHorspool(mentioned in [18]), and dynamic Suffix tree [14] were implemented for comparing with the inverted lists structure. In the searching phase, the searching algorithms of Aho-Corasick [1], SetHorspool (mentioned in [18]), and inverted lists algorithm were implemented. Additionally, the programs for randomize the pattern and the text were also implemented.

All of them were implemented in Java with JavaTM 2 SDK, Standard Edition Version 1.6.22 built in the Netbeans 6.9.1. The abstract data type (ADT) of java.util.Vector was employed for accommodating all structures which were compared. AC-Trie and Reverted-Trie structures were created by the special classes to represent the nodes of Trie and Reverted-Trie, and then they were put into the instances of java.util.Vector. Table τ was created by the java.util.Vector as well, but each instance in the second level of the perfect hasing table (set of positions and set of pattern numbers) was implemented by the java.util.HashTable and the java.util.HashSet structure respectively. For the new proposed algorithm, the variables $SET1$ and $SET2$ were also implemented by java.util.HashTable; as well as, all results were kept in the instances of java.util.ArrayLists.

The data tests of $|\sum|$ were the 52 letters of the English alphabet; 'A' to 'Z' and 'a' to 'z'. The pattern lengths were randomized from 3 to 20 characters, and the average length was 12 characters. The proposed numbers of patterns were 10; 100; 1 000; 10 000; 50 000; and 100 000 and 300 000 (only for the inverted lists algorithm). Each of the pattern numbers was randomly built in 10 files. The texts were randomized from the size of 1 KB, 10 KB, 100 KB, 1 MB, 5 MB, and 10 MB. Also, each of the text sizes was performed in 10 files as well.

For pre-processing tests, each file in each group was read and generated to the data structures one by one. Then the processing time of each file was captured in nano-seconds. Afterwards, each file was built again and both the data structure and the memory usage was captured in Kilo-Bytes. Performing the searching experiments, every pattern file was paired with each text file. For instance, the first file of 10 patterns was paired by the first file of 1 KB, the second file of 10 patterns was paired by the second file of 1 KB, and the other cases were performed in the same way. When the search in each pair of pattern and text size completed, the processing time in nano-seconds was captured. Then, when the 10 pairs of each group of text finished processing, the average time was given.

5.2 Pre-Processing Results

The inverted lists structure was constructed faster and used smaller space than the earlier structures (Aho-Corasick [1] called AC-Trie, SetHorspool in [18] called Reverted-Trie, and the suffix tree[14]).

The inverted lists structure takes the shorter average time than AC-Trie 3.75 folds, the Reverted-Trie 2.33 folds, and the suffix tree 15.69 folds. The resulting details are shown in Table 2, which converts the nano-time to the seconds where '–' means the data structure could not construct (out of the Java heap memory).

Then, the inverted lists structure used less average space than the AC-Trie 18.42%, the Reverted-Trie 20.05 %, and the sufffix tree 92.38 %. In the case of pattern numbers above 1 000, the suffix tree could not create the structure because our computer was out of heap memory in Java while generating the structure. The results are shown in Table 3.

# patterns	AC-Trie	Reverted-Trie	Suffix Tree	Inverted Lists
10	0.161	0.095	0.154	0.030
50	0.152	0.201	0.235	0.113
100	0.401	0.466	0.467	0.278
500	0.566	0.710	19.276	0.351
1 000	1.023	1.905	708.274	0.767
5 000	10.791	8.001	–	5.519
10 000	45.431	20.728	–	6.918
50 000	532.518	110.561	–	43.623
100 000	3 598.131	5 745.879	–	851.156
300 000	–	–	–	1 132.651

Table 2. Processing time (seconds)

# patterns	AC-Trie	Reverted-Trie	Suffix Tree	Inverted Lists
10	4.71	4.93	24.88	4.56
50	4.82	4.96	48.35	4.81
100	4.90	5.10	896.11	4.890
500	5.59	5.67	2 512.46	5.12
1 000	6.21	6.29	–	5.33
5 000	11.10	11.22	–	7.56
10 000	15.83	16.13	–	9.84
50 000	54.57	55.11	–	23.36
100 000	155.84	131.14	–	47.631
300 000	–	–	–	169.584

Table 3. Memory usages (KB)

5.3 Searching Results

The searching times of the inverted lists algorithm (represented by IVL) were more efficient than the SetHorspool algorithm (represented by HP) in average, but took a longer time than the Aho-Corasick (represented by AC). In the case of small pattern numbers and small text sizes, the inverted lists algorithm took an almost equal searching time to that of Aho-Corasick.

In the case of the large pattern numbers and the large text sizes, the proposed algorithm took an almost similar time to SetHorspool in some cases. It should also be noticed that the bottleneck of our algorithm occurs if there are a large number of patterns, in which case then the inverted lists break into serveral groups. Although the intersection can be operated one time per each intersection, it needs the time to analyze the sequence of continuity and the time to check the matching positions. These two points need the time to process and then the searching times are longer than the static algorithms such Aho-Corasick [1] which compares only once per character.

The following tables (4 to 9) show the experimental results which are crossed by the text sizes 1 KB, 10 KB, 100 KB, 1 MB, 5 MB and 10 MB; and the pattern numbers 10; 50; 100; 500; 1 000; 5 000; 10 000; 50 000; 100 000 and 300 000 (only the inverted lists algorithm).

6 DISCUSSIONS AND SUGGESTIONS

This section describes the advantages of the inverted lists structure, opening the research how to improve the inverted lists structure, and suggestions on how to apply this structure to other matching principles.

The primary advantage of the inverted lists structure is that the expected pattern to be matched can be reported over time because this structure keeps the positions and the numbers of patterns. Then the searching results are based not only on yes or no answers, but can also report all numbers and patterns to be matched over

# patterns	AC	HP	IVL
10	0.011	0.045	0.040
50	0.018	0.102	0.042
100	0.020	0.134	0.045
500	0.017	0.201	0.147
1 000	0.039	0.212	0.165
5 000	0.042	0.186	0.167
10 000	0.033	0.165	0.159
50 000	0.029	0.734	0.393
100 000	0.038	0.817	0.498
300 000	–	–	0.987

Table 4. Searching time (seconds) in the given 1 KB text

# patterns	AC	HP	IVL
10	0.066	0.200	0.098
50	0.102	0.755	0.168
100	0.104	1.574	0.141
500	0.112	1.698	0.365
1 000	0.120	1.733	1.001
5 000	0.161	1.695	1.576
10 000	0.210	1.889	1.301
50 000	0.705	2.034	1.696
100 000	1.667	2.701	2.019
300 000	–	–	3.089

Table 5. Searching time (seconds) in the given 10 KB text

time. In contrast, the traditional data structures mentioned in Sections 1 and 2 are not able to handle this aspect because they need to access by sequencing the root of structures.

Considered by the type of each window search, Navarro and Raffinot [16] divide the searching approach to prefix approach (comparing from left to right), suffix approach (comparing from right to left), and factor approach (comparing by calculating the special positions). Then the inverted lists structure is unsequenced to access and compare the characters in the target text. Thus, the target text can be scanned by all mentioned approaches. Moreover, a parallel approach (simultaneous access) could be applied on this structure.

For suggestions, the inverted lists structure could improve the space by grouping or compressing the lists of the pattern sets. For instance, if the pattern numbers are connected in the sequence such as $\langle 1:0:\{1,2,3,4,5,6,7,8,9,10\}\rangle$, they can be grouped as $\langle 1:0:1{-}10\rangle$. Furthermore, this structure could be applied to solve other problems such as approximate matching, regular search, two-dimensional matching, pattern recognition, text indexing and so on.

# patterns	AC	HP	IVL
10	0.301	1.272	1.004
50	0.554	8.012	1.621
100	0.588	13.101	1.589
500	0.634	17.892	4.984
1 000	0.579	15.605	7.312
5 000	1.405	18.243	6.988
10 000	1.464	19.454	9.002
50 000	2.185	20.798	10.102
100 000	4.387	25.391	15.235
300 000	–	–	20.680

Table 6. Searching time (seconds) in the given 100 KB text

# patterns	AC	HP	IVL
10	3.143	24.231	23.623
50	6.286	108.732	30.380
100	5.012	131.076	29.766
500	7.501	155.205	45.291
1 000	7.003	190.532	46.665
5 000	26.784	175.989	49.571
10 000	58.860	179.859	51.651
50 000	90.725	249.101	94.290
100 000	102.198	321.761	134.872
300 000	–	–	356.712

Table 7. Searching time (seconds) in the given 1 MB text

7 CONCLUSION AND PLANNED FUTURE WORKS

A linear time dynamic dictionary matching algorithm, which improves the approach presented in [12], is proposed. This solution adapts the linear time static dictionary matching [30], and especially the inverted lists structure for accommodating the dynamic dictionary. The inverted lists structure is implemented by the perfect hashing table, and it is constructed in optimal time. Furthermore, it is able to insert or delete an individual pattern in minimal time. In theoretical results, this solution takes (1) $O(|P|)$ time for pre-processing where $|P|$ is the sum of all pattern lengths, (2) $O(|p|)$ time for inserting or deleting the pattern where $|p|$ is the length of pattern to be inserted or deleted, and (3) $O(n)$ time for searching in an average and a worst case scenario where n is the length of the given text. In experimental results, the inverted lists structure takes less time and space than the traditional structures; and, the searching time is processed in a linear time. In near future, we will reduce the table space and create the dynamic dictionary matching algorithm using the suffix approach and the factor approach for improving the time complexity.

# patterns	AC	HP	IVL
10	15.118	103.881	98.612
50	30.889	400.266	126.493
100	31.997	766.786	136.431
500	35.150	804.195	139.498
1 000	42.807	869.241	153.521
5 000	129.047	900.480	161.541
10 000	156.598	905.586	213.094
50 000	201.003	1 521.231	300.691
100 000	302.677	1 941.345	341.861
300 000	–	–	399.765

Table 8. Searching time (seconds) in the given 5 MB text

# patterns	AC	HP	IVL
10	40.907	250.583	189.778
50	69.397	882.667	287.018
100	72.976	1 534.896	291.781
500	85.781	1 688.574	320.745
1 000	89.481	2 457.665	331.905
5 000	290.882	2 561.901	348.921
10 000	316.175	2 551.012	449.005
50 000	452.236	2 631.422	631.448
100 000	649.133	2 752.843	739.094
300 000	–	–	876.339

Table 9. Searching time (seconds) in the given 10 MB text

Also, the approximate matching algorithm is being developed by the inverted lists structure.

REFERENCES

[1] Aho, A. V.—Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search. Comm. ACM, 1975, pp. 333–340.

[2] Amir, A.—Farach, M.: Adaptive Dictionary Matching. In Proc. of the 32nd IEEE Annual Symp. on Foundations of Computer Science, 1991, pp. 760–766.

[3] Amir, A.—Farach, M.—Idury, R. M.—La Poutré, J. A.—Schaffer, A. A.: Improved Dynamic Dictionary-Matching. In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms 1993, pp. 392–401.

[4] Amir, A.—Farach, M.—Galil, Z.—Giancarlo, R.—Park, K.: Dynamic Dictionary Matching. Journal of Computer and System Science, Vol. 49, 1994, No. 2, pp. 208–222.

[5] Amir, A.—Farach, M.—Idury, R. M.—La Poutré, J. A.—Schffer, A. A.: Improved Dynamic Dictionary Matching. Information and Computation, Vol. 199, 1995, No. 2, pp. 258–282.

[6] Amir, A.—Landau, G. M.—Lewenstein, M.—Sokol, D.: Dynamic Text and Static Pattern Matching. ACM Transactions on Algorithms, Vol. 3, 2007, No. 2, Article 19, pp. 1–24.

[7] Botelho, F. C.: Near-Optimal Space Perfect Hashing Algorithms. Ph. D. thesis, Federal University of Minas Gerais, Brazil.

[8] Chan, H.-L.—Hon, W.-K.—Lam, T.-W.—Sadakane, K.: Dynamic Dictionary Matching and Compressed Suffix Trees. SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM ymposium on Discrete Algorithms 2005, pp. 13–22.

[9] Commentz-Walter, B.: A String Matching Algorithm Fast on the Average. In Proceedings of the Sixth International Collegium on Automata Languages and Programming 1979, pp. 118–132.

[10] Gongshen, L.—Jianhua, L.—Shenghong, L.: New Multi-Pattern Matching Algorithm. Journal of Systems Engineering and Electronics, Vol. 17, 2006, No. 2, pp. 437–442.

[11] Hong, Y. D.—Ke, X.—Yong, C.: An Improved wu-Manber Multiple Patterns Matching Algorithm. 25th IEEE International Conference on Performance Computing and Communications (IPCCC) 2006, pp. 675–680.

[12] Khancome, C.—Boonjing, V.: Dynamic Dictionary Matching Using Inverted Lists. Proceeding of the Third IASTED International Conference Advances in Computer Science and Technology (ACST 2007), pp. 397–401.

[13] Knuth, D. E.—Morris, J. H.—Pratt, V. R.: Fast Pattern Matching in Strings. SIAM Journal on Computing, Vol. 6, 1997, No. 1, pp. 323–350.

[14] McCreight, E. M.: A Space-Economical Suffix Tree Construction Algorithm. Journal of Algorithms, Vol. 23, 1976, No. 2, pp. 262–272.

[15] Melnik, S.—Raghavan, S.—Yang, B.—Garcia-Molina, H.: Building a Distributed Full-Text Index for the Web. ACM Transactions on Information Systems, Vol. 19, 2001, No. 3, pp. 217–241.

[16] Moffat, A.—Zobel, J.: Self-Indexing Inverted Files for Fast Text Retrieval. ACM Transactions on Information Systems, Vol. 14, 1996, No. 4, pp. 349–379.

[17] Monz, C.—de Rijke, M.: Inverted Index Construction (2002), Availaible on: `http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf`.

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

[19] Pagh, R.: Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions. 2009, availaible on: `www.it-c.dk/people/pagh/papers/hash.pdf`.

[20] Sahinalp, S.—Vishkin, U.: Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm (Extended Abstract). In 37th Annual Symposium on Foundations of Computer Science 1996, pp. 320–328.

[21] Salmela, L.—Tarhio, J.—Kytöjoki, J.: Multipattern String Matching with Q-Grams. ACM Journal of Experimental Algorithmics (JEA), Vol. 11, 2006, Article No. 1.1, pp. 1–19.

[22] Sleator, D. D.—Tarjan, R. E.: A Data Structure for Dynamic Trees. Journal of Computer and System Sciences, Vol. 26, 1983, No. 3, pp. 362–391.

[23] Weiner, P.: Linear Pattern Matching Algorithms. In Proceedings of Symposium on Switching and Automata Theory 1973, pp. 1–11.

[24] Knuth, D. E.: The Art of Computer Programming, Volume 3. Addison-Wesley Publishing Company 1973, pp. 506–549.

[25] Wu, S.—Manber, U.: A Fast Algorithm for Multi-Pattern Searching. Report tr-94-17, Department of Computer Science, University of Arizona, Tucon, AZ 1994.

[26] Yates,R. B.—Neto, B. R.: Modern Information Retrieval. The ACM Press – A Division of the Association for Computing Machinery, Inc., 1999, pp. 191–227.

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

[28] Zobel, J.—Moffat, A.: Inverted Files Versus Signature Files for Text Indexing. ACM Transaction on Database Systems, Vol. 23, 1998, No. 4, pp. 453–490.

[29] Zobel, J.—Moffat, A.: Inverted Files for Text Search Engines. ACM Computing Surveys, Vol. 38, 2006, No. 2, pp. 1–56.

[30] Khancome, C.—Boonjing, V.: Optimal Linear-Time Multi-String Pattern Matching Algorithm. International Journal of Computational Science, Vol. 3, 2009, No. 6, pp. 629–641.

[31] Khancome, C.—Boonjing, V.: Inverted Lists String Matching Algorithms. International Journal of Computer Theory and Engineering, Vol. 2, 2010, No. 3, pp. 1793–8201.

[32] Ćisar, P.—Bošnjak, S.—Maravić Ćisar, S.: EWMA Based Threshold Algorithm for Intrusion Detection. Computing and Informatics, Vol. 29, 2010, No. 6+, pp. 1089–1101.

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

[34] Makula, M.—Beňušková, L.: Interactive Visualisation of Oligomer Frequency in DNA. Computing and Informatics, Vol. 28, 2009, No. 5, pp. 695–710.

[35] Hu, Y.—Wang, P.-F.—Hwang, K.: 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.

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

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

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

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

[40] Jin Kim, H.—Kim, H.-S.—Kang, S.: A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 22, 2011, No. 11, 2011, pp. 1904–1911.

[41] Dai, L.—Xia, Y.: A Lightweight Multiple String Matching Algorithm. International Conference on Computer Science and Information Technology (ICC-SIT'08), Singapore 2008, pp. 611–615.

Chouvalit Khancome is a lecturer in Department of Computer Science, Faculty of Science and Technology, Rajanagarindra Rajabhat University. Also, he is the researcher in Computer Science and Informatics Laboratory of this department. His research areas include text compression, string matching, multiple string matching, and dictionary matching. He received the B. Sc. degree in computer education from Mahasarakam Rajabhat University, Thailand, the M. Sc. degree in computer science from King Mongkut's Institute of Technology, Ladkrabang, Thailand, and the Ph. D. degree in computer science from King Mongkut's Institute of Technology, Ladkrabang, Thailand.

Veera Boonjing is a Professor of Computer Science at the Department of Computer Science, Faculty of Science, KMITL, Thailand. He received the B. Sc. degree in mathematics from Ramkhamhaeng University, Thailand, the M. Sc. degree in computer science from Chulalongkorn University, Thailand, and the Ph. D. degree in decision sciences and engineering systems from Rensselaer Polytechnic Institute, USA.