Data Structure for Dynamic Patterns

Chouvalit Khancome and Veera Boonjing, *Member, IAENG*

***Abstract*—String matching and dynamic dictionary matching are significant principles in computer science. These principles require an efficient data structure for accommodating the pattern or patterns to be searched for in a large given text. Moreover, in the dynamic dictionary matching, the structure is able to insert or delete the individual patterns over time. This research article introduces a new dynamic data structure named *inverted lists* for both principles. The inverted lists data structure, which is derived from the inverted index, is implemented by the perfect hashing idea. This structure focuses on the position of characters and provides a hashing table to store the string patterns. The new data structure is more time efficient than traditional structures. Also, this structure is faster to construct and consumes less memory than others.**

***Index Terms*—Data Structure, String Matching, Multiple String Matching, Suffix Tree, Trie, Bit-parallel, Hashing Table, Dictionary matching, inverted index, inverted list.**

I. Introduction

There are many principles emerging from string processing such as string pattern matching, multiple string pattern matching called static dictionary, dynamic multiple patterns string pattern matching called dynamic dictionary matching. All of them deal with the pattern or patterns of string to be searched for in a large given text. Basically, pattern or patterns are generated to suitable data structures which are then provided for searching.

For solving the problem, string pattern matching deals with single string pattern $p=c_1c_2c_3\ldots c_m$, while dynamic dictionary matching deals with multiple pattern strings $P=\{p^1, p^2,..,p^r\}$. And the patterns in P enable the ability to update individual patterns over time. Traditionally, Trie, Bit-parallel, Hashing table, and Suffix tree are the data structures used for accommodating p or P.

Trie, known as classic data structure for static dictionary, has been used for accommodating patterns for a long time. This structure employs an automaton to contain the set of states labeled by characters of patterns. Many algorithms are based on Trie such as the first linear time (Aho-Corasick [1]—extended from [15]), the sub-linear time (Commentz-Walter[11]—extended from [12]) and SetHorspool (mentioned in [17]). However, when implementing the Trie to applications a large amount of memory is consumed. And when the pattern is updated Trie needs to regenerate the structure with O(|P|) time where |P| is the sum of all pattern lengths.

Bit-parallel data structure employs the sequence of bit to store the patterns. Navarro and Raffinot [17] showed how to apply the single string Shift-Or and Shift-And to Multiple Shift-And[8], Multiple-BNDM[9], and [10]. Nevertheless, this structure is restricted by the word length of computer architecture; furthermore, it requires special methods which are more complex in converting the patterns to the bit form.

The first hashing idea was presented by Karp and Rabin [14] 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 [14] take O($n|P|$) time (exhaustive solution) where n is the length of the given text. A more efficient algorithm presented by Wu and Manber [24] creates the reverted Trie, the shift table, and implements the hashing table for storing the block of patterns to solve the problem. The last solution [25] improves Wu and Manber [24], but it does not support updating the patterns.

Suffix tree is implemented for accommodating the dynamic patterns. Generally, this structure does not directly support the dynamic patterns because it needs to employ the dynamic mechanism of McCreight [16], DS-List [14], or Weiner [23]. Thus, implementing suffix tree to algorithms must attach O($log|P|$) time because the tree structure is embedded by $log|P|$ for data accessing. The first suffix tree algorithm presented by Amir and Farach [2] is the first adaptive algorithm that displayed exhaustive time consumption. Subsequently, [3], [4], [5], [6], [7], [8], [9] and [22] showed the logarithmic algorithms of suffix tree generalization. Therefore, this structure is immediately challenged by how to escape from the factor of logarithmic time ($nlog|P|$). Moreover, the applications which are implemented by suffix tree take more memory than Trie as well.

For solving the problem of information retrieval, the inverted index has been applied; this structure can be adapted to several data structures. The motivation of this research is from the inverted index which focuses on the keywords. But the new structure is based on characters.

This research article proposes to adapt the inverted index [13], [18], and [19] to create a new data structure called inverted lists for accommodating the pattern or patterns. This structure uses O(m) time and O($m+|\lambda|$) space for managing the single pattern string where m is the length of pattern p. For dynamic dictionary matching, the inverted lists structure takes O(|P|) time and O($|\lambda|+|P|$) space where λ are any characters which are exactly used in a finite alphabet $\sum$

Manuscript received December 8, 2009.

Chouvalit Khancome is a PhD student in Computer Science at King Mongkut's Institute of Technology Ladkrabang, Thailand. He is a teacher in the Department of Computer Science, Rajanakarindra Rajabhat University, Thailand; (e-mail: chouvalit@ hotmail.com).

Veera Boonjing is an Associate Professor in Computer Science, Department of Mathematics and Computer Science at King Mongkut's Institute of Technology Ladkrabang, Thailand. Also, he is working in National Centre of Excellence in Mathematics, PERDO, Bangkok, Thailand 10400, (e-mail: kbveera@kmitl.ac.th).

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

and $\lambda \subseteq \sum$. Importantly, this structure can handle the inserting or deleting of the individual pattern in O(|*p*|) time where *p* is the individual pattern to be inserted or deleted. In experimental results, this structure is constructed faster and uses less memory than the traditional data structures.

The remaining sections are organized as follows. Section II shows how to derive the inverted index and the perfect hashing principle for accommodating the new data structure. Section III describes the inverted lists for string pattern matching. Section IV shows the inverted lists for dictionary matching. Section V illustrates the implementations and the experimental results, and section VI is a conclusion.

II. Deriving the Principles

A. Deriving the Inverted Index

The inverted index structure represents the words in the target documents by the form of *<documentID, word:pos>* 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*. The original inverted index [13], [18], and [19] assign all documents as $D=\{D_1 \ldots D_n\}$ where D_i is any document that contains the various keywords with different positions and $1 \le i \le n$.

Each document is analyzed for keeping keywords and their positions. For instance, if the document D_1 has the keywords w_a*:1*,w_b*:2*,w_c*:3*, it can be said that the keyword w_a appears at position 1, w_b appears at position 2, and w_c appears at position 3. Then, all keywords can be rewritten by the form of *word: (posting lists)* where '*posting list*' is *(documentID: word position in that document)*. Thus, all keywords in document D_1 can be rewritten as w_a: (1:1), w_b: (1:2), and w_c: (1:3) respectively.

Afterwards, the keywords and posting lists are converted into the suitable data structures such as B^+tree, suffix tee, and suffix array. This research focuses on the position of characters instead of the keywords. Initially, the document *D* is replaced by the pattern *P*, and each D_i is replaced by p^i. For instance, if there are the patterns *P={aab, aabc, aade}* then the patterns are assigned as D_1=*aab*, D_2=*aabc*, and D_3=*aade*. Then, they are defined by the form of *character :<the occurrence position of character in pattern: the indicated status of the last character of pattern: the number of pattern in P>*; e.g., *a:<1:0:1>*, *<2:0:1>*, *<1:0:2>*, *<2:0:2>*,… . Each list in this form is called the "*individual posting list*". Using this method, all of the individual posting lists can be applied for accommodating the dictionary.

B. Deriving the Perfect Hashing

The most powerful hashing principle is the perfect hashing which takes O(1) time in worst-case performances (shown in [26], [27] and [28]) 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. For this reason, the perfect hashing is chosen for implementing the inverted lists.

The perfect hashing uses the universal key *U* to accommodate all keys for accessing all data in the table and two-level schemes for implementation. The first level is the *n* keys for hashing with chaining to the second level by function *f(n)*, and the second level is the data items associated with the corresponding key of *n*. 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 the groups of posting lists as the data items in the second level where $\lambda \subseteq \sum$.

III. Inverted Lists for Single Pattern

Definition 1 Let $p=c_1c_2c_3\ldots c_m$ be the pattern input, and c_k is any alphabet that occurs in *p* at position *k* where *k=1,2,3,…,m*. The inverted list of c_k can be written by c_k*:<k:0>* if only if *k<m*, or c_k*:<k:1>* if only if *k=m*. Symbolically, c_k*: <k:0>* is represented by *c:* I_{k_0} , and c_k*:<k:1>* is represented by *c:* I_{k_1} .

Example 1. The inverted lists of *p=aabcz*. We have c_1=*a*, c_2=*a*, c_3=*b*, c_4=*c*, and c_5=*z*. The whole inverted list of *p* are *a:<1:0>*, *a:<2:0>*, *b:<3:0>*, *c:<4:0>*, and *z:<5:1>*.

Definition 2 The perfect hashing table which is provided for storing the inverted lists of the single pattern *p* is called the inverted lists table for single patterns and is denoted by τ_s .

Example 2. The table τ_s of pattern *p=aabcz*.

Table 1. The inverted list table τ_s of *p=aabcz*.

$\sum$	I_{k_0} / I_{k_1}	(i.e., the inverted list)
a	I_{1_0} , I_{2_0}	*<1:0>,<2:0>*
b	I_{3_0}	*<3:0>*
c	I_{4_0}	*<4:0>*
z	I_{5_1}	*<5:1>*

Lemma 1 Let I_{k_0} and I_{k_1} be the inverted list of $p=c_1c_2c_3\ldots c_m$. If I_{k_0} and I_{k_1} are stored in the hash table τ_s then accessing I_{k_0} or I_{k_1} takes O(*1*) time where *k=1,2,3,…,m*..

Proof Each inverted list I_{k_0} or I_{k_1} can be retrieved from the table τ_s in O(*1*) time. Given *f(x)* be a hashing function which c_{k_0} is a key to access I_{k_0} , and c_{k_1} is the key to access I_{k_1} where *c* is the character in λ . The table τ_s is implemented by the perfect hashing table. Thus, to retrieve the inverted list I_{k_0} by $f(c_{k_0})$ or to retrieve the inverted list I_{k_1} by $f(c_{k_1})$ takes O(*1*) times by the hashing properties. □

This phase creates the inverted list table τ_s for all alphabets in λ . The next method reads the characters one by one and generates to inverted lists, and each inverted list is added into the table τ_s . Algorithm 1 shows this method.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Algorithm 1.
Input : p=c_1,c_2,c_3,…c_m
Output: Table τ_s of p

1. Create table τ_s
2. j=1
3. while (j<=m) do
4. Create the inverted list of c_j → τ_s at char(c_j)
5. j←j+1
6. end while

Algorithm 1 takes O(*m*) time and O(*m*+|λ|) space shown as the proof in Theorem 1.

Theorem 1 Let $p=c_1c_2c_3…c_m$ be the pattern input, and given τ_s be the inverted list table of single pattern. Generating all characters of *p* to the inverted lists and adding all inverted lists into τ_s takes O(*m*) time, and the table τ_s uses O(*m*+|λ|) space.

Proof For time complexity, the hypothesis is that the whole characters of *p* are generated into the table τ_s. Line 1 creates the table, and line 2 initializes variables, which take O(*1*). Line 3 needs to repeat *m* rounds, and it also takes O(*m*) time. Line 4 is O(*1*) by Lemma 1 while line 5 takes O(*1*) as line 2. Therefore, the preprocessing time take O(*m*) time. □

For space complexity, the table τ_s is created with the size |λ| for all alphabet of $\lambda \subseteq \sum$. Each inverted list of c_k of $p=c_1c_2c_3…c_m$ uses one space per one inverted list then for *k=1* to *k=m* take *m* space as well. Hence, all required spaces of τ_s is O(*m*+|λ|).□

IV. INVERTED LISTS FOR DYNAMIC PATTERNS

Definition 3 Let $P=\{p^1, p^2, p^3,…, p^r\}$ be the set of patterns where p^i is the pattern i^{th} of *m* character $\{c_1c_2c_3…c_m\}$, and $1 \le i \le r$. An individual posting list of a character c_k from the pattern p^i is defined as c_k:<*k:0:i*> if *k*<*m*, or c_k:<*k:1:i*> if *k*=*m*. Symbolically, c_k:<*k:0:i*> is $\varphi_0^{k_i}$, and c_k:<*k:1:i*> is $\varphi_1^{k_i}$ where $1 \le k \le m$.

As in Definition 3, if $P=\{p^1=aab, p^2=aabc, p^3=aade\}$ then the documents are $p^1=a_1a_2b_3$, $p^2=a_1a_2b_3c_4$ and $p^3=a_1a_2d_3e_4$. The individual posting lists of *P* are defined as below:

p^1= *a:<1:0:1>, a:<2:0:1>, b:<3:1:1>,*
p^2=*a:<1:0:2>, a:<2:0:2>, b:<3:0:2>*
c:<4:1:2>, and
p^3= *a:<1:0:3>, a:<2:0:3>, d:<3:0:3>,*
e:<4:1:3>.

Notice that all individual posting lists above can be grouped to a new form such as *a:<1:0:{1,2,3}>*, *<2:0:{1,2,3}>*, *b:<3:1:{1}>*, *<3:0:{2}>*, and so on. Definition 2 shows how to group the posting lists to a new form.

Definition 4 Let l_{max} be the maximum length of patterns in $P=\{p^1,p^2,p^3,…,p^r\}$, and let ε be the position of the same character λ which appears in the various patterns of *P* where $1 \le \varepsilon \le l_{\max}$ and $\lambda \subseteq \sum$. The posting lists of λ 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 $1 \le \{i,l,..., p,q\} \le r$. A group of posting lists of λ can be defined as follows.

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

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

Definition 5 The inverted list of alphabet λ is defined as $I_{\lambda_{\varepsilon,0}}$ if only if the group of posting lists is $\lambda_{\varepsilon,0}$. Similarly, the inverted list of alphabet λ is denoted as $I_{\lambda_{\varepsilon,1}}$ if only if the group of posting lists is $\lambda_{\varepsilon,1}$.

With Definitions 4 and 5, if the posting lists are *a:<1:0:{1,2,3}>*, *a:<2:0:{1,2,3}>* then the groups of posting lists must be written as $I_{a_{1,0}}$ and $I_{a_{2,0}}$ respectively (shown in table 2).

Definition 6 The perfect hashing table which provides for all alphabets over $\sum$ and their corresponding inverted lists of *P* is called the inverted lists table for the dynamic dictionary; this table is denoted as τ_d.

From Definition 6, it can be said that all characters λ are stored in the first column of τ_d, and $I_{\lambda_{\varepsilon,0}}$ and/or $I_{\lambda_{\varepsilon,1}}$ are stored in the second column of τ_d. For instance, if there is *P={aab, aabc, aade}* then *P* can be implemented to the perfect hashing table as the table 2.

Example 3. Table 2 shows the table τ_d of *P={aab, aabc, aade}*.

Table 2. The inverted list table τ_d of *P={aab, aabc, aade}*.

f(λ)	Inverted lists	i.e., the granular inverted lists
a	$I_{a_{1,0}}$, $I_{a_{2,0}}$	*<1:0:{1,2,3}>*, *<2:0:{1,2,3}>*
b	$I_{b_{3,1}}$, $I_{b_{3,0}}$	*<3:1:{1}>*, *<3:0:{2}>*
c	$I_{c_{4,1}}$	*<4:1:{2}>*
d	$I_{d_{3,0}}$	*<3:0:{3}>*
e	$I_{e_{4,1}}$	*<4:1:{3}>*

A. *Inverted lists table construction*

First of all, the empty table τ_d is built for λ, and the entire patterns are then generated to the inverted lists and are added into the table τ_d after the table is constructed. If the inverted lists of target character are already stored in the table, only the number of pattern is added to the corresponding inverted lists; otherwise, a new inverted list is created and added into the table. Algorithm 2 shows this method.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Algorithm 2.
Input: $P=\{p^1,p^2,\ldots,p^r\}$
Output : the table τ_d of P

1. initiate τ_d
2. for $i \leftarrow 1$ to r do
3. for $j \leftarrow$ to m do
4. if Exist(p_j^i)=null then
5. $\tau_d \leftarrow \varphi_0^{j_i}$ or $\varphi_1^{j_i}$
6. else
7. $I_{char(j)_{j,0}}$ or $I_{char(j)_{j,1}} \leftarrow i$
8. end if
9. end for
10. end for

For analyzing time and space, Algorithm 2 is referred as proof. Theorem 2 shows the time complexity and Theorem 3 illustrates the space complexity.

Lemma 2 If there are the inverted lists $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ of λ in τ_d then to access all inverted lists of λ uses O(*1*) time.
Proof Each alphabet λ is a unique character in $\sum$, and λ is implemented as the first level of the perfect hashing table taking O(1) time. The inverted lists $I_{\lambda_{\varepsilon,0}}$ or $I_{\lambda_{\varepsilon,1}}$ 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 the table can be applied to O(1) as all individual items. □

Theorem 2 Let $P=\{p^1,p^2,p^3,\ldots p^r\}$ be the given patterns. All patterns in P are generated into the table τ_d in O(|P|) time where |P| is the sum of all pattern lengths in P.
Proof The proof is that all characters of P are generated to inverted lists and are added into τ_d in O(|P|) time. All of the pattern lengths are denoted as $|p^1|$, $|p^2|$, $|p^3|$,…, $|p^r|$. For the initial step, the table τ_d is built in O(*1*) time. As soon as the table τ_d is built completely, each pattern is scanned by individual character from the first character to the last character. Thus the time to scan equals the number of pattern length; therefore, all patterns are scanned from the pattern 1 to pattern r. 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 O(|P|) time. Meanwhile, to access the table τ_d for storing the inverted list takes O(*1*) time by Lemma 2. Hence, the preprocessing time is proved in O(|P|) time. □

Theorem 3 The table τ_d requires O(| λ |+|P|) space for accommodating the whole inverted lists of P; where |P| is the sum of pattern lengths of P, and τ_d is the inverted lists table.
Proof All patterns in P contain the various characters over λ by the size | λ | where $\lambda \subseteq \sum$, and the table τ_d is implemented as the perfect hashing table. The algorithm is proved as all characters of P are generated to inverted lists and are added into the table τ_d with |P| space. The lengths of P are $|p^1|$, $|p^2|$, $|p^3|$, …, $|p^r|$, and each p^i contains the string $\{c_1c_2c_3\ldots c_m\}$ 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 τ_d is created by | λ | size. Each inverted list is created by the preprocessing phase for all patterns of P; therefore, each inverted list of string $\{c_1c_2c_3\ldots c_m\}$ in each p^i only takes one space per one list. Thus, the space is equal to $|p^1|+|p^2|+|p^3|+\ldots+|p^r|=|P|$ for the second level of perfect hashing table. Hence, the space of τ_d is O(| λ |+|P|). □

B. Pattern Insertion

The pattern insertion deals with the problem of adding all characters of the individual pattern into the inverted table τ_d and maintains the stable dictionary. Let p^ϕ be a new pattern for insertion where ϕ is a unique number that does not appear in the dictionary before inserting the pattern. We begin to search for the existence of p^ϕ in the table τ_d by the function PExist(). The insertion method will be executed if only if the result is null. Next, all inverted lists of p^ϕ are generated and are added into the table as Algorithm 2. This algorithm is illustrated by Algorithm 3 below.

Algorithm 3.
Input : $p^\phi=\{c_1c_2,\ldots,c_m\}$
Output : p^ϕ is stored in τ_d

1. if PExist(p^ϕ) =null then
2. for j$\leftarrow$*1* to m do
3. if Exist(p_j^ϕ)=null then
4. $\tau_d \leftarrow \varphi_0^{j_\phi}$ or $\varphi_0^{j_\phi}$
5. else
6. $I_{char(j)_{j,0}} / I_{char(j)_{j,1}} \leftarrow \phi$
7. end if
8. end for
9. end if

Theorem 4 Let $p=\{c_1c_2c_3\ldots c_m\}$ be the new individual pattern to be inserted into the existing dictionary $P=\{p^1,p^2,p^3,\ldots p^r\}$. The insertion time is O(|p|) where |p| is the length of p.
Proof Algorithm 3 is referred for the proof. Given p^ϕ be a new pattern which contains a string $\{c_1c_2c_3\ldots c_m\}$ where ϕ is an non-existing number of the pattern in P. The length of p^ϕ is m and represented by |p|. In the initial step, line 1 repeats the search for each character from c_1 to c_m, and it takes m operation which is O(|p|) time. The accessing of the inverted list takes O(*1*) time by Lemma 2. For the inner loop, if a new pattern does not exist in P, line 2 will insert the inverted lists from c_1 to c_m. All operations use |p| time, and the hypothesis is proved as well. Meanwhile, line 2, line 4, or line 6 also access the table, and they take O(*1*) by Lemma 2. It can be said that all operations of line 2 take O(|p|) time; therefore, all characters of p^ϕ are converted and added into the existing table in O(|p|) time. □

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

C. Pattern Deletion

Pattern deletion consists of two methods: (1) looks for the required pattern to delete, and (2) repeats and removes the inverted lists of the target pattern one by one. For deletion mechanism, if the corresponding inverted lists have only one posting list, it will be deleted immediately. Otherwise we will delete only an inverted list in the inverted lists group when the pattern number equals σ . The deletion algorithm is described by Algorithm 4 below.

Algorithm 4.
Input: $p=\{c_1c_2c_3\ldots c_m\}$
Output : p is removed from τ_d

1. if(ExistDel(p) = σ) then
2. for $j \leftarrow$ 1 to m do
3. if posting lists in $I_{char(j)_{j,0}} / I_{char(j)_{j,1}} > 1$ then
4. Delete posting lists number equal to σ
5. else
6. Delete $I_{char(j)_{j,0}} / I_{char(j)_{j,1}}$ of char(p_j^{σ})
7. end if
8. end for
9. end if

Example 4. Take $p=aab$ off from $P=\{aab,aabc,aade\}$.

As the example before, the table 2 is referred, and the deletion is begun in line 1. This inspects and returns the number σ for deletion. The characters in the pattern '*aab*' are converted to *a:<1:0:{1}>*, *<2:0:{1}>*, and *b:<3:1:{1}>*. Then the deleting mechanism is started for deleting the inverted lists one by one. By line 4, the inverted list of *a:<1:0:{1}>* and *<2:0:{1}>* are updated as *<1:0:{2,3}>* and *<2:0:{2,3}>* (by taking the number '*1*' off). Otherwise, the inverted list *b:<3:1:{1}>* is deleted by line 6.

Theorem 5 The deletion pattern p from the existing table of P takes O(|p|) time where p is the target pattern to be deleted, and |p| is the length of pattern p.

Proof Referring to Algorithm 3, let p^i be a pattern to be deleted, which p^i contains a string $\{c_1c_2c_3\ldots c_m\}$ with the length m. The length m is denoted by |p|, and 'i' is the number of the existing pattern i^{th} in P. The hypothesis is that all characters of p^i are removed from the inverted list table of P. Line 2 loops to remove c_1 to c_m. Each operation for accessing the inverted list uses O(*1*) by Lemma 2. Thus the main step is initiated by line 2. This step repeats to read one by one from c_1 to c_m and removes the matched inverted lists from τ_d . All operations take |p| time while line 4 or line 6 uses the constant time by Lemma 2. Therefore, to delete all characters of pattern p^i from τ_d takes O(|p|) time. □

V. Implementation and Experimental Results

A. Implementations

We used the Dell Latitude D500 notebook with Intel Pentium M 1.3 GHz, 512 MB of RAM, and running on Windows XP Home as a running application machine. We implemented Aho-Corasick Trie [1] (named AC Trie), Reverted Tire of SetHorspool in [9], dynamic Suffix tree of [16], and our inverted lists. And the abstract data type (ADT) of java.util.Vector in Java language was employed for accommodating all structures.

The $|\sum|$ was 52 letters of English alphabet; 'A' to 'Z' and 'a' to 'z'. We programmed to randomize each pattern with the various lengths of 3 to 20 characters. The programs randomized the pattern groups of 10, 50, 100, 500, 1000, 5000, 10000, and 50000. Each group contained 10 files, and each file was performed to test 10 times and the average was given.

B. Experimental Results

The tests measure the processing time in seconds, and the memory usages in Kilo-Bytes. The processing time of inverted lists construction was faster than the traditional data structures shown in table 3.

Table 3. Comparing processing time (Seconds).

Pattern Number	AC Trie	Reverted Trie	Suffix Tree	Inverted List
10	0.41	0.62	0.35	0.29
50	0.19	0.10	0.16	0.05
100	0.17	0.27	0.59	0.15
500	0.66	0.98	21.67	0.39
1000	1.34	2.08	-	0.86
5000	13.79	9.02	-	4.55
10000	49.43	26.28	-	7.91
50000	550.74	121.11	-	47.63

For using memory, the inverted lists structure used less memory than the others. However, the suffix tree structure was not able to generate in the case of pattern numbers over 500 patterns because Java language used excessive memory. The results are shown by table 4.

Table 4. Comparing memory usage (KB).

Pattern Number	AC Trie	Reverted Trie	Suffix Tree	Inverted List
10	4.74	4.95	24.86	4.87
50	4.83	4.98	48.34	4.89
100	4.92	5.02	896.12	4.90
500	5.60	5.66	2512.56	5.11
1000	6.29	6.30	-	5.32
5000	11.07	11.23	-	7.57
10000	15.86	16.15	-	9.83
50000	54.56	55.15	-	23.38

VI. Conclusion

This research has presented the new data structure called inverted lists for dynamic patterns of string matching and dynamic dictionary matching. The inverted lists structure uses O(m) time and O(m+|λ|) space for accommodating the single pattern string where m is the length of pattern. In the dynamic patterns, this structure takes O(|P|) time and O(|λ|+|P|) space for accommodating multi-patterns string where P is the sum of pattern lengths, and λ represents any characters which are exactly used in a set of a finite alphabet $\sum$. Furthermore, this structure is able to insert or delete the individual pattern in O(|p|) time where p is the individual pattern to be inserted or deleted. In experimental results, this

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

structure is faster and uses less memory than the traditional data structures.

REFERENCES

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

[2] A. Amir and M. Farach. "Adaptive dictionary matching". *Proc. of the 32nd IEEE Annual Symp. On Foundation of Computer Science*, 1991, 760-766.

[3] A. Amir, M. Farach, R.M. Idury, J.A. La Poutr'e, and A.A. Schaffer. "Improved Dynamic Dictionary-Matching". *In Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*. 1993, 392-401.

[4] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré , and A. A. Schäffex. "Improved dynamic dictionary matching". *Information and Computation, 199(2)*. 1995, 258-282.

[5] A, Amir, M. Farach, and Y. Matias. "Efficient randomized dictionary matching algorithms". *In CPM: 3rd Symposium on Combinatorial Pattern Matching*, 1992.

[6] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Manuscript*. 1991.

[7] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Sciences*. 1993.

[8] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. "Dynamic dictionary matching". *Journal of Computer and System Science, 49(2)*. 1994, 208-222.

[9] A. Amir, M. Farach, R. M. Idury, J. A. La Poutre and A. A. Schäffex. "Improved Dynamic Dictionary Matching". *Information and computation*. 119, 1995, 258-282.

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

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

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

[13] C. Monz and M. de Rijke. (11, 02, 2002). *Inverted Index Construction*. [Online]. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf.

[14] D. D. Sleator and R. E. Tarjan. "A data structure for dynamic trees". *Journal of Computer and System Sciences 26(3)*. 1983, 362-391.

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

[16] E.M. McCreight. "A space-economical suffix tree construction algorithm". Journal of Algorithms, 1976, 23(2):262-272.

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

[18] O. R. Zaïane. "CMPUT 391: Inverted Index for Information Retrieval". *University of Alberta*. 2001.

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

[20] S. Melnik, Sriram Raghavan, Beverly Yang and Hector Garcia-Molina. "Building a Distributed Full-Text Index for the Web". ACM Transactions on Information Systems, Vol. 19, No. 3, 2001, 217-241.

[21] T.W. Lam, K.K. To. (03, 11, 2005). "The Dynamic Dictionary Matching Problem Revisited". [Online]. Available : http://citeseer.ist.psu.edu/413873.html.

[22] H-L. Chan, W-K. Hon, T-W. Lam, and K. Sadakane. "Dynamic dictionary matching and compressed suffix trees". SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms. 2005, 13-22.

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

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

[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, 2006, 675-680.

[26] F. C. Botelho. "Near-Optimal Space Perfect Hashing Algorithms". The thesis of PhD. in Computer Science of the Federal University of Minas Gerais, 2008.

[27] R. Pagh. (11, 08, 2009). "Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions". [Online]. Available: www.it.-c.dk/people/pagh/papers/hash.pdf.

[28] Wikipedia (10,07,2009). "Hash function". [Online]. Available: en.wikipedia.org/wiki/Hash_fuction.

ISBN: 978-988-17012-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)