Abstract—This paper proposes a new solution to string matching problem. This solution constructs an inverted list representing a string pattern to be searched for. It then uses a new algorithm to process an input string in a single pass. The preprocessing phase takes 1) time complexity $O(m)$ 2) space complexity $O(1)$ where $m$ is the length of pattern. The searching phase time complexity takes 1) $O(m+\alpha)$ in average case 2) $O(n/m)$ in the best case and 3) $O(n)$ in the worst case, where $\alpha$ is the number of comparing leading to mismatch and $n$ is the length of input text.

Keywords—String matching, inverted list, inverted index, pattern, algorithm.

I. INTRODUCTION

The problem of string matching is to locate all occurrences of a given pattern string $p$ within a given text string $T$. [7] and [8] provide a good review on solutions to the problem. Existing fast solutions, such as [1]-[4], [9], [10], [15], [16], [18], [19], [22] put a pattern $p$ in an automaton for efficiently processing an input text. Among them, the best one takes $O(m)$ time complexity and $O(1)$ space complexity [4] of preprocessing phase; the searching phase time complexity takes $O(n)$ in average case and $O(n/(m+1))$ in the best case [10]. Another solution to string matching is to use a hashing function as proposed by [23]. This solution takes $O(m)$ time complexity and $O(1)$ space complexity. However, it takes $O(mnx)$ time complexity in searching phase. To improve the time complexity of the hashing idea, this article proposes to use an inverted list, a new data structure derived from an inverted index [5], [20], [21] used in information retrieval field, as a data structure for storing a pattern string.

In the following sections, we describe the details of the new solution. Section II gives some basic definitions and details on preprocessing phase. Section III describes the searching phase algorithm and shows its time complexity. The conclusion is in section IV.

II. BASIC DEFINITIONS AND PREPROCESSING PHASE

Let $p$ be a string $c_1c_2\ldots c_m$ within $\sum$ where $\sum$ is all characters over the pattern $p$.

A. Basic Definitions

Definition 1 The keyword $\omega$ of pattern $p$ contains $w_{a_1,0}w_{b_2,0}w_{c_3,0}\ldots w_{u_{m-1},0}$; where $w_{a_{k-1},0}$ or $w_{a_{k+1},0}$ is $c_k$ and $k = 1, 2, \ldots, m$; 1 indicates a status of last character in $p$ and 0 otherwise. Therefore,

$$\omega = w_{a_1,0}w_{b_2,0}w_{c_3,0}\ldots w_{u_{m-1}}.$$ 

Example 1 Given pattern $p=aabcz$, we have $w_{a_1,0}=a$, $w_{b_2,0}=a$, $w_{c_3,0}=b$ , $w_{d_4,0}=c$ and $w_{e_5,0}=z$. Therefore,

$$\omega = a_{1,0}a_{2,0}b_{3,0}c_{4,0}z_{5,1}.$$ 

Definition 2 Given $\omega = w_{a_1,0}w_{b_2,0}w_{c_3,0}\ldots w_{u_{m-1}}$ of $p$. The inverted list $L$ of $\omega$, denoted by $\omega L$, is a set defined as

$$\omega L = \{a:<1:0>,a:<2:0>,b:<3:0>,c:<4:0>,z:<5:1>\}.$$ 

Example 2 From example 1, the inverted list $L$ of $\omega$ is $L_\omega = \{a:<1:0>,a:<2:0>,b:<3:0>,c:<4:0>,z:<5:1>\}$. 

Definition 3 An $I_{A_0}/I_{A_1}$ of $w_i$ is a set containing elements $<i:0>$ or $<i:1>$ where $i$ is the position of $A$.

Definition 4 An inverted-list table $\tau$ is a set of ordered pair $(w_i, I_{A_0}/I_{A_1})$.

Example 3 From example 1 and 2, the table $\tau$ of pattern $p=aabcz$ is shown below.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$&lt;1:0&gt;,&lt;2:0&gt;$</td>
</tr>
<tr>
<td>$b$</td>
<td>$&lt;3:0&gt;$</td>
</tr>
<tr>
<td>$c$</td>
<td>$&lt;4:0&gt;$</td>
</tr>
<tr>
<td>$z$</td>
<td>$&lt;5:1&gt;$</td>
</tr>
</tbody>
</table>

Authors are with Department of Mathematics and Computer Science, Faculty of Science, King Mongkut’s Institute of Technology at Ladkrabang, Thailand (e-mail: chouvalit@hotmail.com, kbleera@kmitl.ac.th).
Theorem 1 The accessing \( I_{\lambda_0} \) or \( I_{\lambda_i} \) in the table \( \tau \) takes \( O(1) \) times.

Proof Let \( f(x) \) be a hashing function, \( w_{\lambda_0} \) be a key for access \( I_{\lambda_0} \) and \( w_{\lambda_i} \) be the key for access \( I_{\lambda_i} \). Suppose the table \( \tau \) implemented by the hash table \([11], [17], [18], [19]\), accessing to \( I_{\lambda_0} \) with \( f(w_{\lambda_0}) \) or accessing \( I_{\lambda_i} \) with \( f(w_{\lambda_i}) \) take \( O(1) \) times. #

B. Preprocessing Phase

The first step of this algorithm is to create the inverted list table for \( \sum \). The next step reads each character from pattern and updates the inverted list. The detailed algorithm is shown in Fig. 1.

![Fig. 1 Preprocessing algorithm](image)

This phase take \( O(m) \) times, we prove in theorem 2. The space complexity uses \( O(1) \), because we use a fixed-sized hash table defined by definition 4.

Theorem 2 Preprocessing phase of string matching using an inverted list take \( O(m) \).

Proof Suppose \( p=c_{1},c_{2},c_{3},...,c_{m} \). Step A creates the table and Step B initializes variables. They take \( O(1) \). Step C repeats \( m \) round taking \( O(m) \) times. Step D is \( O(1) \) by theorem 1. Step E take \( O(1) \) as in step B. Therefore, preprocessing phase take \( O(m) \) #

III. SEARCHING PHASE

The searching phase employs the navigator variable \( N \) as current comparison position; \( \text{SHIFT} \) as the shift window; \( \text{pos} \) as the required position for current matching; “life” as the control loop variable used in each of search window; and SET1, SET2, and SETE as the temporary variables used in matching.

The first character of each search window is compared with the last character in the text followed by taking the inverted list to SETE for reference. If SETE is not empty and matches with the last character, we scan to compare the text from the first to the last character, or if SETE does not contain the last character, we consider the farthest character matching the SETE and scan for matching again. Every comparison takes the inverted list to the temporary variable SET1 or SET2, meanwhile taking the inverted list to these variables. We must also operate SET1 and SET2. The purpose of the operation is to search for the sequence of pattern and check the matching.

After finishing each search window, we move the window to \( \text{SHIFT} \) and begin to search again. This algorithm can move the \( \text{SHIFT} \) beyond the normal shift position. We illustrate in Fig. 2.

![Fig. 2 Searching algorithm](image)

**Lemma 1** Let SET be the sub table with keys \( w_{\lambda_0} \) and \( w_{\lambda_i} \) for accessing \( I_{\lambda_0} \) and \( I_{\lambda_i} \), respectively. The access of \( I_{\lambda_0} \) and \( I_{\lambda_i} \) in SET using \( f(w_{\lambda_0}) \) or \( f(w_{\lambda_i}) \) function takes \( O(1) \) times.

Proof Let SET is the hash table as the theorem 1 that has a key \( w_{\lambda_0} \) and \( w_{\lambda_i} \) for accessing. Therefore, it employs \( f(w_{\lambda_0}) \) and \( f(w_{\lambda_i}) \) for accessing \( I_{\lambda_0} \) and \( I_{\lambda_i} \), respectively. Therefore, it take \( O(1) \) according to theorem 1 #.

**Lemma 2** To get an entry matching a character at \( \text{text}[N] \) from \( \tau \) into SET variables takes \( O(1) \) times.

Proof Let \( \text{text}[N] \) be a character from the text \( T \) which can be represented in terms of key \( w_{\lambda_{pos,0}} \) or \( w_{\lambda_{pos,1}} \). Hence, the access \( I_{\lambda_{pos,0}} \) and \( I_{\lambda_{pos,1}} \) in a table \( \tau \) takes \( O(1) \) following theorem 1 and takes \( I_{\lambda_{pos,0}} \) and \( I_{\lambda_{pos,1}} \) into SET variables are \( O(1) \) following lemma 1 #.

**Definition 5** An operation is a searching for a continuity of \( I_{\lambda_{r,0}} \) and/or \( I_{\lambda_{r,1}} \) in SET1 to \( I_{\lambda_{r,0}} \) and/or \( I_{\lambda_{r,1}} \),
in SET2 considering position of $\varepsilon 2$ prior to next to $\varepsilon 1$. The result is $I_{a_{3}a_{2}a_{1}}$ and/or $I_{a_{2}a_{1}}$.

**Example 4** We show the operation example of SET1 and SET2; where SET1=$\{<2:0>, <1:0>, <3:0>, \}$ and SET2=$\{<1:0>, <3:0>, \}$. The continuity of position 2 to position 3 is $<3:0>$ next to $<2:0>$ and $<1:0>$ prior to $<2:0>$. The result is SET1 = $\{<1:0>, <3:0>, \}$.

**Lemma 3** The operation of SET1, SET2 and SETE takes O(1) times.

**Proof** Let SET1, SET2 and SETE be the SET in lemma 1 such that, SET1 contains $I_{a_{1}a_{0},0}$ and/or $I_{a_{1}a_{0}}$, SET2 contains $I_{a_{2}a_{1},0}$ and/or $I_{a_{2}a_{1}}$, and SETE contains $I_{a_{3}a_{2},0}$ and/or $I_{a_{3}a_{2}}$, Accessing $I_{a_{3}a_{2},0}, I_{a_{3}a_{2}}, I_{a_{3}a_{2},0}, I_{a_{3}a_{2}}$ and/or $I_{a_{3}a_{2}}$, for comparing the operation in the definition 5 take O(1) following lemma 1 #

**Example 5** Given the pattern $p=aabcz$, the text $T=aacbzczefgaabczefgabcdg$, and the inverted list table $r$ of $p=aabcz$ as shown in Table I. The search for $p$ within $T$ according to the algorithm in figure 2 is illustrated as follows.

1. Initialize variables by step A.
   N=5, SET1={$\{1\}$}, SET2={$\{\}$}, pos=1, SETE={$\}$

2. Perform comparison by step C. N=5, SET1={$\{1\}$}, SET2={$\{\}$}, pos=1, SETE={$\{<5:1>\}$}

3. Skip to the farthest from SETE position and uses step D2 and F1.
   N=1, SET1={$<1:0>$}, SET2={$\{\}$}, pos=1, SETE={$\{<5:1>\}$}

4. Skip to the next position by step F2.
   N=2, SET1={$<1:0>$}, SET2={$<1:0>$}, pos=2, SETE={$\{<5:1>\}$}

5. Skip to the next position by step F2.
   N=3, SET1={$<1:0>$}, SET2={$<3:0>$}, pos=3, SETE={$\{<5:1>\}$}

6. Skip to the next position by step F2. N=4, SET1={$<1:0>$}, SET2={$<4:0>$}, pos=4, SETE={$\{<5:1>\}$}

7. Skip to the next position and not access but mask by step F3.
   N=5, SET1={$<1:0>$}, SET2={$<4:0>$}, pos=5, SETE={$\{<5:1>\}$}

8. Initialize variables and go to step B.
   N=10, SET1={$\{\}$}, SET2={$\{\}$}, pos=1, SETE={$\}$

9. Perform comparison by step C. N=10, SET1={$\{\}$}, SET2={$\{\}$}, pos=1, SETE={$\{<5:1>\}$}

10. Start search at N and compare with the first character in pattern by step F1. N=9, SET1={$<1:0>$}, SET2={$\{\}$}, pos=1, SETE={$\{<5:1>\}$}

11. This step does not access but masks by step F3. N=10, SET1={$<1:0>$}, SET2={$\{\}$}, pos=2, SETE={$\{<5:1>\}$}

12. Skip to next position by step F2. N=11, SET1={$<1:0>$}, SET2={$<3:0>$}, pos=3, SETE={$\{<5:1>\}$}
3 The searching phase of string matching using inverted list takes \( O(m + \alpha) \) times in average case. The best case takes \( O(n/m) \) times and takes \( O(n) \) times in worst case.

Theorem 3 The searching phase of string matching using inverted list takes \( O(m + \alpha) \) times in average case. The best case takes \( O(n/m) \) times and takes \( O(n) \) times in worst case.

Proof Let \( |n| \) be the length of \( T \) such \( T = t_1t_2t_3 \ldots t_n \), \( m \) be the length of pattern \( p \), and \( \alpha \) be the number of comparisons leading to mismatch and also included the time of mismatch. Step A uses \( O(1) \) because it initializes variables, Step B to Step H repeat \( m + \alpha \) rounds which uses \( O(m + \alpha) \) time, Step C,D,D2,F1,F2,G and G1 use \( O(1) \) because it access the hash table following lemma 1, Step D1,G2,H use \( O(1) \) to initialize variables, Step F3 uses \( O(1) \) following lemma 3, Step E repeats \( \alpha \) rounds and takes \( O(\alpha) \) meanwhile each of operation takes \( O(1) \) following lemma 1.

Therefore the time complexity of this phase is \( O(m + \alpha) \) times.

The best case of this algorithm happens in the case of mismatching between the last character of search window and the pattern. Hence, the algorithm only handles Step B and Step D1. So the number of comparisons take \( n/m \) rounds lead to \( O(n/m) \) times.

The worst case of this algorithm happens in the case which the text contains the same characters and matches all of search windows. In this case, the algorithm does not go through step G2 and H. So it could not shift beyond the normal SHIFT. Therefore, it takes Step B in \( n \) rounds and leads to \( O(n) \) time.

IV. CONCLUSION

This paper presents a new string matching algorithm adopting an inverted index as an inverted list data structure for storing a target pattern. Storing a pattern into this data structure takes \( O(m) \) time complexity and \( O(1) \) space complexity where \( m \) is the length of pattern. The paper developed a new string matching algorithm with time complexity 1) \( O(m + \alpha) \) in average case 2) \( O(n/m) \) in the best case and 3) \( O(n) \) in the worst case, where \( \alpha \) is the number of comparisons leading to mismatch and \( n \) is the length of input text.

REFERENCES