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 $w_{a_1}$, $w_{b_2}$, $w_{c_3}$, ... $w_{m-1}$; where $w_{a_k}$ or $w_{b_k}$ is $c_k$ and $k = 1, 2, \ldots, m$; 1 indicates a status of last character in $p$ and 0 otherwise. Therefore,

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

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

$$\omega = a1,0a2,0b3,0c4,0z5,1.$$ 

Definition 2 Given $\omega = w_{a_1}w_{b_2}w_{c_3}\ldots w_{m-1}$ of $p$. The inverted list $L$ of $\omega$, denoted by $\omega L$, is a set defined as $\omega L = \{a:1:0, b:2:0, c:3:0, z:5:1\}$. 

Example 2 From example 1, the inverted list $L$ of $\omega$ is

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

Definition 3 An $I_{\lambda_0}/I_{\lambda_1}$ of $\lambda$ is a set containing elements $<i:0>$ or $<i:1>$ where $i$ is the position of $\lambda$.

Definition 4 An inverted-list table $\tau$ is a set of ordered pair $(w_{\lambda}, I_{\lambda_0}/I_{\lambda_1})$.

Example 3 From example 1 and 2, the table $\tau$ of pattern $p=aabcz$ is shown below.
Theorem 1: The accessing $I_{a_k}$ or $I_{b_k}$ in the table $\tau$ takes $O(1)$ times.

Proof: Let $f(x)$ be a hashing function, $w_{a_k}$ be a key for access $I_{a_k}$ and $w_{b_k}$ be the key for access $I_{b_k}$. Suppose the table $\tau$ implemented by the hash table $[11],[17],[18],[19]$, accessing to $I_{a_k}$ with $f(w_{a_k})$ or accessing $I_{b_k}$ with $f(w_{b_k})$ 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.

```
Inverted-List-Table(p=c_1,c_2,...,c_m)
Step A Create table for all alphabet in $\sum$
Step B j=1
Step C while (j<=m)
Step D Create inverted list and add to table at alphabet char($c_j$)
Step E j=j+1
```

Fig. 1 Preprocessing algorithm

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

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

Proof: Suppose $p=c_1,c_2,...,c_m$. 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; SHIFT as the shift window; 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 SHIFT and begin to search again. This algorithm can move the SHIFT beyond the normal shift position. We illustrate in Fig. 2.

```
Inverted-List-Matching(p=c_1,c_2,...,c_m, T=t_1,t_2,...,t_n)
Preprocessing
Create Inverted-List-Table(p)
Searching
Step A N=m, SHIFT=2m, pos=1, SET1=\emptyset, SET2=\emptyset, SETE=\emptyset, life=1
Step B While (SHIFT < n) and (N<=n) Do
Step C Store all member of row(text[N]) to SETE
Step D If SETE=\emptyset
Step D1 N=SHIFT, SHIFT=SHIFT+m
Else
Step D2 Analyze SETE for searching the farthest and set it to N, pos=1, life=1
Step E While SET1 != \emptyset and life=1
Step F If pos=1
Step F1 Store inverted list in of row(text[N]) where inverted list position = pos to SET1, pos=pos+1
Else
Step F2 Store inverted list row(text[N]) where inverted list position = pos to SET2 if pos=position of SETE
Step F3 SET1=SET1 operate SET2 or SETE=SET1 Mask SETE if N=position of SETE and mark success if terminate status = 1 and remove that inverted list had already matched and N=N+1
Step G If SET1 != \emptyset
Step G1 Set pos=maximum inverted list position+1 in SET1 if N=position of SETE and SHIFT <= SHIFT+m or others case pos=life+1
Else
Step G2 life=0
Step H N=SHIFT, SHIFT=SHIFT+m, pos=1, SET1=\emptyset, SET2=\emptyset, SETE=\emptyset, life=1
```

Fig. 2 Searching algorithm

Lemma 1: Let SET be the sub table with keys $w_{a_k}$ and $w_{b_k}$ for accessing $I_{a_k}$ and $I_{b_k}$, respectively. The access of $I_{a_k}$ and $I_{b_k}$ in SET using $f(w_{a_k})$ or $f(w_{b_k})$ takes $O(1)$ times.

Proof: Let SET is the hash table as the theorem 1 that has a key $w_{a_k}$ and $w_{b_k}$ for accessing. Therefore, it employs $f(w_{a_k})$ and $f(w_{b_k})$ for accessing $I_{a_k}$ and $I_{b_k}$, respectively. Therefore, it takes $O(1)$ according to theorem 1.

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

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

Definition 5: An operation is a searching for a continuity of $I_{a_{pos,0}}$ and/or $I_{b_{pos,1}}$ in SET1 to $I_{a_{pos,0}}$ and/or $I_{b_{pos,1}}$
Example 4 We give the operation example of SET1 and SET2, where SET1=\{<2:0>\} 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 \(<2:0>\) and \(<3:0>\). The result is SET1 = \(<1:0>,<2:0>\).

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 \(I_{<1:0>}\) and/or \(I_{<2:0>}\), SET2 contains \(I_{<2:0>}\) and/or \(I_{<3:0>}\) and SETE contains \(I_{<3:0>}\) and/or \(I_{<4:0>}\). Accessing \(I_{<1:0>}, I_{<2:0>}, I_{<2:0>}, I_{<3:0>}, I_{<3:0>}\) and/or \(I_{<3:0>}\) for comparing the operation in the definition 5 take O(1) following lemma 1.

Example 5 Given the pattern \(p=aabcz\), the text \(T=aabczaabcczegbabcdg\), and the inverted list table \(r\) of \(p=aabcz\) as shown in Table 1. 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=\{\}, SET2=\{\}, pos=1, SETE=\{\}

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

3. Skip to the farthest from SETE position and use step D2 and F1. N=1, SET1=\{\<1:0>,\<2:0>\}, SET2=\{\}, pos=1, SETE=\{\<5:1>\}

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

5. Skip to the next position by step F2. N=3, SET1=\{\<1:0>,\<2:0>\}, SET2=\{\<3:0>\}, pos=3, SETE=\{\<5:1>\}

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

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

12. Skip to next position by step F2. N=11, SET1=\{\<1:0>,\<2:0>\}, SET2=\{\<3:0>\}, pos=3, SETE=\{\<1:0>,\<2:0>\}

Please note that the text contains a table and a figure, which are not transcribed here.
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 the worst case takes \( O(n) \) times.

Proof Let \( |n| \) be the length of \( T\) such that \( t_1, t_2, \ldots, t_n \). \( m \) 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, \( B\) to \( H\) repeat \( m + \alpha \) rounds which uses \( O(m + \alpha) \) time, Step C, \( D\), \( D_1\), \( F_2\), \( G\) and \( H\) use \( O(1) \) to initialize variables, \( F\) and \( F_3\) uses \( O(1) \) following lemma 3, Step E repeats \( \alpha \) rounds and takes \( O(\alpha) \) while each of operation takes \( O(1) \) following lemma 1.

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

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 \( D\). 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 \( G_2\) and \( H\). So it could not shift beyond the normal \( SHIFT\). Therefore, it takes \( Step B \) \( n \times \) 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