Inverted Lists String Matching Algorithms

Chouvalit Khancome and Veera Boonjing

Abstract—**This research article presents two algorithms of string pattern matching. These algorithms employ a new data structure called inverted lists structure which is inherited from the inverted index to accommodate a string pattern to be searched. The first solution scans the given text in a single pass for all occurrences of string pattern. The second solution, which improves the first one, takes the comparison times equal to the length of pattern plus the numbers of comparison that lead to be mismatched. For experimental results, these algorithms are efficient in the case of small alphabet sizes.**

Index Terms—**string pattern matching, inverted lists (IVL), inverted index, string algorithm.**

I. Introduction

String pattern matching problem is to locate all occurrences of a string pattern p=c1c2c3…cm in the given text T=t1t2t3…tn over a finite alphabet Σ. Basically, this problem can be solved in two phases; preprocessing and searching. Preprocessing phase requires the efficient data structure to accommodate all characters of p to be searched for. Meanwhile the searching phase compares the text T with the structure of p. Methodology of search is separated to the prefix search, the suffix search, and the factor search, which can be found in [8].

The existing solutions [1]-[4], [9], [10], [15], [16], [18], [19], [22] generate the pattern p to the suitable data structure such as the automata, the shift table, or the bit parallel to decrease the searching time. Among them, the best one [4] takes O(m) time complexity and O(1) space complexity to process the pattern to its structure. The searching complexity of [10] takes O(n) in average case, and O(n/(m+1)) in the best case. In sources [7] and [8] provided a good review on solutions to the problem.

This paper presents two algorithms of string pattern matching; the prefix search and the suffix factor search. These ways guarantee the worst case in O(n) time. We derived the principle of inverted index in [5], [20], and [21] to create the inverted lists to accommodate the pattern p. This structure stores the position of each character of p to the table. The new algorithms process the input pattern in a single pass taking O(m) time and O(m+|Σ|) space where m is the length of pattern, and |Σ| is the size of alphabets. For searching phase, the prefix solution takes O(n) time, and the suffix factor approach takes 1) O(m+α) in average case, 2) O(n/m) in the best case, and 3) O(n) in the worst case; where α is the number of comparisons that lead to be mismatched, and n is the length of the given text. In the experimental result, these algorithms are efficient in the case of small alphabet sizes especially the alphabet size of 2 and 4.

The rests of this paper are organized as follows. The next section describes the basic definitions and the preprocessing phase. The section 3 illustrates the detail of searching algorithms. The section 4 shows the experimental results. The section 5 is the discussion of experimental results and the section 6 is the conclusion and the future works.

Manuscript received September 7, 2009. This work was supported in part by the Software System Engineering Laboratory, Department of Mathematics and Computer Science, Faculty of Science, King Monkut's Institute of Technology at Ladkrabang(KMITL).

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

Veera Boojing 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 inMathematics, PERDO, Bangkok, Thailand 10400, (e-mail: kbveera@kmitl.ac.th).

II. Basic Definitions and Preprocessing Phase

This section shows the basic definitions of what the inverted lists are, and they are also examined by an example. Furthermore, this section gives the preprocessing phase to construct the inverted lists structure and to generate all inverted lists to the table.

A. Basic Definitions

Definition 1 Let $p=c_1c_2c_3...c_m$ be the string pattern, and c_k is a character occurred in p at position k^{th} where $k=1,2,3,...,m$. The inverted list (i.e., IVL) 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 lists of *p* are *a:<1:0>*, *a:<2:0>*, *b:<3:0>*, *c:<4:0>*, and *z:<5:1>*.

Definition 2 The hashing table τ which consists of all characters of $\sum$ and the corresponding inverted lists of *p* is called the inverted lists table.

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

TABLE I. The inverted list table τ

$\sum$	I_{k_0} / I_{k_1}	(i.e., the inverted lists)
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 $p=c_1c_2c_3...c_m$, and they are stored in the hash table τ where $k=1,2,3,...,m$. To access I_{k_0} or I_{k_1} takes O(*1*) time.

Proof This Lemma is proved by each inverted list I_{k_0} or I_{k_1} can be retrieved from the table τ in O(*1*) time. Let *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 $\sum$. The table τ is implemented by applying the hash table in [6], [11], [12], and [13]. 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*) time by the hashing properties. □

B. Preprocessing Phase

This phase creates all inverted lists of p into the table τ. This mechanism reads the character one by one to generate the inverted lists, and they are put to the table τ. We show the details of algorithm as below.

Algorithm 1:Inverted-List Table(*$p=c_1,c_2,c_3,…c_m$*)

```
Step A  Create table $\tau$ for all alphabet in $\sum$
Step B    j←1
Step C  While (j<=m)  Do
Step D      Create the inverted list of $c_j$ → $\tau$ at char($c_j$)
Step E      j←j+1
        End of While
```

This algorithm takes O(*m*) time and O(*m*+|∑|) space shown as Theorem 1.

Theorem 1 All characters of *p* are generated to the inverted lists and added into τ take O(*m*) time, and the table τ uses O(*m*+|∑|) space.

Proof The Step A creates the table, and the Step B initializes the variables in O(*1*). The Step C needs to repeat from c_1 to c_n which takes *m* rounds, and it also gets O(*m*) time. The Step D takes O(*1*) by Lemma 1 while the Step E takes O(*1*) as the Step B. Therefore, overall of the preprocessing time is O(*m*) time. □

For space complexity, the size of τ takes O(|∑|) space for all characters. Each inverted list of ck of p=c1c2c3…cm uses one space for each inverted list. Thus, for k=1 to k=m take m space. Hence, the table of τ required O(m+|∑|) space for accommodating the pattern. □

III. Searching Phase

The searching phase employs several variables; the variable N such a current comparison position; SHIFT is the shift position for the next window search; pos is the required position for the current matching; Life is used to control the loop; SET1, SET2, and SETE are the temporary variables which are used for storing the individual row of table τ.

First and foremost, the OPERATE function is used for the continuity and the matching inspection. The details of function are shown as Algorithm 2.

Algorithm 2: OPERATE (SET1, SET2, N, pos)

```
Step A  Report the successful matching at N if IVL in SET2
        contains <pos+1:1>
Step B  Add every IVL in SET2 that continues from SET1 or
        <1:0> to TEMP and returns TEMP
(i.e., the continuity is the IVL positions in SET2 equal to the
positions in SET1 plus 1.)
```

According to the algorithm above, the OPERATE is used for two solutions; the prefix search approach and the suffix factor search approach for analyzing the continuity and the matching position.

A. Prefix Search Approach

This approach compares the string pattern with the text one by one character from the left to the right. If the comparison is successful, we take the matched inverted lists to SET1 or the next matched inverted lists to the SET2. Afterwards, the algorithm 2 is invoked to operate SET1 and SET2. The Algorithm 3 examines this method in the Step D2.

Algorithm 3: IVL-Prefix-Search(IVL-Table(*p*), *$T=t_1t_2…t_n$*)

```
Step A   N=1 pos=1, SET1=SET2=null
Step B   SET1←(IVL(text[N])=pos) from $\tau$, and N=2
Step C   While (N<= n)  Do
Step D     If SET1 !=null then
Step D1      pos←pos+1 and SET2←(IVL(text[N])=pos or 1)
Step D2      SET1←OPERATE (SET1, SET2, N, pos)
Step D3    Else  pos←1 and  SET1←(IVL(text[N])=pos)
Step E      N←N+1
         End of While
```

Theorem 2 The prefix searching pattern $p=c_1c_2c_3...c_m$ in the text $T=t_1t_2t_3...t_n$ takes O(*n*) time where *n* is the length of *T*.

Proof The hypothesis is for every character in *T* to be scanned. The Step A and the Step B take only O(*1*) time. The Step C will be run from 2 to *n*, and it can be reached the hypothesis as well. Meanwhile the accessing time of Step D, which gets the inverted lists from τ, takes O(*1*) by Lemma 1. All operations are run from t_1 to t_n time, and they also take *n* time. Therefore, the overall time complexity is O(*n*) time. □

B. Suffix Factor Search Approach

This approach is divided to three steps. The first one compares at the last character of pattern that we call the suffix factor. The second one scans the text and compares with the necessary character to be matched from the first character of window search to the suffix factor position. The third one scans the character in the text that beyond the window search if it can. The search employs the variable Life to invoke the scanning method.

For Life=0, the suffix factor is invoked to compare the text, and the inverted lists are taken to SETE. After then, if SETE does not empty and the character in the text is matched then the variable Life=1. If Life=1, N is indicated to the farthest character to be matched and compares the text of N with the inverted lists from N to the position of SETE. In contrast, if SETE does not contain the last character, N is specified to the next farthest character to be matched from the SETE. The text is scanned until the variable N equal the position of SETE. Every comparisons takes the inverted lists to the SET1 or the SET2, and the OPERATE is invoked. After scanning SETE, if it can scan the beyond of window search, the Life=2 and scan the text such a prefix approach.

The figure 1 illustrates this idea, the algorithm 4 presents this approach, and the example 4 illustrates the searching example of this approach.

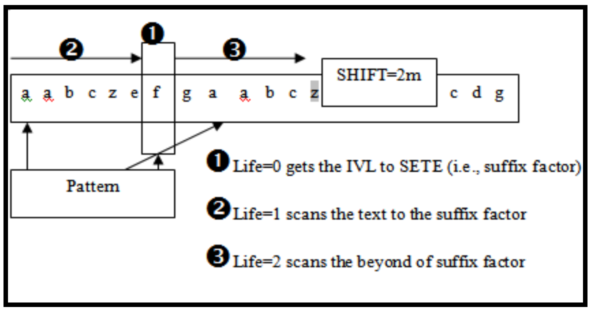
❷
❶
❸
SHIFT=2m
a a b c z e f g a a b c z c d g
Pattern
❶ Life=0 gets the IVL to SETE (i.e., suffix factor)
❷ Life=1 scans the text to the suffix factor
❸ Life=2 scans the beyond of suffix factor

Figure 1. The searching idea.

Algorithm 4: IVL-Factor-Search(IVL-Table(p), $T=t_1t_2\ldots t_n$)

```
Step A   N=m ,SHIFT=2m, pos=1, SET1=SET2=SETE=null,
Step B   While (SHIFT <= n+m) and (N<=n) Do
            /* suffix factor start ❶ */
Step B1      SETE←(IVL(text[N])), Life =0
Step B2      If SETE!=null then N← FARTHES(SETE)
             Else set the next window by SHIFT
          /* scans from farthest of SETE ❷ */
Step C   While (Life=1) and (N< memSETE) and (N<=n-m) Do
Step C1      Scan to compare by IVL-Prefix-Search from the
             farthest position from SETE to SETE-1
Step C2      Report the occurrence if it can scan to position
             of SETE-1 and stop if mismatched
         End of While // ❷ (Step C)
        /* connector to overlapping and prefix search checking*/
Step D   If (Life=1) or (Life=0) and (SET1=null) Then
Step D1      Set the next window by SHIFT
Step D2   Else pos← FARTHES(SET1) and SHIFT++, Life=2
         /* prefix searching for the beyond of SETE position ❸ */
Step E   While (Life=2) and (N<n)
Step E1     Scan to compare by IVL-Prefix-Search and
            SHIFT++ if matched or Life =0 if mismatched
Step E2   Report the occurrence if pos=m
         End of While //❸ (Step D)
Step B3 Set the next window by SHIFT and SHIFT←SHIFT+m
        End of While // ❶ (Step B)
```

Theorem 3 The suffix factor search takes 1) $O(m+\alpha)$ time in average case, 2) O(n/m) time in the best case, and 3) O(n) time in worst case where m is the length of pattern input, n is the length of T, and α is the number of comparisons that lead to be mismatched.

Proof In average case, the Step A uses O(1) because it initializes variables. The Step B1 and all of Step C repeats at most m-1 round if the characters in pattern are matched. If the characters in the pattern are matched in some character or mismatched, the comparison time is at most α rounds. Thus the comparison time of the Step B and the Step C take $m+\alpha$ times, and they also take $O(m+\alpha)$ time. Meanwhile all operations of inverted lists take O(1) time by Lemma 1. The other steps compare and skip the variables which take at most $m+\alpha$ rounds as the Step B. Therefore the time complexity is $O(m+\alpha)$.☐

The best case could happen whenever all last character of window search has no the inverted lists to be matched. Hence, the algorithm only handles Step B to Step B2. Therefore, the comparison times take n/m rounds leading to O(n/m). ☐

The worst case could occur if the text T contains the same character and the pattern is matched in all windows of search. In this case, the algorithm needs to scan all characters of T that leads to O(n) time. ☐

Example 3. Searching p=$aabcz$ in the text T=$aabczefgaabczefgabcdg$.

1. Initiate the variables by Step A.
Set N=5, SET1=SET2=SETE=null, SHIFT=10.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

2. Take the inverted lists by Step B1. Set N=5, SET1=null, SET2=null, pos=1, SETE={<5:1>}, and Life =1.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

3. Skip to the farthest from SETE position and uses the Step B2 and C1. Set N=1, SET1={<1:0>,<2:0>}, SET2=null, pos=1, SETE={<5:1>}, and Life =1.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

4. Skip to the next position by Step C1. Set N=2, SET1={<1:0>,<2:0>}, SET2={<1:0>,<2:0>}, pos=2, SETE={<5:1>}, and Life =1.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

SET1← OPERATE(SET1, SET2, N, pos) then SET1={<1:0>,<2:0>}.

5. Skip to the next position by Step C1. Set N=3, SET1={<1:0>, <2:0>}, SET2={<3:0>}, pos=3, SETE={<5:1>} and Life =1.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

SET1← OPERATE(SET1, SET2, N, pos) then SET1={<3:0>}.

6. Skip to the next position by the Step F2. Set N=4, SET1={<1:0>, <2:0>}, SET2={<4:0>}, pos=4, SETE={<5:1>}, and Life =1.

```
a a b c z e f g a a  b  c  z  e  f  g  a  b  c  d  g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z             SHIFT=10
```

SET1←OPERATE(SET1, SET2, N, pos) then SET1={<4:0>} and reports the occurrence at the position of SETE, SET1=null.

7. Use Step D and Step D1 for the next window search.

8. Initiate the variables by Step B3. Set N=10, SET1=null, SET2=null, pos=1, SETE=null and Life =0.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=15

9. Perform the comparison by Step B1. Set N=10, SET1=null, SET2=null, pos=1, SETE={<1:0>,<2:0>}, and Life =0.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=15

This comparison is not match thus SETE is analyzed to look for the farthest and set N=farthest from SETE. Set N=9 and skip to the Step E.

10. Start the search at N and compare with the first character in pattern by Step E1. Set N=9, SET1={<1:0>,<2:0>}, SET2=null, pos=1, SETE={<1:0>,<2:0>}, and Life =1.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=15

11. This step does not access to the structure, and skips to the next position. Set N=10, SET1={<1:0>,<2:0>}, SET2={}, pos=2, SETE={<1:0>,<2:0>}, and Life =1.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=15

SET1← OPERATE(SET1, SET2, N, pos) then SET1={<1:0>,<2:0>} .

12. Skip to the next position. Set N=11, SET1={<1:0>,<2:0>}, SET2={<3:0>}, pos=3, SETE={<1:0>,<2:0>}, and Life =2.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=15

SET1← OPERATE(SET1, SET2, N, pos) then SET1={<3:0>}, and SHIFT←SHIFT+1, SHIFT=16.

13. Skip to the next position. Set N=12, SET1={<3:0>}, SET2={<4:0>}, pos=4, SETE={<1:0>,<2:0>}, and Life =2.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=17

SET1← OPERATE(SET1, SET2, N, pos) then SET1={<4:0>}, and SHIFT←SHIFT+1, SHIFT=17.

14. Skip to the next position. Set N=13, SET1={<4:0>}, SET2={<5:1>}, pos=5, SETE={<1:0>,<2:0>}, and Life =2.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a a b c z SHIFT=18

SET1← OPERATE(SET1, SET2, N, pos) which SET1={<5:1>} and reports the occurrence at the position of SETE, and SHIFT←SHIFT+1, SHIFT=18. Remove the inverted list from SET1 and SET1=null. Stop the window search and skip to Step B3.

15. Initiate the variables and skip to Step B. Set N=18, SET1=null, SET2=null, pos=1, SETE=null, and Life =0.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SHIFT=23 a a b c z

16. Perform the comparison by the Step B1. Set N=18, SET1=null, SET2=null, pos=1, SETE={<3:0>}, and Life =0.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SHIFT=23 a a b c z

This comparison is not match, and the SETE N=farthest position from SETE and N=16.

17. Start the search at N and compare with the first character of pattern. Set N=16, SET1=null, SET2=null, pos=1, SETE={<3:0>}, and Life =1.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SHIFT=23 a a b c z

This comparison is not match then the SET1=null. Stop the search and skip to the Step B3.

18. Initiate the variables and skip to the Step B. N=23, SET1=null, SET2=null, pos=1, SETE=null, and Life =0.

a a b c z e f g a a b c z e f g a b c d g
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
SHIFT=28

SHIFT > n+m and N>n, and the search is finished.

IV. Experimental results

We used the Dell Latitude D500 notebook with Intel Pentium M 1.3 GHz and 512 MB of RAM running the Windows XP Home as the equipment for setting the experiments. The implemented algorithms to applications are by Java with JSE version 1.6.

The data tests were randomized from the text Σ with the sizes of 2, 4, 8, 16 and 32. The $|\Sigma|$=2 were the binary digits; $|\Sigma|$=4 were the capital English letters from 'A' to 'D'; $|\Sigma|$=8 were the capital English letters from 'A' to 'H'; $|\Sigma|$=16 were the capital English letters from 'A' to 'P'; and $|\Sigma|$=32 were the capital English letters from 'A' to 'Z' and the small letters from 'a' to 'f'. The patterns were randomized from Σ by the lengths of 2, 4, 8, 16 and 32 characters. The texts were randomized from each Σ in the size of 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB.

The following figures show the experimental results where BM is the symbol of Boyer-Moor algorithm, BF is the symbol of Brute-Force algorithm, IVLSF is the symbol of inverted lists in the suffix approach algorithm, IVLPF is the symbol of the inverted lists in prefix approach algorithm and KMP is the symbol of Knuth-Morris-Pratt algorithm.

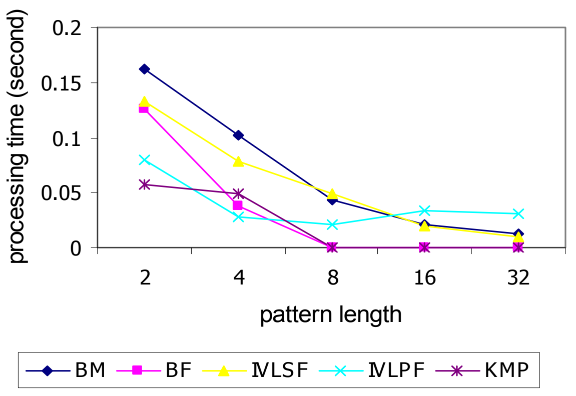

Figure 2. Processing time when |∑|=2 and the text size = 1 KB.

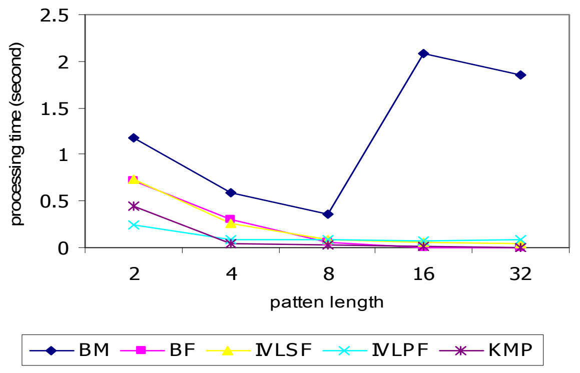

Figure 6. Processing time when |∑|=2 and the text size = 4 KB.

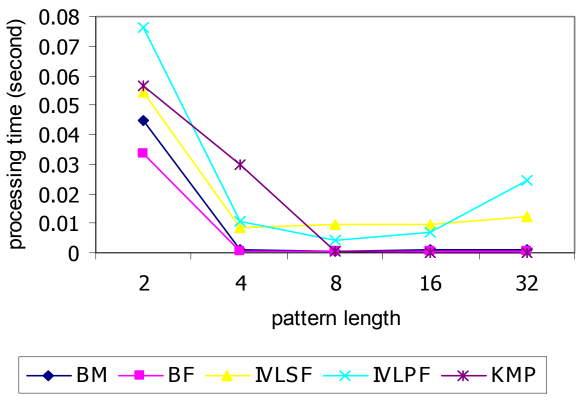

Figure 3. Processing time when |∑|=4 and the text size = 1 KB.

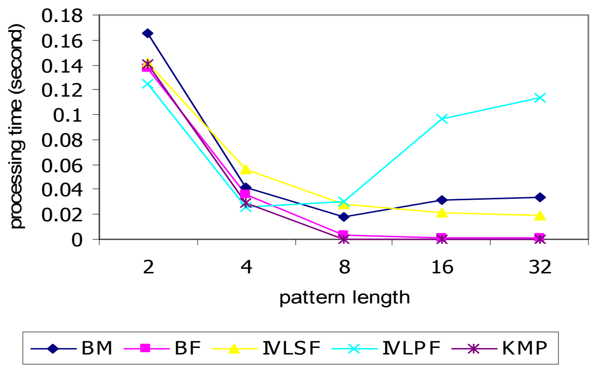

Figure 7. Processing time when |∑|=4 and the text size = 4 KB.

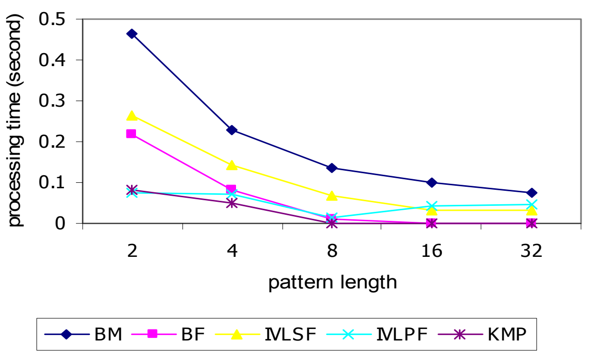

Figure 4. Processing time when |∑|=2 and the text size 2 KB.

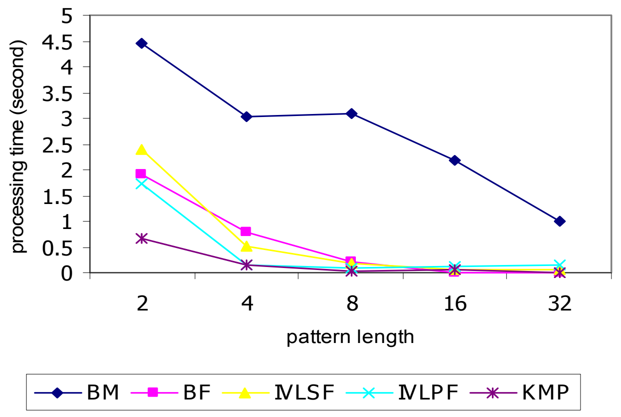

Figure 8. Processing time when |∑|=2 and the text size = 8 KB.

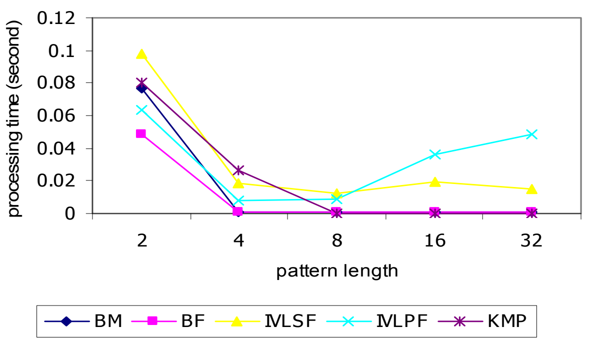

Figure 5. Processing tem when |∑|=4 and the text size = 2 KB.

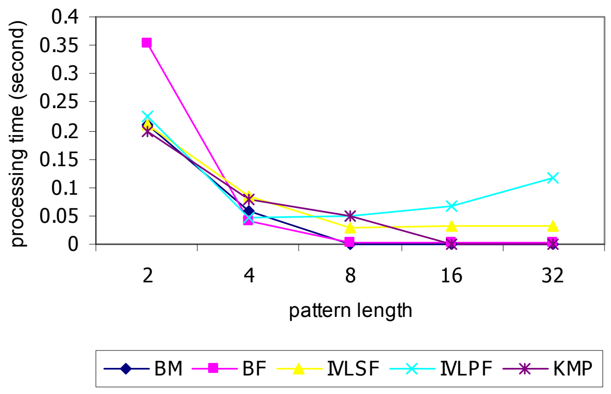

Figure 9. Processing time when |∑|=4 and the text size = 8 KB.

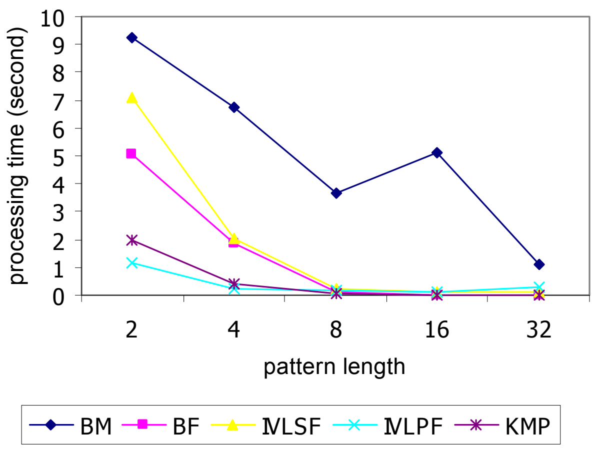

Figure 10. Processing time |∑|=2 and the text size = 16 KB.

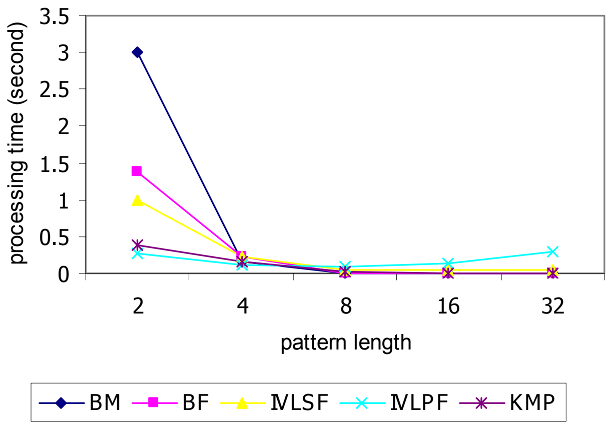

Figure 11. Processing time |∑|=4 and the text size = 16 KB.

V. Discussion

According to the experimental results, the new algorithms are efficient in |∑|=2 and |∑|=4 especially the large texts. These algorithms are faster than the others in the small alphabet sizes because the new algorithms are less compare time than the others. Our algorithms are suitable for the binary because the other algorithms are more worst-case than ours. The bottle neck of our algorithm is that the OPERATE needs to filter the inverted lists too much if a pattern is long and an alphabet size is large.

VI. Conclusion and Future works

This article presented two string pattern matching algorithms: the prefix search and the suffix factor search. Both solutions employ the inverted lists to store the target pattern. These algorithms process the string pattern in O(m) time and O(m+|∑|) space where m is the length of pattern, and |∑| is the size of alphabets. In searching phase, the prefix solution takes O(n) time. The suffix factor approach takes 1) O(m+α) in average case scenario, 2) O(n/m) in the best case, and 3) O(n) in the worst case scenario; where α is the number of comparisons that lead to be mismatched, and n is the length of the given text. In experimental results, the new algorithms are fast in the small alphabet size especially the binary digit cases.

For future works, we are attempting to decrease the character inspection by the shift table, and we are improving the methodology of keeping the inverted lists by storing for one letter per one inverted list.

References

[1] R.S. Boyer and J. S. Moore, “A fast string searching algorithm”, Communications of the ACM. 20, 1997, pp. 762-772.
[2] M. Chrochemore and C. Handcart, “Automata for Matching Patterns”, in Handbook of Formal Languages, Volume 2, Linear Modeling: Background and Application, G. Rozenberg and A.Salomaa ed.,Springer-Verlag,Berlin. 1997, Ch. 9, pp. 399-462.
[3] M. Chrochemore, “Off-line serial exact string searching”, in Pattern Matching Algorithms, A. Apostolico and Z. Galil ed., Oxford University Press. Chapter 1, pp. 1-53.
[4] M. Crochemore, L. Gasieniec, and W. Rytter, “Constant-space string-matching in sublinear average time**”,** Compression and Complexity of Sequences 1997. 1997**,** pp. 230 – 239.
[5] C. Monz and M. de Rijke. (2006, August, 12) Inverted Index Construction. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf.
[6] M. Escardo, (2008, October 15), Complexity considerations for hash tables Available: http://www.cs.bham.ac.uk/~mhe/foundations2/node92.html.
[7] C. Charras and T. Lecroq. (2008, October 10). Handbook of Exact String Matching. Available: www-igm.univ-mlv.fr/~lecroq/string/string.pdf.
[8] G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings. The press Syndicate of The University of Cambridge. 2002. pp. 15-40.
[9] Z. Galil, R. Giancarlo, “On the exact complexity of string matching upper bounds”, SIAM Journal on Computing, 21(3). 1992, pp. 407-437.
[10] H. Kesong, W. Yongcheng and C. Guilin, “Research on A Faster Algorithm for Pattern Matching”, Proceedings of the fifth International workshop on Information retrieval with Asian languages. 2000, pp. 119-124.
[11] wikipedia, (2007, November 15), Hash table. Available: http://en.wikipedia.org /wiki/ Hash_table.
[12] K. Loudon, (2008, November 24), Hash Tables. Available: www.oreilly.com/catalog/ masteralgoc/chapter/ch08.pdf.
[13] V. H. DINH, (2006, November 24), Hash Table. Available: http://libetpan.sourceforge.net /doc/API/API/x161.html.
[14] J. Law, “Book reviews: Review of "Flexible pattern matching in strings: practical on-line algorithms for text and biological sequences by Gonzolo Navarro and Mathieu Raffinot." Cambridge University Press 2002”. ACM SIGSOFT Software Engineering Notes, Volume 28 Issue 2. 2003, pp. 1-36.
[15] G. Navarro, M. Raffinot, “Fast and flexible string matching by combining bit-parallelism and suffix automata”, December 2000 Journal of Experimental Algorithmics (JEA), Volume 5. 2000.
[16] D.E. Knuth, J.R. Morris, and J. H. Pratt, “Fast pattern matching in strings”. SIAM Journal on Computing 6(1). 1977, pp. 323-350.
[17] M. S. Ager, O. Danvy and H. K. Rohde, “Fast partial evaluation of pattern matching in strings”. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 28 Issue 4. 2006, pp. 3-9.
[18] M. S. Ager, O. Danvy and H. K. Rohde, “On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation”. Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation. 2002, pp. 32-46.
[19] J. R. Morris, J. H. Pratt, “A linear pattern-matching algorithm”, Technical Report 40, University of California, Berkeley. 1970.
[20] O. R. Zaïane. (2001, September 15), “CMPUT 391: Inverted Index for Information Retrieval”, University of Alberta. Available: http://www.cs.ualberta.ca/~zaiane/courses /cmput39-03/.
[21] R. B. Yates and B. R. Neto. “Mordern Information Retrieval”, The ACM press. A Division of the Association for Computing Machinery, Inc. 1999, pp. 191-227.
[22] I. Simon, “String matching and automata”, in Results and Trends in Theoetical Computer Science, Graz, Austria, J. Karhumaki, H. Maurer and G. Rozenerg ed., Lecture Notes in Computer Science 814, Springer- Verlag, Berlin. 1994, pp. 386-395.

IACSIT
International Association of Computer Science and Information Technology
WWW.IACSIT.ORG