World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:1, 2007

String Matching using Inverted Lists

Chouvalit Khancome, and Veera Boonjing

***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+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 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(mxn) 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.

Authors are with Department of Mathematics and Computer Science, Faculty of Science, King Monkut's Institute of Technology at Ladkrabang, Thailand (e-mail: chouvalit@hotmail.com, kbveera@kmitl.ac.th).

A. Basic Definitions

Definition 1 The keyword ω of pattern p contains $w_{a_{1,0}}\ w_{b_{2,0}}\ w_{c_{3,0}}\ldots w_{\cdots m,1}$; where $w_{n_{k,0}}$ or $w_{n_{k,1}}$ is c_k and k = 1, 2, …, m; 1 indicates a status of last character in p and 0 otherwise. Therefore,

$$\omega = w_{a_{1,0}}\ w_{b_{2,0}}\ w_{c_{3,0}}\ldots w_{\cdots m,1}.$$

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 $w_{e_{5,1}}$ =z. Therefore,

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

Definition 2 Given $\omega = w_{a_{1,0}}\ w_{b_{2,0}}\ w_{c_{3,0}}\ldots w_{\cdots m,1}$ of p. The inverted list L of ω, denoted by L_ω, is a set defined as

$$L_\omega = \{ w_a :< 1:0 >, w_b :< 2:0 >, w_c :< 3:0 > ,\ldots, w_{\cdots} :< m:1 > \}.$$

Example 2 From example 1, the inverted list L of ω is

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

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

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

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

TABLE I
THE INVERTED LIST TABLE τ OF P=AABCZ

w_λ	$I_{\lambda_0}/I_{\lambda_1}$
a	<1:0>,<2:0>
b	<3:0>
c	<4:0>
z	<5:1>

Theorem 1 The accessing I_{λ_0} or I_{λ_1} in the table τ takes O(1) times.

Proof Let f(x) be a hashing function, w_{λ_0} be a key for access I_{λ_0} and w_{λ_1} be the key for access I_{λ_1}. Suppose the table τ implemented by the hash table [11], [17], [18], [19], accessing to I_{λ_0} with f(w_{λ_0}) or accessing I_{λ_1} with f(w_{λ_1}) 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=c1,c2,c3,…cm)
Step A  Create table for all alphabet in ∑
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 take 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 $p=c_1,c_2,c_3,\ldots,c_m$. 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=c1,c2..cm, T=t1t2…tn)
Preprocessing
  Create Inverted-List-Table(p)
Searching
Step A   N=m ,SHIFT=2m, pos=1, SET1=φ, SET2=φ, SETE=φ, 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 = φ
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 != φ 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
               SET1← 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 != φ
Step G1          Set pos=maximum inverted list position+1 in SET1 if
                 N>=position of SETE and SHIFT ← SHIFT+1 or
                 others case pos←pos+1
               Else
Step G2          life =0
Step H     N=SHIFT, SHIFT=SHIFT+m, pos=1, SET1=φ,
           SET2=φ, SETE=φ, life = 1
```

Fig. 2 Searching algorithm

Lemma 1 Let SET be the sub table with keys w_{λ_0} and w_{λ_1} for accessing I_{λ_0} and I_{λ_1}, respectively. The access of I_{λ_0} and I_{λ_1} in SET using f(w_{λ_0}) or f(w_{λ_1}) function takes O(1) times.

Proof Let SET is the hash table as the theorem 1 that has a key $w_{\lambda_{\varepsilon,0}}$ and $w_{\lambda_{\varepsilon,1}}$ for accessing. Therefore, it employs f(w_{λ_0}) and f(w_{λ_1}) for accessing I_{λ_0} and I_{λ_1}, respectively. Therefore, it take O(1) according to theorem 1 #

Lemma 2 To get an entry matching a character at text[N] from τ 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 $w_{\lambda_{pos,0}}$ or $w_{\lambda_{pos,1}}$. Hence, the access $I_{\lambda_{pos,0}}$ and $I_{\lambda_{pos,1}}$ in a table τ 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 $I_{\lambda q_{\varepsilon 1,0}}$ and/or $I_{\lambda q_{\varepsilon 1,1}}$ in SET1 to $I_{\lambda b_{\varepsilon 2,0}}$ and /or $I_{\lambda b_{\varepsilon 2,1}}$

in SET2 considering position of $\varepsilon 2$ prior to/next to $\varepsilon 1$. The result is $I_{\lambda b_{\varepsilon 2,0}}$ and /or $I_{\lambda b_{\varepsilon 2,1}}$

Example 4 We show 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>. 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 $I_{\lambda_{\varepsilon 1,0}}$ and/or $I_{\lambda_{\varepsilon 1,1}}$, SET2 contains $I_{\lambda_{\varepsilon 2,0}}$ and /or $I_{\lambda_{\varepsilon 2,1}}$ and SETE contains $I_{\lambda_{\varepsilon 3,0}}$ and /or $I_{\lambda_{\varepsilon 3,1}}$. Accessing $I_{\lambda_{\varepsilon 1,0}}, I_{\lambda_{\varepsilon 1,1}}, I_{\lambda_{\varepsilon 2,0}}, I_{\lambda_{\varepsilon 2,1}}$, $I_{\lambda_{\varepsilon 3,0}}$ and /or $I_{\lambda_{\varepsilon 3,1}}$ for comparing the operation in the definition 5 take O(1) following lemma 1 #

Example 5 Given the pattern p=aabcz, the text T=aabczefgaabczefgabcdg, and the inverted list table τ 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={}, SET2={}, pos=1, SETE={}

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

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 step D2 and F1. N=1, SET1={<1:0>,<2:0>}, SET2={}, pos=1, SETE={<5:1>}

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 F2. N=2, SET1={<1:0>,<2:0>}, SET2={<1:0>,<2:0>}, pos=2, SETE={<5:1>}

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←SET1 operate SET2, SET1={<1:0>,<2:0>} by step F3.

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

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←SET1 operate SET2, SET1={<3:0>} by step F3.

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

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←SET1 operate SET2, SET1={<4:0>} by step F3.

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>}

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←SET1 mask SETE, SET1={<5:1>} matched 1 and remove <5:1> from SET1 and SETE. So SET1={} stop this search window and go to next window search by step G2 and H.

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

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 comparison by step C. N=10, SET1={}, SET2={}, pos=1, SETE={<1:0>,<2:0>}

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
Not match but analyze SETE looking for the farthest and set N=farthest from SETE, therefore set N=9 and go to step D2.

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>}

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 but masks by step F3. N=10, SET1={<1:0>,<2:0>}, SET2={}, pos=2, SETE={<1:0>,<2:0>}

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
SET1←SET1 mask SETE, so SET1={<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>}

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←SET1 operate SET2, so SET1={<3:0>} by step F3 and if N>position of SETE SHIFT←SHIFT+1, SHIFT=16 by step G1.

13. Skip to next position by step F2. N=12, SET1={<3:0>}, SET2={<4:0>}, pos=4, SETE={<1:0>,<2:0>}

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=17

SET1←SET1 operate SET2, so SET1={<4:0>} by step F3 and if N>position of SETE SHIFT←SHIFT+1, SHIFT=17 by step G1.

14. Skip to next position by step F2. N=13, SET1={<4:0>}, SET2={<5:1>}, pos=5, SETE={<1:0>,<2:0>}

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=18

SET1←SET1 operate SET2, so SET1={<5:1>} matched 2 by step F3 and if N>position of SETE SHIFT←SHIFT+1, SHIFT=18. Remove inverted list from SET1 so SET1={} and stop search window and go to step G2 and H.

15. Initialize variables and go step B. N=18, SET1={}, SET2={}, pos=1, SETE={}

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 comparison by step C. N=18, SET1={}, SET2={}, pos=1, SETE={<3:0>}

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

Not matched and analyze SETE N=farthest position from SETE, so N=16 by step D2.

17. Start search at N and compare with the first character in pattern by step F1. N=16, SET1={}, SET2={}, pos=1, SETE={<3:0>}

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

Not matched and SET1={} stop search window and go to step G2 and H.

18. Initialize variables and go step B. N=23, SET1={}, SET2={}, pos=1, SETE={}

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

N and SHIFT > n finished searching.

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 α 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 α rounds and takes O(α) meanwhile 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 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+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 leading to mismatch and n is the length of input text.

References

[1] B., R. S., Moore, J.S., "A fast string searching algorithm", Communications of the ACM. 20, 1997, pp. 762-772.

[2] M. Chrochemore, Handcart C., "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; W. Rytter, "Constant-space string-matching in sublinear average time**",** Compression and Complexity of Sequences 1997. Proceedings, 1997**,** pp. 230 – 239.

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

[6] M. Escardo, (2006, October 15), *Complexity considerations for hash tables* Available: http://www.cs.bham.ac.uk/~mhe/foundations2/node92.html

[7] C. Charras and T. Lecroq. (2006, 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.

[9] Galil, Z., Giancarlo, R., "On the exact complexity of string matching upper bounds**"**, SIAM Journal on Computing, 21(3), 1992, pp. 407-437.

[10] H. Kesong, W. Yongcheng, 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, (2006, November 15), *Hash table*. Available: http://en.wikipedia.org/wiki/Hash_table.
[12] K. Loudon, (2006, 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.
[16] D.E. Knuth, JR. Morris, , J.H., Pratt, V.R., "Fast pattern matching in strings". SIAM Journal on Computing 6(1), 1997, pp. 323-350.
[17] M. S. Ager, O. Danvy, 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, 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] JR. Morris, J.H., Pratt, V.R., *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.
[23] R.M. Karp, M.O. Rabin., Efficient randomized pattern matching algorithms, IBM Journal on Research Development 31(2), 1987, pp. 249-260.