Collaborative Report DC-96-10

Proceedings of the Prague Stringologic Club Workshop '96

Edited by Jan Holub

November 1996

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Sponsors

We thank the Digital Equipment Corporation, Czech Republic and Hewlett-Packard, Czech Republic for their support of the Prague Stringologic Club Workshop '96.

Table of contents

Preface

In 1996 a group of people solving string problems decided to found the Prague Stringologic Club in order to group people researching in the same field. The founder members were M. Bloch, J. Holub, J. Kolář, B. Melichar and P. Mužátko.

The first action of the Prague Stringologic Club was to organize a workshop. This workshop was called the Prague Stringologic Club Workshop'96 and several researchers working in the field of strings were invited to give their talks. The workshop was held on August 14, 1996 at the Department of Computer Science and Engineering of Czech Technical University in Prague.

Here are the proceedings of this workshop.

Jan Holub, editor

An Efficient Multi-Attribute Pattern Matching Machine

Kazuaki Ando, Masami Shishibori and Jun-ichi Aoe

Department of Information Science & Intelligent Systems
Faculty of Engineering
Tokushima University
2-1 Minami-Josanjima-Cho
Tokushima-Shi 770
Japan

e-mail: {ando, aoe}@is.tokushima-u.ac.jp

Abstract. We describe an efficient multi-attribute pattern matching machine to locate all occurrences of any of a finite number of the sequence of rule structures (called matching rules) in a sequence of input structures. The proposed algorithm enables us to match set representations containing multiple attributes. Therefore, in proposed algorithm, confirming transition is decided by the relationship, whether the input structure includes the rule structure or not. It consists in constructing a finite state pattern matching machine from matching rules and then using the pattern matching machine to process the sequence of input structures in a single pass. Finally, the pattern matching algorithm is evaluated by theoretical analysis and the evaluation is supported by the simulation results with rules for the extraction of keywords.

Key words: string pattern matching, set representation, multi-attribute pattern matching, finite state pattern matching machine, matching algorithm

1 Introduction

String pattern matching [Aho75, Aho90, Knut77, Aoe84, Fan93, Boye77] is an important component of many areas in science and information processing. A string pattern matching machine has been applied to various fields such as the lexical analysis of a compiler [Aho86], voice recognition [Take93], bibliographic search [Aho75], spelling checking [Pete80], text editing and so on. Aho and Corasick describe a simple, efficient string pattern matching machine [Aho75] (hereafter called C machine) to locate all occurrences of finite number of keywords in a text string in a single pass. However, in the AC machine, the input is restricted to characters. In addition, a multi-attribute input is very useful for many applications, such as, extraction of keywords [Kimo91, Ogaw93], document processing [Ikeh93], and so on. Moreover, the multi-attribute pattern matching is necessary for the realization of higher pattern matching.

This paper describes an efficient multi-attribute pattern matching machine (hereafter called MAPM machine) to locate all occurrences of any of a finite number of

matching rules in a sequence of input structures and a method for constructing the multi-attribute pattern matching machine. The proposed algorithm enables us to match set representations containing multiple attributes.

In the following chapters, the multi-attribute pattern matching scheme is described in detail. Chapter **2** explains an efficient string pattern matching machine based on Aho and Corasick. Chapter **3** describes the multi-attribute matching rules for the MAPM machine and an efficient algorithm for these matching rules. Chapter **4** explains the construction of the goto, output and failure functions for the MAPM machine. Chapter **5** shows the theoretical estimations and experimental evaluations for the MAPM machine. Finally, in Chapter **6** the future research is discussed.

2 The Aho-Corasick Algorithm

This chapter explains an efficient string pattern matching machine, where a finite state string pattern matching machine based on Aho and Corasick [Aho75] locates all occurrences of any of a finite number of keywords in a text string.

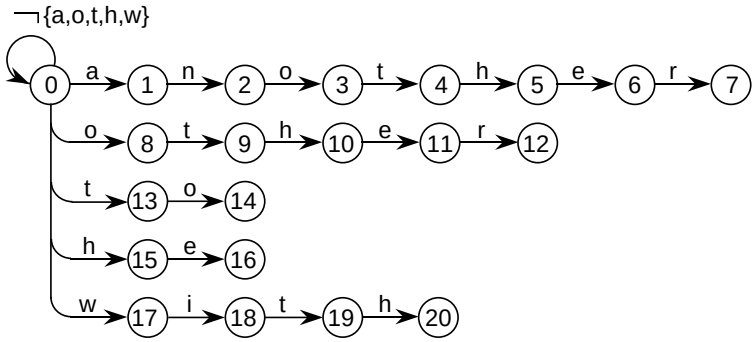
Let K_SET be a finite set of strings which we shall call keywords and let TX be an arbitrary string which we shall call the text string. The AC machine is a program which takes as input the text string TX and produces as output the locations in TX at which the keywords (elements of K_SET) appear as substrings. The AC machine is constructed as a set of states. Each state is represented by a number. The state number 0 represents an initial state.

With I as the set of input symbols, the behavior of the AC machine is defined by next three functions:

goto function $g : S \times I \rightarrow S \cup \{fail\}$,
failure function $f : S \rightarrow S$,
output function $output : S \rightarrow A$, subset of K_SET.

Figure 1 shows the functions, from Aho *et. al.*, used by the machine AC for the set of keywords K_SET = {"another", "other", "to", "he", "with"}. Here, ¬{'a', 'o', 't', 'h', 'w'} denotes all input symbols other than 'a', 'o', 't', 'h' or 'w'.

The directed graph in Figure 1 (a) represents the goto function. For example, the transition labeled 'a' from state 0 to state 1 indicates that $g(0,\text{'a'}) = 1$. The absence of the arc indicates *fail*. The AC machine has the property that $g(0,\text{'}\sigma\text{'}) \neq fail$ for all input symbols σ. The behavior of the AC machine is summarized below.



(a) The goto function.

s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
f(*s*)	0	0	8	9	10	11	12	0	13	15	16	0	0	8	0	0	0	0	13	15

(b) The failure function.

s	*output* (*s*)
6	{ he }
8	{ another, other }
11	{ he }
12	{ other }
14	{ to }
16	{ he }
20	{ with }

(c) The output function.

Figure 1: The function of the AC machine.

Algorithm 1: The AC machine.
[Input]: A target text $TX = c_1c_2...c_n$, where each c_i, for $1 \leq i \leq n$, is an input symbol and an AC machine with goto function g, failure function f, and output function $output$.
[Output]: Locations at which the keywords occur in TX.
[Method]:

```
begin
  s ← 0;
  for i ← 1 until n do
    begin
      while g(s, c_i) = fail do s ← f(s);
      s ← g(s, c_i);
      if output(s) ≠ φ then
        print i, output(s);
    end
end
```

(Example 1) Consider the behavior of the AC machine that uses the functions in Figure 1 to process the text string "stothe". Since $g(0,\text{'s'}) = 0$, the machine enters the state 0. Since $g(0,\text{'t'}) = 13$, $g(13,\text{'o'}) = 14$, the AC machine enters state 14,

advances to the next input symbol and emits $output(14)$, indicating that it has found the keywords "to" at the end of position 3 in the text string. In state 14 with the input symbol 't', the AC machine makes two state transitions in its operating cycle. Since $g(14,\text{'t'}) = fail$, the AC machine enters the state $8 = f(14)$. At this point, since $g(8,\text{'t'}) = 9$, the AC machine enters state 9. Hereafter since $g(9,\text{'h'}) = 10$, $g(10,\text{'e'}) = 11$, the AC machine enters state 11 and computes the matching operation after detecting keyword "he". (END)

The AC algorithm consists in constructing a finite state pattern matching machine from the keywords and then using the machine to process the text string in a single pass. Construction of the AC machine takes a time proportional to the sum of the lengths of the keywords.

3 A Multi-Attribute Pattern Matching Algorithm

This chapter explains an algorithm of an efficient multi-attribute pattern matching machine (called MAPM machine). The MAPM machine is a extension of the Aho-Corasick Algorithm. Section **3.1** describes the multi-attribute matching rules for the MAPM machine. In Section **3.2**, an efficient algorithm of these matching rules is presented.

3.1 Matching rules for the MAPM machine

Let $ATTR$ be the *attribute name* and let $VALUE$ be the *attribute value*. Let R be a finite set of pairs $(ATTR, VALUE)$, then we shall call R a *rule structure*. For example, the following attributes are considered.

STR : string, that is, word spelling.
CAT : category, or, a part of speech.
SEM : semantic information such as concepts and categories.

For example, the rule structure R of "doctor" is defined using the above attributes as follows:

$$R = \{(STR, \text{"doctor"}), (CAT, Noun), (SEM, Human)\}$$

If $RULE$ is the *matching rule* consisting of a sequence of rule structures, $RULE$ is defined as follows:

$$RULE = R_1 R_2 ... R_n (1 \leq n)$$

Let R_SET be a set of $RULE$.

(Example 2) Consider the following example of a R_SET.

$R\_SET = \{RULE_1, RULE_2\}$
$RULE_1 = R_1 R_3, RULE_2 = R_2 R_3$
$R_1 = \{(STR, \text{"male"})\}$
$R_2 = \{(STR, \text{"female"})\}$
$R_3 = \{(SEM1, Male), (SEM2, Imago)\}$

$RULE_1$ is a rule to detect tautology expressions and $RULE_2$ is a rule to detect contradictory expressions. For example, $RULE_1$ can detect the expression of "male man" or "male bull" and so on, $RULE_2$ can detect the expression of "female ram" or "female stallion" and so on. By using the multiple attributes for pattern matching, it is easy to define an abstraction rule. (END)

Input structures to be matched by rule structures are also defined by the same set representation. N is used as the notation for input structures to distinguish them from R. In order to consider the abstraction of the rule structure, matching of the rule structure R and the input structure N are decided by the relationship such that N includes R ($N \supseteq R$).

3.2 A Matching Algorithm

Let α be a sequence of the input structures. The MAPM machine is a program which takes as input α and produces as output the locations in a α at which every $RULE$ in R_SET appears as subsequences of structures. The MAPM machine consists of a set of states. Each state is represented by a number. One state (usually 0) is designated as the initial state. The MAPM machine processes a α by successively reading the input structure N in a α, making state transitions and occasionally emitting an output.

Let S_e be a set of states and let J be a set of the rule structure R, then the behavior of the MAPM machine is defined by next three functions:

goto function $g_e : S_e \times J \rightarrow S_e \cup \{fail\}$,
failure function $f_e : S_e \rightarrow S_e$,
output function $output_e : S_e \rightarrow A_e, subset\ of R\_SET$.

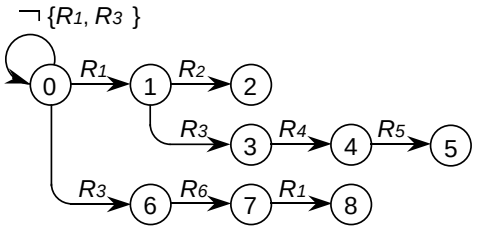
Figure 2 shows the functions used by the machine MAPM for a $R\_SET = \{RULE_1, RULE_2, RULE_3, RULE_4\}$. Here $\neg$ indicates all input structures except R_1 and R_3.

As the AC machine, the goto function g_e maps a pair consisting of a state and an input structure into a state or the message $fail$. The directed graph in Figure 2 (a) represents the goto function. The failure function f_e maps a state into a state. The failure function is constructed whenever the goto function reports $fail$. Certain states are designated as output states which indicate that a $RULE$ has been found. The output function formalizes this concept by associating R_SET (possible empty) with every state.

In the MAPM machine, a confirming transition is decided by the relationship, whether the input structure N includes the rule structure R or not. Therefore, in order to confirm a transition from certain state to next_state using the relationship, we define a function CHECK($state, N$) as follows:

[Function CHECK($state, N$)]
For each transition $g_e(state, R) = next\_state$ in the goto graph, if some transition labeled R which satisfies the relationship ($N \supseteq R$) exists, then it returns all $next\_state$, otherwise it returns $fail$.

(Example 3) Suppose that $g_e(s_1, R_1) = s_2$ and $g_e(s_1, R_2) = s_3$ are defined in the goto graph and R_1 and R_2 are defined as follows:



(a) The goto function.

s	1	2	3	4	5	6	7	8
f(*s*)	0	0	6	9	0	0	0	1

(b) The failure function.

s	*output*$_e$ (*s*)
2	{ *R1*, *R2* }
5	{ *R1*, *R3*, *R4*, *R5* }
7	{ *R3*, *R6*}
8	{ *R3*, *R6*, *R1*}

(c) The output function.

R_SET = { *RULE1*, *RULE2*, *RULE3*, *RULE4* }
RULE1 = *R1 R2* *RULE2* = *R1 R3 R4 R5* *RULE3* = *R3 R6* *RULE4* = *R3 R6 R1*
R1 = { (*CAT*, *Adjective*) } *R2* = { (*CAT*, *Noun*), (*SEM1*, *Flower*) } *R3* = { (*CAT*, *Noun*), (*SEM1*, *Human*), (*SEM2*, *Female*) } *R4* = { (*STR*, "like"), (*CAT, Verb*) } *R5* = { (*CAT*, *Noun*) } *R6* = { (*STR*, "are"), (*CAT*, *Verb*) }

(d) *R_SET* = { *RULE1*, *RULE2*, *RULE3*, *RULE4* }.

Figure 2: The functions of the MAPM machine for R_SET.

$$R_1 = (SEM, Human)$$
$$R_2 = (CAT, Noun))$$

Consider the behavior of the function CHECK to process the following N_1 and N_2.

$$N_1 = (STR, \text{“mother”}), (CAT, Noun), (SEM, Human)$$
$$N_2 = (STR, \text{“beautiful”}), (CAT, Adjective)$$

For N_1 the function CHECK returns s_2 and s_3 to satisfy $R_1 \subseteq N_1$ and $R_2 \subseteq N_1$. In the case of N_2 , $R_1 \subseteq N_2$ and $R_2 \subseteq N_2$ are not satisfied, therefore the function CHECK returns $fail$. (END)

Algorithm 2 summarizes the behavior of the MAPM machine.

Algorithm 2: A Multi-Attribute Pattern Matching machine (MAPM machine).
[Input]: A sequence of input structures $\alpha = N_1, N_2, ..., N_n$, where each N_i is an input structure and a MAPM machine with goto function g_e, failure function f_e, and output function $output_e$.
[Output]: Locations at which sequences of structure were extracted.
[Method]:

```
begin
  queue1 ← 0;
  for i ← 1 until n do
    begin
      queue2 ← empty;
      while queue1 ≠ φ do
        begin
          temp ← empty;
          let state be the next state in queue1;
          queue1 ← queue1 − state;
          while CHECK(state, Ni) = fail do state ← fe(state);
          temp ← CHECK(state, Ni);
          queue2 ← queue2 ∪ temp;
          while temp ≠ φ do
            begin
              let element be the next state in temp;
              temp ← temp − element;
              if outpute(element) ≠ φ then
                begin
                  print i;
                  outpute(element)
                end
            end
        end;
      queue1 ← queue2
    end
end.
```

As we have mentioned before, in this algorithm, confirming transitions are decided by the relationship ($N \supseteq R$). Therefore, it has the possibility that some R such that N includes R exist among those transitions. In order to solve this problem, Algorithm 2 stores all states returned by the function CHECK in a first-in-first-out list denoted by the variable $queue_1$, and the MAPM machine continues to process for each state in $queue_1$.

(Example 4) Figure 2 shows the functions used by the MAPM machine for a $R\_SET = RULE_1, RULE_2, RULE_3, RULE_4$. Consider the behavior of the MAPM machine that uses the functions in Figure 2 to process the sequence of the input structures $\alpha = N_1N_2N_3N_4N_5N_6$.

N_1 = (STR, "beautiful"), ($CAT, Adjective$)
N_2 = (STR, "rose"), ($CAT, Noun$), ($SEM1, Flower$)

N_3 = (STR, "and"), ($CAT, Conjunction$)
N_4 = (STR, "pretty"), ($CAT, Adjective$)
N_5 = (STR, "girl"), ($CAT, Noun$), ($SEM1, Human$), ($SEM2, Female$)
N_6 = (STR, "are"), ($CAT, Verb$)

Since N_1 includes R_1 and N_2 includes R_2, CHECK(0,N_1) = 1 and CHECK(1, N_2) = 2, the MAPM machine enters state 2, advances to the next input structure and emits the $output_e(2)$, indicating that it has found the $RULE_1$ at the end of position 2 in the α. Since N_3 doesn't include R_1 or R_3, CHECK(0, N_3) = 0, the MAPM machine enters the state 0. Since N_4 includes R_1 and N_5 includes R_3, CHECK(0, N_4) = 1, CHECK(1, N_5) = 3, the MAPM machine enters state 3. In state 3 with the input structure N_6, the MAPM machine makes two state transitions in this operating fashion. Since N_6 doesn't include R_4, CHECK(3, N_6) = $fail$, the MAPM machine enters the state 6 = $f_e(3)$. At this point, since N_6 includes R_6, CHECK(6, N_6) = 7, the MAPM machine enters state 7, emits $output_e(7)$, indicating that it has found the $RULE_3$ at the end of position 6 in the α. (END)

The MAPM algorithm consists in constructing a finite state pattern matching machine from matching rules and then using the machine to process the sequence of input structures in a single pass.

4 Construction of Goto, Output and Failure Function for the MAPM Machine

This chapter explains the construction of the goto, output and failure function for the MAPM machine. Although the construction of the MAPM machine is similar to the construction of the AC machine [Aho75], the point which changes a transition label for the goto function to a set is different from the AC machine. Therefore, the decision of the same transition when the goto, failure and output functions are constructed is defined as follows:

[Condition for the equivalency of rule structures]
If the number of elements of rule structure R_1 and R_2 are equal and each of the elements ($ATTR, VALUE$) of R_1 and R_2 are equal, then we define R_1 equal to R_2 ($R_1 = R_2$).

In order to examine the equivalency of rule structures, we define a function ARC($state, R$). The function ARC returns a $next\_state$ that satisfies the condition for the equivalency of rule structures. The following shows the function ARC.

[Function ARC($state, R_2$)]
For each transition $g_e(state, R_1) = next\_state$ in the goto graph, if $R_1 = R_2$, then the function ARC returns a $next\_state$, otherwise it returns $fail$.

(Example 5) Suppose that $g_e(state_1, R_1) = state_2$ is defined in the goto graph and R_1 is defined as:

$$R_1 = (CAT, Noun), (SEMHuman)$$

Consider the behavior of the function ARC to process the following rule structures R_2 and R_3:

$$R_2 = (CAT, Noun), (SEM, Human)$$
$$R_3 = (CAT, Noun)$$

For R_2, ARC returns $state_2$ to satisfy the condition for the equivalency of rule structures $R_1 = R_2$. For R_3, the function CHECK doesn't satisfy the condition for the equivalency of rule structures $R_1 = R_3$ and therefore returns $fail$. (END)

Algorithm 3 summarizes the method for the construction of the goto and output functions for the MAMP machine. Algorithm 4 summarizes the method for the construction of the failure and output functions for the MAMP machine.

Algorithm 3: Construction of the goto function.
[Input]: Set of $RULE\ R\_SET = RULE_1, RULE_2, ..., RULE_k$.
[Output]: Goto function g_e and a partially computed output function $output_e$.
[Method]: We assume $output_e(s)$ is empty when state s is first created, and $g_e(s, R) = fail$ if R is undefined or if $g_e(s, R)$ has not yet been defined. The procedure $enter(RULE)$ inserts into the goto graph a path that spells out $RULE$.

```
begin
   newstate ← 0;
   for i ← 1 until k do enter(RULE_i);
   for all R such that ARC(0, R) = fail do g_e(0, R) ← 0;
end

procedure enter(R_1, R_2, ..., R_m)
begin
   state ← 0; j ← 1;
   while ARC(state, R_j) ≠ fail do
      begin
         state ← ARC(state, R_j);
         j ← j + 1
      end
   for p ← j until m do
      begin
         newstate ← newstate + 1;
         g_e(state, R_p) ← newstate;
         state ← newstate;
      end
   output_e(state) ← R_1, R_2, ..., R_m
end
```

Algorithm 4: Construction of the failure function.
[Input]: Goto function g_e and output function $output_e$ from Algorithm 3.
[Output]: Failure function f_e and output function $output_e$.
[Method]:

```
begin
  queue ← empty;
  for each R such that ARC(0, R) = s ≠ 0 do
    begin
      queue ← queue ∪ s;
      f_e(s) ← 0
    end
  while queue ≠ empty do
    begin
      let r be the next state in queue;
      queue ← queue − r;
      for each R such that ARC(r, R) = s ≠ fail do
        begin
          queue ← queue ∪ s;
          state ← f_e(r);
          while ARC(state, R) = fail do state ← f_e(state);
          f_e(s) ← ARC(state, R);
          output_e(s) ← output_e(s) ∪ output_e(f_e(s))
        end
    end
end
```

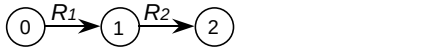
(Example 6) Consider the construction of the MAPM machine for the $R\_SET = RULE_1, RULE_2, RULE_3, RULE_4$ in Figure 2 (d). In the first place, the states and the goto function are determined according to Algorithm 3. Second, the failure function is computed according to Algorithm 4. The output function is constructed according to both Algorithms. The decision of the same transition when the goto, failure and output functions are constructed is decided by the equivalency of the rule structures.

Firstly, consider the construction of goto function. Adding the first rule $RULE_1$ to the graph, $g_e(0, R_1) = 1$ and $g_e(1, R_2) = 2$ are constructed as shown in Figure 3 (a). At this point, the output "R_1, R_2" is associated with state 2.

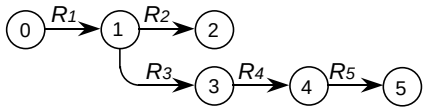
Adding the second rule $RULE_2$, Figure 3 (b) is obtained. Since ARC(0, R_1) = 1, notice that when the rule structure R_1 in $RULE_2$ is added there is already a transition labeled "R_1" from state 0 to state 1. Therefore it is not needed to add another transition labeled "R_1" from state 0 to state 1. After that, $g_e(1, R_3) = 3$, $g_e(3, R_4) = 4$ and $g_e(4, R_5) = 5$ are constructed as shown Figure 3 (b). The output "R_1, R_3, R_4, R_5" is associated with state 5.

Adding the third rule $RULE_3$, $g_e(0, R_3) = 6$ and $g_e(6, R_6) = 7$ are constructed as shown in Figure 3 (c). The output "R_3, R_6" is associated with state 7.

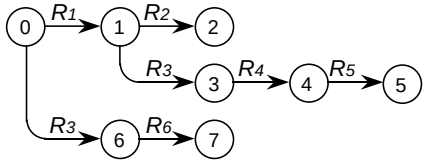
Adding the last rule $RULE_4$, Figure 3 (d) is obtained. The output "R_3, R_6, R_1" is associated with state 8. Here, since ARC(0, R_3) = 6 and ARC(6, R_6) = 7, we have been able to use the existing transition labeled R_3 from state 0 to state 6 and the



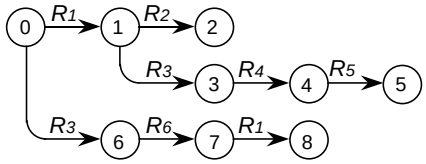
(a) The graph for *RULE1*.



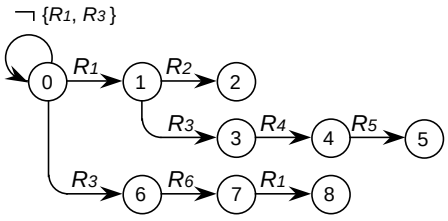
(b) The graph for *RULE1* and *RULE2*.



(c) The graph for *RULE1*, *RULE2* and *RULE3*.



(d) The graph for *RULE1*, *RULE2*, *RULE3* and *RULE4*.



(e) The graph for *R_SET*.

Figure 3: The construction of the goto graph for R_SET.

existing transition labeled R_6 from state 6 to state 7. To complete the construction of the goto function, a loop from state 0 to state 0 on all the input structures other than "R_1" and "R_3", is added to the graph. Finally, Figure 3 (e) is obtained.

Next, consider the construction of the failure function. To compute the failure function from Figure 3 (e), $f_e(1) = f_e(6) = 0$ is set since state 1 and 6 are the states of depth 1. Then the failure function for state 2, 3 and 7, the states of depth 2, is computed. To compute $f_e(2)$, $state = f_e(1) = 0$ is set; and since ARC(0, R_2) = 0, $f_e(2) = 0$ is determined. To compute $f_e(3)$, $state = f_e(1) = 0$ is set, and since ARC(0, R_3) = 6, $f_e(3) = 6$ is determined. To compute $f_e(7)$, state = $f_e(6) = 0$ is set; and since ARC(0, R_6) = 0, $f_e(7) = 0$ is determined. Continuing in this fashion, the failure function shown in Figure 2 (b) is obtained.

Finally, the goto, failure and output functions are constructed as shown in Figure 2 (b) and (c). (END)

5 Evaluation

In this chapter, the theoretical estimations and experimental evaluations for the MAPM machine are presented. Section **5.1**, describes the theoretical estimations for the MAPM machine, and in Section **5.2**, it is evaluated by applying it to the extraction of keywords [Kimo91, Ogaw93].

5.1 Theoretical Estimations

Suppose that the time complexity for confirming a transition in the MAPM machine is O(1). Let m be the length of sequence of input structures α. The time complexity of matching Algorithm 2 by the MAPM machine is O(m), because the matching cost of the AC machine is independent of the number of matching rules (keywords). However, the precise complexity for confirming a transition depends on the cost of the function CHECK and $queue_1$ in Algorithm 2.

Consider the time complexity of the function CHECK. By using the order of attributes names, sets of input and rule structures can be represented as the sorted-list whose nodes are denoted by (attribute-name, attribute-value, pointer). Similarly, the goto function is represented by the list structure. Let K be the kinds of attribute names. In the function CHECK, the time complexity for judging the relationship, whether the input structure includes the rule structure or not, is similar to the cost ($K + K = 2K$) of merging two sorted_lists of maximum length K into one list. Therefore, the time complexity of judging the relationship is O(K). Suppose that B is the maximum number of transitions leaving from certain state s. Then, the time complexity of the function CHECK is O(KB). Although this cost is more than that of the AC machine, an expression ability of rules for the MAPM machine is higher than the rule for the AC machine.

From the above observation, consider the time complexity of confirming a transition. Let D be the maximum number of the states in $queue_1$. The complexity is O(KBD), because $queue_1$ has all states returned by the function CHECK in Algorithm 2.

The time complexity for constructing the AC machine is proportional to the total length h of keywords. On the other hand, the time complexity for the construction of the MAPM machine is O(hK) in the worst-case, because confirming transitions depends on the function ARC and the time complexity for determining the equivalency of the rule structures in the function ARC is the same cost O(K) as the function CHECK.

5.2 Experimental Evaluations

The MAPM machine is evaluated by applying it to the extraction of keywords [Kimo91, Ogaw93]. For experimental evaluations, the MAPM machine has been implemented on a DELL OptiPlex GXMT5133 and it has been written in the C language.

In order to evaluate the efficiency of the proposed algorithm, we defined 112 rules for the extraction of keywords. Table 1 shows the information about the rules for the extraction of keywords. The information about $RULE$ are the values for each

Total number of kinds of attribute names	5
Information about $RULE$	
Total number	112
Average length	2.5
Average number of kind of attributes names	1.9
Construction time of the MAPM machine [sec]	0.543

Table 1: Information about $RULE$ and construction time.

	Text1	Text2	Text3	Average
Number of words	411	233	197	280.0
Matching time [ms]	20.9	10.3	7.4	12.9
Number of extracted keywords	18	15	15	16.0

Table 2: The results of the simulation.

rule structure. From the average number, 1.9, of kinds of attribute names, it turns out that the attribute of each structure was abstracted effectively. It seems that the construction time (CPU time), 0.543 second, is practical.

Table 2 shows the results of the simulation using the above rule. To perform the simulation of the extraction of keywords, the following three texts were used.

Text1: General document, such as letter, journal, etc.
Text2: Abstract of a paper.
Text3: Document of patents.

From the average matching time in Table 2, the efficiency of the proposed algorithm could be verified. As shown by the theoretical estimations, the time complexity of the MAPM machine depends on the cost of the function CHECK. In the simulation, the attribute of each structure was abstracted effectively, such that the average of number of kinds of attribute names is 1.9. Consequently, good results could be obtained.

6 Conclusions

We have described an efficient method for multi-attribute pattern matching in this paper. A multi-attribute pattern matching is useful for many applications and the proposed algorithms enable the realization of higher pattern matching. The presented algorithm are evaluated by theoretical estimation and the experimental evaluation is supported by simulation results for the extraction of keywords.

In the proposed algorithms, it takes time to judge whether the input structure includes the rule structure or not. Therefore, as future extension, we are considering an efficient data structure and an efficient decision algorithm for the judging of the relationship, whether the input structure includes the rule structure or not. We

believe the proposed method is very useful for any existing and future computing system that would require an efficient multi-attribute pattern matching.

References

[Aho75] A.V. Aho – M.J. Corasick: Efficient String Matching : An Aid to Bibliographic Search. *Comm. ACM*, Vol.18, No.6, pp.333-340, 1975.

[Aho90] A.V. Aho: Algorithms for Finding Patterns in Strings. in J. Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier Science Publishers, pp.275-300, 1990.

[Aho86] A.V. Aho – R. Sethi – J.D. Ullman: Compilers Principles, Techniques and Tools. Reading MA: Addison-Wesley, ch. 2, 1986.

[Knut77] D.E. Knuth – J.H. Morris, Jr. – V.R. Pratt: Fast pattern matching in strings. *SIAM J.Comput.*, Vol.6, pp.323-350, June 1977.

[Kimo91] H. Kimoto: Automatic Indexing and Evaluation of Keywords for Japanese Newspapers. Ttrans. of the Institute of Electronics, Information and Communication Engineers of Japan, D-I Vol.J74-D-I, No.8, pp.556-566, Aug. 1991. (in Japanese)

[Aoe84] J. Aoe – Y. Yamamoto – R. Shimada: A Method for Improving String Pattern Matching Machine. *IEEE Trans. Software. Eng.*, Vol.10, No.6, pp.116-120, 1984.

[Fan93] J.-J. Fan – K.-Y. Su: An efficient algorithm for matching multiple patterns. *IEEE Trans. Knowledge and Data Eng.*, KDE-5, 2, pp.339-351, 1993.

[Pete80] J. L. Peterson: Computer Programs for Spelling Correction. Lecture Notes in Computer Science, New York: Springer-Verlag, 1980.

[Boye77] R.S. Boyer – J.S. Moore: A fast string pattern matching algorithm. *Comm. ACM*, Vol.20, No.10, pp.762-772, 1977.

[Ikeh93] S. Ikehara – E. Ohara – S. Takagi: Natural Language Processing for Japanese Text Revision Support System. Journal of Information Processing Society of Japan, Vol.34, No.10, pp.1249-1258, Oct. 1993. (in Japanese)

[Ogaw93] Y. Ogawa – M. Mochinushi – A. Bessho: A Compound Keyword Assignment Method for Japanese Texts. Research Report, Information Processing Society of Japan, 93-NL-97-15, pp.103-109, Sep. 1993. (in Japanese)

[Take93] Y. Takebayashi: Natural Language Processing in Speech Understanding and Dialogue. Journal of Information Processing Society of Japan, Vol.34, No.10, pp.1287-1296, Oct. 1993. (in Japanese)

Efficiency of AC-Machine and SNFA in Practical String Matching

Martin Bloch

Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University,
Karlovo nám. 13,
121 35 Prague 2,
Czech Republic

e-mail: bloch@cslab.felk.cvut.cz

Abstract. A note on practical experience with on Aho-Corasick-machine and SNFA (Searching NFA) estimating the construction aspects and run cost. It is shown that SNFA can be more practical than AC-machine.

Key words: string matching, AC algorithm, nondeterministic finite automaton, practical text searching

1 Introduction

Let P be finite set of pattern character strings of the finite length and let T be a text, i.e. string of characters of length t. The task is to find all occurrences of all patterns in the text using an abstract matching machine. We can use different machines optimizing *space* or *time* or $cost = space * time$ of their run phase. All these machines are defined by P completely. We can recognize two cases of P or pattern machines: static or dynamic. In static case set P is constant. In dynamic case patterns can be included into or deleted from the set P. Let us pay attention to two machines SNFA (Searching Nondeterministic Finite Automaton) and AC-machine (Aho-Corasick [AC75]) only.

Both these machines include G-trie as their basic structure. The G-trie is root $\rightarrow$ leaf oriented tree, in which every node is uniquely labeled by prefix $\in$ Prefix(P) and trajectory from root to the node is also labeled by this prefix. The G-depth of a node is length of trajectory from root to the node. Prefix(P) is set of all prefixes of all patterns in P. The G-trie represents goto function.

Undoubtedly, AC-machine represents great theoretical leap in stringology proving linear run time depending just on t and not depending on P. (AC-machine is specific case of implementation of DFA (Deterministic Finite Automaton) and makes less than $2t$ transitions after reading text T). This lovely miracle was obtained by two ingenious tricks:

- The G-trie is overlaid by leaf $\rightarrow$ root oriented fail F-tree containing nonlabeled fail arrows. (G and F have the same set of nodes and common root.) Each node (except common root) is equipped by fail arrow which points to the node which is labelled by longest suffix of label of the node. The F-tree represents failure function. The F-depth of a node is length of trajectory to the root.
- The root contains labeled loops for all characters in given alphabet which do not label any arrows going from the root to another node.

The space complexity of the AC-machine is linear dependent on the number of nodes in the trie. It can be implemented in different ways (saving the linearity) but practically requiring more space (say, linked list or direct access table implementation). Very good practical method designed by [AYS88] is based on the interspersed direct access tables. (Imagine a set of combs with missing teeth to be somehow assembled in line overlaying such a way that no tooth would mask another one and the length of this assembly is almost minimal).

This method preserves fast unit time state transition and space linearity with little space overhead caused by unused slots. The overhead under 5% can be easily reached in large practical cases [Hla96]. The only severe problem in the implementation of AC/Aoe is necessity of solving the "teeth conflict" by moving one comb to another place. Aoe et al. solve this task by moving the smaller comb having smaller number of teeth (that sounds somehow logically). Hladík moves the newer comb what spares the time necessary for counting teeth in both combs and allows to use the memory in a better way. It is not known whether Hladík's method has higher overhead than Aoe's method.

The construction of the goto function, ie. G-trie, is rather simple problem and moreover it can be created dynamically. The construction of fail function is not difficult for static case when the trie is scanned width-first. It is difficult to maintain the dynamic AC-machine because after addition or deletion of any pattern the fail function should be recalculated completely. In order to avoid this tedious recalculation, inverse arrow for every fail arrows should be implemented but it increases memory requirements further. The machine equipped with such inverse arrows is denoted as ACi-machine.

2 Comparison of AC-machine and SNFA

SNFA is a very specific case of NFA and its construction is simpler than AC-machine but its run is principally slower. SNFA is just the G-trie amended by set of labelled loops for all characters in given alphabet in its root. Therefore the root is the only source of nondeterminism.

Let us imagine that the general NFA is a sort of a playboard and its run is a sort of a game played by team of pawns or dwarfs. At most one pawn can stay at one node each time. After reading of an input character every pawn goes via all goto arrows labelled by the read character. If there is no such an arrow the pawn is removed. When more pawns are encountered in one node they join into one pawn. The game is over when there is no pawn on the playboard or when the input is exhausted.

SNFA starts with just one pawn in the root node. This root pawn permanently stays in the root because of the set of root loops and plays the role of a bee queen

yielding other pawns. Pawns cannot encounter each other at one node (the playboard is a trie) and therefore no problem arise with the joining operation. Two pawns never stay at the nodes with the same G-depth. The maximum number of pawns is the length of the longest pattern in P plus one.

AC-machine is a DFA and therefore just one pawn plays the game starting in the root node, too. This pawn cannot be removed because he uses the fail arrow to save his life when the input character does not match any goto arrow. In the worst case the pawn falls down into the root and the root node provides his immortality.

There is a certain remarkable similarity between SNFA and AC-machine. AC-machine pawn determines where pawns would stay in equivalent SNFA (having identical G-trie). This is given by the fail arrow trajectory from any node to the root which shows where all pawns would stay in SNFA. The number of pawns is given by the F-depth of such node (plus one for permanent pawn in the root). The maximum number of pawns is therefore limited by the depth of the F-tree.

3 Experiments

Several tests have been performed on two large practical pattern sets:

CA Queries for SDI retrieval in the Chemical Abstracts data base. Pattern are chemical terms mainly.

WN Word Net thesaurus of the English language prepared at Princeton University containing nouns, verbs, adjectives and adverbs.

The following table shows main characteristics of pattern sets, G-tries and F-trees.

Pattern set	**CA** Chem. Abstracts SDI queries	**WN** WordNet thesaurus
no. of patterns	13872	174678
no. of nodes	70315	882831
avg. G-depth	10.8	10.2
max. G-depth	47	63
avg. F-depth		4.3
max. F-depth	8	9

The following table shows typical branching of the G-trie that can be useful for its implementation.

Node type	**output degree**	**percentage**
fork	>1	8
single	1	77
leaf	0	15

In order to estimate the run efficiency of the AC-machine and the SNFA, following characteristics have been introduced:

RSR is a relative space requirement given by the ratio of number of arrows to the number of nodes. For the G-trie is RSR=1, for the AC-machine is RSR=2, for the ACi-machine RSR=3.

ATC is the arrow transition coefficient defined as the average number of transitions via G-arrows or F-arrows caused by one input character.

ANP is an average number of pawns taking part in the game.

RRT is a relative run time consumed. RRT = ATC * ANP.

RRC is a relative run cost where RRC = RSR * ATC * ANP.

The following table shows these run characteristics:

Pattern set	**CA**		**WN**	
Machine	SNFA	AC	SNFA	AC
RSR	1	2	1	2
ATC	1	1.42	1	1.56
ANP	3.3	1	2.8	1
RRT=ATC*ANP	3.3	1.42	2.8	1.56
RRC=RSR*ATC*ANP	3.3	2.84	2.8	3.12

4 Conclusions

It follows from previous table that time ratio RRT_{SNFA}/RRT_{AC} is 2.32 or 1.79 and therefore AC is about twice faster. Nevertheless, the cost ratio RRC_{SNFA}/RRC_{AC} varies from 1.16 to 0.90 and therefore SNFA run can be sometimes cheaper than AC run. This intimates that for some practical cases the SNFA is not so bad. Taking into account its simpler construction, the lower storage requirements and the dynamic ability it can be preferred in practical cases where time requirements have not the absolute priority.

References

[AC75] Aho, A. V. - Corasick, J. M.: Efficient string matching: An aid to bibliographic search, CACM, 18, June 1975, no. 6, pp. 333-340.

[AYS88] Aoe, J. - Yasutome, S. - Sato, T.: An efficient digital search algorithm by using a double-array structure, IEEE 1988, pp. 472-479.

[Blo89] Bloch, M.: Optimalizace vyhledávání vzorku v textových řetězech (Optimization of pattern retrieval in text strings), Doctoral thesis, Dept. of Comp. Sci. and Eng., Fac. of Electrical Eng., Czech Tech. Univ., Prague 1989, p. 79, (in Czech).

[Hla96] Hladík, J.: Vyhledávací stroj AC/Aoe (Retrieval machine AC/Aoe.), Diploma thesis, Dept. of Comp. Sci. and Eng., Fac. of Electrical Eng., Czech Tech. Univ., Prague 1996, p.136, (in Czech).

Reduced Nondeterministic Finite Automata for Approximate String Matching

Jan Holub

Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University,
Karlovo nám. 13,
121 35 Prague 2,
Czech Republic

e-mail: holub@cs.felk.cvut.cz

Abstract. We will show how to reduce the number of states of nondeterministic finite automata for approximate string matching with k mismatches and nondeterministic finite automata for approximate string matching with k differences in the case when we do not need to know how many mismatches or differences are in the found string. Also we will show impact of this reduction on Shift-Or based algorithms.

Key words: finite automata, approximate string matching

1 Introduction

Problem of approximate string matching can be described in the following way: Given a text string $T = t_1 t_2 \cdots t_n$, a pattern $P = p_1 p_2 \cdots p_m$, and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of a substring X in the text string T such that the distance $D(P, X)$ between the pattern P and the string X is less than or equal to k. In this article we will consider two types of distances called Hamming distance and Levenshtein distance.

The Hamming distance, denoted by D_H, between two strings P and X of equal length is the number of positions with mismatching symbols in the two strings. We will refer to approximate string matching as string matching with k mismatches whenever D is the Hamming distance. The Levenshtein distance, denoted by D_L, or edit distance, between two strings P and X, not necessarily of equal length, is the minimal number of editing operations insert, delete and replace needed to convert P into X. We will refer to approximate string matching as string matching with k differences whenever D is the Levenshtein distance. Clearly the Hamming distance is a special case of the Levenshtein distance in which we submit only replace as an editing operation.

A nondeterministic finite automaton (NFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is a finite set of input symbols, δ is a state transition function from $Q \times (A \cup \{\varepsilon\})$ to the power set of Q, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is

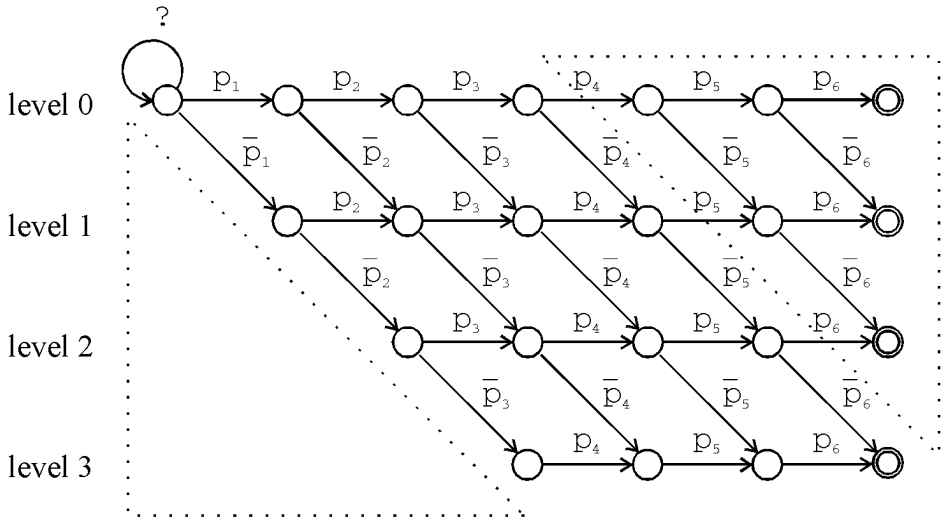


Figure 1: *NFA* for approximate string matching with k mismatches.

the set of final states. In the following, we will use the alphabet $A = \{s_1, s_2, \cdots, s_{|A|}\}$. If $p \in A$ then $\bar{p}$ is the complement set $A - \{p\}$, in our case. A question mark ? will represent any character of the alphabet.

2 String Matching with k Mismatches

2.1 Nondeterministic Finite Automaton

A *NFA* for string matching with k mismatches for pattern P, as it was presented in [Me95] and [Da92], is shown in Figure 1. In this figure $m = 6$ and $k = 3$. The sequence of states of the level 0 of the *NFA* contains states that correspond to the given pattern without any mismatch, the sequence of states of the level 1 contains states that correspond to the given pattern with one mismatch, ...etc. At the end of each level there is a final state. This state says that the pattern was found with $0, 1, \ldots$ etc. mismatches, respectively.

There are three kinds of transition:

A transition representing matching character in pattern P and character in the text T, $t_i = p_j$. In Figure 1 this transition is marked by the arrow directed to the right. It leads to a state with the same number of mismatches as the old state, to a state of the same level.

A transition representing an editing operation *replace*, $t_i = \overline{p_j}$. In Figure 1 this transition is marked by the arrow directed to the right-down, it leads to a state with a number of mismatches one greater than the former state, to a state on one level lower.

A transition representing that the *NFA* always stays in the initial state. In Figure 1 this transition is marked by the self loop in the level 0.

This *NFA* accepts all strings having a postfix X such that $D_H(P, X) \leq k$.

The number of states of *NFA* for string matching with k mismatches is $(k+1)(m+1-\frac{k}{2})$.

2.2 Reducing the number of states

There can be some situations in which we want to know all occurrences of the given pattern in the input text with at most k mismatches but we are not interested in knowing the number of mismatches in the found string. The states that are needed only to recognize how many mismatches are in the found string, form a right angle triangle in upper right corner of the *NFA*, as marked by the dotted line in Figure 1. On the opposite side of the *NFA* there is the complement triangle of missing states. If we omit these two triangles we obtain a simplified *NFA* with $(k+1)(m+1-k)$ states.

Now the final states have been changed. If the *NFA* is in the final state at the end of level j it means that the pattern P can be found with at least j and at most k mismatches. A problem appears at the end of the input text. If less than $k-j$ characters remain in the input text then the final state of level j in the *NFA* does not mean that the pattern can be found with at least j and at most k mismatches, because the transition for either mismatch or replace needs to read one input character. So if the *NFA* is in final state at the end of level j and the position in input text i is at most $n-k+1$ it means that the pattern P can be found with at least j and at most k mismatches.

2.3 Shift-Or Algorithm

The initial version of this algorithm was presented in [BG92]. There are also modifications of this algorithm called Shift-Add [BG92] and Shift-And [WM92].

The Shift-Or algorithm uses m bit state vectors R^j which represent rows of the *NFA*, as it was presented in [Ho96], and a mask table D that for each character has an m bit vector in which bits corresponding to positions of the character in the pattern are set to 0 and other bits are set to 1. This table is used for operation matching. If the *NFA* is in a state (i,j), where i, $0 \leq i \leq m$, is a depth of state in the *NFA* and j, $0 \leq j \leq k$, is a level of state, then i-th bit of the vector R^j contains 0. If the *NFA* is not in a state (i,j), i-th bit of the vector R^j contains 1. The right angle triangle in lower left corner of the *NFA*, as marked by the dotted line in Figure 1, can be defined as the first j bits of each vector R^j and in the vectors R^j it is represented by 0s.

The vector R^0 is defined by formula $R^0_{i+1} = shl(R^0_i) or (D[t_{i+1}])$, where R^0_i is the old value and R^0_{i+1} is a new value of R^0 corresponding to position i in the text T, and represents exact string matching. *or* is bitwise operation OR and *shl* is bitwise operation left shift, that moves bits of the vector to the left and fills the last bit of the vector with an 0. Vectors for approximate pattern matching with k mismatches are defined by the formula $R^j_{i+1} = (shl(R^j_i) or (D[t_{i+1}])) and (shl(R^{j-1}_i))$, where j denotes the number of substitutions. At the beginning of the search the vectors R^j are filled up by 1s.

The fact, that the pattern has been found with at most j mismatches in position i, is detected by appearing 0 at the end of the vector R^j_i.

The Shift-Or algorithm computes new states of the *NFA* in a parallel way. It computes whole sequence of states of one level at once. The bitwise operation $shl(R^j_i)$ moves all the states of one level to the left and inserts 0 in first position. It represents the transition of matching, each state moves to the state corresponding to the next position in the pattern P. This operation is only for matching, so we have to eliminate states that do not match. That is performed by the bitwise operation $or(D[t_{i+1}])$,

that takes the mask vector corresponding to character t_{i+1} in the text T and replaces all 0s in mismatching positions by 1s. The result of this operation is that only states in matching positions have been moved to the next states of the same level.

The second item of the above formula is $shl(R_i^{j-1})$. The item represents the editing operation *replace*. It takes the sequence of states of level $j-1$ corresponding to $j-1$ mismatches, moves it one position to the left and makes the bitwise operation *and* between result of the first item of the formula and this sequence of states. Here, the bitwise operation $and(shl(R_i^{j-1}))$ adds states coming from the level corresponding to one less mismatches than in level j. It is clear that this item has no meaning for the states of the sequence of states without mismatches. Thus this item is present only in formulae for computing vectors R^j, where $0 < j \leq m$. To make Shift-Or algorithm faster this bitwise operation representing *replace* is performed even in the positions matching the input character. It simplifies the formula without change of behaviour of Shift-Or algorithm.

2.4 Simplified Shift-Or Algorithm

The reduced *NFA* can be described by vectors R^j, that are k bit shorter than the original ones. Of course, the formulae for computation of new vectors R^j have changed too. The new formulae are $R_{i+1}^0 = shl(R_i^0) or (D_j[t_{i+1}])$ for exact matching and $R_{i+1}^j = (shl(R_i^0) or (D_j[t_{i+1}])) and\ R_i^{j-1}$ for j mismatches. The mask table D_j is a part of mask table D being $m-k$ bit long and starting at position j in D. There are two ways how to represent mask tables D_j. The first way is to compute needed column of this mask table by shifting the column of the original mask table D whenever it is needed and another is to store shifted mask tables D_j, $0 \leq j \leq k$. The first way has higher time complexity and the second has higher space complexity.

A problem similar to the problem that appears at the end of the input text described above appears at the beginning of the input text. The previous problem has appeared because of omitting states in a *NFA* and this problem has appeared because of omitting first j bits of vector R^j. If the input text starts with a string that is equal to the last $m-k$ characters of the pattern then the vector R^k will report found pattern with at most k mismatches after reading $m-k$ input character, but it is clear that the pattern can be found after reading at least m input characters.

Because of the problems at the beginning of the input text and at the end of the input text we can say that the vector R_i^j can report that the pattern can be found with at most k mismatches only if $m-k+j \leq i \leq n-k+j$.

This simplification reduces the length of the state vectors R^j and simplifies the formula for computation of the state vectors with one or more mismatches. One operation shl is omitted, but on the other hand a new operation appears. This operation is shift of one column of the mask table D. This operation can be omitted too, if we accept higher space complexity of characteristic vector representation.

3 String Matching with k Differences

In string matching with k differences there are two new editing operations. The new editing operations are *insert* and *delete*. The operation *insert* puts some character in a text and the operation *delete* removes some character from a text. It is clear that

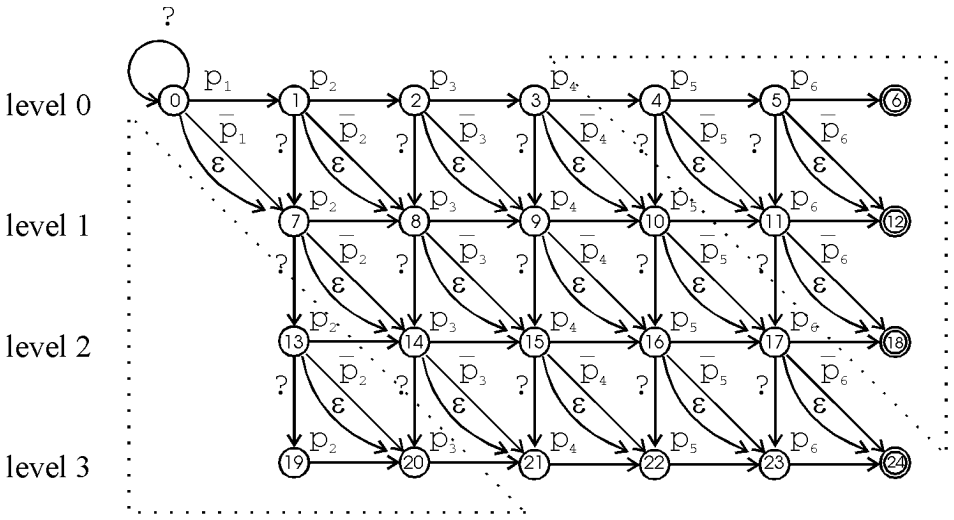


Figure 2: *NFA* for approximate string matching with k differences.

after adding these two editing operations into the set of editing operations, the string found with k differences need not be of the same length as the pattern P.

3.1 Nondeterministic Finite Automaton

A *NFA* for string matching with k differences for the pattern $P = p_1 p_2 \cdots p_m$, as it was presented in [Me96-1], is shown in Figure 2. In this figure $m = 6$ and $k = 3$. The sequence of states of level 0 of the *NFA* contains states that correspond to the given pattern without any differences, the sequence of states of level 1 contains states that correspond to the given pattern with one difference, ...etc. At the end of each level there is a final state. This state says that the pattern was found with 0,1,...etc. differences, respectively. There are two new arrows. The first is directed to the down and represents editing operation *insert*. The second one is directed to the right-down and represents ε transition of editing operation *delete*. This *NFA* accepts all strings having a postfix X such that $D_L(P, X) \leq k$.

The number of states of the *NFA* for string matching with k differences is $m * (k + 1) + 1$.

The initial state of *NFA* in Figure 2 is state 0, but also, as presented in [HU79], all states to which *NFA* can move from the initial state without reading any input character are also initial states. So initial state includes all states that are located on the diagonal starting from the state 0. Since *NFA* is all the time also in initial state it is in states 0, 7, 14 and 21. At the beginning of the *NFA* there are several states bordered by the dotted line. The *NFA* moves to all these states after reading the first $k - 1$ input characters. Then the *NFA* stays all the time also in these states but the *NFA* can move from these states only in initial states and so these states are redundant.

The *NFA* can move to these states by transitions representing editing operation *insert*. So these states represent situations that at most $k-1$ characters were inserted before the string and we do not care how many characters were inserted before the string.

If we denote editing operation *replace* r, *insert* i, *delete* d and matching m we can

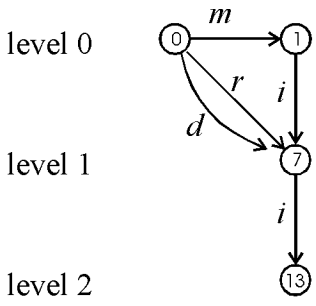


Figure 3: The transitions in *NFA* for approximate string matching with k differences.

describe ways how to move to these states by these four characters. For example we can move to the state numbered by 13 by following sequences of operations: $d+i$ or $r+i$ or $m+i+i$ as it is shown in Figure 3.

The sequence of operations *delete* and *insert* has the same result as operation *replace* so we can write $d+i=r$. The sequence of operations *replace* and *insert* has the same result as sequence of operations *insert* and *replace* so we can write $r+i=i+r$. The sequence of operations *matching* and *insert* leading from some of initial states has the same result as operations *insert* and *replace* so we can write $m+i=i+r$.

Now we can rewrite the sequences of operations how to move to state numbered by 13: $d+i=r$, $r+i=i+r$, $m+i+i=i+r+i=i+i+r$. If we look at these sequences of operations we can see that all the sequences of operations needed to move to the state numbered by 13 can be replaced by operation *replace* because we do not care about the characters inserted before the string so we do not need the state numbered by 13.

Such replacing of sequences of operations can be done for all states bordered by the dotted line so all states inside the bordered area are redundant and can be omitted.

The *NFA* for approximate string matching with k differences reduced by the way described above has the same number of states as the *NFA* for approximate string matching with k mismatches and it is $(k+1)(m+1-\frac{k}{2})$ states, as shown in section 2.1. It is clear that only reduced *NFA* for approximate string matching with k differences, that have $k>1$, have less states then nonreduced *NFA*.

The *NFA* for approximate string matching with k differences reduced as described above is shown in Figure 4.

3.2 Reducing the number of states

The simplification described in section 2.2 for approximate string matching with k mismatches can be applied to approximate string matching with k differences as well.

The states that are needed only to recognize how many differences are in a found string, form also a right angle triangle in upper right corner of the *NFA*, as marked by the dotted line in Figure 4. On the opposite side of the *NFA* there is the complement triangle of states omitted because of the reduction. If we omit these two triangles we obtain a simplified *NFA* with $(k+1)(m+1-k)$ states.

Now final states have also been changed. If the *NFA* is in final state at the end

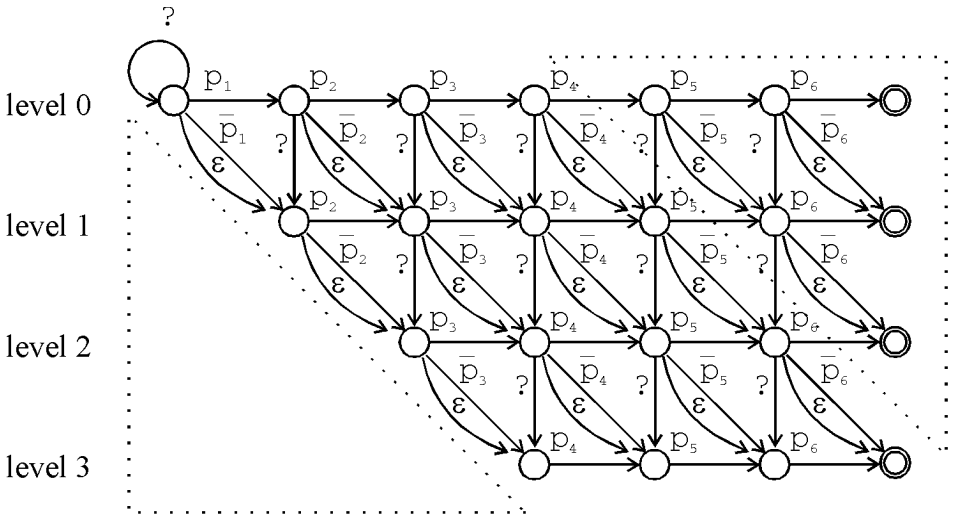


Figure 4: Reduced *NFA* for approximate string matching with k differences.

of level j it means that the pattern can be found with at least j and at most with k differences. The problem that appears at the end and at the beginning of the input text in approximate string matching with k mismatches does not exist in the case of approximate string matching with k differences. It is due to ε transitions that the *NFA*, to move from one state to another, does not need to read any input character. ε transitions represent editing operation *delete* so we can delete the first k characters or the last k characters of the pattern. That is why this simplification does not need the limits used in case of approximate string matching with k mismatches.

3.3 Shift-Or Algorithm

The Shift-Or algorithm for string matching with k differences has two new items in formulae for computing vectors R^j. The new items of the formula are $shl(R^{j-1}_{i+1})$, representing a transition of the editing operation *delete*, and R^{j-1}_i, representing a transition of editing operation *insert*. The formula for string matching with k differences is $R^j_{i+1} = (shl(R^j_i) or (D[t_{i+1}])) and (shl(R^{j-1}_i)) and (shl(R^{j-1}_{i+1})) and (R^{j-1}_i)$. It can be reduced to $R^j_{i+1} = (shl(R^j_i) or (D[t_{i+1}])) and (shl(R^{j-1}_i and\ R^{j-1}_{i+1})) and (R^{j-1}_i)$. At the beginning of searching the vectors R^j are filled up in such a way, that first j bits from the left contain 0s and other bits contain 1s.

In the case of approximate string matching with k differences we do not omit any 0s at the beginning of the table as in the case of approximate string matching with k mismatches.

The third item, $shl(R^{j-1}_{i+1})$, represents the editing operation *delete*. If there is some character deleted, we have to skip it and continue behind it. In the *NFA* in Figure 4, the skipping of the deleted character is marked by ε transition and continuation is marked by transition representing matching. In the Shift-Or algorithm there are those two operations in an inverted order. At first the operation representing matching is executed. It was already executed in computing of the vector R^{j-1}_{i+1}. By that we got to the next character in the pattern P. The following operation is ε transition. It is represented by operation *shl* that gets us to the next character in the pattern P and to the level of the states corresponding to differences one higher. It seems the

case of deleting the first character in the pattern P was not involved. But that is only an illusion. Such a case is covered by an initial setting of the vectors R^j. At the beginning of searching the vectors R^j are filled up in such a way that the first j bits from the left contain 0s and other bits contain 1s. The initial filling up by 0s is given by the ε transitions coming from the initial state with a self loop. It means that at the beginning of searching there are j deleted characters, $0 \leq j \leq k$.

The fourth item in the formula is R_i^{j-1}, that represents the editing operation *insert*. In Figure 4, the transition representing operation *insert* is marked by an arrow directed down. It means that the state in the *NFA* stays in the same depth but moves to the level of the states corresponding to one more differences.

3.4 Simplified Shift-Or Algorithm

The new *NFA* can be described also by vectors R^j, that are k bit shorter then the original ones. The new formulae are $R_{i+1}^0 = shl(R_i^j) or (D_j[t_{i+1}])$ for exact matching and $R_{i+1}^j = (shl(R_i^j) or (D_j[t_{i+1}])) and\ R_i^{j-1} and\ R_{i+1}^{j-1} and (shr\ R_i^{j-1})$ for j differences, where shr is bitwise operation right shift. The mask table D_j is also a part of mask table D being $m-k$ bit long and starting at position j in D. The ways how to represent mask tables D_j are the same as for approximate string matching with k mismatches.

This simplification reduces the length of the state vectors R^j but does not simplify the formula for computation of the state vectors with one or more differences as in previous section.

Conclusions

In this article we have shown that in the case that we are not interested in knowing the number of errors in the found string we can reduce *NFA* for aproximate string matching. In the case of approximate string matching with k mismatches this reduction not only reduces length of vectors of Shift-Or based algorithms but also simplifies formulae for computing these vectors. In the case of approximate string matching with k differences it only reduces length of the vectors.

Another way how to use *NFA* is to transform it into deterministic finite automaton. Decrease of states in reduced deterministic finite automata is described in [Me96-2].

References

[BG92] Baeza-Yates, R., Gonnet, G. H.: A new approach to text searching. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 74 – 82.

[Da92] Darabont, T.: Approximate string matching with k mismatches. Master's thesis, Czech Technical University, 1992.

[Ho96] Holub, J.: Approximate string matching in text. Master's thesis, Czech Technical University, 1996.

[HU79] Hopcroft, J. E., Ullman, J. D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, Massachusetts.

[Me95] Melichar, B.: Approximate string matching by finite automata. Computer Analysis of Images and Patterns, LNCS 970, Springer, Berlin 1995, pp. 342 – 349.

[Me96-1] Melichar, B.: String matching with k differences by finite automata. Proceedings of the 13th ICPR, Vol. II, August 1996, pp. 256 – 260.

[Me96-2] Melichar, B.: Space Complexity of Linear Time Approximate String Matching. In this volume.

[WM92] Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 83 – 91.

Space Complexity of Linear Time Approximate String Matching

Bořivoj Melichar

Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University,
Karlovo nám. 13,
121 35 Prague 2,
Czech Republic

e-mail: melichar@cs.felk.cvut.cz

Abstract. Approximate string matching is a sequential problem and therefore it is possible to solve it using finite automata. Nondeterministic finite automata are constructed for string matching with k mismatches and k differences. The corresponding deterministic finite automata are base for approximate string matching in linear time. Then the space complexity of both types of deterministic automata is calculated. Moreover, reduced versions of nondeterministic automata are taken into account and the space complexity of their deterministic equivalents is calculated.

Key words: approximate string matching, finite automata, space complexity

1 Introduction

Approximate string matching can be described in the following way:
Given a text string $T = t_1 t_2 \cdots t_n$, a pattern $P = p_1 p_2 \cdots p_m$, and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of a substring X in the text string T such that the distance $D(P, X)$ between the pattern P and the string X is less than or equal to k. In this paper we will consider two types of distances called Hamming distance and Levenshtein distance.

The Hamming distance, denoted by D_H, between two strings P and X of equal length is the number of positions with mismatching symbols in the two strings. We will refer to approximate string matching as *string matching with k mismatches* whenever D is the Hamming distance. The Levenshtein distance, denoted by D_L, or edit distance, between two strings P and X, not necessarily of equal length, is the minimal number of editing operations **insert**, **delete** and **replace** needed to convert P into X. We will refer to approximate string matching as *string matching with k differences* whenever D is the Levenshtein distance.

Approximate string matching is a sequential problem and therefore it is possible to solve it using finite automata. Two variants of nondeterministic finite automata are constructed for string matching with k mismatches and for string matching with k differences ([Me95], [Me96]).

There are two ways how to use these automata as a base for the matching algorithm:

1. To simulate the nondeterministic automaton in a deterministic way.

2. To construct an equivalent deterministic automaton.

Several known algorithms use simulation of nondeterministic automata in a deterministic way [BG92], [MW92], [Uk85], [WM92]. The simulation leads to the time complexity which is greater than linear. The only exception are SHIFT-OR based algorithms ([BG92], [WM92]) which simulate the nondeterministic automata in linear time for small m and k using bit vectors. The advantage of the simulation of nondeterministic automata is the low space complexity.

Use of deterministic finite automata leads to the linear time complexity for all m and k. The drawback of this approach is a high expected space complexity. Therefore we try to find the space complexity of deterministic finite automata for matching which is less pessimistic than in [Uk95].

A nondeterministic finite automaton (NFA) is a 5 - tuple $M = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is a finite set of input symbols, δ is a state transition function from $Q \times (A \cup \{\varepsilon\})$ to the power set of Q, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states.

A finite automaton is deterministic (DFA) if $\delta(q, a)$ has exactly one element for any $q \in Q$ and $a \in A$ and $\delta(q, \varepsilon) = \emptyset$ for any $q \in Q$.

In the following, we will use the alphabet $A = \{s_1, s_2, \cdots, s_{|A|}\}$.
If $p \in A$ then $\bar{p}$ is the complement set $A - \{p\}$, in our case.

2 String Matching with k Mismatches

First, we construct a nondeterministic finite automaton M_H for a given pattern $P = p_1 p_2 \cdots p_m$, alphabet $A = \{s_1, s_2, \cdots, s_{|A|}\}$, and $k \leq m$. This automaton is depicted in Fig. 1.

Each state $q \in Q$ has a label (i, j), where i, $0 \leq i \leq k$, is a level of q, and j, $0 \leq j \leq m$, is a depth of q. In the automaton M_H, there are $k + 1$ levels of states sequences. Every level ends in one of the final states $(0, m), (1, m), \cdots, (k, m)$. These final states are accepting states of strings with $0, 1, 2, \cdots, k$ mismatching symbols, respectively. The sequence of states of the level 0 corresponds to the given pattern without any mismatch. Levels $1, 2, \cdots, k$ correspond to the strings with $1, 2, \cdots, k$ mismatching symbols, respectively. From each nonfinal state of level j, $0 \leq j < k$, there exists a transition to the state of the level $j + 1$, which means, that a mismatch occurs. Moreover, there is a self loop in the state $(0, 0)$ for every symbol of the alphabet A. This automaton accepts all strings having a postfix X such that $D_H(P, X) \leq k$. The number of states of the automaton M_H is
$(k + 1)(m + 1 - \frac{k}{2}) = (m + 1) + (m) + (m - 1) + \cdots + (m - k + 1)$.

Because this finite automaton is nondeterministic, it is necessary to construct an equivalent deterministic finite automaton (DFA_H) using the standard algorithm [AU71,2].

Let us use the number of items of the transition table of DFA_H as a measure of the space complexity of the algorithm of string matching with k mismatches. This

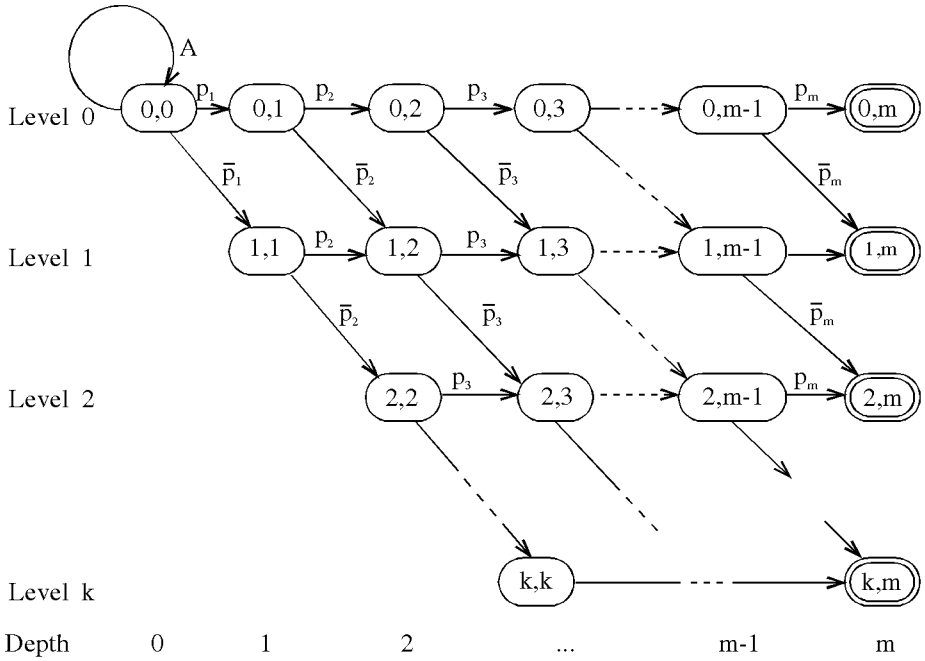


Figure 1: Nondeterministic finite automaton M_H.

number of items is the number of states (number of rows) of DFA_H multiplied by number of columns.

For the evaluation of the number of states of the DFA_H in question we will use the following lemma.

Lemma 1 *Let M_H be a nondeterministic finite automaton for given pattern $P = p_1 p_2 \cdots p_m$, $k \leq m$ (see Fig. 1). Let DFA_H be the deterministic finite automaton constructed by the standard algorithm for M_H. Then each state of the automaton DFA_H contains at most one state of M_H with depth j, $0 \leq j \leq m$.*

Proof. The standard construction of the deterministic finite automaton equivalent to the nondeterministic one is based on the "parallel simulation" of the nondeterministic automaton.

The assertion of the lemma can be formulated in this way:

(∗) The nondeterministic automaton M_H can reach at most one state at each depth during parallel simulation.

This assertion can be proven by induction on the length of input string l, $0 \leq l \leq n$. For $l = 0$ assertion (∗) holds, because M_H is in state $(0, 0)$. Let us assume, that assertion (∗) is true for all $l \leq n'$. That means that M_H is in some number of states $(i_1, j_1), (i_2, j_2), \cdots, (i_q, j_q)$, where all $j_r, 0 \leq r \leq q$, are different. For the state (i_r, j_r), $j_r < m, i_r < k$, there are two possible transitions:

1. to the state (i_r, j_{r+1}) in case when the input symbol matches the symbol p_{r+1} of the pattern,

2. to the state (i_{r+1}, j_{r+1}) in case when no match occurs.

For the state $(0,0)$ there is moreover the selfloop. For the state (i_r, j_r), where $i_r = k$, there is possible only transition when match occurs. For the state (i_r, j_r), where $j_r = m$, there is no transition possible. From this follows that at most one state in the depth j_{r+1} will be reached from each state (i_r, j_r) and the assertion holds. This completes the proof of the lemma.

□

From the Lemma 1 follows the method of computation of the maximum number of states of the deterministic automaton DFA_H.

There are $p+1$ states in each depth p of the nondeterministic automaton M_H for $0 \leq p \leq k-1$. Moreover, there are $k+1$ states in M_H for each depth p, $k \leq p \leq m$.

The number of subsets of these states can be computed as a product of numbers of states of all depths between 1 and m increased by one, for the case, when no state of particular depth is present in the subset. Therefore the maximum number of states of the deterministic automaton DFA_H is

$$3 * 4 * \cdots * (k+1) * (k+2)^{m-k+1} = \frac{(k+1)!}{2} * (k+2)^{m-k+1}.$$

The number of states of DFA_H is

$$\mathcal{O}((k+1)! * (k+2)^{m-k+1}).$$

For the computation of the number of columns of the transition table of the DFA the following lemma is useful.

Lemma 2 *Let $P = p_1 p_2 \cdots p_m$ be a pattern. Let $M_H = (Q, A, \delta, q_0, F)$ be nondeterministic automaton for P and $k \geq 0$. Let $X = \{x : x \in A, x \neq p_i, 1 \leq i \leq m\}$. Then for a deterministic automaton $DFA_H = (Q_D, A, \delta_D, q_{0D}, F_D)$ constructed for M_H holds: for all $q \in Q_D$ exists $p \in Q_D$ such that $\delta_D(q, x) = p$ for all $x \in X$.*

Proof. The set X is a subset of A containing symbols not used in the pattern P. The automaton M_H has for all symbols $x \in X$ identical columns in the transition table. Thus $\delta(q, x) = \{p_1, p_2, \cdots, p_r\}$ and $\delta(q, y) = \{p_1, p_2, \cdots, p_r\}$ holds for all $q \in Q$ and all pairs $x, y \in X$. Due to the construction of the deterministic automaton, for $q \in Q_D$ and $q = \{q_1, q_2, \cdots, q_S\}$, it holds

$$\delta_D(\{q_1, q_2, \cdots, q_S\}, x) = \bigcup_{i=1}^{s} \delta(q_i, x)$$

and

$$\delta_D(\{q_1, q_2, \cdots, q_S\}, y) = \bigcup_{i=1}^{s} \delta(q_i, y).$$

Because
$\delta(q_i, x) = \delta(q_i, y)$, $1 \leq i \leq S$, then $\delta_D(\{q_1, q_2, \cdots, q_S\}, x) = \delta_D(\{q_1, q_2, \cdots, q_S\}, y)$.

□

From this lemma the consequence follows: If the pattern has length m then no more than m different symbols from an alphabet A may appear in it. For all other symbols, both deterministic and nondeterministic automata behave in the same way. It means, that the subset $X \subset A$ of symbols not present in the pattern may be replaced by some $x \in X$ and the size of alphabet will be $m+1$.

From it follows, that the space complexity of the deterministic automaton DFA does not depend on the size of alphabet if it is large enough.

It follows from this discussion, that the number of columns of the transition table of DFA is $a = min(|A|, m+1)$.

The total space complexity of deterministic automaton DFA_H is:

$$\mathcal{O}((k+1)!(k+2)^{m-k+1} * min(|A|, m+1)).$$

The nondeterministic automaton M_H can be reduced as described in [Ho96]. This reduction leads to the nondeterministic automaton RM_H having just $(m+1-k)$ states at each level. States

$$\begin{array}{l}(0, m-k), (0, m-k+1), \cdots, (0, m),\\ (1, m-k+1), (1, m-k+2), \cdots, (1, m),\\ \vdots\\ (k-1, m)\end{array}$$

can be removed when we need not know the number of mismatches in the found string. Moreover states $(0, m-k+1), (1, m-k), \cdots, (k, m)$ will be final states.

Because Lemma 1 is valid for RM_H as well as for M_H, it is possible to use similar approach for computation of maximum number of states of deterministic finite automaton $RDFA_H$ constructed for RD_H. Let us assume $k \leq \frac{m}{2}$. In this case the automaton RM_H has $p+1$ states in each depth p for $0 \leq p \leq k+1$. There are $k+1$ states in each depth $p, k \leq p \leq m-k-1$. Moreover, there are $m-p+1$ states in RM_H for each depth p, $m-k+1 \leq p \leq m$.

From this follows, that the maximum number of states of the reduced deterministic automaton $RDFA_H$ is

$$NS(RDFA_H) = 3 * 4 * \cdots * (k+1) * (k+2)^{m-2k+1} * (k+1) * \cdots * 3 * 2 =$$

$$= \frac{1}{2}((k+1)!)^2 * (k+2)^{m-2k+1}..$$

If $\frac{m}{2} < k < m$ then the situation is different. In this case the maximal number of states of RM_H in one depth is lower than $k+1$ and it is equal to $m-k+1$. The expression $NS'(RDFA_H)$ for the evaluation of number of states of deterministic automaton has the form:

$$NS'(RDFA_H) = 3 * 4 * \cdots * (m-k+1) * (m-k+2)^{2k-m+1}(m-k+1) * \cdots * 3 * 2 =$$

$$= \frac{1}{2}((m-k+1)!)^2 * (m-k+2)^{2k-m+1}$$

Because Lemma 2 is also valid for reduced automaton RM_H, the space complexity of the reduced deterministic finite automaton $RDFA_H$ is:

$$\mathcal{O}(((k+1)!)^2 * (k+2)^{m-2k+1} * min(|A|, m+1)),$$

when $k < \frac{m}{2}$ and

$$\mathcal{O}(((m-k+1)!)^2 * (m-k+2)^{2m-k+1} * min(|A|, m+1))$$

when $\frac{m}{2} < k < m$.

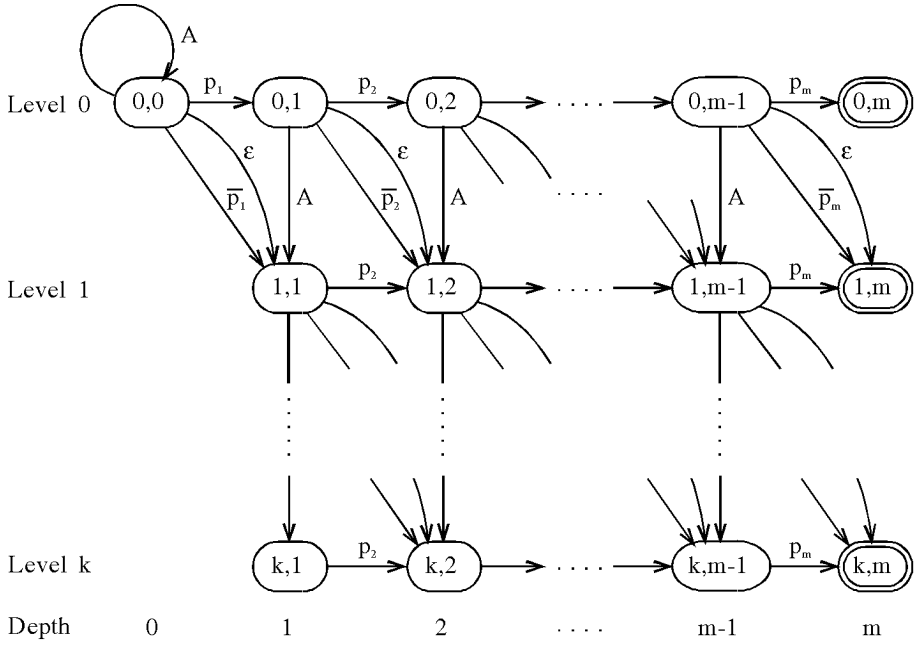


Figure 2: Nondeterministic finite automaton M_L.

3 String Matching with k Differences

We will construct a nondeterministic finite automaton M_L for a given pattern $P = p_1 p_2 \cdots p_m$, alphabet $A = \{s_1, s_2, \cdots, s_{|A|}\}$, and $k \leq m$. This automaton is depicted in Fig. 2. Each state $q \in Q$ has a label (i, j), where $i, 0 \leq i \leq k$, is the level of q, and $j, 0 \leq j \leq m$, is the depth of q.

The automaton is composed of $k+1$ levels of state sequences. Every level ends in one of the final states $(0, m), (1, m), \cdots, (k, m)$ which accept strings with $0, 1, \cdots, k$ differences, respectively. In each level, with exception of the level 0, there are m states with depth $1, 2, \cdots, m-1, m$, where the depth of a state is its "distance" from the state $(0, 0)$ of level 0. In the level 0, there are $m+1$ states and the state $(0, 0)$ has the depth equal to 0.

The transitions between adjacent levels correspond to the edit operations **insert**, **replace** and **delete** in the following way:

1. The transitions corresponding to the operation **insert** are "vertical" transitions from each nonfinal state of level j, $0 \leq j < k$, with the exception of the initial state, to the state of level $j+1$ with the same depth for all symbols of the alphabet A.

2. The transitions corresponding to the operation **replace** are "diagonal" transitions from each nonfinal state (i, j) of level j, $0 \leq j < k$ to the state $(i+1, j+1)$ of level $j+1$. The label of such transition is the complement of the label of transition from state (i, j) to state $(i, j+1)$.

3. The transitions corresponding to the operation **delete** are "diagonal" ε-transitions from each nonfinal state (i, j) of level j, $0 \leq j < k$, and depth less than m to the state $(i+1, j+1)$ of the level $j+1$.

Finally, there are self loops in the state $(0,0)$ for all symbols of the alphabet A. This automaton accepts all strings with postfix X such that $D_L(P,X) \leq k$. The automaton has $m(k+1)+1$ states.

As in the case of the automaton for mismatching problem, we will construct an equivalent deterministic automaton DFA_L for M_L.

The approach we use for evaluation of the space complexity of the DFA_L is based on the notion of ε-diagonals. Because we can leave out ε-diagonals below the initial ε-diagonal starting with state $(0,0)$ for reasons described in [Ho96], the nondeterministic automaton M_L contains $(m+1)$ ε-diagonals containing the following states:

number of diagonal	set of states	number of states
0	$(0,0),(1,1),\cdots,(k,k)$	$k+1$
1	$(0,1),(1,2),\cdots,(k,k+1)$	$k+1$
2	$(0,2),(1,3),\cdots,(k,k+2)$	$k+1$
$\vdots$		
$m-k$	$(0,m-k),(1,m-k+1),\cdots,(k,m)$	$k+1$
$\vdots$		
$m-2$	$(0,m-2),(1,m-1),(2,m)$	3
$m-1$	$(0,m-1),(1,m)$	2
m	$(0,m)$	1

For the computation of number of states of the equivalent deterministic automaton DFA_L for M_L we will use the following lemma.

Lemma 3 *Let M_L be a nondeterministic finite automaton for given pattern $P = p_1p_2\cdots p_m$, $k \leq m$ (see Fig. 2). Let DFA_L be the deterministic finite automaton constructed by the standard algorithm for M_L. Then each state of the automaton DFA_L contains at most one state of M_L from each ε-diagonal.*

Proof. The proof is based on the observation, that if automaton M_L reaches some state (p,q) at the ε-diagonal d, then, due to ε-transitions, it reaches all next states $(p+1,q+1),(p+2,q+2),\cdots,(p+k,q+k)$ of the same ε-diagonal. Therefore we can select such state (p,q) of each diagonal, where the level p is minimal, as a "representative" of the set of all next states at the same ε-diagonal.

□

The number of states of deterministic automaton DFA_L for M_L we can compute as a product of the number of states at diagonals $1,2,\cdots,m$ increased by one because some states of DFA_L may contain no state from particular diagonal. The diagonal 0 plays special role, because automaton M_L is always in the state $(0,0)$ due to the selfloop in the initial state.

Because the number of "full" ε-diagonals having length $k+1$ others than diagonal 0 is $m-k$ and there is k "short" diagonals having length $k,k-1,\cdots,1$, respectively, the number of different subsets of the representatives (the maximum number of states of DFA_L) is given by

$$NS(DFA_L) = (k+2)^{m-k} * \frac{(k+1)!}{2}$$

Using the previous result on number of rows of transition table we can express the space complexity of the deterministic finite automaton for approximate string matching with k-differences as:

$$\mathcal{O}((k+2)^{m-k} * (k+1)! * min(|A|, m+1)).$$

The nondeterministic automaton M_L can also be reduced as described in [Ho96]. This reduction leads to the reduced automaton RM_L having "full" ε-diagonals only. The number of states of reduced deterministic automaton $RDFA_L$ we can express as

$$NS(RDFA_L) = (k+2)^{m-k}$$

and the space complexity of the reduced deterministic automaton for approximate string matching with k-differences is

$$\mathcal{O}((k+2)^{m-k} * min(|A|, m+1)).$$

4 Conclusion

The main result presented here is upper bound of space complexity of four variants of deterministic finite automata for approximate string matching. While the number of states of nondeterministic finite automata is $O(k * m)$ in all cases, the number of states of corresponding deterministic automata is much lower then $O(2^{k*m})$. In the case of string matching with k differences, the presented space complexity is still pessimistic and the computing of more realistic upper bound is open problem.

References

[AU71,2] Aho, A., Ullman, J,: The theory of parsing, translation and compiling. Vol. I: Parsing, Prentice Hall, Englewood Cliffs, New York 1972.

[BG92] Baeza-Yates, R., Gonnet, G. H.: A new approach to text searching. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 74 – 82.

[Ho96] Holub, J.: Reduced nondeterministic finite automata for approximate string matching. In this volume.

[Me95] Melichar, B.: Approximate string matching by finite automata. Computer Analysis of Images and Patterns, LNCS 970, Springer, Berlin 1995, pp. 342 – 349.

[Me96] Melichar, B.: String matching with k differences by finite automata. Proceedings of the 13th ICPR, Vol. II, August 1996, pp. 256 – 260.

[MW92] Manber, U., Wu, S.: Approximate pattern matching. BYTE, November 1992, pp. 281 – 292.

[Uk85] Ukkonen, E.: Finding approximate patterns in strings. Journal of algorithms 6, 1985, pp. 132 – 137.

[WF74] Wagner, R., A., Fischer, M., J.: The string-to-string correction problem. Journal of the ACM, January 1974, Vol. 21, No. 1, pp. 168 – 173.

[WM92] Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 83 – 91.

Approximate Regular Expression Matching

Pavel Mužátko

Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University,
Karlovo nám. 13,
121 35 Prague 2,
Czech Republic

Abstract. We extend the definition of Hamming and Levenshtein distance between two strings used in approximate string matching so that these two distances can be used also in approximate regular expression matching. Next, the methods of construction of nondeterministic finite automata for approximate regular expression matching considering both mentioned distances are presented.

Key words: regular expression, finite automata, approximate string matching

1 Introduction

The notions from the theory of approximate string matching will be used for describing the problem of approximate regular expression matching. Approximate string matching is defined as follows: A text T, a pattern P, and an integer k are given. All occurrences of a substring X should be found such that the distance $D(P, X)$ between the string X and the pattern P is less or equal to k.

There are two basic types of distances called Hamming distance and Levenshtein distance. The Hamming distance (notation D_H) between two strings of equal length is the number of positions with mismatching symbols in this two strings. The Levenshtein or edit distance (notation D_L) between two strings P and X, not necessarily of equal length, is the minimal number of editing operations insert, delete, and replace needed to convert P into X.

If the Hamming distance is used then the approximate string matching is referred as string matching with k mismatches. If the Levenshtein distance is used then the approximate string matching is referred as string matching with k differences. Similarly, the notions regular expression matching with k mismatches and regular expression matching with k differences will be used.

2 Definition of Regular Expressions

Definition 1
A regular expression V over an alphabet A is defined as follows:

1. $\emptyset, \varepsilon, a$ are regular expressions for all $a \in A$.

2. If x, y are regular expressions over A then:

   (a) $(x + y)$ (union)

   (b) $(x.y)$ (concatenation)

   (c) $(x)^*$ (closure)

   are regular expressions over A.

Definition 2
A value h(x) of a regular expression x is defined as follows:

1. $h(\emptyset) = \emptyset, h(\varepsilon) = \{\varepsilon\}, h(a) = \{a\}$,

2. $h(x + y) = h(x) \cup h(y)$,
   $h(x.y) = h(x).h(y)$,
   $h(x^*) = (h(x))^*$.

The value of a regular expression is a regular language, a set of patterns. Unnecessary parentheses in regular expressions can be avoided by the convection for precedence of regular operations. The highest precedence has the closure operator, the lowest precedence has the union operator.

3 Regular Expression Matching with k Mismatches

The Hamming distance D_H^R between a regular expression V with a value $h(V)$ and a string X can be defined by using the Hamming distance D_H between two strings as follows:
$D_H^R = \min_{w \in h(V) \wedge |w| = |X|} D_H(w, X)$

Now, the construction of a nondeterministic finite automaton accepting patterns with the postfix generated by a given regular expression with k mismatches is presented.

Let a regular expression V over an alphabet A is given, and $M = (Q_0, A, q_0, \delta_0, F_0)$ is a nondeterministic finite automaton accepting the language $L = h(V)$. Let the automaton M has m states. The automaton $M_H^R = (Q, A, q_0, \delta, F)$ accepting patterns with the postfix from $h(V)$ with k mismatches will be constructed by interconnecting the $k + 1$ clones $M_0, ..., M_k$ of the automaton M.
Each state of the automaton M_H^R is labeled by $q_{i,j}$, where i is the number of the clone, $0 \leq i \leq k$, j is the number of the state inside the clone M_i, $0 \leq j \leq m - 1$. The mapping δ of the automaton M_H^R will be defined in the following way:

1. All transitions defined in the automata $M_0, ..., M_k$ will be also included in the automaton M_H^R.

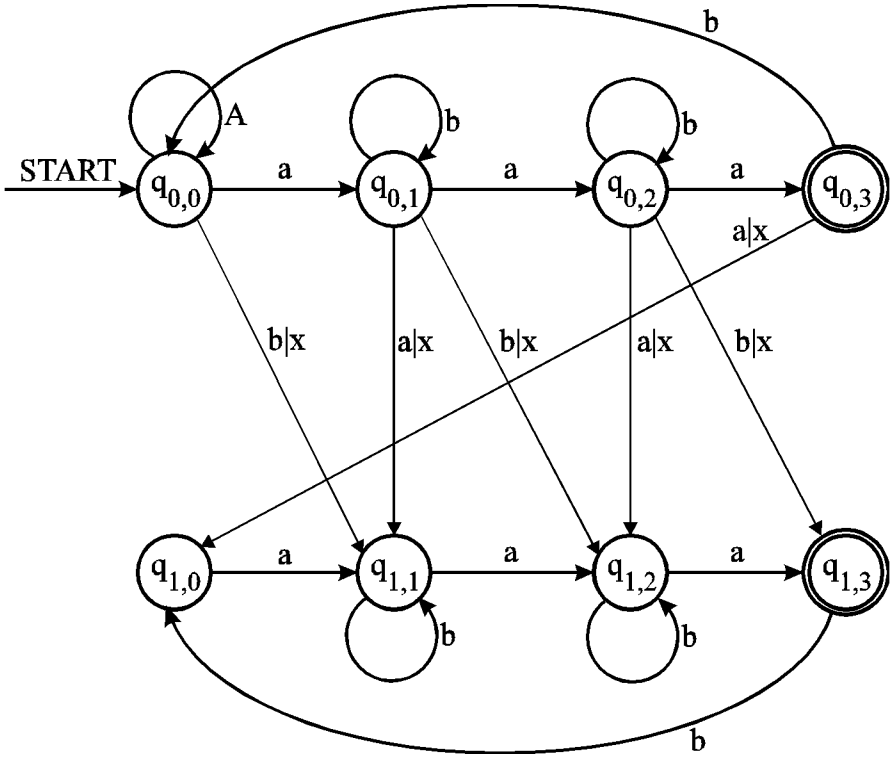


Figure 1: Nondeterministic automaton H_1.

2. Error transitions will be added. For each state $q_{i,j}$ $(0 \leq i \leq k-1,\ 0 \leq j \leq m-1)$ and for each such a symbol $a \in A$, for which $\delta(q_{i,j}, a)$ is defined, define $\delta(q_{i,j}, \overline{a}) = \delta(q_{i+1,j}, a)$, where $\overline{a}$ denotes all symbols from the alphabet A except the symbol a.

3. A self loop for all symbols from the alphabet A will be added for the state $q_{0,0}$.

The initial state of the automaton M_H^R is the state $q_{0,0}$. The set of final states $F = F_0 \cup F_1 \cup ... \cup F_k$.

The number of states of the automaton M_H^R is $m(k+1)$.

Example 1

A transition diagram of a nondeterministic automaton H_1 accepting with 1 mismatch patterns with the postfix described by the regular expression $V = ab^*ab^*a(bab^*ab^*a)^*$ over the alphabet $A = \{a, b, x\}$ can be found in Fig. 1. This automaton accepts all strings with a postfix X such that $D_H^R(V, X) \leq 1$. The result of searching in the text $aabxabaa$ can be described as follows: $aab_{(1)}x_{(1)}a_{(1)}ba_{(1)}a_{(0,1)}$. The number in parentheses shows the number of mismatches occurred when a final state of the automaton H_1 was reached.

4 Regular Expression Matching with k Differences

The Levenshtein distance D_L^R between a regular expression V with a value $h(V)$ and a string X can be defined by using the Levenshtein distance D_L between two strings

as follows:
$D_L^R = \min_{w \in h(V)} D_L(w, X)$

Let V be again a regular expression over an alphabet A and M is a nondeterministic finite automaton accepting the language $L = h(V)$. We will construct a nondeterministic finite automaton M_L^R accepting with k differences all patterns with the postfix from $h(V)$. This automaton will be as in the previous case constructed by interconnecting the $k+1$ clones $M_0, ..., M_k$ of the automaton M.
Each state of the automaton M_L^R is again labeled by $q_{i,j}$, where i is the number of the clone, $0 \leq i \leq k$, j is the number of the state inside the clone M_i, $0 \leq j \leq m-1$. The mapping δ of the automaton M_L^R is defined in the following way:

1. All transitions defined in the automata $M_0, ..., M_k$ will be also included in the automaton M_L^R.

2. Replace transitions will be added. For each state $q_{i,j}$ $(0 \leq i \leq k-1,\ 0 \leq j \leq m-1)$ and for each such a symbol $a \in A$, for which $\delta(q_{i,j}, a)$ is defined, define $\delta(q_{i,j}, \overline{a}) = \delta(q_{i+1,j}, a)$, where $\overline{a}$ denotes all symbols from the alphabet A except the symbol a.

3. Delete transitions will be added. For each state $q_{i,j}$ and for each symbol $a \in A$ $(0 \leq i \leq k-1,\ 0 \leq j \leq m-1)$ $\delta(q_{i,j}, \varepsilon) = \delta(q_{i+1,j}, a)$.

4. Insert transitions will be added. For each state $q_{i,j}$ $(0 \leq i \leq k-1,\ 0 \leq j \leq m-1)$, and for each symbol $a \in A$ $\delta(q_{i,j}, a) = q_{i+1,j}$. All replace transitions between states, where insert transitions are also defined (e.g. the replace transitions between the states $q_{i,j}$ and $q_{i+1,j}$), can be removed.

5. A self loop for all symbols from the alphabet A will be added for the state $q_{0,0}$.

The initial state of the automaton M_L^R is the state $q_{0,0}$. The set of final states $F = F_0 \cup F_1 \cup ... \cup F_k$.

The number of states of the automaton M_L^R is $m(k+1)$.

Example 2

A transition diagram of a nondeterministic automaton L_1 accepting with maximaly 1 difference patterns with the postfix defined by the regular expression $V = ab^*ab^*a(bab^*ab^*a)^*$ over the alphabet $A = \{a, b, x\}$ can be found in Fig. 2. Delete transitions are depicted as dashed lines. This automaton accepts all strings with a postfix X such that $D_L^R(V, P) \leq 1$.

The result of searching in the text $abxaa$ can be described as follows:
$abxa_{(R)}a_{(R,I)}$.
The symbol in parentheses determines the operation needed to convert some pattern from $h(V)$ to the string read when a final state was reached.

The notation (R, I) has the following meaning:
The string $abbaa \in h(V)$ can be converted to the string $abxaa$ by using one replace operation. The string $abaa \in h(V)$ can be converted to the string $abxaa$ by using one insert operation.

Example 3

Let us consider the input text $abbbabab$. We are interested in finding all occurrences of

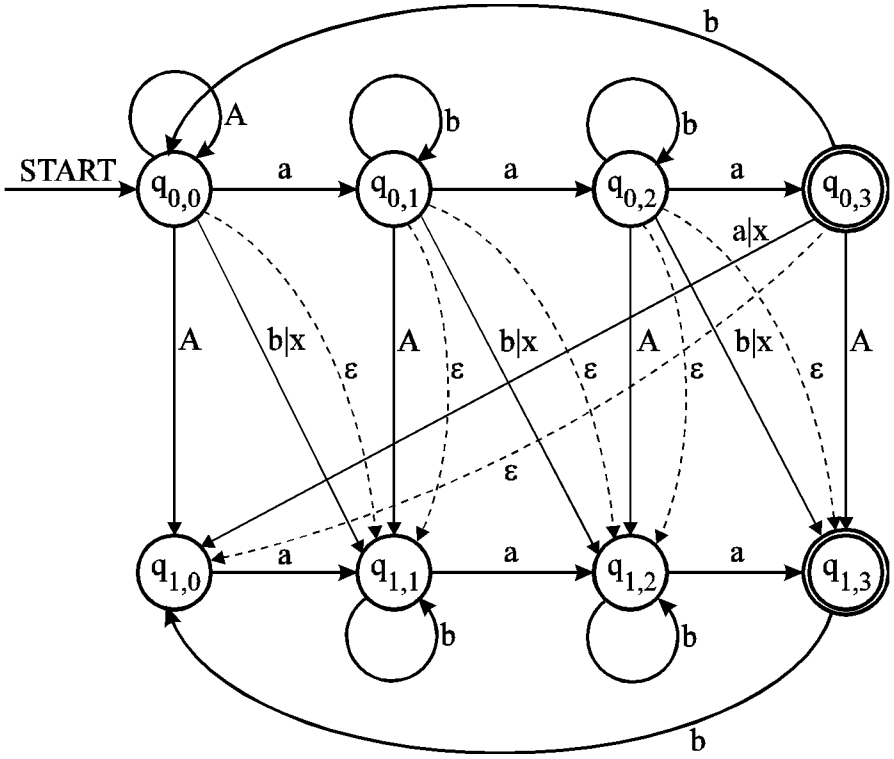


Figure 2: Nondeterministic automaton L_1.

strings with the postfix X such that $D_L^R(V, X) \leq 1$, where V is the regular expression from the previous example. The automaton L_1 will be used. The result can be described as follows:

$abbba_{(R,D)}b_{(R,D)}a_{(0,R,D,I)}b_{(R,D,I)}$.

The symbol 0 denotes the occurrence of a string from $h(V)$.

5 Conclusion

Both the nondeterministic automata M_H^R and M_L^R have to be deterministicaly simulated for practical purpose. But during the process of creating of equivalent deterministic finite automata the number of states can rise exponentialy, while the deterministic simulation of a nondeterministic automaton is of a high time complexity. It seems that this problem can be solved by constructing of a hybrid deterministic-nondeterministic finite automaton, but the problem is still open.

References

[1] Aho, A., Ullman, J.: The Theory of Parsing, Translation, and Compiling. Vol. I: Parsing, Prentice Hall, Englewood Cliffs, New York 1992.

[2] Melichar, B.: Approximate string matching by finite automata. In: Computer Analysis of Images and Patterns, LNCS 970, Springer 1995, pp. 132 – 137.

Fast Full Text Search Using Tree Structured [TS] File

Takashi SATO

Dept. Arts and Sciences (Computer Science)
Osaka Kyoiku University
4-698-1 Asahigaoka, Kashiwara, 582 Japan

e-mail: sato@cs.osaka-kyoiku.ac.jp

Abstract. The author proposes a new data structure (TS–file) in order to make a fast search for an arbitrary string in a large full text stored in secondary storage. The TS–file stores the location of every string of length L (the level) in the text. Using this, we can efficiently search for, not only strings of length L but also those shorter than or longer than L. From an analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length l_k is shorter than or equal to L, and $2(L - l_k + 1)$ otherwise. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS–file, a compressed TS–file is introduced in order to lower storage costs for practical use without losing search speed. The experimental results on compression using UNIX online manuals and network news show that the space overhead of the TS–file against the text searched is from 17% (when $L = 3$) to 212% (when $L = 12$) which is small enough for practical use.

Key words: data storage and indexing, gram based index, full text search, no false drop, TS–file

1 Introduction

The capability to search for strings which are not specified in advance is required more and more recently in the various ways of processing online data such as documents, articles, books, manuals, news, dictionaries and so on. When a text becomes huge, methods which search the full text directly[1]–[4] are not practical. So auxiliary data structures are used in order to speed up the search[5]–[8]. A signature file[9],[10] is a typical data structure for such purposes and it is widely used in practical applications[11]–[13]. However, when we consider the recent status of secondary storage which is rapidly increasing in space per drive and decreasing in cost per bit, faster and more flexible string searches are needed more than those which require less space.

In this paper, we propose a new data structure called a *TS–file* (Tree Structured file) and a set of algorithms using this in order to make arbitrary string searches especially fast. In a previous paper, using a compressed data file we proposed an algorithm which is efficient when the length of the search string is rather long[14].

The method in this paper is most suitable when the length of the search string is rather short. The basic ideas of the TS–file is to store the location of every string of length L in the text. Using a TS–file, not only strings of length L but also those shorter than or longer than L can be searched efficiently.

A retrieval system using transposed files based on single characters, pairs of adjacent characters and longer strings of adjacent characters has been reported for searches of Japanese text[15]. Since the size of the Japanese character set is large, multiple data structures are provided for these combinations of character classes. This system is analogous to our method for each $L = 1, 2, \cdots$, however, our method prepares only one data structure and it has a unique L value.

Since one can find arbitrary strings using the TS–file alone, the proposed method is more accurate than the one using signature files or PAT tree[7] by which one can only know the possibility of existence. The proposed data structure is not made from a word by word index stored as an inverted index[16]. In other words, it does not depend on any specific language styles, for example, in which words are separated by blanks and so on. So, the proposed method is applicable to a wide variety of pattern matches which includes bit strings and genetic information.

From the analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length l_k is shorter than or equal to L, and $2(L - l_k + 1)$ otherwise. This is far less than in the case of signature file search. The proposed algorithm is one of the fastest for arbitrary string searches. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS–file, we introduce a compressed TS–file by (1)making the data structure a tree to remove unused slots (*null* pointers), and (2)storing differences between adjacent elements if possible in order to lower storage costs for practical use without losing search speed.

Experiments using UNIX online manuals (up to 6.1Mbyte) and network news (up to 500Mbyte) as source text show that the searches are very fast and their time is less than 200msec in most cases and several hundred milli seconds even when keys have many matches or their length is much longer than the level. The overhead of storing the TS–file compared to text size is 30% when $L = 4$ and 66% when $L = 6$ for UNIX online manuals and 47% when $L = 4$ and 212% when $L = 12$ for network news. These overheads are small enough for practical use.

2 Definitions

The alphabet is denoted by Σ. σ $(=|\Sigma|)$ denotes the size of the alphabet. The TS–file stores the location of every string of a given length (called a *gram*) in the text. This length is called the *level*, L. A string sought is a *key*, k, whose length is l_k. A key is constituted of characters c_i. So a key is denoted by $k = c_1 c_2 \cdots c_{l_k}$ $(c_i \in \Sigma)$. The length of the *text* searched is n characters. The text is assumed to be large compared with the size of main memory and is stored on secondary storage. Data are transferred to and from the main memory in blocks of size B words. The units of memory are *word*, *half word* and *character*. The number of characters per word is w. In typical cases, 1 word = 4 byte and 1 character = 1 byte so that $w = 4$.

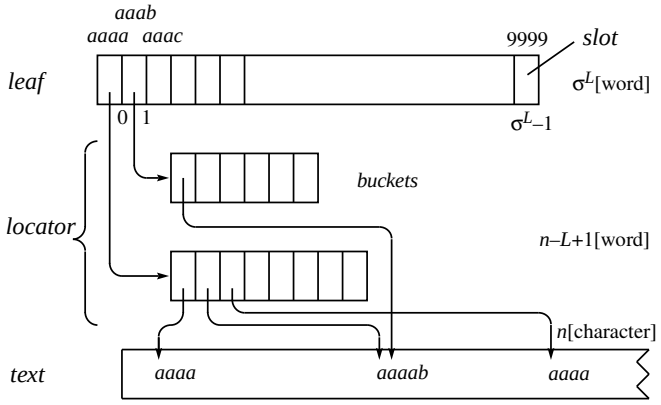


Figure 1: TS-file: Basic Structure ($L = 4$).

3 Basic TS–file and Its Storage Cost

3.1 Basic Data Structures

The basic TS–file consists of a *leaf* and a *locator*. Because the number of combinations of strings with length L is σ^L, the leaf has addresses of L digits in the σ-ary system $(0, 1, \cdots, \sigma^L - 1)$(see Fig. 1). We call each leaf address a *slot*. Each slot has a pointer which points to a bucket in the locator or *null*. Each bucket has pointers which point to locations in the text. The bucket which is pointed to by a slot stores the locations where the strings corresponding to the slot are found in the text. If there is no corresponding string in the text, the pointer in the slot is *null*.

3.2 Storage Cost

The size of the leaf is σ^L words assuming 1 word/slot. The size of the locator is $n - L + 1 \simeq n$ words assuming 1 word/pointer because there are $n - L + 1$ strings of length L in the text. Summing these up, the storage cost becomes

$$\sigma^L + n \quad [\text{word}]. \tag{1}$$

When slots used (i.e. not *null*) are sparse, we collect only used slots. Then slots become two words each because each slot should contain a slot value also. In this case a leaf is at most $2n$ words because the number of slots does not exceed n. Taking account of this collection of used slots, the storage cost becomes less than

$$min\{2n, \sigma^L\} + n \quad [\text{word}]. \tag{2}$$

If we can store information without having to align word boundaries, we can store data in every bit. The slot value is expressed in $L\lceil \log_2 \sigma \rceil$ bits and the pointer is expressed in $\lceil \log_2 n \rceil$ bits, so the above cost becomes as follows.

$$min\{n(L\lceil \log_2 \sigma \rceil + \lceil \log_2 n \rceil), \sigma^L \lceil \log_2 n \rceil\} + n\lceil \log_2 n \rceil \quad [\text{bit}]. \tag{3}$$

4 Search Algorithm and Its Cost

This section shows concrete algorithms based on the data structure introduced in **3**. We obtain a search cost by estimating the number of block transfers between main

memory and secondary storage.

4.1 Search Algorithm

The algorithm is explained in terms of three cases according to the relation between l_k and L.
(A) $l_k = L$

(1) Obtain a slot address for k by

$$s = \sum_{i=1}^{l_k} \mathrm{ord}(c_i)\sigma^{L-i}, \tag{4}$$

where 'ord' represents an arbitrary function which maps each character uniquely onto $0, 1, \cdots, \sigma - 1$.

(2) Find a bucket of the locator which stores pointers to the same strings as k by following the pointer in the slot obtained in (1).

(3) List the locations where k appears in the text by following the contents in the bucket found in (2).

(B) $l_k < L$

(1) Obtain lower and upper limit of slots (s_1 and s_2 respectively) since the slots to be searched are consecutive.

$$s_1 = \sum_{i=1}^{l_k} \mathrm{ord}(c_i)\sigma^{L-i}, \tag{5}$$

$$s_2 = \sum_{i=1}^{l_k} (\mathrm{ord}(c_i) + \delta_{i,l_k})\sigma^{L-i} - 1, \tag{6}$$

where $\delta_{i,i} = 1, \delta_{i,j} = 0 (i \neq j)$.

(2) Obtain buckets of the locator which are pointed to by the pointers in the slots of between s_1 and s_2.

(3) List the locations where k appears in the text by following contents in the buckets found in (2).

(C) $l_k > L$

(1) Obtain $l_k - L + 1$ slots from the following equation.

$$s_j = \sum_{i=1}^{L} \mathrm{ord}(c_{i+j})\sigma^{L-i} \quad (j = 0, \cdots, l_k - L). \tag{7}$$

(2) Obtain buckets of the locator which are pointed to by the pointers in the slots $s_j (j = 0, \cdots, l_k - L)$

(3) Find candidate locations where k may appear by following contents in the buckets found in (2). Candidate locations for the appearance of the key k in the text are the offsets of the obtained buckets contents minus j.

(4) List the locations where k truly appears by intersecting the sets of candidates for each j.

When $l_k < L$, we compute the set sum of locations in the buckets pointed to by the slots corresponding to strings containing the key. When $l_k > L$, we have to compute the set product of locations in the buckets pointed to by the slots contained in the key.

[Example1] Fig. 2 (a), (b) and (c) show how to follow the pointers in leaves and locators of a $L = 4$ TS–file when the key is 'text', 'ftr' and 'search' respectively. Since the buckets of the locator are stored sequentially, they are drawn in one box and separated by double lines. □

4.2 Search Cost

We estimate the cost to execute the algorithms in **4.1**. Because the TS–file is on the secondary storage, the search cost becomes the number of block transfers (fetches) from secondary storage to main memory.

(A) $l_k = L$

One fetch is required to read a slot computed from equation (4). When the number of matches for the key is M, $\lceil M/B \rceil$ fetches are required in order to read all matches in the locator. Summing these up, we obtain the cost f^a_{eq}.

$$f^a_{eq} = 1 + \lceil M/B \rceil \tag{8}$$

The fetches required to find a first match is $f^1_{eq} = 2$ because only the first block of the locator has to be read.

(B) $l_k < L$

The algorithm fetches consecutive slots from s_1 to s_2 and buckets of the locator pointed to by these slots. Not only slots but also the buckets pointed to by the consecutive slots are expected to be stored in adjacent regions of secondary storage. Then the number of fetches to find all matches becomes

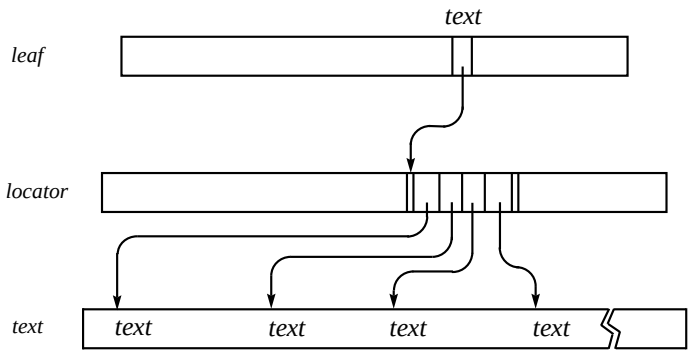
$$\begin{aligned} f^a_{lt} &= \lceil (s_2 - s_1 + 1)/B \rceil + \lceil \sum_{i=s_1}^{s_2} M_i/B \rceil, \\ &= \lceil \sigma^{L-l_k}/B \rceil + \lceil \sum_{i=s_1}^{s_2} M_i/B \rceil, \end{aligned} \tag{9}$$

where M_i is the number of matches for slot i. The first term of this equation becomes quite small under the compression proposed in **6.2** because it comes from a sequential scan of the leaf. The fetches required to find a first match is $f^1_{lt} = 2$ because only the first block of the locator for the slot s_1 has to be read.

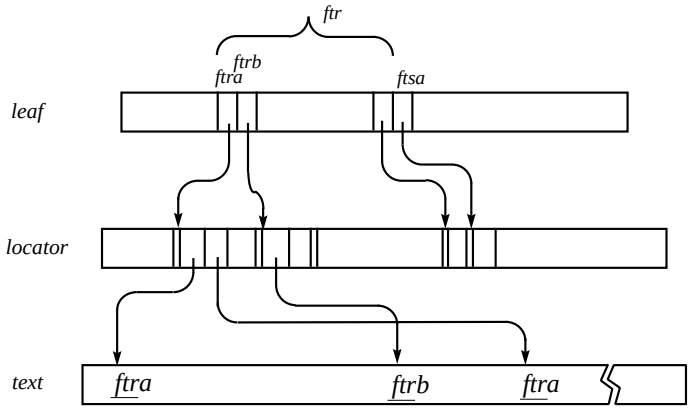
(C) $l_k > L$

Although the algorithm fetches $l_k - L + 1$ slots and the related parts of the locator, they are not necessarily adjacent on the secondary storage. So the number of fetches to find all matches becomes

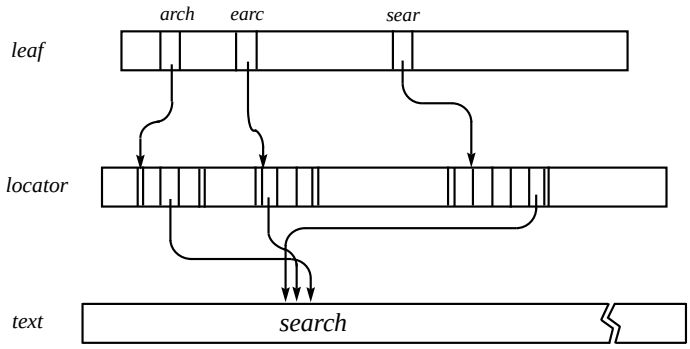
$$f^a_{gt} = l_k - L + 1 + \sum_{j=1}^{l_k - L + 1} \lceil M_j/B \rceil. \tag{10}$$



(a) $l_k = L, key =$ *"text"* $(l_k = L = 4)$.



(b) $l_k < L, key =$ *"ftr"* $(l_k = 3, L = 4)$.



(c) $l_k > L, key =$ *"search"* $(l_k = 6, L = 4)$.

Figure 2: Example of TS–file search

The fetches required to find a first match is $f_{gt}^1 = 2(l_k - L + 1)$.

We have analyzed the cost in three cases from the relations between l_k and L; the search cost is independent of n as long as the number of matches (M, M_i, M_j) are constant. And we have to pay attention to the fact that slots are accessed sequentially when $l_k < L$, but randomly when $l_k > L$.

5 Comparison with Other Methods

In this section, we review signature files, which are widely used for full text searches, and PAT trees which provide fast retrieval times. Both are not based on word indices and can be retrieved by arbitrary strings. We then compare these with the TS–file.

5.1 Signature Files

In section **6**, we present a way of compressing TS–files without losing the features of a fast full text search. Since we have to handle many parameters there, we will compare our method with search using signature files from the viewpoint of search speed before section **6**. The signature file is a typical example of an auxiliary data structure which is used in order to make a full text search fast. In this section, we compare the method proposed in the previous sections with the method of signature files, and we show the former is much faster than the latter. The signature file is known as an effective method for fast search and has been studied extensively. Because there are many forms of signature files, we first outline the signature file which is the object of comparison with the method proposed. Next we estimate the search cost when the signature file is used and compare this with the results discussed in section **4**.

Since the TS–file can search for a string in a text which is not necessarily composed of words, we assume that the signature files with which we are comparing are also made from grams (i.e. strings of characters) and not words. We divide the text into logical blocks of D characters. For each logical block, we make a bit vector of length b bits. We make each bit vector as follows. The bit vector is bit string of all zeros initially. For each pair of adjacent characters from the beginning of the logical block, one obtains a number between $0, 1, \cdots, b-1$ by applying an appropriate hash function. The i-th position of the bit vector is set to '1' when the number is i. Because the number of such pairs is $(D-1)$, the hash function sets bits to '1' in this bit vector $(D-1)$-times. Although the hash function should be selected carefully so as to distribute '1's randomly and uniformly, we cannot avoid collisions. Combining the bit vectors of all blocks, we get a signature file. As we assume n is the length of the text, the number of logical blocks is $\lceil n/D \rceil$. We use α for the ratio of b to D $(\alpha = b/D)$. Then the size S in bytes of the signature file becomes

$$S = b/8 \times \lceil n/D \rceil \simeq \alpha n/8 \quad \text{[byte]}. \tag{11}$$

In order to make the search fast, this signature file is sliced column wise when the bit vectors to be looked at are stored row wise. S is divided into b sub-files whose size is $S_b = S/b \simeq \alpha n/8b = n/8D$.

Next we consider a key search using the above sliced signature file. Since the key length is l_k, we compute the locations where '1' is set $(l_k - 1)$ times from the pairs of adjacent characters in the key. As we assume $l_k \ll D$, hash collisions are negligible,

so the number of these locations is approximately $(l_k - 1)$. Searching the $(l_k - 1)$ sub-files which store the location in the bit vectors corresponding to the locations of 1's computed from the key, we return as candidates the rows which have 1's in all these sub-files. In this case the number of fetches f^a_{sg} of the signature sub-files becomes

$$\begin{aligned} f^a_{sg} &= S_b(l_k - 1)/B \\ &\simeq n(l_k - 1)/(8DB) \end{aligned} \tag{12}$$

We know that this search returns the numbers of logical blocks which may contain the key. So we have to access the text blocks directly and examine them in order to know whether the key truly exists or not and what are the offsets in these blocks if it exists.
[Example2] When $n = 10^8, B = 1024, D = b = 256, L = 6, l_k = 7, M_1 = M_2 = 100$, the numbers of fetches of the two methods are

$$\begin{aligned} f^a_{sg} &= 286 \\ f^a_{gt} &= 4. \end{aligned}$$

In this case, we see that the method proposed in this paper is about seventy times faster than the method using signature files. □
f^a_{gt} does not change with n, however, f^a_{sg} grows in proportion to n (i.e. $O(n)$).

5.2 PAT Trees

A PAT tree[7] is a Patricia tree constructed over all the possible strings (called sistring) formed by starting at a given position and continuing to the end of a text. A Patricia tree[17, 18] is a digital tree where the individual bits of the keys are used to decide on the branching. A zero bit will cause a branch to the left subtree, a one bit will cause a branch to the right subtree. Hence Patricia trees are binary digital trees. In addition, Patricia trees have in each internal node an indication of which bit of the query is to be used for branching. This may be given as a count of the number of bits to skip. This allows internal nodes with single descendants to be eliminated, and thus all internal nodes of the tree produce a useful branching, that is, both subtrees are non-null.

Patricia trees store key values at external nodes; the internal nodes have no key information, just the skip counter and the pointers to the subtrees. The external nodes in a PAT tree are sistrings, that is, integer displacements. For a text of size n, there are n external nodes in the PAT tree and $n - 1$ internal nodes.

Retrieval using a PAT tree follows edges from a root towards leaves by considering skip counts in nodes. The contents of leaves in a subtree which follow edges where the bit comparisons end are candidate places for the key. Since there may have been bits skipped which should have been compared, we have to access the text and confirm whether we can find the key at that place or not. This method can not avoid false drops.

The depth of a leaf is the number of comparisons required to distinguish its sistring from others. The average depth is $d = \log_2 n$. It is natural to assume that PAT trees are stored in secondary memory because the texts being searched are large and therefore also stored in secondary memory.

We store skip counts of a complete binary tree of e levels * and pointers from nodes at the lowest level of the tree into a one block space in memory. If we assume these are represented in a one word space, the size of data in one block is $2^{e+1} - 1$[word]. Then the number of levels which fit into a block is at most $e = \log_2(B+1) - 1$. When we cache one block which stores the root of a PAT tree in main memory, the average number of block accesses to a leaf in order to find a pattern which may match with the key is at least $\lceil d/e \rceil - 1$. Including one access to the text for confirmation, this becomes $\lceil d/e \rceil$.

The whole subtree lying under a point where the bit comparisons end has to be searched in order to find all patterns which match with the key in a text. If we assume that the point is at level t, the average number of nodes in the subtree becomes 2^{d-t}. Dividing by the number of words in a block, the number of block accesses becomes $2^{d-t}/B$ †

[Example 3] When $n = 2^{30} (\simeq 10^9)$, $B = 1024$, $d = 30, e = 9$, the number of disk accesses to find a pattern which matches the key is 4, and the number of accesses to find all patterns when $t = 15$ is 32. □

We compare these results with the TS–file when $l_k \leq L$. The number of accesses to find one matching string is two in the case of the TS–file. To find all strings matching a key, we scan a part of the locator of the TS–file. It contains the same data as the leaf of a subtree of the PAT tree which starts at a point at level t but its size is half that of the PAT subtree. Based on this, we estimate that the search speed of the TS–file is twice as fast as that of the PAT tree in this case. On the other hand, when $l_k \gg L$, a PAT tree becomes advantageous in terms of the search speed.

The problem with a PAT tree is its high construction cost. In order to make a PAT tree, all the sistrings in the text have to be sorted. When n sistrings whose average length is $n/2$ are sorted in main memory whose work size is p blocks by multi-way merge sort, the number of disk accesses is $f^s_{PAT} = (n/Bw)^2 \lceil \log_p (n/Bw)^2/2 \rceil$ ‡. On the other hand in the case of a TS–file it is $f^s_{TS} = 2(nL/Bw)\lceil \log_p (nL/Bw) \rceil$.
[Example 4] When $n = 2^{25}, B = 1024, w = 8, L = 12, p = 2048$,

$$\begin{aligned} f^s_{PAT} &\simeq 50,332,000 \\ f^s_{TS} &\simeq 131,000. \end{aligned} \tag{13}$$

The sort used in making a TS–file is about 400 times faster than that for a PAT tree. □

6 Compression of the TS–file

Although we can search for arbitrary strings fast by the method explained in the earlier sections, the TS–file often becomes too large to store in practice. So we want to compress the data structure without losing the features of fast full text searches. We explain compression methods for the locator and the leaf respectively in the following.

*in a PAT tree a level is a set of nodes which are at the same depth from the root.

†This is the case when every block is 100% filled up. If we use $\log_e 2 \times 100\%$, an average value reported in the literature [7], the number of accesses becomes $2^{d-t}/B \log_e 2$.

‡In general, the number of disk accesses required to sort x blocks of data by p–way merge sort is $2x\lceil \log_p x \rceil$.

6.1 Compression of the Locator

6.1.1 Using Block Numbers for Locations

In this paper, we are not concerned with searches making use of the text structure (for example SMGL or Hyper text) [19],[20]. When we search a large text, however, it is rather rare that it have no structure. Usually texts have logical blocks at least. So, numbering each logical block in sequence, we adopt the method in which the numbers are output as the result of the search instead of the locations (pointers) where the key appears in the text. Because the number of pointers in one logical block is (the number of characters in the block $-L+1$), the number expressed should decrease considerably if we use the logical block numbers instead of pointers as output results. Using block number, a half word may be enough in most cases even if the pointer is one word in these cases. For example, in section **7** we use a UNIX online manual whose size is 6.11 Mbyte. Although we use 4 bytes (1 word) to express pointers, 2 bytes (half word) is enough to express the number of logical blocks (manual pages) which is 2707 in total. So we halve the amount of storage. Moreover, since the same strings often appear in the same logical block, duplicated logical block numbers for the same string should be removed. If we assume c is the average number of duplications, the compression rate α_c by this compression method in storing the locations where the key appears becomes

$$\alpha_c = 1/2c. \tag{14}$$

When the locations are stored by block number, $-j$ in the algorithm of **4.1**(c)(3) should be removed, and the result gives only the possibility of existence. Now we have to access the logical blocks of the text whether the key is truly there or not. However, we have confirmed experimentally that if L is large enough ($L \geq 6$), false drops are very rare in practice.

6.1.2 Run-length Encoding

According to **6.1.1**, each bucket of the locator contains the numbers of the logical blocks which include a given string of length L. If the differences between adjacent numbers are small, we can store them in less storage space by sorting these numbers and storing their differences. This method is called *run-length encoding*[16].

For example, if 7bits is used to express the difference and 1 bit to express whether the next data is differenced or not, 1 byte (=8bits) is enough to store a block number when the difference is less than 128. We can halve the memory required for the locator compared with the case when 2 bytes is used for each logical block number. Assuming that p_d is the probability of being able to store by difference, the compression factor α_d by this method becomes

$$\alpha_d = 1 - p_d/2. \tag{15}$$

When we apply both **6.1.1**,**6.1.2** methods, the total compression rate for the locator becomes $\alpha_c\alpha_d$.

6.2 Compression of the Leaf

6.2.1 Using a Tree Structure

Although in **3.1** we prepared slots for every sub string of length L, the results of the experiments in **7** show that the usage rate of slots is not high. As we stated in **3.2**, we can remove slots that are not used in order to save space. If these slots are simply removed and the leaf is compacted, we can no longer compute, using just the sub string, the slots where the desired pointer to the locator is stored. So another auxiliary data structure should be added in order to access slots quickly. Firstly, in each slot we store not only a pointer to the locator but also a slot value computed from the sub string. Though each slot size increases from 1 word to 2 words, assuming a slot value can be expressed in 1 word, considerable compression is expected as a result because slots which have *null* pointers can be removed. If u_s is the usage rate of the slots, the compression factor becomes

$$\alpha_s = 2u_s. \tag{16}$$

Secondly, we prepare the *root*, a data structure which stores the values of regularly spaced slots (interval=b_f) of the leaf in order to guarantee fast accesses to the required slots of the leaf. In order to make leaf access only one block, b_f is set as

$$b_f = B/2 \tag{17}$$

from the fact that the physical block size is B (transfer unit between main and secondary memory) and each slot is 2 words (one for the slot value and one for a pointer to the leaf). Using this data structure, two fetches are required to access the leaf, as long as accessing the root is one fetch. Since the slots of the leaf are accessed in only one fetch for the basic TS–file, one more fetch is required in this case. But from the fact that physical data accesses are not required when we retrieve keys successively because the root is cached in main memory, the influence on search performance is negligible. If b_c is the number of blocks that can be used as the root cache, the whole root can be put in cache when the usability of slots u_s is

$$u_s \leq b_c b_f B/2\sigma^L = b_c B^2/4\sigma^L. \tag{18}$$

Although in most cases, a two-level structure (root and leaf) is enough, we can make the root into a multi-level structure, i.e. a tree structure, if we can not put the whole root in main memory. Fig. 3 shows this tree structure.

6.2.2 Compression of Slots

When we adopt the structure for the leaf proposed in **6.2.1**, each slot is composed of a slot value and a pointer to a bucket of the locator. Among these, the former is compressed to 1 byte by the run-length encoding as **6.1.2**, when the difference of neighboring slot values is less than 128. Because buckets are stored in the order of corresponding slot value, the pointers which point to buckets can be also compressed by run-length encoding.

Assuming p_s and $p_p (0 \leq p_s, p_p \leq 1)$ are the possibilities of storing differences for slot values and pointers respectively, the compression factor by this method becomes

$$\alpha_p = 1 - (3/8)(p_s + p_p). \tag{19}$$

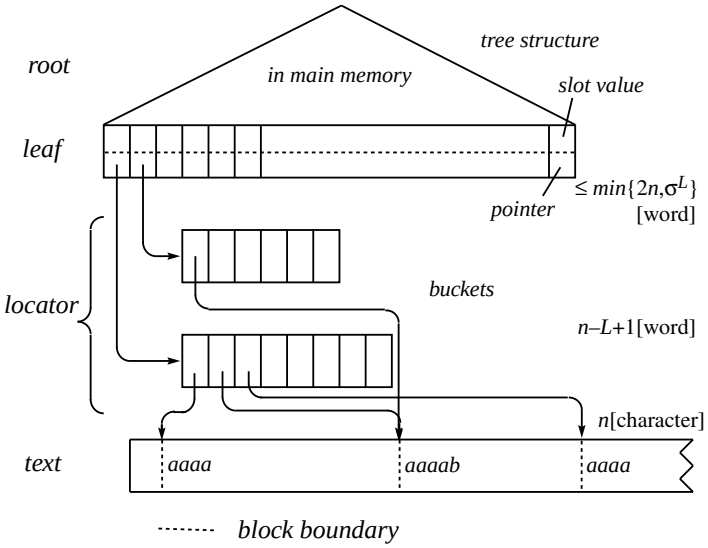


Figure 3: Compressed TS–file.

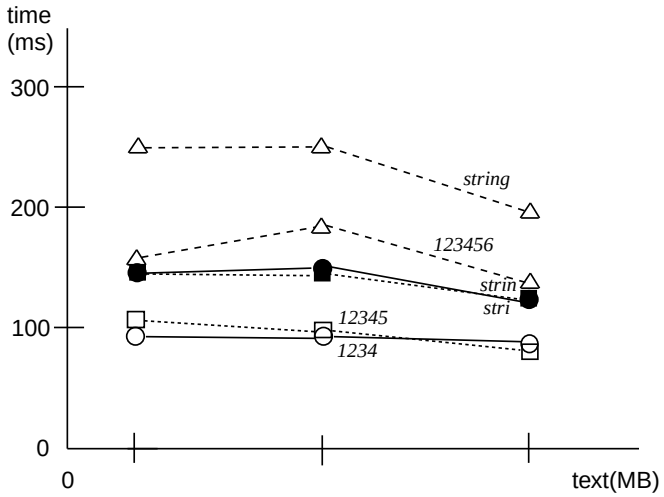
If both **6.2.1**,**6.2.2** are applied, the size of the leaf becomes $\sigma^L \alpha_s \alpha_p$ and the root becomes $\sigma^L \alpha_s / b_f$.

7 Experimental Results

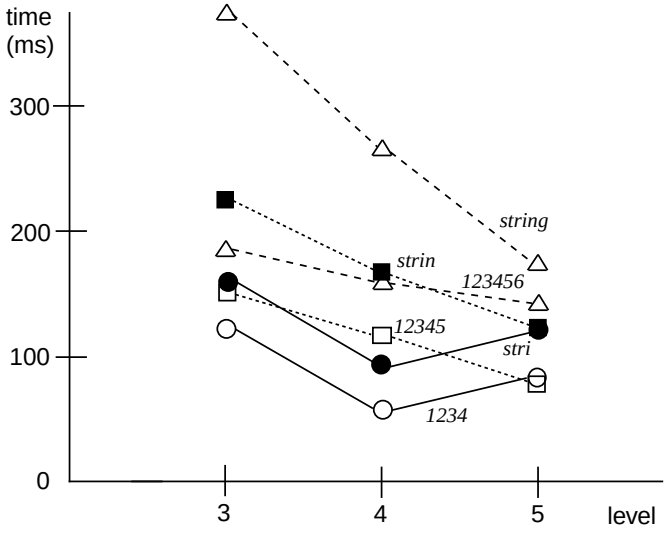
In order to confirm that the proposed data structure and algorithm provide very fast searches and that the compression methods proposed are effective we did experiments and measured the parameters appearing in **6**. The computer mainly used is a SUN Microsystems Sparc Server 630 (28.5MIPS) and the text searched is a UNIX online manual. In **7.3** where second stage compression is applied, we also show the results using network news, whose total size is 100, 300, 500Mbyte, as a source text. Each search time is measured under the condition that the whole root is cached and no parts of the leaf and the locator are cached in main memory initially.

7.1 Basic Structure

Experiments were done for the basic TS–file of **3** first (see Table 1). Fig. 4(a) shows the relationship between text size and search time, and (b) shows the relationship between level and search time. To make the alphabet size small, upper-case letters are converted to lower-case and letters other than numerical or alphabetical are all converted to blanks. That is, Σ={0,1,$\cdots$,9,a,b,$\cdots$,z,$\sqcup$} and $\sigma = 37$. Although this manual consists of 2707 separate files (called *manual pages*), we concatenated them into one large non-structured file. In order to change the text size in the experiments, four different size texts were made from the concatenation of 200, 500, 1000, 2707 pages of online manual respectively. As shown in Table 1, the maximum size of these texts is 6.11 Mbyte. Because $37^6 \leq 2^{32}$ and 1 word = 4 bytes in this case, the slot values can be expressed in one word if $L \leq 6$. The pointers to the locator are expressed in one word. The size of the locator and the leaf of each level are shown in Table 1. In the first experiment, we measured the search time for a key set of '1234',



(a) Text size – Time (Level: 5)



(b) Level – Time (Text: 6.1MB)

Figure 4: Basic

'12345', '123456', 'stri', 'strin', 'string'. Among these strings, '1234', '12345', '123456' are used as low selectivity examples, on the other hand, 'stri', 'strin', 'string' are used as relatively hight selectivity examples. The numbers of matches are shown in the 'unix manual' column of Table 2. Since no false drop occurs in basic TS–file search, we don't have to access the text in order to determine the locations where the key appears. Because the case $l_k < L$ is faster than the case $l_k > L$, L is preferably set somewhat larger than the average key length expected. The number of slots, however, grows exponentially with L, so we should take care that the leaf does not become too large by referring to the analysis of **3**. The size of the locator is not related to L. It is four times the text size because one pointer is four times longer than one character (=1 byte). In this experiment, however, it is less than three times as large because the pointers which point to characters which are neither alphabetic nor numeric are not stored in order to save storage space.

level		4				5				signature file			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
text(MB)		.751	1.89	2.80	6.11	.751	1.89	2.80	6.11	.751	1.89	2.80	6.11
leaf(MB)		7.29	7.29	7.29	7.29	270	270	270	270	.092	.232	.343	.750
locator(MB)		2.16	5.43	8.05	17.5	2.16	5.43	8.05	17.5				
time (ms)	1234	69.7	78.1	73.3	62.3	93.2	89.0	93.9	89.8	.42s	.82s	1.2s	2.2s
	12345	111	104	157	119	112	90.0	95.9	81.1	.35s	.67s	.98s	1.9s
	123456	205	159	234	166	163	197	187	133	.34s	.59s	.83s	1.7s
	stri	84.3	88.9	83.4	93.6	148	120	149	120	.39s	.77s	1.1s	2.2s
	strin	191	171	173	170	148	109	146	121	.41s	.78s	1.1s	2.1s
	string	278	262	267	269	257	214	241	184	.41s	.78s	1.1s	2.1s

Table 1: Experimental Results 1 – Basic Structure

	unix manual (page#)				news (MB)		
	200	500	1000	2707	100	300	500
1234	3	5	6	13	125	516	771
12345	2	3	3	8	51	136	223
123456	2	3	3	6	24	87	137
stri	57	151	248	601	9798	15184	21817
strin	40	95	143	376	867	3288	4993
string	40	95	143	376	867	3285	4988
database	–	–	–	–	1082	4540	6706
cryptograph	–	–	–	–	9	75	156

Table 2: Number of Matches

The search time measured is that for finding all addresses of matched strings. It is very fast as predicted in the analysis of **4** and it is less than 300 msec. In particular, it is faster, less than 150msec, when $l_k < L$. Search time does not increase with text size. The time required to search these texts using the signature files ($b = 256$) described in **5.1** is also recorded for reference. In the table 's' indicates that this is the only search measured in seconds. The size of the signature files is written between the rows for leaf and locator in this table. We can read the relationship between l_k, L and measured time qualitatively although, because the times measured are short and apt to include measurement errors, it doesn't necessarily agree with the analysis.

Although a fast search is accomplished with this basic TS–file, the locator and the leaf which constitute the TS–file are quite large and storing them is burdensome.

7.2 Compression by Block Number and Tree Structure

For the first stage compression of the TS–file, page numbers, which are expressed in half word (=2 bytes), instead of locations are put in the locator, and a tree structure is made in order to remove unused slots of the leaf. Table 3(a) shows the experimental results in this case. The size of the locator, leaf and root are also measured in the table. The usage rate u_s of the slots and the average duplication count c of the same string in a logical block of the text which relate to the compression rate of the leaf is measured also. The size of the locator is decreased by 1/4 to 1/10 and the leaf is decreased as per equation (16). The manual pages don't have to be concatenated in this experiment. Each manual page corresponds to a logical block in **6** and the output is page numbers. Search time was measured for the same key set as in **7.1**. But it should be noted that when $l_k > L$ the time measured is until determining the possibility of existence. According to another experiment there are no false drops

for the search strings in the table. The result of these experiments shows that the searches are quite a lot faster than those using the basic TS–file, because the amount of data accessed is reduced due to compression.

level		4				5				6			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
$u_s \times 100(\%)$		1.04	1.47	1.81	2.84	.0688	.104	.130	.214	.0034	.0057	.0072	.0122
c		3.06	2.95	2.92	2.74	2.28	2.21	2.22	2.12	1.90	1.84	1.87	1.82
root(kB)		.612	.864	1.06	1.67	1.49	2.26	2.82	4.64	2.76	4.54	5.77	9.80
leaf(kB)		156	220	272	426	381	578	722	1187	704	1163	1477	2509
locator(kB)		353	919	1377	3197	474	1232	1814	4128	570	1475	2151	4828
time (ms)	1234	48.1	58.5	78.4	63.5	61.4	61.2	68.6	64.8	61.1	52.5	72.9	81.2
	12345	48.2	80.1	82.1	66.0	62.7	60.9	68.5	63.8	61.5	52.1	62.2	91.8
	123456	49.0	80.9	77.1	73.6	59.3	83.2	69.4	65.3	69.5	50.3	69.0	89.6
	stri	83.2	94.2	97.9	116	90.2	107	111	115	108	77.3	118	124
	strin	97.1	122	126	151	90.2	116	108	109	114	76.6	126	138
	string	158	187	205	242	94.3	136	149	205	114	76.8	126	135

(a) Compression – 1

level		4				5				6			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
p_s		.902	.918	.931	.951	.773	.801	.811	.837	.633	.680	.690	.711
p_p		1.00	.972	.959	.937	1.00	.992	.988	.977	1.00	.997	.996	.991
p_d		.886	.923	.919	.912	.794	.846	.840	.824	.684	.746	.738	.717
root(kB)		.612	.864	1.06	1.67	1.49	2.26	2.82	4.60	2.75	4.54	5.77	9.80
leaf(kB)		44.8	64.1	79.2	124	128	189	234	380	273	431	544	907
locator(kB)		197	495	744	1739	286	710	1053	2426	375	925	1358	3096
space overhead(%)		32.3	29.6	29.4	30.5	55.3	47.7	46.0	46.0	86.7	72.0	68.1	65.7
time (ms)	1234	81.9	62.4	77.7	67.7	68.9	60.0	69.0	65.7	67.9	69.3	66.4	101
	12345	82.3	62.9	79.6	56.7	67.3	61.7	57.2	64.6	65.5	69.7	63.6	101
	123456	72.2	65.7	79.0	74.6	66.3	61.2	71.2	70.2	63.5	66.6	66.5	101
	stri	79.0	85.7	78.0	111	100	93.0	94.1	124	96.8	65.6	119	121
	strin	92.1	112	105	162	110	102	90.4	127	110	74.6	128	135
	string	126	160	187	241	111	106	161	219	111	74.7	128	135

(b) Compression – 2

Table 3: Experimental Results 2

Since a larger L increases the chance of $l_k \leq L$, it decreases search time. Moreover if $L \geq l_k$ we know the page numbers which contain the key without accessing the text because then there are no false drops. So a larger L is more advantageous as long as storage space permits it.

7.3 Compression by Run-length Encoding

For the second stage of compression, we did an experiment in which the locator and the leaf are compressed by run-length encoding (see Table 3(b)). Fig. 5(a) shows the relationship between text size and search time, and (b) shows the relationship between level and search time. The size of the compressed locator and leaf agree with the values which are computed from the equations in **6** with the original size and the compressing probability (p_s, p_p, p_d) measured. When $L = 4$ and the text is 2707 pages, the size of the TS–file (sum of the locator, leaf and root) is 1.86Mbyte which is 30% overhead against 6.11Mbyte (the text size). This ratio is 65.7% when $L = 6$ (see space overhead row of the table). Fig. 6(a) shows how TS–file is compressed by the first and second compression. We also show how search time changes by these compressions in (b).

level		4	6	12	sig
root(MB)		.0191	.115	1.20	—
leaf(MB)		1.23	10.6	111	—
locator(MB)		46.1	67.9	99.7	—
space overhead(%)		47.4	78.6	212	12.3
time (ms)	1234	95.8	105	84.3	41s
	12345	184	115	83.5	36s
	123456	271	92.4	92.0	37s
	stri	106	163	183	37s
	strin	206	133	167	38s
	string	331	134	165	37s
	database	363	290	86.6	38s
	cryptograph	611	505	78.0	13s

Table 4: Experimental Results 3
100MB (Compression–2)

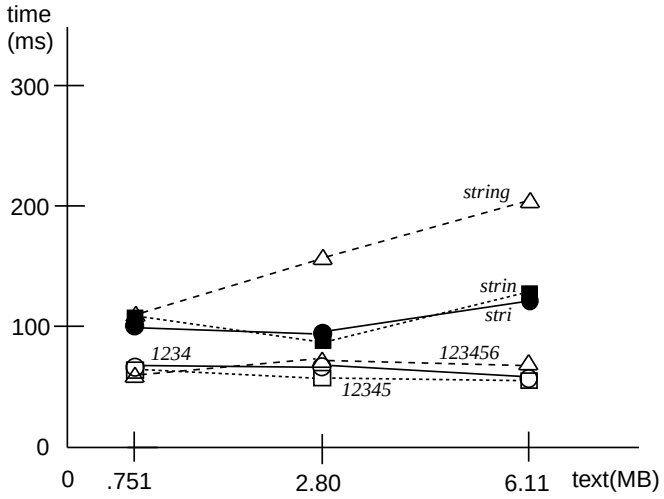
For this compression only, we also tried using 100Mbyte of network news as text (see Table 4). Fig. 7 shows the relationship between level and search time. Creating TS–files for the levels of 4,6 and 12, we measured the space and search time. Slots have twice the length, i.e. 8 bytes, for $L = 12$ only. we added 'database' and 'cryptograph' ($l_k = 8$ and 12 respectively) to the previous list of search strings. The number of matches are shown in the 'news' column of Table 2. 17.5% false drops were observed for the search string 'strin' and level $L = 4$; however, no false drop was observed for any other combinations of search strings and levels.

The ratio of the size of the TS–file to the text size is less than half (47.4%) when $L = 4$ and about twice (212%) when $L = 12$ (see space overhead in Table 4). Considering the average length of words is from five to seven, $L = 12$ is a sufficiently large level. The size of the root is 1.2Mbyte even when $L = 12$, which may easily be put in the main memory. From the fact that ordinary key word indices, which cannot be used for arbitrary string searches, often become bigger than the text and that secondary storage devices are increasing their space and decreasing the price per byte recently, the TS–file is sufficiently small for practical use.

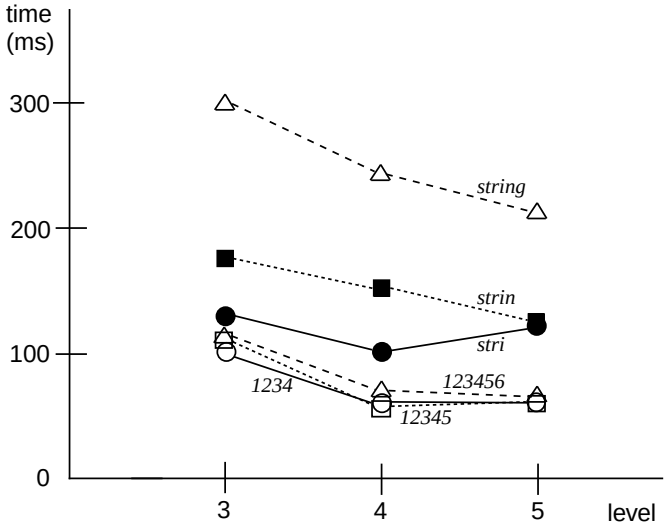
The search time measured is less than 200msec for the strings whose length is less than six when $L = 6$ and for all strings searched when $L = 12$. The search time is very fast for the arbitrary string searches of the 100Mbyte text.

Searches using signature files ($b = 256$) are also recorded for reference. Although the overhead in size (12.3%) is smaller than that of a TS–file, the search speed is very slow.

We also used 300 and 500Mbyte network news as a text with level $L = 12$. We use different computers in this case. A Silicon Graphics Challenge is used to make the TS–files, and a Silicon Graphics Indy R4600 (62.8 Specint92) is used to search for keys. We measured the time required to search for the first matches as well as that for all match (see Table 5). Fig. 8(a) and (b) show the relationship between level and search time for all and first match respectively. It is proved by this experiment that the search is fast even when the text size is quite large and its time depends only on the number of matches.

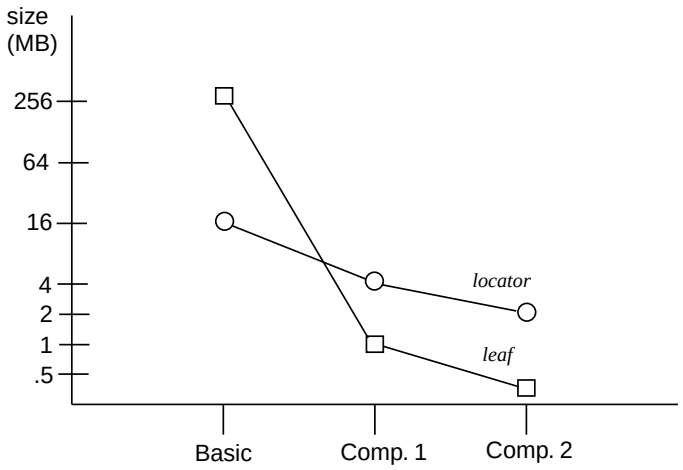


(a) Text size – Time (Level: 5).

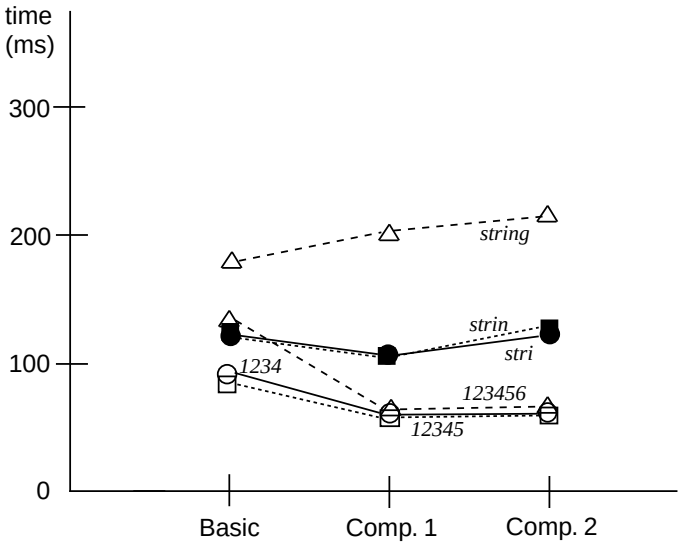


(b) Level – Time (Text: 6.1MB).

Figure 5: Compression



(a) Size of leaf and locator.



(b) Time (Text: 6.1MB; Level: 5).

Figure 6: Basic, Comp. 1 and Comp. 2

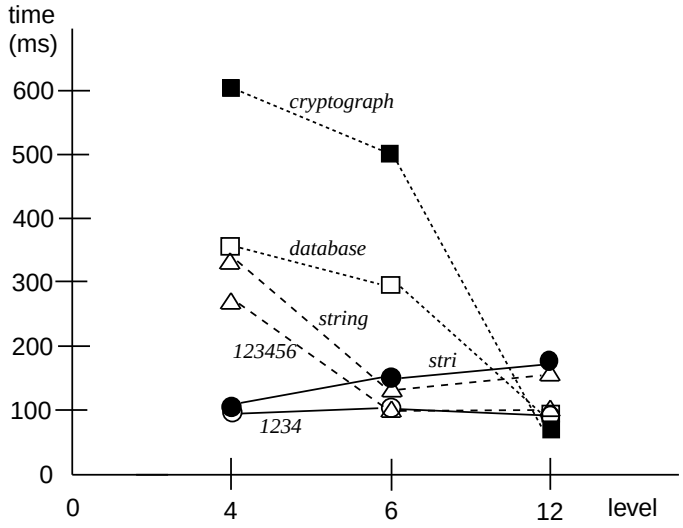


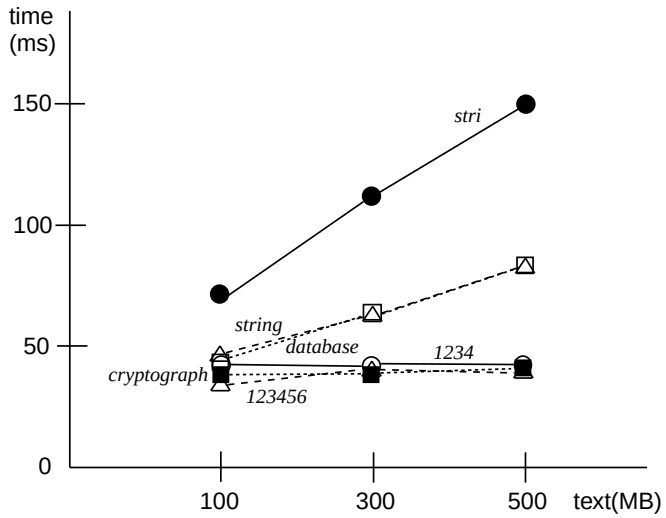
Figure 7: 100MB text Level – Time.

text(MB)		100	300	500
root(MB)		1.20	3.94	7.71
leaf(MB)		111	263	555
locator(MB)		99.7	248	442
space overhead(%)		212	172	201
first match (ms)	1234	33.2	33.1	37.3
	12345	36.6	37.1	36.8
	123456	31.0	37.6	37.2
	stri	40.2	39.2	38.1
	strin	34.6	38.7	41.9
	string	35.7	34.2	38.8
	database	39.1	40.6	37.3
	cryptograph	36.1	36.4	34.6
all match (ms)	1234	41.8	42.1	42.4
	12345	39.0	33.1	35.7
	123456	34.9	41.7	39.2
	stri	72.1	120	152
	strin	43.3	57.7	75.3
	string	47.9	63.1	78.1
	database	45.1	63.9	78.3
	cryptograph	39.6	39.8	41.8

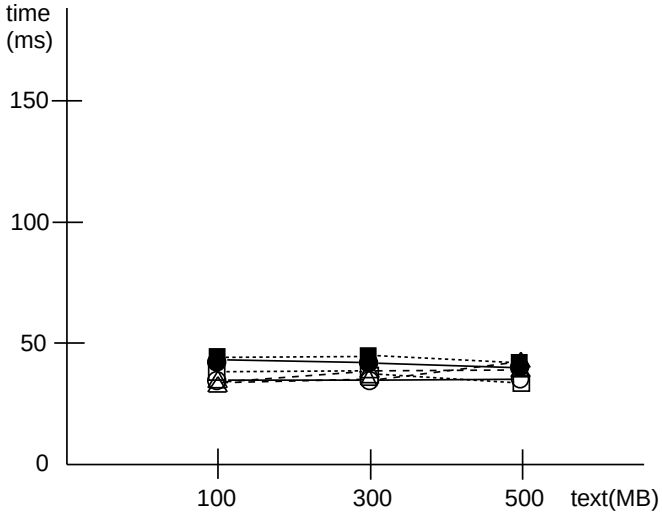
Table 5: Experimental Results 4
$L = 12$ (Compression–2)

8 Conclusions

In this paper we introduced the TS–file, a new gram based data structure for fast full text search, and we gave a set of concrete search algorithms using the TS–file. Because the proposed method can find an arbitrary string using the TS–file alone, the proposed method is more accurate than the one using signature files by which one



(a) all match.



(b) first match.

Figure 8: Text size – Time (Level: 12)

can only know the possibility of existence. We also showed that this method is much faster than searches using signature files by analysis. From the experimental results, we confirmed that sub string matches of rather short strings, which are common in practice, can be done very fast. We were able to reduce the size of the TS–file without losing search speed by introducing two stage compression methods by which the storage required became sufficiently small for practical use.

References

[1] D. E. Knuth, J. H Morris. and V. R. Pratt: "Fast pattern matching in strings", *SIAM J. Comput.*, **6**, 2, pp.322–350 (1977–06).

[2] R. S. Boyer and J. S. Moore: "A fast string searching algorithm", *Commun. ACM*, **20**, 10, pp.762–772 (1977–10).

[3] A. V. Aho and M. J. Corasick: "Efficient string matching : An aid to bibliographic search", *Commun. ACM*, **18**, 6, pp.333–340 (1975–06).

[4] R. M. Karp and M. O. Rabin: "Efficient randomized pattern-matching algorithms", *IBM J. Res. Develop.*, **31**, 2, pp.249–260 (1987).

[5] C. Faloutsos: "Access methods for text", *ACM Comput. Surveys*, **17**, 1, pp.49–74 (1985–03).

[6] J. Aoe: *Computer algorithms – String pattern matching strategies*, IEEE Computer Society Press (1994).

[7] G. Gonnet, R. Baeza-Yates and T. Snider: New indices for text: Pat trees, in *Information retrieval: data structure & algorithms* chapter 5, W. Frakes and R. Baeza-Yates Ed., pp. 66–82 (1992).

[8] H. Shang: *Trie methods for text and spatial data on secondary storage*, Ph.D Dissertation, Faculty for Graduate Studies and Research of McGill University (1995).

[9] M. C. Harrison: "Implementation of the substring test by hashing", *Commun. ACM*, **14**, 12, pp.777–779 (1971-12).

[10] C. Faloutsos: "Signature files: Design and performance comparison of some signature extraction methods", *Proc. 1985 ACM SIGMOD International Conference on Management of Data*, pp.63–82.

[11] D. L. Lee and C. Leng: "A partitioned signature file structure for multiattribute and text retrieval", *Proc. 6th IEEE International Conference on Data Engineering*, Feb. 1990, pp.389–397.

[12] T. Kinoshita, J. Aoe and T. Sato: "A method for speeding up a hash search using a substring test", *Trans. Inst. Electron. Inf. Commun. Eng. D–I*, **J73–D–I**, 5, pp.535–538(1990–05) [in Japanese].

[13] W. W. Chang and H. J. Schek: "A signature access method for starburst database system", *Proc. 15th International Conference on VLDB 1989*, pp.145–153.

[14] T. Sato: "Fast string pattern matching by using compressed data files", *Trans. Inst. Electron. Inf. Commun. Eng. D–I* , **J73–D–I**, 4, pp.451–452 (1990–04) [in Japanese].

[15] Y. Ogawa and M. Iwasaki: A new character-based indexing method using frequency data for Japanese documents, *In Proc. 18th ACM SIGIR Conf.*, Seattle, USA, July 9–13 1995, pp. 121–129.

[16] J. Zobel, A. Moffat and R. Sacks-Davis: "An efficient indexing technique for full-text database systems", *Proc.18th International Conference on VLDB 1992*, pp.352–362.

[17] D. Morrison: PATRICIA – Practical algorithm to retrieve information coded in alphanumeric, *J.ACM,*, **15**, 4„ pp. 514–534 (1968).

[18] D. Knuth: *The art of computer programming: Vol.3 sorting and searching*, Addison-Wesley, Reading, Mass., pp. 490–499 (1973).

[19] C. Clifton and H. Garcia-Molina: "Indexing a hypertext database", *Proc.16th International Conference on VLDB 1990*, pp.36–49.

[20] R. Sacks-Davis, T. Arnold-Moore and J. Zobel: "Database systems for structured documents", *Proc. Int. Symp. Advanced Database Technologies and Their Integration ADTI'94*, Nara, Japan, Oct. 26–28 1994, pp.272–283.

[21] T.Sato: "Fast Full Text Search with Free Word Using TS–file", *Proc. 19th ACM SIGIR Conf.*, Zurich, Switzerland, Aug. 18–22 1996, p.342.

A Collection of New Regular Grammar Pattern Matching Algorithms

Bruce W. Watson

Ribbit Software Systems Inc.
IST Technologies Research Group
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

Abstract. A number of new algorithms for regular grammar pattern matching is presented. The new algorithms handle patterns specified by regular grammars — a generalization of multiple keyword pattern matching and single keyword pattern matching, both considered extensively in and [14, Chapter 4] and in [18].

Among the algorithms is a Boyer-Moore type algorithm for regular grammar pattern matching, answering a variant of an open problem posed by A.V. Aho in 1980 [2, p. 342]. Like the Boyer-Moore and Commentz-Walter algorithms, the generalized algorithm makes use of shift functions which can be precomputed and tabulated.

It appears that many of the new algorithms can be efficiently implemented.

Key words: pattern matching, algorithms, regular grammars, regular pattern matching, algorithmics, string algorithms

1 Introduction

The pattern matching problem is: given a regular pattern grammar (for a formal definition see Section 2) and an input string S (over an alphabet V), find all substrings of S which correspond to the language denoted by some production in the grammar. Several restricted forms of this problem have been solved (all of which are discussed in detail in [14, Chapter 4], and in [3, 18]):

- The Knuth-Morris-Pratt [12] and Boyer-Moore [5] algorithms solve the problem when there is only a single production and its right-hand side has no nonterminals — it is in V^* (the single keyword pattern matching problem).
- The Aho-Corasick [1] and Commentz-Walter [6, 7] algorithms solve the problem when all productions in the grammar have right-hand sides without nonterminals — all of them are in V^* (this is the multiple keyword pattern matching problem). The Aho-Corasick and Commentz-Walter algorithms are generalizations of the Knuth-Morris-Pratt and Boyer-Moore algorithms respectively.

To date, very few regular grammar pattern matching algorithms have been developed. Only recently, the generalized Boyer-Moore algorithm was developed [14, 16, 17]*.

It should be noted that there do exist other regular pattern matching algorithms with good performance — for example, the one which was developed by R. Baeza-Yates [4, 10]. Those algorithms are not, however, considered here since they either use regular expressions† or they require some precomputation on the input string, and are therefore not suited to the type of application presented in this paper.

This paper is structured as follows:

- Section 2 gives the problem specification, and a naïve algorithm.
- Section 3 gives a family of algorithms which process the input string in a left-to-right manner.
- Section 4 gives a family of algorithms which process the input string in a right-to-left manner. These algorithms are not symmetrical with the ones in Section 3, due to our asymmetrical choice of right-linear grammars for our regular pattern grammars.
- Section 5 presents the conclusions of this paper.

Before continuing with the developement of the algorithms, we first give some of the definitions required for reading this paper.

1.1 Mathematical preliminaries

Most of the following definitions are standard ones relating to regular grammars and languages.

Definition 1.1 (Alphabet): An *alphabet* is a finite, non-empty set of symbols. □

Throughout this paper, we will assume some fixed alphabet V.

Definition 1.2 (Functions pref and suff): For a given string z, define $\mathbf{pref}(z)$ (respectively $\mathbf{suff}(z)$) to be the set of prefixes (respectively suffixes), including string z and the empty string ε, of z. □

Definition 1.3 (String manipulation operators): Since we will be manipulating the individual symbols of strings, and we do not wish to resort to such low-level details as indexing, we define the following four operators (all of which are infix operators, taking a string as the left operand, a natural number as the right operand, and yielding a string):

- $w \nwarrow k$ is the $k\,\mathbf{min}\,|w|$ left-most symbols of w.
- $w \nearrow k$ is the $k\,\mathbf{min}\,|w|$ right-most symbols of w.

*That research was performed jointly with Richard E. Watson of the Department of Mathematics, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada; he can now be reached at `rwatson@RibbitSoft.com`.

†Although regular grammars and regular expressions have the same descriptive power, they can yield algorithms which are substantially different.

- $w \swarrow k$ is the $|w| - k \,\mathbf{max}\, 0$ right-most symbols of w.
- $w \searrow k$ is the $|w| - k \,\mathbf{max}\, 0$ left-most symbols of w.

□

Definition 1.4 (Regular pattern grammar): A *(right) regular grammar* (also known as a *right linear grammar*) is defined to be a three tuple, (V, N, P), where:

- V is our alphabet, known as the *terminal* alphabet.
- N is an alphabet, known as the *nonterminal* alphabet.
- $P \subseteq N \times (V^* \cup V^*N)$ is a finite and nonempty set of (right linear) *productions*. We usually write a given production (A, w) as $A \longrightarrow w$. We also define left-hand side and right-hand side functions **lhs** and **rhs** (respectively) such that $\mathbf{lhs}(A \longrightarrow w) = A$ and $\mathbf{rhs}(A \longrightarrow w) = w$.

  We also define a function **vpart** on right-hand sides as follows: $\mathbf{vpart}(w)$ is the longest prefix of w containing only symbols in V; that is, we drop the nonterminal on the right, if there is one. More formally, for a right-hand side x (for $x \in V^*$) we have $\mathbf{vpart}(x) = x$; for a right-hand side xB (for $B \in N$, $x \in V^*$) we have $\mathbf{vpart}(xB) = x$.

□

We assume some fixed regular grammar (V, N, P) throughout this paper. Note that, unlike usual grammars (for parsing, etc.), we do not have a "start symbol". We have chosen to use right regular grammars, instead of left regular grammars (which have the same descriptive power), because they are symmetrical (under reversal); with of this choice, we must treat both left-to-right and right-to-left algorithms.

Definition 1.5 (Languages of strings and productions): We define function $\mathcal{L}$, mapping strings in $V^* \cup V^*N$ to regular languages over V, as follows (for $w \in V^*$):

$$\mathcal{L}(w) = \{w\}$$

and (for $B \in N$)

$$\mathcal{L}(wB) = \{w\}\mathcal{L}(B)$$

We extend function $\mathcal{L}$ to map productions and nonterminals to the regular languages they denote as follows (for $A \longrightarrow w \in P$):

$$\mathcal{L}(A \longrightarrow w) = \mathcal{L}(w)$$

and

$$\mathcal{L}(A) = (\cup\, w : p \in P \wedge A \in \mathbf{lhs}(p) : \mathcal{L}(p))$$

Since this definition may be recursive, we naturally take the usual fixed-point definition, which always yields regular languages. □

Property 1.6 (Language of a production): We have the following useful property of the language of a production $p \in P$:

$$\mathcal{L}(p) \subseteq \mathbf{vpart}(\mathbf{rhs}(p))V^*$$

□

Intuitively, the above property holds because all words in the language denoted by some production p have $\mathbf{vpart}(\mathbf{rhs}(p))$ as their prefix.

Throughout this paper, we adopt the convention of extending a given function which takes elements of some set D so that it takes elements of 2^D (sets of elements taken from D). (Typically, this is used to extend a function which takes one production, giving a function which takes a set of productions.)

Definition 1.7 (Chain rules): Productions of the form $A \longrightarrow B$ (for $A, B \in N$) are known as *chain rules*. (When B has been recognized as the left-hand side of a production matching a substring, production $A \longrightarrow B$ has been matched as well.) For this reason, we define function $\mathbf{crule} \in 2^P \longrightarrow 2^P$ (where 2^P denotes the set of all subsets of our production set P) as:

$$\mathbf{crule}(U) = \{\, A \longrightarrow B \mid A \longrightarrow B \in P \wedge B \in \mathbf{lhs}(U) \,\}$$

We define function $\mathbf{crule}^*$ to be the reflexive and transitive closure of function **crule**.

□

2 Problem specification and a naïve algorithm

We begin this section with a precise specification of the regular grammar pattern matching problem.

Definition 2.1 (Regular pattern matching problem): Given an input string $S \in V^*$, and our regular grammar, establish postcondition RPM :

$$O = \{\, (l, p) \mid lu = S \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$$

□

Intuitively, this means that we are registering all productions which match some substring of S, along with the left context (in S) of the match location. (That is, for simplicity, we are registering our matches by their begin-point.) We will use the notation O_x to refer to the set of productions in O which match with left context x (a prefix of S). More formally, $O_x = \{\, p \mid (x, p) \in O \,\}$.

We can now give our naïve (and nondeterministic) algorithm

Algorithm 2.2:

$O := \emptyset$;
for $l, u : lu = S \rightarrow$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
rof{ RPM }

□

Note that we are still making some assumptions about our ability to evaluate membership in $\mathcal{L}(p)$.

In the next two sections, we consider adding more determinism to our first algorithm. We will use the property that substrings of S can be characterized as "prefixes of suffixes" of S or as "suffixes of prefixes" of S.

3 Left-to-right algorithms

We begin by deciding to traverse input string S from left-to-right in the following algorithm.

Algorithm 3.1:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
od$\{\ RPM\ \}$

□

This algorithm is still far from practical to implement, and we now consider adding an inner repetition to implement the update of O. The inner repetition can consider prefixes of u in either increasing order or decreasing order. We begin by considering the former.

Algorithm 3.2:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{lv\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

The test $v \in \mathcal{L}(p)$ is particularly difficult to implement — indeed, almost as difficult as the problem we are trying to solve. There does not appear to be an easy manner in which to improve the algorithm, and we abandon its development here.

The following (alternative) algorithm considers prefixes of u in order of decreasing length.

Algorithm 3.3:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $v, r := u, \varepsilon;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge u \in \mathcal{L}(p) \,\};$
  do $v \neq \varepsilon \rightarrow$
    $v, r := v \searrow 1, (v \nearrow 1)r;$
    $O := O \cup \{lv\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od{ *RPM* }

□

Improving this algorithm appears to be as difficult as the previous one. As a result, we do not pursue either of them any further.

3.1 A recursive algorithm

We can also develop a recursive algorithm which traverses input string S from left-to-right. Before presenting the recursive version, we first examine another imperative algorithm traversing S from left-to-right. It considers productions p and decompositions $u, r : ur = S$ such that $\mathbf{vpart}(p) \in \mathbf{suff}(u)$. For this algorithm only, we make two trivial restrictions to patterns and the input string: $S \neq \varepsilon$ and there is no production p such that $\mathbf{rhs}(p) = \varepsilon$. We also define the following predicate, which makes the subsequent algorithms more concise:

Definition 3.4 (Predicate R): Predicate R takes an argument in $N \times V^* \times (N \cup V) \times V^*$:

$$R(A, w, a, u) \equiv A \longrightarrow wa \in P \wedge w \in \mathbf{suff}(u)$$

□

Algorithm 3.5:

$u, r := \varepsilon, S;\ O := \emptyset;$
do $r \neq \varepsilon \rightarrow$
  { $ur = S$ }
  { Deal with productions without a nonterminal in the RHS. }
  $O := O \cup \{\, (u \searrow |w|, A \longrightarrow w(r \nwarrow 1)) \mid R(A, w, r \nwarrow 1, u) \,\};$
  { Deal with productions with a nonterminal in the RHS. }
  $O := O \cup \{\, (u \searrow |w|, A \longrightarrow wB) \mid R(A, w, B, u) \wedge B \in N \wedge$
    $\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset \,\};$
  $u, r := u(r \nwarrow 1), r \swarrow 1;$
od{ *RPM* }

□

We now present the recursive version of the same algorithm. Beginning with $O = \emptyset$, the procedure invocation $mat(\varepsilon, S)$ will yield postcondition RPM.

Algorithm 3.6:

proc $mat(u, r) \rightarrow$
$\{\ ur = S \wedge r \neq \varepsilon\ \}$
{ Deal with productions without a nonterminal in the RHS. }
$O := O \cup \{\ (u \searrow |w|, A \longrightarrow w(r \nwarrow 1)) \mid R(A, w, r \nwarrow 1, u)\ \}$;
{ Deal with productions with a nonterminal in the RHS. }
$O := O \cup \{\ (u \searrow |w|, A \longrightarrow wB) \mid R(A, w, B, u) \wedge B \in N \wedge$
$\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset\ \}$;
if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1)$
$[\!]\ (r \swarrow 1) = \varepsilon \rightarrow$ **skip**
fi
corp

□

This algorithm is guaranteed to terminate, since S is a finite string, and the division between u and r (in S) is monotonically moving from left-to-right with each recursive call.

The algorithm can be made more efficient (in the next few pages) by moving the second update of O below the recursive call.

The expression $\mathbf{suff}(u)$ occurs in several places within the update of O. We could introduce a new parameter such that $U : U = \mathbf{suff}(u)$. In the above algorithm, all tests for membership in $\mathbf{suff}(u)$ involve prefixes of $\mathbf{rhs}(P)$. For this reason, we can introduce the more general $U : \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))$. This is established with $U = \{\varepsilon\}$ in the initial invocation of mat. We now derive the new value for U in the recursive invocation of mat, based upon the old value:

$\mathbf{suff}(u(r \nwarrow 1)) \cap \mathbf{pref}(\mathbf{rhs}(P))$
$=$ $\{$ property of **suff** $\}$
$(\mathbf{suff}(u)(r \nwarrow 1) \cup \{\varepsilon\}) \cap \mathbf{pref}(\mathbf{rhs}(P))$
$=$ $\{ \cup$ distributes over $\cap$; $\varepsilon \in \mathbf{pref}(\mathbf{rhs}(P))\ \}$
$(\mathbf{suff}(u)(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\}$
$=$ $\{\ \{wa\} \cap \mathbf{pref}(W) \neq \emptyset \equiv (\{w\} \cap \mathbf{pref}(W))\{a\} \cap \mathbf{pref}(W) \neq \emptyset\ \}$
$((\mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P)))(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\}$
$=$ $\{$ definition of $U\ \}$
$(U(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\}$

The domain of U is finite, and it conceptually represents a *state*. There are some very efficient means of representing this particular domain, which is related to the Aho-Corasick state function. The new value of U can be more easily computed using the following function.

Definition 3.7 (Function τ): We define function $\tau \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \times (N \cup V) \longrightarrow 2^{\mathbf{pref}(\mathbf{rhs}(P))}$ as

$$\tau(U, a) = (Ua \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\emptyset\}$$

□

This function is easily precomputed, since it corresponds to the forward trie constructed from the right-hand sides of productions. The updates of O can be made much simpler with the introduction of the following function (most of which can also be precomputed):

Definition 3.8 (Function *Output*): Function $Output \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \times V^* \longrightarrow V^* \times 2^{\mathbf{pref}(\mathbf{rhs}(P))}$ is defined as

$$Output(U, u) = (\cup\, A, w : A \longrightarrow w \in P \wedge w \in U : \{u \searrow |w|\} \times \mathbf{crule}^*(A \longrightarrow w))$$

□

These two functions are used in the following version of our algorithm:

Algorithm 3.9:

---

proc $mat(u, r, U) \rightarrow$
  $\{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
  $O := O \cup Output(\tau(U, r \nwarrow 1), u);$
  if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
  $[\!]$ $(r \swarrow 1) = \varepsilon \rightarrow$ **skip**
  fi;
  $O := O \cup (\cup\, B : B \in N \wedge \mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset : Output(\tau(U, B), u))$
corp

□

The second update of O still contains the predicate $\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset$. With this update appearing after the recursive call to mat, all productions matching at u will already appear in O. The predicate can therefore be replaced with $B \in \mathbf{lhs}(O_u)$ in the following algorithm:

Algorithm 3.10:

---

proc $mat(u, r, U) \rightarrow$
  $\{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
  $O := O \cup Output(\tau(U, r \nwarrow 1), u);$
  if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
  $[\!]$ $(r \swarrow 1) = \varepsilon \rightarrow$ **skip**
  fi;
  $O := O \cup (\cup\, B : B \in N \wedge B \in \mathbf{lhs}(O_u) : Output(\tau(U, B), u))$
corp

□

For our final improvement, we change the algorithm such that at any given $u, r : ur = S$, we only register local matches (those p matching at u). Any other (nonlocal) matches are returned by the procedure to be locally registered at their appropriate local u level. To make the new version concise, we redefine our output function as follows:

Definition 3.11 (Function *Output*): Function $Output \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \longrightarrow Naturals \times 2^P$ now registers matching productions with the number of levels that they must be passed back for matching

$$Output(U) = (\cup\ A, w : A \longrightarrow w \in P \wedge w \in U : \{|w|\} \times \mathbf{crule}^*(A \longrightarrow w))$$

□

Since this function does not depend upon u (unlike our earlier definition), we can fully precompute it. The new matching scheme is presented in the final recursive algorithm (in which we introduce variable *matches* to hold the intermediate matches):

Algorithm 3.12:

---

proc $mat(u, r, U) \rightarrow$
$\quad \{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
$\quad matches := Output(\tau(U, r \nwarrow 1));$
$\quad \mathbf{if}\ (r \swarrow 1) \neq \varepsilon \rightarrow matches := matches \cup mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
$\quad [\!]\ (r \swarrow 1) = \varepsilon \rightarrow \mathbf{skip}$
$\quad \mathbf{fi};$
$\quad matches := matches \cup (\cup\ B : B \in N \wedge$
$\qquad\qquad B \in \mathbf{lhs}(matches_0) : Output(\tau(U, B)));$
$\quad O := O \cup \{u\} \times matches_0;$
$\quad mat := (\cup\ i : 0 < i : \{i - 1\} \times matches_i)$
corp

□

For efficiency reasons, we also consider two methods by which we can represent the set *matches*. Both of the methods rely upon the fact that P is a finite set, and that the integers (in the first components of variable *matches*) are in $[0, (\mathbf{MAX}\ p : p \in P : |p|) - 1]$ (the upperbound is one less than the length of the longest production).

1. Use signature $matches \in P \longrightarrow 2^{[0,(\mathbf{MAX}\ p:p\in P:|p|)-1]}$. In other words, map each production to a set of levels that it must be passed up to match locally. The representation can use an array (indexed by an integer associated with each production in P) of bit vectors (each of length $(\mathbf{MAX}\ p : p \in P : |p|)$) indicating the number of levels back that the production must be passed for local registration. All of the updates of *matches* can be done using bit vector operations (bitwise-OR). Finding which productions have matched locally is done by looking up those productions whose corresponding bit vector has the 0 bit set. Computing the final return value of the procedure is done by bit shifting all of the entries in the representation of *matches*.

2. Use signature $matches \in [0, (\mathbf{MAX}\ p : p \in P : |p|) - 1] \longrightarrow 2^P$. In this representation, we map each level (to be passed on the return) to the set of productions which match at that level. The representation can use an array (indexed by level number) of bit vectors (each of length $|P|$). Again, all of the updates of *matches* can be done using bit vector operations (bitwise-OR). Finding which productions have matched locally is done by looking up those productions in entry 0 of *matches*. The return value can be computed trivially if the array is represented as a circular array (in which the current 0 position is determined by a separate pointer); in this case, the return value only involves updating the pointer.

These representations would also yield interesting representations for function *Output*.

4 Right-to-left algorithms

In this section, we consider algorithms which traverse input string from right-to-left in general. Our first algorithm consists of a single repetition:

Algorithm 4.1:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
od{ *RPM* }

□

The update of O in the repetition must still be implemented. This will be addressed shortly in Section 4.1. Another possible implementation is to use a nested repetition to traverse the prefixes of u in order of increasing length, as in the following algorithm:

Algorithm 4.2:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od{ *RPM* }

□

In Section 4.2, this algorithm will be used as the starting point for the derivation of a particularly efficient new algorithm.

The inner repetition of the above algorithm could also be structured to consider prefixes of u in order of decreasing length:

Algorithm 4.3:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := u, \varepsilon;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge u \in \mathcal{L}(p) \,\};$
  do $v \neq \varepsilon \rightarrow$
    $v, r := v \searrow 1, (v \nearrow 1)r;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

It does not appear that there are any straightforward methods for improving the efficiency of Algorithm 4.3.

4.1 Improving the single-repetition algorithm

In this section, we make some improvements to Algorithm 4.1. The update of O can be made much simpler by introducing a new variable W:

$$W = \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\}$$

The resulting algorithm is:

Algorithm 4.4:

$l, u := S, \varepsilon;$
$W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \varepsilon \in \mathcal{L}(x) \,\};$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\};$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

The initialization of W can be greatly simplified using the chain-rule relation **crule**:

Algorithm 4.5:

$l, u := S, \varepsilon;$
$W := \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid \mathbf{rhs}(p) = \varepsilon \,\}));$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\};$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

We now need an efficient implementation of the update of W. We can derive the update as follows, starting with the new value (after the updates of l and u):

$$
\begin{array}{ll}
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}((l \nearrow 1)u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{property of } \mathbf{pref} \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (\{\varepsilon\} \cup (l \nearrow 1)\mathbf{pref}(u)) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \cap \text{ distributes over } \cup \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge ((l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \vee \varepsilon \in \mathcal{L}(x)) \,\} \\
= & \quad \{\, \text{split quantification} \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \varepsilon \in \mathcal{L}(x) \,\} \\
 & \cup \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{use } \mathbf{crule}^* \text{ for first quantification} \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{change of bound variable in second quantification: } x = (l \nearrow 1)x' \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}((l \nearrow 1)x') \neq \emptyset \,\} \\
= & \quad \{\, a\mathbf{pref}(u) \cap \mathcal{L}(ax') \neq \emptyset \Rightarrow \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset; \text{ first conjunct} \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset \,\} \\
= & \quad \{\, az \in \mathbf{suff}(W) \Rightarrow z \in \mathbf{suff}(W) \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \\
 & \qquad\qquad \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset \,\} \\
= & \quad \{\, \text{invariant on } W \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup (l \nearrow 1)\{\, x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge x' \in W \,\}
\end{array}
$$

Since the domain of W is finite, we can define a state set, and an initial state.

Definition 4.6 (State set): State set is defined as $Q = \mathbf{suff}(\mathbf{rhs}(P))$. The initial state q_0 is defined to be

$$\{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid \mathbf{rhs}(p) = \varepsilon \,\}))$$

□

Using this state set, we can also define a *transition* function (using the derivation above).

Definition 4.7 (Function σ): Transition function $\sigma \in Q \times V \longrightarrow Q$ is defined as

$$\sigma(q, a) = q_0 \cup a\{\, y \mid ay \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge y \in q \,\}$$

□

The state set and the transition function can be used in the final algorithm:

Algorithm 4.8:

$l, u := S, \varepsilon;$
$W := q_0;$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \sigma(W, l \nearrow 1);$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

This algorithm can be simplified somewhat by defining an output function for the update of O, and by applying Aho-Corasick-like simplification of the state set.

4.2 Improving an algorithm with two repetitions

We begin with Algorithm 4.2, duplicated here:

Algorithm 4.9:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

In this algorithm, as we consider prefixes of u of increasing length, we can make use of some information already stored in the set O. We will use the variable v to keep track of partial matches corresponding to right-hand sides of productions. Once we have a completed right-hand side, the match can be registered, along with any other matches induced by chain rules. We consider the two possible forms of right-hand sides separately.

We begin by rewriting the set

$$\{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$$

(used in the inner repetition's update of O in the algorithm above, and catering to the simpler form of right-hand side) as

$$\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\})$$

We now turn to the second form of right-hand side. In the following derivation, we rely upon the fact that the outer repetition considers string S from right-to-left. We

would like to register a match when there is some nonterminal $A \in \mathbf{lhs}(O_{lv})$ (that is, A is the left-hand side of some production matching in r, with left context lv) and vA is the right-hand side of some production. More formally, the set of such matches is

$$\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\})$$

We use these two formulas in the following algorithm:

Algorithm 4.10:

$l, u := S, \varepsilon;\ O := \{S\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\});$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\});$
  od
od$\{\ RPM\ \}$

□

The twin updates of O in the inner repetition arise from the fact that we have two different types of right-hand sides to consider.

In the above algorithm, we note that, once $v \notin \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$, it is not possible to find a further match by extending v on the right. It is thus possible to terminate the inner repetition once further iterations are futile. This is done by extending the inner repetition guard to $r \neq \varepsilon$ **cand** $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$.

This change also happens to give us the inner repetition invariant $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$, which is initially true by the redundant initialization of v. This invariant encompasses the information which we will later use to improve the algorithm. For this reason, we would also like to have this as an invariant of the outer repetition. This can be done by adding the initialization $v, r := \varepsilon, \varepsilon$ at the beginning of the program. All of these improvements yield the following algorithm:

Algorithm 4.11:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
  do $r \neq \varepsilon$ **cand** $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\});$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\});$
  od
od$\{\ RPM\ \}$

□

The evaluation of the inner repetition guard can be done by using a trie.

Definition 4.12 (Function τ_{red}): The *trie* for (the finite set of keywords) $\mathbf{rhs}(P)$ (over combined alphabet $N \cup V$) is function $\tau_{red} \in \mathbf{pref}(\mathbf{rhs}(P)) \times V \longrightarrow \mathbf{pref}(\mathbf{rhs}(P)) \cup \{\bot\}$ defined by

$$\tau_{red}(w,a) = \begin{cases} aw & \text{if } wa \in \mathbf{pref}(\mathbf{rhs}(P)) \\ \bot & \text{if } wa \notin \mathbf{pref}(\mathbf{rhs}(P)) \end{cases}$$

This function is known as the *reduced trie* — a compressed version of the function τ given in Definition 3.7. □

Using the trie, we rewrite the conditional conjunct $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$ as

$$\tau_{red}(v, r \nwarrow 1) \neq \bot$$

(This hinges upon the fact that $\mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \subseteq \mathbf{pref}(\mathbf{rhs}(P))$ and $S \in V^*$.) To make the algorithm more concise, we also define the following output function:

Definition 4.13 (Output function): Function $Out \in \mathbf{pref}(\mathbf{rhs}(P)) \longrightarrow 2^P$ is defined by

$$Out(w) = \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = w \,\})$$

□

Function Out is easily precomputed. It is obvious that we can use Out for the first update of O in the inner repetition. We can use this function, along with the reverse trie, to rewrite the second update of O in the inner repetition, as follows:

$$\begin{array}{ll} & \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\}) \\ = & \quad \{\, \text{definition of } \tau_{red} \,\} \\ & \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \tau_{red}(v,A) \wedge A \in \mathbf{lhs}(O_{lv}) \,\}) \\ = & \quad \{\, \text{definition of } Out \,\} \\ & Out(\{\, \tau_{red}(v,A) \mid A \in \mathbf{lhs}(O_{lv}) \,\}) \end{array}$$

Using these two functions yields the following algorithm:

Algorithm 4.14:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times Out(\varepsilon);$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times Out(\varepsilon);$
  do $r \neq \varepsilon$ **cand** $\tau_{red}(v, r \nwarrow 1) \neq \bot \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times Out(v);$
    $O := O \cup \{l\} \times Out(\{\, \tau_{red}(v,A) \mid A \in \mathbf{lhs}(O_{lv}) \,\} \setminus \{\bot\});$
  od
od$\{\ RPM\ \}$

□

4.3 Greater shift distances

In a manner analogous to the Commentz-Walter and Boyer-Moore algorithm derivations in [14, Chapter 4] or [18, 20], we can use the invariant $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$ on subsequent iterations of the outer repetition to make a shift of $k \geq 1$ symbols by replacing the assignment $l, u := l \searrow 1, (l \nearrow 1)u$ by $l, u := l \searrow k, (l \nearrow k)u$.

As with the Commentz-Walter and Boyer-Moore algorithms, we would like an ideal shift distance — the shift distance to the nearest match to the left (in input string S). Formally, this distance is given by: $(\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n)$. Any shift distance less than this is also acceptable, and we define a safe shift distance (similar to that given in [14, Chapter 4] and in [18, 20]).

Definition 4.15 (Safe shift distance): A shift distance k satisfying

$$1 \leq k \leq (\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n)$$

is a *safe shift distance*. We call the upperbound (the quantification) the *maximal safe shift distance* or the *ideal shift distance*. □

Using a safe shift distance, the update of l, u then becomes $l, u := l \searrow k, (l \nearrow k)u$. In order to compute a safe shift distance, we will weaken predicate $\mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset$ (which we call the *ideal shift predicate*) in the range of the maximal safe shift distance quantification. This technique of using predicate weakening to find a more easily computed shift distance was introduced in [18] and used in [14, 20]. The weakest predicate, *true*, yields a shift distance of 1 — which, in turn, yields our last algorithm. We now find a weakening of the ideal shift predicate which is stronger than *true*, but still precomputable.

In the following weakening, we will first remove the dependency of the ideal shift predicate on r and then l. The particular weakening that we derive will prove to yield precomputable shift tables. Assuming $1 \leq n \leq |l|$ and the (implied) invariant $u = vr$, we begin with the ideal shift predicate:

$$\begin{array}{ll}
& \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset \\
\equiv & \quad \{\, \text{invariant: } u = vr \,\} \\
& \mathbf{pref}((l \nearrow n)vr) \cap \mathcal{L}(P) \neq \emptyset \\
\Rightarrow & \quad \{\, \text{discard lookahead to } r\text{: } r \in V^*, \text{ monotonicity of } \mathbf{pref} \text{ and } \cap \,\} \\
& \mathbf{pref}((l \nearrow n)vV^*) \cap \mathcal{L}(P) \neq \emptyset \\
\Rightarrow & \quad \{\, \text{domain of } l \text{ and } n\text{: } n \leq |l|, \text{ so } (l \nearrow n) \in V^n \,\} \\
& \mathbf{pref}(V^n vV^*) \cap \mathcal{L}(P) \neq \emptyset \\
\equiv & \quad \{\, \text{property of } \mathbf{pref} \text{ (see [14, Chapter 2])} \,\} \\
& V^n vV^* \cap \mathcal{L}(P)V^* \neq \emptyset \\
\Rightarrow & \quad \{\, \text{property of } \mathcal{L}(P)\text{: } \mathcal{L}(P) \subseteq \mathbf{vpart}(\mathbf{rhs}(P))V^* \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P))V^*V^* \neq \emptyset \\
\equiv & \quad \{\, V^*V^* = V^* \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset \\
\equiv & \quad \{\, \text{property of languages (see [14, Chapter 2])} \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset \vee V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset
\end{array}$$

Note that we have removed the dependence upon l, meaning that we can remove the upper bound on n in the **MIN** quantification. Given the last line above, we have the following approximation:

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n) \\ \geq & \quad \{\,\text{derivation above, disjunction in the resulting range predicate}\,\} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n v V^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset : n) \\ & \mathbf{min}(\mathbf{MIN}\ n : 1 \leq n \wedge V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset : n) \end{array}$$

This last line above can be written more concisely with the introduction of a pair of auxiliary functions.

Definition 4.16 (Functions b_1, b_2): We define two functions b_1, b_2 with signatures $b_1, b_2 \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \longrightarrow \mathit{Naturals}$ (the domain comes from the fact that $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$) as:

$$\begin{array}{rcl} b_1(x) & = & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n x V^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset : n) \\ b_2(x) & = & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset : n) \end{array}$$

These two functions are, in fact, reversed versions of the Commentz-Walter shift functions (known as d_1 and d_2) for (finite) keyword set $\mathbf{vpart}(\mathbf{rhs}(P))$. Their precomputation is extremely well understood, and is detailed in [18, 20]. The precomputation algorithm involves the trie (for $\mathbf{rhs}(P)$) τ_{red}, introduced earlier. □

Using the auxiliary functions, our approximation of the ideal shift distance is $b_1(v)\,\mathbf{min}\,b_2(v)$. Using the new shift distance yields our final algorithm (with new variable h to hold the shift distance):

Algorithm 4.17:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times \mathit{Out}(\varepsilon);$
do $l \neq \varepsilon \rightarrow$
  $h := b_1(v)\,\mathbf{min}\,b_2(v);$
  $l, u := l \searrow h, (l \nearrow h)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \mathit{Out}(\varepsilon);$
  do $r \neq \varepsilon$ **cand** $\tau_{red}(v, r \nwarrow 1) \neq \perp \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \mathit{Out}(v);$
    $O := O \cup \{l\} \times \mathit{Out}(\{\,\tau_{red}(v, A) \mid A \in \mathbf{lhs}(O_{lv})\,\} \setminus \{\perp\});$
  od
od$\{\ RPM\ \}$

□

4.3.1 Specializing the pattern matching algorithm

By restricting the form of the regular grammars, we can specialize Algorithm 4.17 to obtain a reversed version of the Commentz-Walter and the Boyer-Moore algorithms.

The most straightforward specialization is to restrict the productions to be of the form $A \longrightarrow w$ for $w \in V^*$ and each nonterminal appears as at most one left-hand side. From this restriction, we have $\mathbf{vpart}(\mathbf{rhs}(P)) = \mathbf{rhs}(P)$. In this case, the set of productions essentially represents a finite set of keywords $\mathbf{rhs}(P)$ (the left-hand sides are redundant). We can then delete the second update of O in the inner repetition, since it is used exclusively for productions with a nonterminal as the right-most symbol of the right-hand side. The resulting algorithm is the reversal to the Commentz-Walter algorithm without lookahead. (For a presentation of the Commentz-Walter algorithm, see [14, Section 4.4] or [19].)

We can similarly restrict the set of productions to consist of a single production $A \longrightarrow w$ for $w \in V^*$. In this case, we obtain a variant of the Boyer-Moore algorithm. (For a number of variants of the Boyer-Moore algorithm, see [14, Section 4.5] and [11].)

4.3.2 Improving the algorithm

We now briefly mention two approaches to improving this algorithm (both of which are discussed in more detail in [14, Chapters 4 and 5]):

- In the derivation of a weakened range predicate, we eliminated any (right) lookahead into string r by replacing it with V^*. We could have retained a single symbol of lookahead, by replacing r with $(r \nwarrow 1)V^*$. We could then have further manipulated the predicate and defined a third shift function.

- Also in the derivation, we discarded any (left) lookahead into l by replacing $l \nearrow n$ with V^n. We could have kept a single symbol of lookahead by replacing $l \nearrow n$ with $V^{n-1}(l \nearrow 1)$. This would also have yielded a different shift function.

5 Conclusions

We have succeeded in deriving and presenting a number of new algorithms for the regular grammar pattern matching problem. The interesting, and possibly efficient, algorithms include a generalization of the Boyer-Moore algorithm, a recursive algorithm (which resembles a generalized Aho-Corasick algorithm), and an algorithm based on a type of finite automaton.

Interestingly, the Boyer-Moore type algorithm presented here was only derived after a regular tree pattern matching version of the algorithm was developed [15].

Future directions include implementing all of the algorithms and collecting benchmarking data.

Acknowledgements:

I would like to thank Richard Watson, Dr. Kees Hemerik, Dr. Gerard Zwaan, and Prof. Dr. Frans Kruseman Aretz for their technical assistance during the development of these algorithms. A great deal of feedback was also provided by the participants at the Prague Stringologic Club Workshop '96 in Prague, in particular the organizers Prof. Dr. Melichar, Dr. Martin Bloch, and Ing. Jan Holub (who also provided a great deal of help with the typesetting). I thank Drs. Nanette Saes for proofreading and offering suggestions for improvement of this paper.

References

[1] Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search, *Comm. ACM,* **18**(6) (1975) 333–340.

[2] Aho, A.V.: Pattern matching in strings, in: Book, R.V., ed., *Formal Language Theory: Perspectives and Open Problems.* (Academic Press, New York, 1980) 325–347.

[3] Aho, A.V.: Algorithms for finding patterns in strings, in: van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Vol. A.* (North-Holland, Amsterdam, 1990) 257–300.

[4] Baeza-Yates, R.: *Efficient Text Searching.* (Ph.D dissertation, University of Waterloo, Canada, May 1989).

[5] Boyer, R.S., Moore, J.S.: A fast string searching algorithm, *Comm. ACM,* **20**(10) (1977) 62–72.

[6] Commentz-Walter, B.: A string matching algorithm fast on the average, in: Maurer, H.A., ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer-Verlag, Berlin, 1979) 118–132.

[7] Commentz-Walter, B.: A string matching algorithm fast on the average, Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[8] Crochemore, M., Rytter, W.: *Text Algorithms.* (Oxford University Press, Oxford, England, 1994).

[9] Fredkin, E.: Trie memory, *Comm. ACM* **3**(9) (1960) 490–499.

[10] Gonnet, G.H., Baeza-Yates, R.: *Handbook of Algorithms and Data Structures (In Pascal and C).* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[11] Hume, S.C., Sunday, D.: Fast string searching, *Software—Practice and Experience* **21**(11) (1991) 1221–1248.

[12] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[13] Watson, B.W.: The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/pattm/`.

[14] Watson, B.W.: *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995). Contact `watson@RibbitSoft.com`.

[15] Watson, B.W.: A Boyer-Moore (or Watson-Watson) type algorithm for regular tree pattern matching, in: Aarts, E.H.L., ten Eikelder, H.M.M., Hemerik, C., Rem, M., eds., *Simplex Sigillum Veri: Een Liber Amicorum voor prof.dr.*

F.E.J. Kruseman Aretz (Eindhoven University of Technology, ISBN 90-386-0197-2, 1995) 315–320.

[16] Watson, B.W.: A new algorithm for regular grammar pattern matching, *Proceedings of the Fourth European Symposium on Algorithms,* Barcelona, Spain, 1996.

[17] Watson, B.W., Watson, R.E.: A Boyer-Moore type algorithm for regular expression pattern matching, Computing Science Report 94/31, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `watson@RibbitSoft.com`.

[18] Watson, B.W., Zwaan, G.: A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992. Available by e-mail from `watson@RibbitSoft.com` or `wsinswan@win.tue.nl`.

[19] Watson, B.W., Zwaan, G.: A taxonomy of sublinear multiple keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `wsinswan@win.tue.nl`.

[20] Watson, B.W., Zwaan, G.: A taxonomy of sublinear multiple keyword pattern matching algorithms, to appear in: *Science of Computer Programming,* (1996).

[21] Zwaan, G.: Sublinear pattern matching, in: Aarts, E.H.L., ten Eikelder, H.M.M., Hemerik, C., Rem, M., eds., *Simplex Sigillum Veri: Een Liber Amicorum voor prof.dr. F.E.J. Kruseman Aretz* (Eindhoven University of Technology, ISBN 90-386-0197-2, 1995) 335–350.