Programming Techniques
Glenn Manacher Editor

Efficient String Matching: An Aid to Bibliographic Search

Alfred V. Aho and Margaret J. Corasick
Bell Laboratories

This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text. The algorithm consists of constructing a finite state pattern matching machine from the keywords and then using the pattern matching machine to process the text string in a single pass. Construction of the pattern matching machine takes time proportional to the sum of the lengths of the keywords. The number of state transitions made by the pattern matching machine in processing the text string is independent of the number of keywords. The algorithm has been used to improve the speed of a library bibliographic search program by a factor of 5 to 10.

Keywords and Phrases: keywords and phrases, string pattern matching, bibliographic search, information retrieval, text-editing, finite state machines, computational complexity.

CR Categories: 3.74, 3.71, 5.22, 5.25

Authors' present addresses: A. V. Aho, Bell Laboratories, Murray Hill, N.J. 07974. M. J. Corasick, The MITRE Corporation, Bedford, Mass. 01730.

1. Introduction

In many information retrieval and text-editing applications it is necessary to be able to locate quickly some or all occurrences of user-specified patterns of words and phrases in text. This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords and phrases in an arbitrary text string.

The approach should be familiar to those acquainted with finite automata. The algorithm consists of two parts. In the first part we construct from the set of keywords a finite state pattern matching machine; in the second part we apply the text string as input to the pattern matching machine. The machine signals whenever it has found a match for a keyword.

Using finite state machines in pattern matching applications is not new [4, 8, 17], but their use seems to be frequently shunned by programmers. Part of the reason for this reluctance on the part of programmers may be due to the complexity of programming the conventional algorithms for constructing finite automata from regular expressions [3, 10, 15], particularly if state minimization techniques are needed [2, 14]. This paper shows that an efficient finite state pattern matching machine can be constructed quickly and simply from a restricted class of regular expressions, namely those consisting of finite sets of keywords. Our approach combines the ideas in the Knuth-Morris-Pratt algorithm [13] with those of finite state machines.

Perhaps the most interesting aspect of this paper is the amount of improvement the finite state algorithm gives over more conventional approaches. We used the finite state pattern matching algorithm in a library bibliographic search program. The purpose of the program is to allow a bibliographer to find in a citation index all titles satisfying some Boolean function of keywords and phrases. The search program was first implemented with a straightforward string matching algorithm. Replacing this algorithm with the finite state approach resulted in a program whose running time was a fifth to a tenth of the original program on typical inputs.

2. A Pattern Matching Machine

This section describes a finite state string pattern matching machine that locates keywords in a text string. The next section describes the algorithms to construct such a machine from a given finite set of keywords.

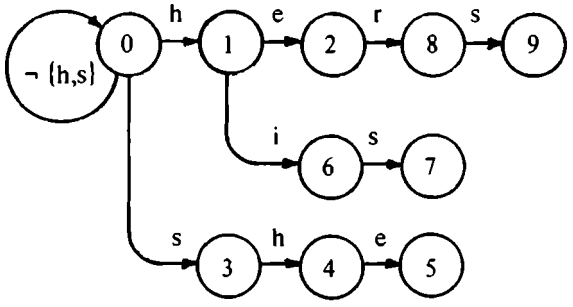
In this paper a *string* is simply a finite sequence of symbols. Let $K = \{y_1, y_2, \ldots, y_k\}$ be a finite set of strings which we shall call *keywords* and let x be an arbitrary string which we shall call the *text string*. Our problem is to locate and identify all substrings of x which are keywords in K. Substrings may overlap with one another.

A pattern matching machine for K is a program which takes as input the text string x and produces as output the locations in x at which keywords of K appear as substrings. The pattern matching machine consists of a set of *states*. Each state is represented by a number. The machine processes the text string x by successively reading the symbols in x, making state transitions and occa-

sionally emitting output. The behavior of the pattern matching machine is dictated by three functions: a goto function g, a failure function f, and an output function *output*.

Figure 1 shows the functions used by a pattern matching machine for the set of keywords {he, she, his, hers}.

Fig. 1. Pattern matching machine.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	*output*(i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

One state (usually 0) is designated as a *start* state. In Figure 1 the states are 0, 1, . . . , 9. The goto function g maps a pair consisting of a state and an input symbol into a state or the message *fail*. The directed graph in Figure 1(a) represents the goto function. For example, the edge labeled h from 0 to 1 indicates that $g(0, \text{h}) = 1$. The absence of an arrow indicates *fail*. Thus, $g(1, \sigma) = fail$ for all input symbols σ that are not e or i. All our pattern matching machines have the property that $g(0, \sigma) \neq fail$ for all input symbols σ. We shall see that this property of the goto function on state 0 ensures that one input symbol will be processed by the machine in every machine cycle.

The failure function f maps a state into a state. The failure function is consulted whenever the goto function reports *fail*. Certain states are designated as output states which indicate that a set of keywords has been found. The output function formalizes this concept by associating a set of keywords (possibly empty) with every state.

An *operating cycle* of a pattern matching machine is defined as follows. Let s be the current state of the machine and a the current symbol of the input string x.

1. If $g(s, a) = s'$, the machine makes a *goto transition*. It enters state s' and the next symbol of x becomes the current input symbol. In addition, if *output*(s') $\neq$ *empty*, then the machine emits the set *output*(s') along with the position of the current input symbol. The operating cycle is now complete.

2. If $g(s, a) = fail$, the machine consults the failure function f and is said to make a *failure transition*. If $f(s) = s'$, the machine repeats the cycle with s' as the current state and a as the current input symbol.

Initially, the current state of the machine is the start state and the first symbol of the text string is the current input symbol. The machine then processes the text string by making one operating cycle on each symbol of the text string.

For example, consider the behavior of the machine M that uses the functions in Figure 1 to process the text string "ushers." Figure 2 indicates the state transitions made by M in processing the text string.

Fig. 2. Sequence of state transitions.

u	s	h	e	r	s
0	0	3	4	5	8	9
				2		

Consider the operating cycle when M is in state 4 and the current input symbol is e. Since $g(4, \text{e}) = 5$, the machine enters state 5, advances to the next input symbol and emits *output*(5), indicating that it has found the keywords "she" and "he" at the end of position four in the text string.

In state 5 on input symbol r, the machine makes two state transitions in its operating cycle. Since $g(5, \text{r}) = fail$, M enters state $2 = f(5)$. Then since $g(2, \text{r}) = 8$, M enters state 8 and advances to the next input symbol. No output is generated in this operating cycle.

The following algorithm summarizes the behavior of a pattern matching machine.

Algorithm 1. Pattern matching machine.

Input. A text string $x = a_1 a_2 \cdots a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function *output*, as described above.

Output. Locations at which keywords occur in x.

Method.

```
begin
  state ← 0
  for i ← 1 until n do
    begin
      while g(state, a_i) = fail do state ← f(state)
      state ← g(state, a_i)
      if output(state) ≠ empty then
        begin
          print i
          print output(state)
        end
    end
end
```

Each pass through the **for**-loop represents one operating cycle of the machine.

Algorithm 1 is patterned after the Knuth-Morris-Pratt algorithm for finding one keyword in a text string [13] and can be viewed as an extension of the "trie" search discussed in [11]. Hopcroft and Karp (unpublished) have suggested a scheme similar to Algorithm 1 for finding the first occurrence of any of a finite set of keywords in a text string [13]. Section 6 of this paper discusses a deterministic finite automaton version of Algorithm 1 that avoids all failure transitions.

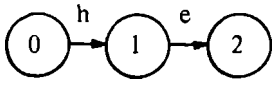
3. Construction of Goto, Failure, and Output Functions

We say that the three functions g, f, and *output* are *valid* for a set of keywords if with these functions Algorithm 1 indicates that keyword y ends at position i of text string x if and only if $x = uyv$ and the length of uy is i.

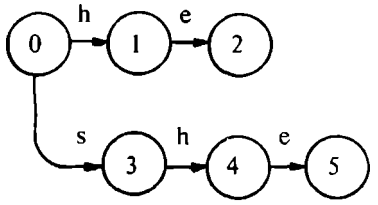
We shall now show how to construct valid goto, failure and output functions from a set of keywords. There are two parts to the construction. In the first part we determine the states and the goto function. In the second part we compute the failure function. The computation of the output function is begun in the first part of the construction and completed in the second part.

To construct the goto function, we shall construct a *goto graph*. We begin with a graph consisting of one vertex which represents the state 0. We then enter each keyword y into the graph, by adding a directed path to the graph that begins at the start state. New vertices and edges are added to the graph so that there will be, starting at the start state, a path in the graph that spells out the keyword y. The keyword y is added to the output function of the state at which the path terminates. We add new edges to the graph only when necessary.

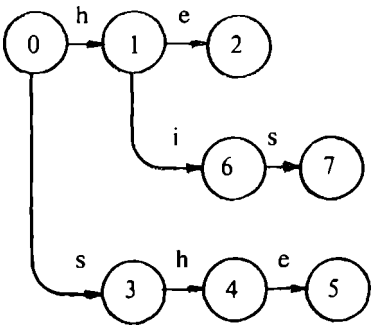
For example, suppose {he, she, his, hers} is the set of keywords. Adding the first keyword to the graph, we obtain:



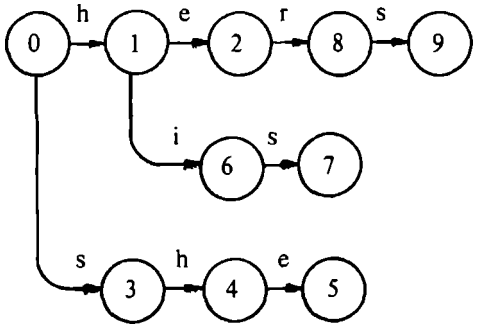
The path from state 0 to state 2 spells out the keyword "he"; we associate the output "he" with state 2. Adding the second keyword "she," we obtain the graph:



The output "she" is associated with state 5. Adding the keyword "his," we obtain the following graph. Notice that when we add the keyword "his" there is already an edge labeled h from state 0 to state 1, so we do not need to add another edge labeled h from state 0 to state 1. The output "his" is associated with state 7.



Adding the last keyword "hers," we obtain:



The output "hers" is associated with state 9. Here we have been able to use the existing edge labeled h from state 0 to 1 and the existing edge labeled e from state 1 to 2.

Up to this point the graph is a rooted directed tree. To complete the construction of the goto function we add a loop from state 0 to state 0 on all input symbols other than h or s. We obtain the directed graph shown in Figure 1(a). This graph represents the goto function.

The failure function is constructed from the goto function. Let us define the *depth* of a state s in the goto graph as the length of the shortest path from the start state to s. Thus in Figure 1(a), the start state is of depth 0, states 1 and 3 are of depth 1, states 2, 4, and 6 are of depth 2, and so on.

We shall compute the failure function for all states of depth 1, then for all states of depth 2, and so on, until the failure function has been computed for all states (except state 0 for which the failure function is not defined). The algorithm to compute the failure function f at a state is conceptually quite simple. We make $f(s) = 0$ for all states s of depth 1. Now suppose f has been computed for all states of depth less than d. The failure function for the states of depth d is computed from the failure function for the states of depth less than d. The states of depth d can be determined from the nonfail values of the goto function of the states of depth $d-1$.

Specifically, to compute the failure function for the states of depth d, we consider each state r of depth $d-1$ and perform the following actions.

1. If $g(r, a) = fail$ for all a, do nothing.

2. Otherwise, for each symbol a such that $g(r, a) = s$, do the following:

(a) Set $state = f(r)$.

(b) Execute the statement $state \leftarrow f(state)$ zero or more times, until a value for *state* is obtained such that $g(state, a) \neq fail$. (Note that since $g(0, a) \neq fail$ for all a, such a state will always be found.)

(c) Set $f(s) = g(state, a)$.

For example, to compute the failure function from Figure 1(a), we would first set $f(1) = f(3) = 0$ since 1 and 3 are the states of depth 1. We then compute the failure function for 2, 6, and 4, the states of depth 2. To compute $f(2)$, we set $state = f(1) = 0$; and since $g(0, e) = 0$, we find that $f(2) = 0$. To compute $f(6)$, we set $state = f(1) = 0$; and since $g(0, i) = 0$, we find that $f(6) = 0$. To compute $f(4)$, we set $state = f(3) = 0$; and since $g(0, h) = 1$, we find that $f(4) = 1$. Continuing in this fashion, we obtain the failure function shown in Figure 1(b).

During the computation of the failure function we also update the output function. When we determine $f(s) = s'$, we merge the outputs of state s with the outputs of state s'.

For example, from Figure 1(a) we determine $f(5) = 2$. At this point we merge the output set of state 2, namely {he}, with the output set of state 5 to derive the new output set {he, she}. The final nonempty output sets are shown in Figure 1(c).

The algorithms to construct the goto, failure and output functions from K are summarized below.

Algorithm 2. Construction of the goto function.
Input. Set of keywords $K = \{y_1, y_2, \ldots, y_k\}$.
Output. Goto function g and a partially computed output function *output*.
Method. We assume $output(s)$ is empty when state s is first created, and $g(s, a) = fail$ if a is undefined or if $g(s, a)$ has not yet been defined. The procedure $enter(y)$ inserts into the goto graph a path that spells out y.

```
begin
    newstate ← 0
    for i ← 1 until k do enter(y_i)
    for all a such that g(0, a) = fail do g(0, a) ← 0
end

procedure enter(a_1 a_2 ··· a_m):
begin
    state ← 0; j ← 1
    while g(state, a_j) ≠ fail do
        begin
            state ← g(state, a_j)
            j ← j + 1
        end
    for p ← j until m do
        begin
            newstate ← newstate + 1
            g(state, a_p) ← newstate
            state ← newstate
        end
    output(state) ← {a_1 a_2 ··· a_m}
end
```

The following algorithm, whose inner loop is similar to Algorithm 1, computes the failure function.

Algorithm 3. Construction of the failure function.
Input. Goto function g and output function *output* from Algorithm 2.
Output. Failure function f and output function *output*.
Method.

```
begin
    queue ← empty
    for each a such that g(0, a) = s ≠ 0 do
        begin
            queue ← queue ∪ {s}
            f(s) ← 0
        end
    while queue ≠ empty do
        begin
            let r be the next state in queue
            queue ← queue − {r}
            for each a such that g(r, a) = s ≠ fail do
                begin
                    queue ← queue ∪ {s}
                    state ← f(r)
                    while g(state, a) = fail do state ← f(state)
                    f(s) ← g(state, a)
                    output(s) ← output(s) ∪ output(f(s))
                end
        end
end
```

The first **for**-loop computes the states of depth 1 and enters them in a first-in first-out list denoted by the variable *queue*. The main **while**-loop computes the set of states of depth d from the set of states of depth $d-1$.

The failure function produced by Algorithm 3 is not optimal in the following sense. Consider the pattern matching machine M of Figure 1. We see $g(4, e) = 5$. If M is in state 4 and the current input symbol a_i is not an e, then M would enter state $f(4) = 1$. Since M has already determined that $a_i \neq e$, M does not then need to consider the value of the goto function of state 1 on e. In fact, if the keyword "his" were not present, then M could go directly from state 4 to state 0, skipping an unnecessary intermediate transition to state 1.

To avoid making unnecessary failure transitions we can use f', a generalization of the *next* function from [13], in place of f in Algorithm 1. Specifically, define $f'(1) = 0$. For $i > 1$, define $f'(i) = f'(f(i))$ if, for all input symbols a, $g(f(i), a) \neq fail$ implies $g(i, a) \neq fail$; define $f'(i) = f(i)$, otherwise. However, to avoid making any failure transitions at all, we can use the deterministic finite automaton version of Algorithm 1 given in Section 6.

4. Properties of Algorithms 1, 2, and 3

This section shows that the goto, failure, and output functions constructed by Algorithms 2 and 3 from a given set of keywords K are indeed valid for K.

We say that u is a *prefix* and v is a *suffix* of the string uv. If u is not the empty string, then u is a *proper* prefix. Likewise, if v is not empty, then v is a *proper* suffix.

We say that string u *represents* state s of a pattern matching machine if the shortest path in the goto graph from the start state to state s spells out u. The start state is represented by the empty string.

Our first lemma characterizes the failure function constructed by Algorithm 3.

LEMMA 1. *Suppose that in the goto graph state s is represented by the string u and state t is represented by the string v. Then, $f(s) = t$ if and only if v is the longest proper suffix of u that is also a prefix of some keyword.*

PROOF. The proof proceeds by induction on the length of u (or equivalently the depth of state s). By Algorithm 3 $f(s) = 0$ for all states s of depth 1. Since each state of depth 1 is represented by a string of length 1, the statement of the lemma is trivially true for all strings of length 1.

For the inductive step, assume the statement of Lemma 1 is true for all strings of length less than j, $j > 1$. Suppose $u = a_1 a_2 \cdots a_j$ for some $j > 1$, and v is the longest proper suffix of u that is a prefix of some keyword. Suppose u represents state s and $a_1 a_2 \cdots a_{j-1}$ represents state r. Let $r_1, r_2, \cdots, r_n$ be the sequence of states such that

1. $r_1 = f(r)$,
2. $r_{i+1} = f(r_i)$ for $1 \leqslant i < n$,
3. $g(r_i, a_j) = fail$ for $1 \leqslant i < n$, and
4. $g(r_n, a_j) = t \neq fail$.

(If $g(r_1, a_j) \neq fail$, then $r_n = r_1$.) The sequence $r_1, r_2, \cdots, r_n$ is the sequence of values assumed by the variable *state* in the inner **while**-loop of Algorithm 3. The statement following that **while**-loop makes $f(s) = t$. We claim that t is represented by the longest proper suffix of u that is a prefix of some keyword.

To prove this, suppose v_i represents state r_i for $1 \leqslant i \leqslant n$. By the inductive hypothesis v_1 is the longest proper suffix of $a_1 a_2 \cdots a_{j-1}$ that is a prefix of some keyword; v_2 is the longest proper suffix of v_1 that is a prefix of some keyword; v_3 is the longest proper suffix of v_2 that is a prefix of some keyword, and so on.

Thus v_n is the longest proper suffix of $a_1 a_2 \cdots a_{j-1}$ such that $v_n a_j$ is a prefix of some keyword. Therefore $v_n a_j$ is the longest proper suffix of u that is a prefix of some keyword. Since Algorithm 3 sets $f(s) = g(r_n, a_j) = t$, the proof is complete. □

The next lemma characterizes the output function constructed by Algorithms 2 and 3.

LEMMA 2. *The set output(s) contains y if and only if y is a keyword that is a suffix of the string representing state s.*

PROOF. In Algorithm 2 whenever we add to the goto graph a state s that is represented by a keyword y we make *output*$(s) = \{y\}$. Given this initialization, we shall show by induction on the depth of state s that *output*$(s) = \{y \mid y$ is a keyword that is a suffix of the string representing state $s\}$.

This statement is certainly true for the start state which is of depth 0. Assuming this statement is true for all states of depth less than d, consider a state s of depth d. Let u be the string that represents state s.

Consider a string y in *output*(s). If y is added to *output*(s) by Algorithm 2, then $y = u$ and y is a keyword. If y is added to *output*(s) by Algorithm 3, then y is in *output*$(f(s))$. By the inductive hypothesis, y is a keyword that is a suffix of the string representing state $f(s)$. By Lemma 1, any such keyword must be a suffix of u.

Conversely, suppose y is any keyword that is a suffix of u. Since y is a keyword, there is a state t that is represented by y. By Algorithm 2, *output*(t) contains y. Thus if $y = u$, then $s = t$ and *output*(s) certainly contains y. If y is a proper suffix of u, then from the inductive hypothesis and Lemma 1 we know *output*$(f(s))$ contains y. Since Algorithm 3 considers states in order of increasing depth, the last statement of Algorithm 3 adds *output*$(f(s))$ and hence y to *output*(s). □

The following lemma characterizes the behavior of Algorithm 1 on a text string $x = a_1 a_2 \cdots a_n$.

LEMMA 3. *After the jth operating cycle, Algorithm 1 will be in state s if and only if s is represented by the longest suffix of $a_1 a_2 \cdots a_j$ that is a prefix of some keyword.*

PROOF. Similar to Lemma 1. □

THEOREM 1. *Algorithms 2 and 3 produce valid goto, failure, and output functions.*

PROOF. By Lemmas 2 and 3. □

5. Time Complexity of Algorithms 1, 2, and 3

We now examine the time complexity of Algorithms 1, 2, and 3. We shall show that using the goto, failure and output functions created by Algorithms 2 and 3, the number of state transitions made by Algorithm 1 in processing a text string is independent of the number of keywords. We shall also show that Algorithms 2 and 3 can be implemented to run in time that is linearly proportional to the sum of the lengths of the keywords in K.

THEOREM 2. *Using the goto, failure and output functions created by Algorithms 2 and 3, Algorithm 1 makes fewer than $2n$ state transitions in processing a text string of length n.*

PROOF. In each operating cycle Algorithm 1 makes zero or more failure transitions followed by exactly one goto transition. From a state s of depth d Algorithm 1 can never make more than d failure transitions in one operating cycle.[1] Thus the total number of failure transitions must be at least one less than the total number of goto transitions. In processing an input of length n Algorithm 1 makes exactly n goto transitions. Therefore the total number of state transitions is less than $2n$. □

The actual time complexity of Algorithm 1 depends on how expensive it is:

1. to determine $g(s, a)$ for each state s and input symbol a,

[1] As many as d failure transitions can be made. [13] shows that, if there is only one keyword in K, $O(\log d)$ is the maximum number of failure transitions which can be made in one operating cycle.

2. to determine $f(s)$ for each state s,
3. to determine whether $output(s)$ is empty, and
4. to emit $output(s)$.

We could store the goto function in a two-dimensional array, which would allow us to determine the value of $g(s, a)$ in constant time for each s and a. If the size of the input alphabet and the keyword set are large, however, then it is far more economical to store only the nonfail values in a linear list [1,11] for each state. Such a representation would make the cost of determining $g(s, a)$ proportional to the number of nonfail values of the goto function for state s. A reasonable compromise, and one which we have employed, is to store the most frequently used states (such as state 0) as direct access tables in which the next state can be determined by directly indexing into the table with the current input symbol. Then for the most frequently used states we can determine $g(s, a)$ for each a in constant time. Less frequently used states and states with few nonfail values of the goto function can be encoded as linear lists.

Another approach would be to store the goto values for each state in the form of a binary search tree [1, 12].

The failure function can be stored as a one-dimensional array so that $f(s)$ can be determined in constant time for each s.

Thus, the non-printing portion of Algorithm 1 can be implemented to process a text string of length n in cn steps, where c is a constant that is independent of the number of keywords.

Let us now consider the time required to print the output. A one-dimensional array can be used to determine whether $output(s)$ is empty in constant time for each s. The cost of printing the output in each operating cycle is proportional to the sum of the lengths of the keywords in $output(s)$ where s is the state in which Algorithm 1 is at the end of the operating cycle. In many applications $output(s)$ will usually contain at most one keyword, so the time required to print the output at each input position is constant.

It is possible, however, that a large number of keywords occur at every position of the text string. In this case Algorithm 1 will spend a considerable amount of time printing out the answer. In the worst case we may have to print all keywords in K at virtually every position of the text string. (Consider an extreme case where $K = \{a, a^2, a^3, \ldots, a^k\}$ and the text string is a^n. Here a^i denotes the string of i a's.) Any other pattern matching algorithm, however, would also have to print out the same number of keywords at each position of the text string so it is reasonable to compare pattern matching algorithms on the basis of the time spent in recognizing where the keywords occur.

We should contrast the performance of Algorithm 1 with a more straightforward way of locating all keywords in K that are substrings of a given text string. One such way would be to take in turn each keyword in K and successively match that keyword against all character positions in the text string. The running time of this technique is at best proportional to the product of the number of keywords in K times the length of the text string. If there are many keywords, the performance of this algorithm will be considerably worse that that of Algorithm 1. In fact it was the time complexity of the straightforward algorithm that prompted the development of Algorithm 1. (The reader may wish to compare the performance of the two algorithms when $K = \{a, a^2, \ldots, a^k\}$ and the text string is a^n.)

Finally let us consider the cost of computing the goto, failure, and output functions using Algorithms 2 and 3.

THEOREM 3. *Algorithm 2 requires time linearly proportional to the sum of the lengths of the keywords.*

PROOF. Straightforward. □

THEOREM 4. *Algorithm 3 can be implemented to run in time proportional to the sum of the lengths of the keywords.*

PROOF. Using an argument similar to that in Theorem 2, we can show that the total number of executions of the statement $state \leftarrow f(state)$ made during the course of Algorithm 3 is bounded by the sum of the lengths of the keywords. Using linked lists to represent the output set of a state, we can execute the statement $output(s) \leftarrow output(s) \cup output(f(s))$ in constant time. Note that $output(s)$ and $output(f(s))$ are disjoint when this statement is executed. Thus the total time needed to implement Algorithm 3 is dominated by the sum of the lengths of the keywords. □

6. Eliminating Failure Transitions

This section shows how to eliminate all failure transitions from Algorithm 1 by using the next move function of a deterministic finite automaton in place of the goto and failure functions.

A deterministic finite automaton [15] consists of a finite set of states S and a next move function δ such that for each state s and input symbol a, $\delta(s, a)$ is a state in S. That is to say, a deterministic finite automaton makes exactly one state transition on each input symbol.

By using the next move function δ of an appropriate deterministic finite automaton in place of the goto function in Algorithm 1, we can dispense with all failure transitions. This can be done by simply replacing the first two statements in the **for**-loop of Algorithm 1 by the single statement $state \leftarrow \delta(state, a_i)$. Using δ, Algorithm 1 makes exactly one state transition per input character.

We can compute the required next move function δ from the goto and failure functions found by Algorithms 2 and 3 using Algorithm 4. Algorithm 4 just precomputes the result of every sequence of possible failure transitions. The time taken by Algorithm 4 is linearly proportional to the size of the keyword set. In practice, Algorithm 4 would be evaluated in conjunction with Algorithm 3.

The next move function computed by Algorithm 4 from the goto and failure functions shown in Figure 1 is tabulated in Figure 3.

The next move function is encoded in Figure 3 as follows. In state 0, for example, we have a transition on h to state 1, a transition on s to state 3, and a transition on any other symbol to state 0. In each state, the dot stands

Algorithm 4. Construction of a deterministic finite automaton.

Input. Goto function g from Algorithm 2 and failure function f from Algorithm 3.

Output. Next move function δ.

Method.

```
begin
  queue ← empty
  for each symbol a do
    begin
      δ(0, a) ← g(0, a)
      if g(0, a) ≠ 0 then queue ← queue ∪ {g(0, a)}
    end
  while queue ≠ empty do
    begin
      let r be the next state in queue
      queue ← queue − {r}
      for each symbol a do
        if g(r, a) = s ≠ fail do
          begin
            queue ← queue ∪ {s}
            δ(r, a) ← s
          end
        else δ(r, a) ← δ(f(r), a)
    end
end
```

for any input character other than those above it. This method of encoding the next move function is more economical than storing δ as a two-dimensional array. However, the amount of memory required to store δ in this manner is somewhat larger than the corresponding representation for the goto function from which δ was constructed since many of the states in δ each contain transitions from several states of the goto function.

Using the next move function in Figure 3, Algorithm 1 with input "ushers" would make the sequence of state transitions shown in the first line of states of Figure 2.

Using a deterministic finite automaton in Algorithm 1 can potentially reduce the number of state transitions by 50%. This amount of saving, however, would virtually never be achieved in practice because in typical applications Algorithm 1 will spend most of its time in state 0 from which there are no failure transitions. Calculating the expected saving is difficult, however, because meaningful definitions of "average" set of keywords and "average" text string are not available.

7. An Application to Bibliographic Search

Algorithm 1 is attractive in pattern matching applications involving large numbers of keywords, since all keywords can be simultaneously matched against the text string in just one pass through the text string. One such application in which this algorithm has been successfully used arose in a library bibliographic search program which locates in a cumulative citation index all citations satisfying some Boolean function of keywords.

The data base used for this retrieval system is the cumulated machine-readable data used for *Current Technical Papers*, a fortnightly citation bulletin produced for internal

Fig. 3. Next move function.

		input symbol	next state
	state 0:	h	1
		s	3
		·	0
	state 1:	e	2
		i	6
		h	1
		s	3
		·	0
state 9:	state 7:	state 3: h	4
		s	3
		·	0
	state 5:	state 2: r	8
		h	1
		s	3
		·	0
		state 6: s	7
		h	1
		·	0
		state 4: e	5
		i	6
		h	1
		s	3
		·	0
		state 8: s	9
		h	1
		·	0

use by the technical libraries of Bell Laboratories. These citations are gathered from journals, covering a broad classification of technical interests. In the summer of 1973 there were three years of cumulated data, representing about 150,000 citations with a total length of about 10^7 characters.

With this search system a bibliographer can retrieve from the data base all titles satisfying some Boolean combination of keywords. For example, the bibliographer can ask for all titles in the data base containing both the keywords "ion" and "bombardment." The bibliographer can also specify whether a keyword is required to be preceded and/or followed by a punctuation character such as space, comma, semicolon, etc. A specification of this nature can explicitly deny matching on keywords embedded in the text. For example, it is often reasonable to accept the word "ions" as a match for the substring "ion." However, it is usually unreasonable to accept a word such as "motion" as a match on that keyword. The implementation permits specification of acceptance with full embedding, left embedding, right embedding, or none at all. This provision creates no difficulty for Algorithm 1 although the use of a class of punctuation characters in the keyword syntax creates some states with a large number of goto transitions. This may make the deterministic finite automaton implementation of Algorithm 1 more space consuming and less attractive for some applications.

An early version of this bibliographic search program

employed a direct pattern matching algorithm in which each keyword in the search prescription was successively matched against each title. A second version of this search program was implemented, also in FORTRAN, in which the only difference was the substitution of Algorithms 1, 2 and 3 for the direct pattern matching scheme. The following table shows two sample runs of the two programs on a Honeywell 6070 computer. The first run involved a search prescription containing 15 keywords, the second a search prescription containing 24 keywords.

	15 keywords	24 keywords
old	.79	1.27
new	.18	.21

CPU Time in Hours

With larger numbers of keywords the improvement in performance became even more pronounced. The figures tend to bear out the fact that with Algorithm 1 the cost of a search is roughly independent of the number of keywords. The time spent in constructing the pattern matching machine and making state transitions was insignificant compared to the time spent reading and unpacking the text string.

8. Concluding Remarks

The pattern matching scheme described in this paper is well suited for applications in which we are looking for occurrences of large numbers of keywords in text strings. Since no additional information needs to be added to the text string, searches can be made over arbitrary files.

Some information retrieval systems compute an index or concordance for a text file to allow searches to be conducted without having to scan all of the text string [7]. In such systems making changes to the text file is expensive because after each change the index to the file must be updated. Consequently, such systems work best with long static text files and short patterns.

An interesting question from finite automata theory is: Given a regular expression R of length r and an input string x of length n, how quickly can one determine whether x is in the language denoted by R? One method for solving this problem is first to construct from R a nondeterministic finite automaton M and then to simulate the behavior of M on the input x. This gives an $O(rn)$ solution [1].

Another approach along these lines is to construct from R a nondeterministic finite automaton M, then to convert M into a deterministic finite automaton M' and then to simulate the behavior of M' on x. The only difficulty with this approach is that M' can have on the order of 2^r states. The simulation of M' on the other hand is linear in n of course. The overall complexity is $O(2^r + n)$.

Using Algorithm 4 we can construct a deterministic finite automaton directly from a regular expression R in time that is linear in the length of R. However, the regular expression is now restricted to be the form $\Sigma^*(y_1 + y_2 + \cdots + y_k)\Sigma^*$ where Σ is the input symbol alphabet. By "concatenating" a series of deterministic finite automata in tandem, we can extend this result to regular expressions of the form $\Sigma^* Y_1 \Sigma^* Y_2 \cdots \Sigma^* Y_m \Sigma^*$ where each Y_i is a regular expression of the form $y_{i1} + y_{i2} + \cdots + y_{ik_i}$.

A related open question is what new classes of regular sets can be recognized in less than $O(rn)$ time. Along these lines, in [5] it is shown that regular expressions of the form $\Sigma^* y \Sigma^*$ where y is a keyword with "don't care" symbols can be recognized in $O(n \log r \log\log r)$ time.

Acknowledgements

The authors are grateful to A. F. Ackerman, A. D. Hall, S. C. Johnson, B. W. Kernighan, and M. D. McIlroy for their helpful comments on the manuscript. This paper was produced using the typesetting system developed by Kernighan and Cherry [9]. The assistance of B. W. Kernighan and M. E. Lesk in the preparation of the paper was appreciated.

Received August 1974; revised January 1975

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Booth, T.L. *Sequential Machines and Automata Theory.* Wiley, New York, 1967.
3. Brzozowski, J.A. Derivatives of regular expressions. *J. ACM* **11**:4 (October 1964), 481-494.
4. Bullen, R.H., Jr., and Millen, J.K. Microtext - the design of a microprogrammed finite state search machine for full-text retrieval. *Proc. Fall Joint Computer Conference,* 1972, pp. 479-488.
5. Fischer, M.J., and Paterson, M.S. String matching and other products. Technical Report 41, Project MAC, M.I.T., 1974.
6. Gimpel, J.A. A theory of discrete patterns and their implementation in SNOBOL4. *Comm. ACM* **16**:2 (February 1973), 91-100.
7. Harrison, M.C. Implementation of the substring test by hashing. *Comm. ACM* **14**:12 (December 1971), 777-779.
8. Johnson, W.L., Porter, J.H., Ackley, S.I., and Ross, D.T. Automatic generation of efficient lexical processors using finite state techniques. *Comm. ACM* **11**:12 (December 1968), 805-813.
9. Kernighan, B.W., and Cherry, L.L. A system for typesetting mathematics. *Comm. ACM* **18**:3 (March 1975), 151-156.
10. Kleene, S.C. Representation of events in nerve nets. In *Automata Studies,* C.E. Shannon and J. McCarthy (eds.), Princeton University Press, 1956, pp. 3-40.
11. Knuth, D.E. *Fundamental Algorithms,* second edition, The Art of Computer Programming **1**, Addison-Wesley, Reading, Mass., 1973.
12. Knuth, D.E. *Sorting and Searching,* The Art of Computer Programing **3**, Addison-Wesley, Reading, Mass., 1973.
13. Knuth, D.E., Morris, J.H., Jr., and Pratt, V.R. Fast pattern matching in strings. TR CS-74-440, Stanford University, Stanford, California, 1974.
14. Kohavi, Z. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1970.
15. McNaughton, R., and Yamada, H. Regular expressions and state graphs for automata. *IRE Trans. Electronic Computers* **9**:1 (1960), 39-47.
16. Rabin, M.O., and Scott, D. Finite automata and their decision problems. *IBM J. Research and Development* **3**, (1959), 114-125.
17. Thompson, K. Regular search expression algorithm. *Comm. ACM* **11**:6 (June 1968), 419-422.