Information Processing Letters 21 (1985) 219–227
North-Holland

18 November 1985

INCREMENTAL STRING MATCHING

Bertrand MEYER *

Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A.

Communicated by J. Nievergelt
Received 1 February 1985
Revised 27 March 1985

The problem studied in this paper is to search a given text for occurrences of certain strings, in the particular case where the set of strings may change as the search proceeds.

A well-known algorithm by Aho and Corasick applies to the simpler case when the set of strings is known beforehand and does not change. This algorithm builds a transition diagram (finite automaton) from the strings, and uses it as a guide to traverse the text. The search can then be done in linear time.

We show how this algorithm can be modified to allow incremental diagram construction, so that new keywords may be entered at any time during the search. The incremental algorithm presented essentially retains the time and space complexities of the non-incremental one.

Keywords: String matching, word processing, bibliographic search, indexing, program correctness, analysis of algorithms

1. Introduction

The problem of searching a text for all occurrences of one or more strings (hereafter called *search strings*) has been well researched. Several algorithms have been published [3,1,2].

This paper considers a variant of the problem which, to our knowledge, has not been addressed by previous publications: the case when the set of search strings may change as the search proceeds. In fact, in the practical application that led to this work, the search strings are found in the text itself as it is being searched.

In the next section we describe that application, an interactive book indexing program. In Section 3 we give the algorithm by Aho and Corasick which serves as a basis for our solution. Section 4 shows why this algorithm does not readily apply to the incremental case. A solution is proposed in Section 5; its correctness is proved in Section 6 and its efficiency analyzed in Section 7.

2. An index program

We first present the concrete occasion for which we developed the algorithm below. Of course, there may be other applications of incremental string searching.

The occasion is a program for making book indexes. An index is a sorted list of all the 'interesting' words which appear in a text, each word being accompanied by a sorted list of the pages where it occurs.

As anyone who has tried knows, preparing indexes is a tedious and error-prone task, and it is natural to look for computerized aids. Much of the work (searching for interesting words in the text, sorting the lists) can indeed be done automatically; but one critical step requires human intervention: deciding which words are 'interesting'

* On leave from Electricité de France, 1 Avenue du Général de Gaulle, 92141 Clamart, France.

and which are not. We call the person who will make this decision the *indexer*, the best indexer is usually the author.

The best place to look for 'interesting' words is of course the text itself, which we assume to be available as a computer file. Our indexing program thus has a phase called the *collector* which presents the indexer with the text and asks him to select words for indexing.

The collector is an interactive program. It displays the text one screen at a time; each word appearing on a screen belongs to one of the following three categories:

- 'rejected' words, which the indexer has already designated as not interesting,
- 'retained' words, which the indexer has designated as interesting (these words appear underlined on the screen as the collector is being executed),
- 'undecided' words, whose fate has not yet been sealed (these appear highlighted).

Thus, whenever a screenfull of text is displayed, the indexer must choose to either reject or retain each 'undecided' word on the screen. This decision process is the essential object of the collector.

From the program point of view, then, what the collector must do is to search each successive portion of text for occurrences of words belonging to the union of the 'rejected' and 'retained' sets. Both these sets change as new words are being classified by the indexer: thus, the string searching method must allow for incremental construction of the set of search strings.

3. A non-incremental algorithm

A very efficient algorithm by Aho and Corasick [1] applies to the case when there is more than one search string. The principle of this algorithm is that one first builds a 'transition diagram' from the set of search strings, and then traverses the text using this diagram as a guide. Thus, in the standard algorithm all search strings must be known at the outset.

Since our algorithm is based on a modification of Aho and Corasick's, we shall first present the key aspects of theirs. Our presentation is slightly different from the one in their original paper; it is close to the one we gave in [4].

3.1. Data structures

Aho and Corasick's algorithm uses three data structures as internal representation of a search string set: a tree T, called 'goto function' in [1] (it is in fact a trie); an 'output table' O; and a 'failure function' F. Together, these three structures constitute what may be termed the 'transition diagram' associated with the search string set. We describe them in turn.

3.1.1. The tree

The branches of the tree T are labeled by characters. Tree T is associated in a natural way with the set of search strings: for example, the search string set {A, CAN, AN} may yield the tree of Fig. 1.

An important property of this tree is that each node has an associated character string. For example, in the above tree, node 0 is associated with the empty string, node 3 with string CA, etc. From now on, *we will not make the distinction between a node and the associated string*; for example, we say that string AN is a suffix of node 4 (that is, of the associated string CAN), or that string CAT does not appear in the above tree (that is, no node of the tree is associated with this string).

If CC is the character set and the nodes are numbered from 0 to N, i.e., we define

type Node = 0..N;

then the tree may be represented as a two-dimensional array

T : **array**[Node, C] **of** Node;

$$\begin{tikzcd}
0 \arrow[r, "C"] \arrow[d, "A"] & 2 \arrow[r, "A"] & 3 \arrow[r, "N"] & 4 \\
1 \arrow[d, "N"] & & & \\
5 & & &
\end{tikzcd}$$

Fig. 1. A string matching tree.

where the child of node n through branch labeled c is T[n, c]. By convention, the root is numbered 0 and T[n, c] is 0 if there is no branch leading from node n with label c. We will not distinguish between the tree and the associated array; note that, in practice, the array will usually be sparse, requiring a suitable implementation (hashed, linked etc.).

3.1.2. The output table

For any node n, the output set of n, written O[n], is the set of suffixes of n which are search strings (we shall carefully distinguish between the *suffixes* of a string, which include the string itself, and its *proper suffixes*, which do not). In Fig. 2, corresponding to the search string set {A, CAN, AN}, the output sets (some of which are empty) have been written next to the corresponding nodes.

3.1.3. The failure function

The failure function F is defined on all nodes except the root. For such a node n, F[n] is the longest proper suffix of n that appears in tree T. F[n] may be node 0, the root (i.e., the empty string). It is important to note that the inverse of F is a tree spanning T, with node 0 as its root.

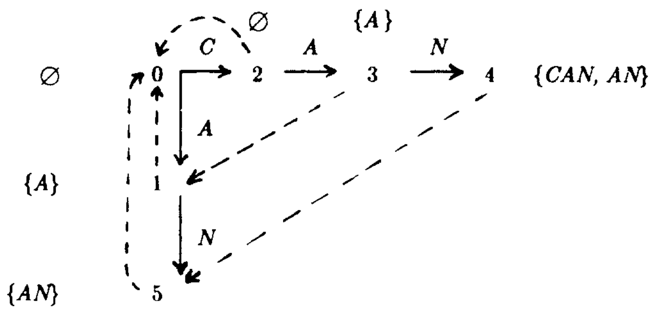

Fig. 2. A completed transition diagram.

The dashed lines in Fig. 2 represent the failure function for the search string set {A, CAN, AN}.

3.2. The string searching algorithm

Assuming a transition diagram consisting of the above three data structures has been constructed, the string searching algorithm is a simple traversal of the text, guided by the transition diagram:

```
procedure Recognize (text: in STRING)
        --Search text for strings represented by T, O, and F.
        n : NODE;
begin
        n := 0;        --Start at the root
        for c in text loop      --Examine next character
                while n ≠ 0 and T[n, c] = 0 loop      --Find worthy successor node
                        n := F[n]
                end while;
                n := T[n, c];
                Report all strings in O[n] as occurring at this point of the text
        end for
end procedure Recognize
```

This algorithm clearly makes one T transition per character of the text. It is proved in [1] that the number of F transitions is at most ℓ, the length of the text. Thus, the time complexity of the searching algorithm is $O(\ell)$.

3.3. Constructing the transition diagram

To execute the above algorithm, the three data structures T, O, and F must have been built from the search string set S. This is done in two steps:

Build_tree; *Build_failure*

The first step builds T and initializes O; the second builds F and completes O.

We describe these two steps below. We assume that the data structures T, O, and F are global to all the procedures given and have been properly dimensioned (an upper bound for N is *lsum*, the sum of the lengths of the search strings) and initialized (all values of T and F to zero, all values of O to the empty set).

3.4. Constructing the tree

The first step may be done as follows:

```
procedure Build_tree;
    last := 0;      --Number of the last entered node
    for s in S loop      --enter search string s into tree
        Enter_in_tree(s);
    end for
end procedure Build_tree
```

where *Enter_in_tree*(local to *Build_tree*) is given below.

```
procedure Enter_in_tree(s : in STRING);
        --Enter new search string s into tree T.
    n, n′ : NODE;
begin
    n := 0;      --Start at root
    for c in s loop      --Examine next character
        n′ := T[n, c];
        if n′ = 0 then      --Create new node
            last := last + 1; n′ := last;
            Enter_child(n, n′, c)
        end if;
        n := n′
    end for;
    Enter_output(n, s)
end procedure Enter_in_tree
```

The auxiliary procedures *Enter_child* and *Enter_output* each consist of a simple assignment; the only reason for pulling them out of the body of *Enter_in_tree* is to ease adaptation to the ‘incremental’ case later.

```
procedure Enter_child(n, n′ : in NODE; c : in CHARACTER);
    --Add to the tree a branch from n to n′ labeled c.
begin
   T[n, c] := n′
end procedure Enter_child;
```

```
**Procedure** *Enter_output*(n : **in** NODE; s : **in** STRING);
        --*Define the output set of* n *as consisting of the sole string* s.
**begin**
    O[n] := {s}
**end procedure** *Enter_output*
```

3.5. Constructing the failure function

For any node n other than the root, let *lps*(n) be the longest proper suffix of n which appears in the tree. To complete the transition diagram, procedure *Build_failure* must set F[n] to *lps*(n) for every non-root node n, and complete O[n] accordingly.

To do this, the *Build_failure* algorithm uses a loop that considers all nodes of T in order of increasing length of the associated strings (i.e., first the root, then its children, then their children etc.).

```
procedure Build_failure;
    Precondition: T is a tree associated with the given string set
    Postcondition: For all nodes i ≠ 0, F[i] = lps(i)
    --Build the failure function F corresponding to T.
begin
    for n in NODE in order of increasing length loop    --Compute F for the children of n
        --loop invariant
            --For any child i of a node previously considered, F[i] = lps(i)
        --end invariant
        for c in CC such that T[n, c] ≠ 0 loop
            Complete_failure(n, c);    --Compute F[T[n, c]]
        end for
    end for
end procedure Build_failure
```

with *Complete_failure* defined as follows:

```
procedure Complete_failure(n : in NODE; c : in CHARACTER);
    --Precondition: For any i ≠ 0 which is either n or a shorter node, F[i] = lps(i)
    --Compute F[T[n, c]].
    n', m, m' : NODE;
begin
    n' := T[n, c]; m := n;
    repeat
    m := F[m]
        --loop invariant
            --m is a proper suffix of n,  .
            --and for any proper suffix p of n longer than m in the tree, T[p, c] = 0
        --end invariant
        until m = 0 or T[m, c] ≠ 0
    end repeat;
    m' := T[m, c];
    F[n'] := m';
    O[n'] := O[n'] ∪ O[m']
end procedure Complete_failure
```

(Note that the invariant of a **repeat**... **until** loop is written at the end of the loop body since it may not be satisfied until after the first iteration.)

The correctness of *Build_failure* will follow from the loop invariant of that procedure since any node but the root is a child of another.

To show that this invariant is indeed preserved by the loop body, assume that $n \neq 0$ and that $F(i) = lps(i)$ for all nodes i considered before n. We have to show that, for any child n′ of n, say $n' = T[n, c]$, execution of *Complete_failure*(n, c) results in $F[n'] = lps(n')$.

Using the notation xy for the concatenation of x and y (where x is a string and y is a string or a single character), we have $n' = nc$. Thus, the longest proper suffix of n′ in the tree is either $m' = mc$, where m is the longest proper suffix of n in the tree such that T[m, c] is not 0, or 0 if there is no such m.

Now the list of all proper suffixes of n which appear in the tree is precisely, in order of decreasing length, the list of nodes examined successively by *Complete_failure*, namely F[n], F[F[n]], F[F[F[n]]], etc. This is a direct consequence of the inductive assumption: since n is not the root, n is the child of a previously considered node: thus, F[n] is the longest proper suffix of n in T.

It is essential for this correctness argument that the set of nodes be explored in order of increasing length. This can be ensured in three different ways:

- The above *Build_tree* algorithm may be modified so that the number assigned to any node is smaller than the number assigned to nodes on the following level in the tree. To do this, *Build_tree* should consider successive character positions in all search strings rather than successive search strings (that is, the order of the embedded loops in *Build_tree* should be reversed). Then, the loop in *Build_failure* will just consider nodes in order from 0 to N. This is the solution presented in [4].
- Another solution is to add a topological sort step between *Build_tree* and *Build_failure* which will renumber the nodes according to the rule stated above.
- The solution given in [1] uses a FIFO queue of nodes in *Build_failure* to make sure that nodes are processed in order of increasing levels, without imposing a special node numbering.

3.6. Efficiency

It is shown in [1] that construction of the complete transition diagram, as given above, takes no more than O(*lsum*) time and space, where *lsum* is the sum of the lengths of the search strings.

4. The problem with incremental construction

Let us now assume that instead of being all known beforehand, the search strings become available as the search proceeds.

There is no particular problem with the construction of the tree; we can execute *Enter_in_tree*(s) as each new string comes along. The real difficulty is associated with the failure function (and with the associated 'completion' of the output table).

From a practical point of view, it should be noted that the failure function is only useful when some search strings may be proper suffixes of others. Referring to the book indexing application mentioned in Section 2, this will not occur if all index entries correspond to *words*, always enclosed in delimiters in the text. If such is the case, the *Build_tree* procedure as given above is sufficient, and one may apply Aho and Corasick's method without a failure function. In many cases, however, one needs to have *phrases* as well as single words in index entries, so that a search string may be the proper suffix of another; for example, an index to the present paper might include entries for both strings *suffix* and *proper suffix*. If such is the case, one must find a way to build the F function incrementally, as new search strings are entered into T.

Now, when a new node $n' = T[n, c]$ is entered, one must compute F[n′]; this in itself raises no difficulty since the longest proper suffix of n′ in the tree must be of the form T[m, c] for some proper suffix m of n, and we may assume inductively as before that all proper suffixes of n are accessible through F. But this is not the whole

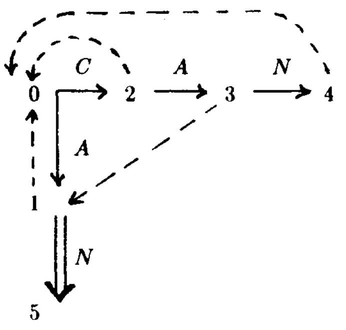

Fig. 3. Node insertion.

story: when adding n′ we may also have to update the failure value of *existing* nodes.

Fig. 3 illustrates the problem. The dashed lines represent the current value of the failure function. Assume we initially had the two search strings A and CAN and we add AN, corresponding to the transition represented by the double arrow. Then, F[4] must be updated to point to the new node 5.

The next section presents a solution to this problem.

5. The incremental algorithm

When a new node n′ = T[n, c] is entered, the value of F[x′] must be changed for some existing node x′ if and only if n′ is the longest proper suffix of the string associated with x′. This may only be the case if x′ = T[x, c] for a node x such that $n \in F^{+}[x]$, where $^{+}$ denotes (nonreflexive) transitive closure.

Thus, to be able to find all the nodes whose F value needs to be updated, we must keep a record IF of the inverse function of F. Note that F is single-valued but IF may be multi-valued. As indicated in Section 3.1.3, IF is a tree.

If we have access to IF, then to enter a new search string we use the procedure *Enter_in_tree* as given in Section 3.4. We only need to change procedures *Enter_output* and *Enter_child*. Both now have a slightly different specification and become recursive, as follows. We assume that IF, as the other data structures, is properly initialized: IF[n] should be initially empty for all nodes n.

```
procedure Enter_output(n : in NODE; s : in STRING);
        --Add s to the output set of any node which has n as a suffix.
begin
        O[n] := O[n] ∪ {s};
        for x in IF[n] loop Enter_output(x, s) end for;
end procedure Enter_output;

procedure Enter_child(n, n′ : in NODE; c : in CHARACTER);
        --Add branch from n to n′ labeled c;
        --update failure and inverse functions to account for the insertion of n′.
begin
        T[n, c] := n′;
        Complete_failure(n, c);              --Compute F[n′]
        IF[F[n′]] := IF[F[n′]] ∪ {n′};       --Update IF for m′ = F[n′]
        Complete_inverse(n, n′, c);          --Compute IF[n′] and change to n′ the corresponding
                                               values of F
end procedure Enter_child
```

Procedure *Complete_failure* is as before (Section 3.5). Procedure *Complete_inverse* finds all nodes which will 'fail to' a new node and is defined as follows:

```
procedure Complete_inverse(y, n' : in NODE; c : in CHARACTER);
    --Recursive Precondition : n' is a suffix of yc
    --and T[βn, c] = 0 for any proper suffic βn of y in the tree (where n' = nc)
    --Record n' as new failure value for all x' such that yc = lps(x')
    x, x' : NODE;
begin
    for x in IF[y] loop
            --{y = lps(x)}
        if T[x, c] ≠ 0 then
            x' := T[x, c];      --[yc is a proper suffix of x'; thus so is n'}
            --Remove previous failure value of x'; install new one.
            IF[F[x']] = IF[F[x']] − {x'};      --Set difference
            F[x'] := n'; IF[n'] := IF[n'] ∪ {x'};
        else
            --Try recursively with nodes having x as proper suffix
            Complete_inverse(x, n', c);
        end if
    end for
end procedure Complete_inverse
```

6. Correctness

To prove the correctness of the above incremental algorithm, we first note that termination of the two recursive calls follows from the fact that IF is a tree, and that the changes brought to T and O when a new search string is inserted are the same ones that Aho and Corasick's algorithm would have performed. Thus, we concentrate on the partial correctness of the modifications to F and IF.

It suffices to prove that an execution of *Enter_in_tree* as given above leaves the following two properties of the transition diagram invariant:
- (INV) IF is the inverse of F.
- (SUFF) For any node $x \neq 0$, $F[x] = lps(x)$ (that is, F[x] is the longest proper suffix of x in the tree).

Property (INV) is trivially invariant since any modification to F in *Enter_in_tree* is accompanied by the corresponding modification to IF and conversely.

To show that property (SUFF) is invariant, we study in what way function *lps* changes when $n' = T[n, c]$ is inserted and check that F follows suit. There may be two reasons for change:

(1) The definition of *lps* has to be extended for n' in accordance with (SUFF); the call to *Complete_failure* takes care of this case.

(2) The value of *lps*[y'] for some previously existing nodes y' may now become n'. These nodes are those which have n', i.e., nc, as a proper suffix, and have no longer proper suffix in the tree.

Assuming inductively that (SUFF) was satisfied before the insertion, any such y' is of the form αnc (see Fig. 4), such that no node of the form βnc, where β is a proper suffix of α, exists in the tree.

Thus, $y' = T[y, c]$, where $y = \alpha n$. As expressed by the 'recursive precondition' to procedure *Complete_inverse*, the required y nodes are exactly those which are considered (in order of increasing length) by the successive recursive calls to that procedure, starting with $y = n$.

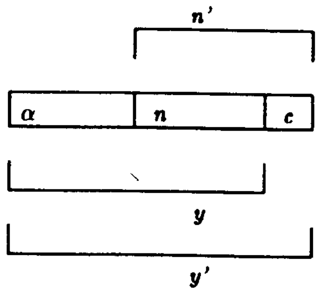

Fig. 4. Strings and suffixes.

7. Efficiency

The recognition algorithm (procedure *Recognize*) is not affected by the modification; nor is the performance of the part of the algorithm which builds the tree. We must consider the impact of the modification on the building of the failure and output functions.

Regarding space efficiency, since we must store IF[n] for every node n (presumably as a linked list), the space requirement for the representation of the failure function is doubled, remaining O(*lsum*).

Regarding time, we first notice that for each search string the operations performed by *Complete_failure* are a subset of those performed in the original algorithm; there may be fewer operations because some proper suffixes may not have been entered yet. Thus, the total time for this procedure will be *O*(*lsum*). For a given search string set, the operations performed by *Enter_output* (adding elements to the output sets) are also the same in the incremental algorithm as in the original, although the former may do them in a different order and will follow inverse F chains.

The case of *Complete_inverse* is more delicate since this procedure may follow void chains which the direct algorithm would never have explored. When inserting n′ = T[n, c], the maximum number of nodes which may be searched in this fashion is the number of elements in the set $IF^+[n]$. Thus, the maximum number of extra operations is

$$K * \sum_{n \in \text{Node}} \textit{descendants}(n),$$

where K is the size of the character set CC and *descendants*(n) is the number of proper descendants of node n in the IF tree.

It is easily proved that, for any tree with M nodes and height h,

$$\sum_{n \in \text{Node}} \textit{descendants}(n) \leqslant M * h.$$

Here, IF has the same nodes as T, thus $M \leqslant \textit{lsum}$, and $h \leqslant \textit{lmax}$, where *lmax* is the maximum search string length.

Thus, the overall complexity of the algorithm is now $O(K * \textit{lmax} * \textit{lsum})$, which is identical to the original *O*(*lsum*) if we consider K and *lmax* as constants. In normal practical cases, the constant factors to apply are much smaller than the above analysis would seem to imply.

Acknowledgment

We are grateful to Man-Tak Shing and Don Brady for useful comments and for pointing out errors in a previous version of this paper.

References

[1] A.V. Aho and M.J. Corasick, Fast pattern matching: An aid to bibliographic search, Comm. ACM 18 (6) (1975) 333–340.

[2] R.S. Boyer and J. Strother Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.

[3] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, TR CS-74-440, Stanford Univ., CA, 1974.

[4] B. Meyer and C. Baudoin, Méthodes de Programmation (Eyrolles, Paris, 1978) (new edition, 1984).