A SPACE-ECONOMICAL SUFFIX TREE CONSTRUCTION ALGORITHM

BY EDWARD M. McCREIGHT

XEROX
PALO ALTO RESEARCH CENTER

A SPACE-ECONOMICAL SUFFIX TREE CONSTRUCTION ALGORITHM

BY EDWARD M. McCREIGHT

CSL-75-3 APRIL 1975

The first section presents a new algorithm for constructing auxiliary digital search trees to aid in exact-match substring searching. This algorithm has the same asymptotic running-time bound as previously published algorithms, but is more economical in space. The second section discusses some implementation considerations. The third section presents new work dealing with how to modify these search trees in response to incremental changes in the strings they index (the update problem).

Key Words and Phrases:

pattern matching algorithms, searching, search trees, context search, substring search, analysis of algorithms

CR Categories:

3.74, 4.34, 5.32

XEROX
PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD/PALO ALTO/CALIFORNIA 94304

INTRODUCTION

A number of computer applications need a basic function which locates a specific substring of characters within a longer main string. The most obvious such application is context searching within a text editor. Other applications include automatic command completion by the keyboard-handling executive of an operating system, and limited pattern-matching used in speech recognition <Kim 74>. This basic function is also useful as a building block in the construction of more sophisticated pattern matches.

The naive algorithm to implement this function simply attempts to match the substring against the main string in all possible positions. It is straightforward but can be slow, since for example the program might reverify the fact that position 17 in the main string is the character 'a' almost as often as the number of characters in the substring (consider the substring 'aaaaaaab'). An asymptotically more efficient algorithm was discovered by Knuth, Pratt, and Morris in 1970 <Knu 74>. It involves preprocessing the substring into a search automaton and then feeding the main string into the search automaton, one character at a time. In both of these algorithms the average search time is at least linear in the length of the main string.

If one were expecting to do many substring searches in the same main string, it would be worthwhile to build an auxiliary index to that main string to aid in the searches. A useful index structure which can be constructed in time linear in the length of the main string, and yet which enables substring searches to be completed in time linear in the length of the substring was first discovered by Weiner <Wei 73>.

In addition, his auxiliary index structure permits one easily to answer several new questions about the main string itself. For example, what is the longest substring of the main string which occurs in two places? in k places? One can also transmit (or store) a message with excerpts from the main string in minimum time (or space) by a dynamic-programming process which for each position of the message finds the longest excerpt of the message which begins there and is a substring of the main string. This latter application motivated Weiner's original discovery.

Section 2 presents Algorithm M, an algorithm for constructing an index structure functionally equivalent to Weiner's, but requiring about 25% less data space. Section 3 discusses several implementation alternatives and includes a detailed space analysis. Section 4 shows that minor changes to the main string usually result in minor changes to the auxiliary index to the main string, and that incrementally updating the index is usually more efficient than recomputing it. This result is particularly relevant to the context of a dynamically changing data base, as with a text editor.

ALGORITHM M

This section is an exposition of a new algorithm, called Algorithm M, for mapping a finite string S of characters into an auxiliary index to S in the form of a digital search tree T whose paths are the suffixes of S, and whose terminal nodes correspond uniquely to positions within S. The algorithm requires that:

> S1. The final character of S may not appear elsewhere in S.

If a string does not satisfy S1, it can be extended to a string which does by padding it with a new character. For example, the string abab is not acceptable, but it can be padded to the acceptable string ababc. If a string S satisfies S1, then no suffix of S is a prefix of a different suffix of S. This results in the existence of a terminal node in T for each suffix of S, since any two suffixes of S eventually go their separate ways in T.

Let n represent the length of the string S. To enable Algorithm M to operate in time and space linear in n, three constraints are placed on the form of T. Together their effect is that a tree T representing S is a multiway Patricia tree <Knu 73> and thus contains at most n nonterminal nodes.

> T1. An arc of T may represent any nonempty substring of S.
>
> T2. Each nonterminal node of T, except the root, must have *at least two* son arcs.
>
> T3. The string represented by an arc α of T must differ in

its *first* character from the string represented by any brother arc of *α*.

As an example, Algorithm M would map the string ababc (hereafter called S) into the tree

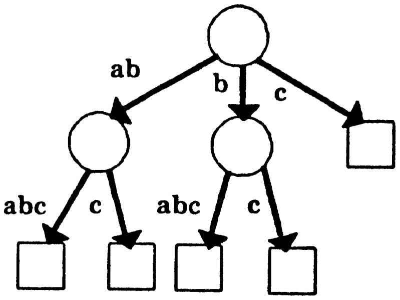


Figure 1

Because of constraints T2 and T3, this mapping is unique up to order among siblings. A few definitions and conventions are appropriate here.

Let Σ be the alphabet of characters used in S. Roman letters will be used to denote single characters of Σ, while Greek letters will denote (possibly empty) finite sequences or strings of characters from Σ. In our depictions of trees a straight line will denote a single arc and a wavy line will denote a nonempty sequence of arcs whose detail is being suppressed as irrelevant.

A partial path is defined as a (downward) connected sequence of tree arcs which begins at the root of the tree.

A path is defined as a partial path which terminates at a terminal node.

Constraints S1, T2, and T3 guarantee that a partial path may be named unambiguously by concatenating the strings on its arcs.

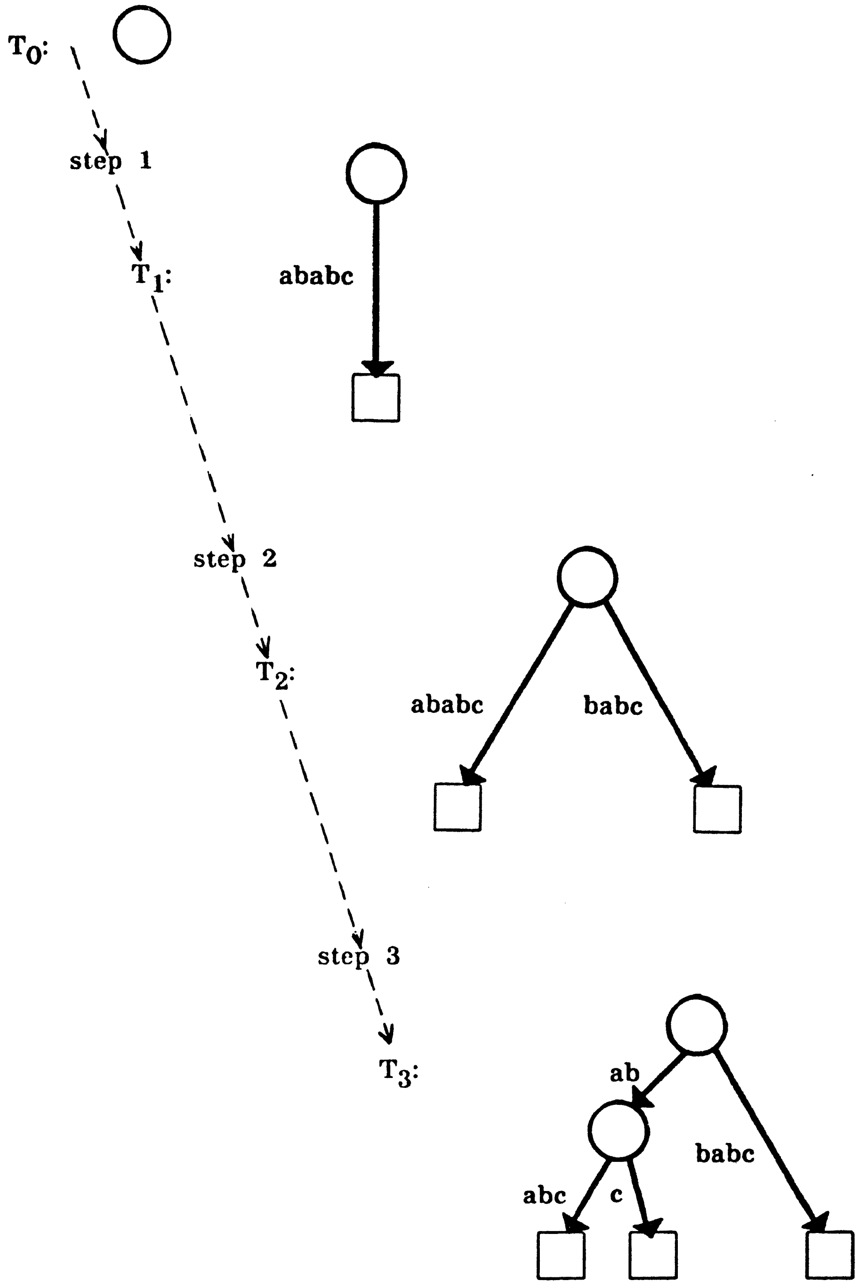
The locus of a string is the node at the end of the partial path (if any) named by the string.

An extension of a string α is any string of which α is a prefix.

The extended locus of a string α is the locus of the shortest extension of α whose locus is defined.

The contracted locus of a string α is the locus of the longest prefix of α whose locus is defined.

Algorithm M begins with an empty tree T_0 and enters paths corresponding to the suffixes of S one at a time, from longest to shortest. The tree T corresponding to our example string S (ababc) would be constructed by the algorithm in the following steps, one step per suffix of S:


T0:
step 1
T1:
ababc
step 2
T2:
ababc
babc
step 3
T3:
ab
babc
abc
c

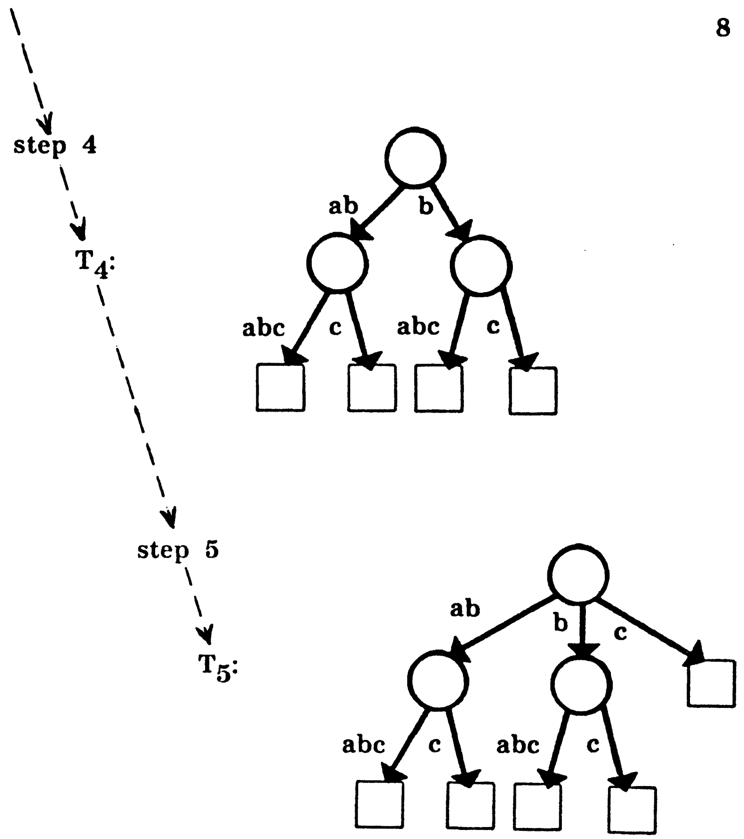


Figure 2

We define suf_i to be the suffix of S beginning at character position i. (Position 1 is defined to be the leftmost character of S, so suf_1 is S.) During step i, the algorithm inserts a path corresponding to the string suf_i into the tree T_{i-1} to produce the tree T_i. We define $head_i$ as the longest prefix of suf_i which is also a prefix of suf_j, for some $j < i$. Equivalently, $head_i$ is the longest prefix of suf_i whose extended locus exists within the tree T_{i-1}. We define $tail_i$ as suf_i - $head_i$. In our example, suf_3 = abc, $head_3$ = ab and $tail_3$ = c. Constraint S1 assures us that $tail_i$ is not empty. To insert suf_i into the tree T_{i-1}, the extended locus of $head_i$ in T_{i-1} is found, a new nonterminal node is constructed to split the incoming arc and become the locus of $head_i$ if necessary, and finally a new arc labelled $tail_i$ is constructed from that nonterminal node to a new terminal node.

For example, consider step 3 which transforms T_2 to T_3 in Figure 2. The algorithm must insert suf_3. By tracing this string within T_2, it sees that $head_3$ is ab, that the extended locus of ab is the leftmost terminal node, and that its incoming arc (labelled ababc) must be split. The algorithm splits the arc into two parts, labelled ab and abc, by inserting a new nonterminal node. A new arc labelled c, or $tail_3$, is then added from that nonterminal node to a new terminal node.

If the algorithm is to be efficient, an efficient data structure for the representation of trees must be used. Since the arcs of a tree T represent substrings of S (by constraint T1), we can represent the character string associated with an arc of T by a pair of integers denoting its starting and ending positions in S. Thus, the actual internal form of the tree in Figure 1 might be:

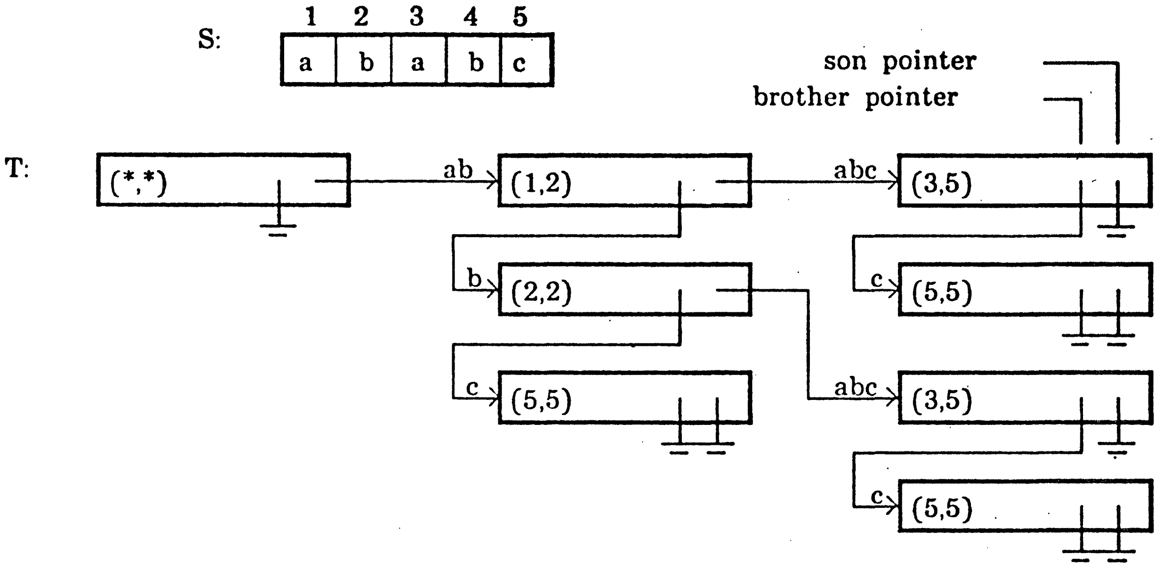


Figure 3

Given this representation, it should be clear from the example that at each step i, after the algorithm has somehow found the extended locus in T_{i-1} of head_i, the introduction of a new nonterminal and a new arc corresponding to tail_i takes at most constant time. If the algorithm could find the extended locus of head_i in at most constant time (averaged over all steps), then it would run in time linear in n, the length of S.

The algorithm does this by exploiting the following relationship between the strings head_{i-1} and head_i.

Lemma 1: If head_{i-1} can be written as $x\delta$ for some character x and some (possibly empty) string δ, then δ is a prefix of $head_i$.

Proof: By induction on i. Suppose $head_{i-1} = x\delta$. This means that there is a j, $j < i-1$, such that $x\delta$ is a prefix both of suf_{i-1} and of suf_{j-1}. Thus δ is a prefix both of suf_j and of suf_i. Therefore by the definition of *head*, δ is a prefix of $head_i$.

To exploit this relationship, auxiliary links are added to our tree structure. From each nonterminal node which is the locus of $x\delta$, where x is a character and δ is a string, a suffix link is introduced pointing to the locus of δ. (Note that the locus of δ is never within the subtree rooted at the locus of $x\delta$.) Depicting suffix links by dashed lines, the new representation of the tree in Figure 1 is:

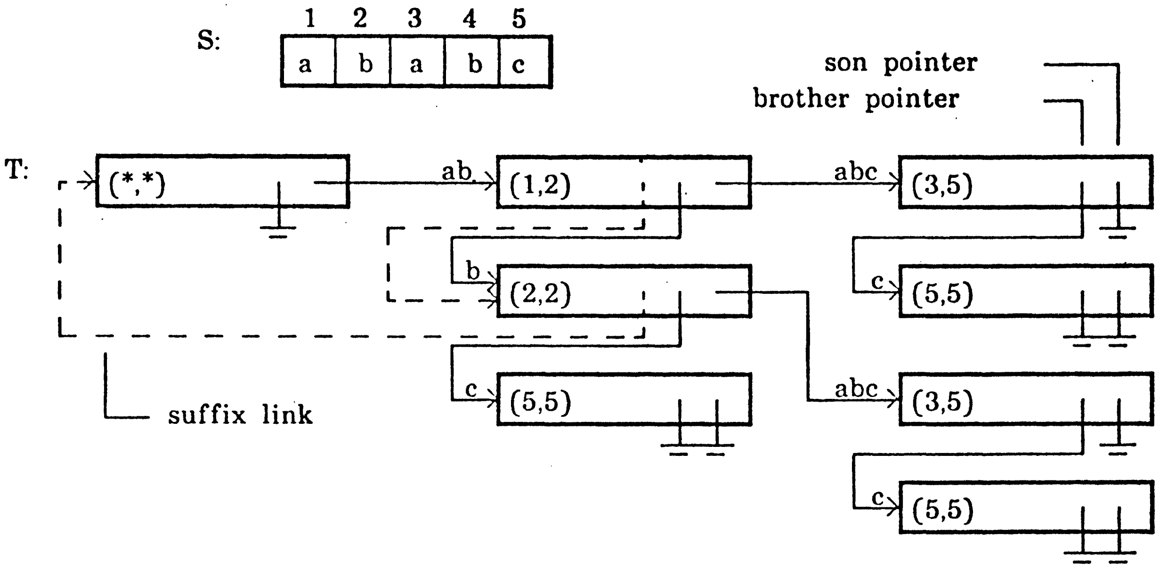


Figure 4

These links enable the algorithm in step i to do a short-cut search for the locus of $head_i$ beginning at the locus of $head_{i-1}$, which it has visited in the previous step.

The following semi-formal presentation of the algorithm will prove by induction on i that

P1: in tree T_i only the locus of $head_i$ could fail to have a valid suffix link, and that

P2: in step i the algorithm visits the contracted locus of $head_i$ in T_{i-1}.

Properties P1 and P2 clearly obtain if i=1. Now suppose i>1. In step i the algorithm does the following:

Substep A: First the algorithm identifies three strings χ, α, and β, with the following properties:

(1) $head_{i-1}$ can be represented as $\chi\alpha\beta$.

(2) The locus of $\chi\alpha$ is the contracted locus of $head_{i-1}$ in T_{i-2}.

(3) χ is a string of at most one character which is empty only if α is empty.

Lemma 1 guarantees that $head_i$ may be represented as $\alpha\beta\gamma$ for some (possibly empty) string γ. The algorithm now chooses the appropriate case of the following two:

α empty: The algorithm calls the root node c and goes to substep B. Note that c is defined as the locus of α.

α nonempty: By definition, the locus of $\chi\alpha$ must have existed in the tree T_{i-2}. By P1 the suffix link of that (nonterminal) locus node must be defined in tree T_{i-1}, since the node itself must have been constructed before step i-1. By P2 the algorithm visited that node in step i-1. The algorithm follows its suffix link to a nonterminal node called c and goes to substep B. Note that c is defined as the locus of α.

Substep B: This is called "rescanning", and is the key idea of Algorithm M. Since $\alpha\beta\gamma$ is defined as $head_i$, by the definition of *head* we know that the extended locus of $\alpha\beta$ exists in the tree T_{i-1}. This means that there is a sequence of arcs downward from node c (the locus of α) which spells out some extension of β. To rescan β the algorithm finds the child arc ρ of c which begins with the first character of β and leads to a node which we shall call f. It compares the lengths of β and ρ. If β is longer, then a recursive rescan of β-ρ is begun from node f. If β is the same length or shorter, then β is a prefix of ρ, the algorithm has found the extended locus of $\alpha\beta$, and the rescan is complete. It has been accomplished in time linear in the number of nodes encountered. A new nonterminal node is constructed to be the locus of $\alpha\beta$ if one does not already exist. The algorithm calls d the locus of $\alpha\beta$ and goes to substep C. (Note that substep B constructs a new node to be the locus of $\alpha\beta$ only if γ is empty.)

Substep C: This is called "scanning". If the suffix link of the locus of $\chi\alpha\beta$ is currently undefined, the algorithm first defines that link to point to node d. This and inductive hypothesis establish the truth of P1 in T_i. Then the algorithm begins searching from node d deeper into the tree to find the extended locus of $\alpha\beta\gamma$. The major difference between scanning and rescanning is that in rescanning the length of β is known beforehand (because it has already been scanned), while in scanning the length of γ is not known beforehand. Thus the algorithm must travel downward into the tree in response to the characters of $tail_{i-1}$ (of which γ is a prefix) one by one from left to right. When the algorithm "falls out of the tree" (as constraint S1 guarantees that it must), it has found the extended locus of $\alpha\beta\gamma$. The last node of T_{i-1} encountered in this downward trek of rescanning and scanning is the contracted locus of $head_i$ in T_{i-1}; this establishes the truth of P2. A new nonterminal node is constructed to be the locus of $\alpha\beta\gamma$ if one does not already exist. Finally a new arc labelled $tail_i$ is constructed from the locus of $\alpha\beta\gamma$ to a new terminal node. Step i is now finished.

We now introduce a more involuted example string S which is capable of illustrating the full complexity of the algorithm. Let S be $b^5abab^3a^2b^5c$, and consider step 14. Figure 5 depicts T_{13} with some detail missing and with node labels which will be applied in step 14. In step 14 the algorithm must insert suf_{14} (b^5c) into T_{13}. First it equates $\chi\alpha$ to ab, so χ is a and α is b. Thus, since $\chi\alpha\beta$ is abbb ($head_{13}$), β must be bb. In substep A the algorithm observes α to be nonempty and therefore follows the suffix link from the locus of ab (labelled as node x in figure 5) to the locus of b, calling that node c. In substep B the algorithm rescans β from node c. This rescanning encounters one intermediate node f, and stops at the locus of bbb, calling that node d. Substep C then begins at node d and scans downward in the tree to discover γ. In this example, it will discover that γ is bb, and will construct a new nonterminal node as the locus of $head_{14}$, which is $\alpha\beta\gamma$, or b^5.

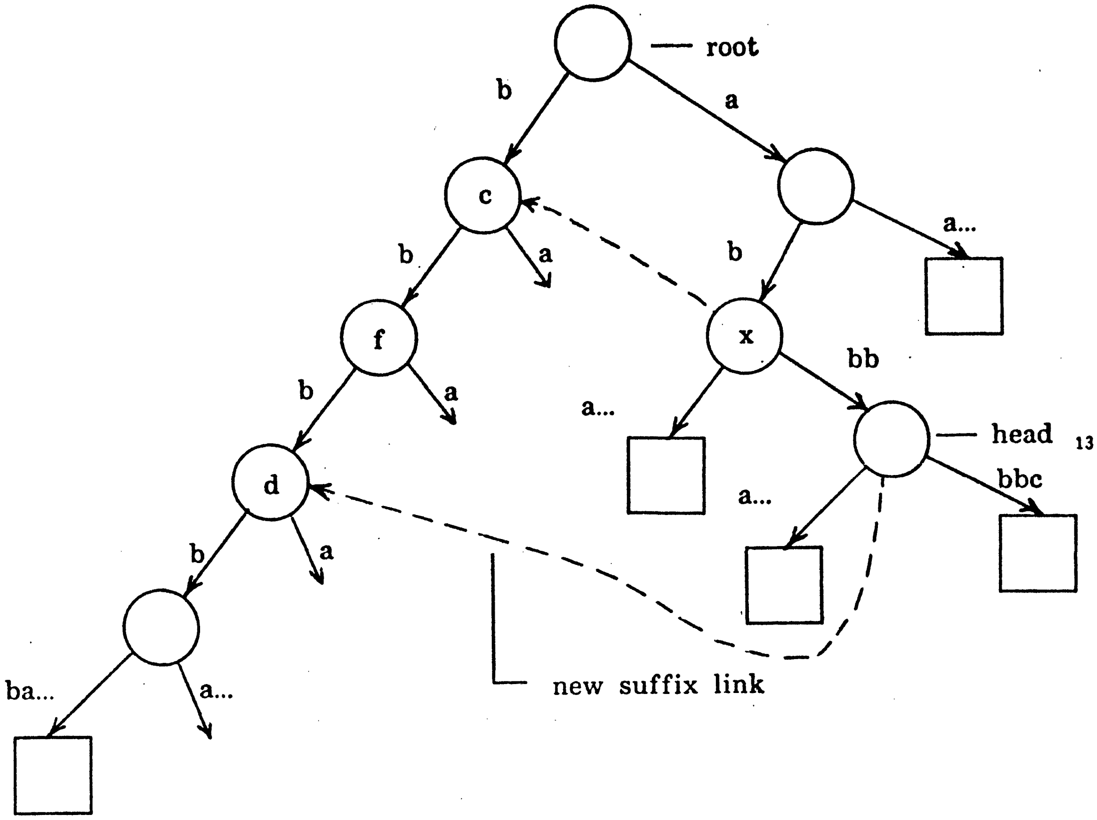


Figure 5

The time spent in scanning and rescanning must now be analyzed. At each step, rescanning and scanning is done on a suffix of S. Let $\underline{res_i}$ be defined as the shortest suffix of S to which the rescan and scan operations are confined during step i ($\beta\gamma$ followed by $tail_i$). Observe that for every intermediate node f encountered during the rescan of β, there will be a nonempty string (ρ) which is contained in res_i but not in res_{i+1}. Therefore length(res_{i+1}) is at most length(res_i) - int_i, where int_i is the number of intermediate nodes encountered while rescanning during step i. By repeated substitution we see that $\sum_{i=1}^{n} \text{int}_i$ is at most n, since length(res_n) = 0 and length(res_0) = n. Thus the total number of intermediate nodes encountered while rescanning during the operation of the algorithm is at most n.

In step i the number of characters which must be scanned to locate $head_i$ (the length of γ) is length($head_i$) - length($head_{i-1}$) + 1. Thus the total number of characters scanned during the operation of the algorithm is $\sum_{i=1}^{n}$ (length($head_i$) - length($head_{i-1}$) + 1), which collapses to n + length($head_n$) - length($head_0$), or n.

Algorithm M executes in n distinct steps, each of which takes constant time except for rescanning and scanning. We have just seen that rescanning and scanning each add time at most linear in n to the total running time. Therefore, Algorithm M operates within a time bound linear in the length of its input string S.

HASH CODED IMPLEMENTATION

Given the tree representation in figure 4, the running time of Algorithm M is potentially linearly dependent on the size of the alphabet of the input string. For example, consider the tree resulting from the string

abcdefghijklmnopqrstuvwxyz

The root node of this tree has 26 son arcs, and to establish each new son arc it is necessary to verify that the root does not already have a son arc starting with the same character. In general, if the input string consisted of n different characters, at least $(n^2-n)/2$ atomic search steps would be required. The length of the string explains one factor of n, but the other factor of n is attributable only to the size of the alphabet. Perhaps we can find a different tree representation for which running time degrades more gracefully with alphabet size.

The tree representation in figure 4 is fairly efficient in space but slow to search. A different representation might associate with each nonterminal node a table of pointers, with one pointer for each character of the input alphabet. This would be fast to search, but slow to initialize and prohibitive in size for a large alphabet. Alternatively one might use ordered lists or balanced trees <Knu 73>.

Encoding the arcs of T as entries in a hash table appears to be the best representation of all. It is very compact and (in the average case) reasonably speedy. Although it is unknown ahead of time how many son arcs a given nonterminal node will have, the total number of arcs in T has an upper bound of 2n (at most two arcs are added per step of Algorithm M). We now consider the data structure design in some detail in order to obtain precise storage bounds.

At each step of Algorithm M at most one nonterminal node and exactly one terminal node are created. We name those nodes by the number of the step in which they were created. A nonterminal node created during step k is named node k. A terminal node created during step k is named node n+k. In this way we can determine from the number assigned to a node whether it is a terminal or nonterminal node.

The hash table implements a function f from the set of ordered pairs of the form (nonterminal node, character) to the set of nodes, with the property that $f(n_s, x) = n_d$ if there is an arc in T from the nonterminal node n_s to the node n_d which begins with the character x, and $f(n_s, x) = 0$ otherwise. Using a hashing algorithm of the family proposed by Lampson (<Knu 73>, exercise 6.4.13), this table can be represented in $2n(\log_2 n+\log_2|\Sigma|+2)$ bits, where Σ is the size of the input alphabet.

We must, of course, represent the input string S. This can be done with a vector of $n \log_2|\Sigma|$ bits. In addition, when traversing an arc in T the algorithm must be able to deduce its starting and ending position in S. For this we introduce a table L which maps each nonterminal node into the length of the tree arc leading into that nonterminal node. Suppose we are about to traverse the arc α from node n_s to n_d. We have been keeping track of d_s, the number of characters in the partial path to n_s. The arc α begins at position n_s+d_s in S, and ends at position $n_s+d_s+L(n_d)$. We compute d_d as $d_s+L(n_d)$. The table L can be represented in $n \log_2 n$ bits.

Finally we introduce a table S to represent the suffix links of T. The table S maps each nonterminal node into the number of the node to which its suffix link points. It can be represented in n $\log_2 n$ bits.

In summary, using this design we can represent T in $4n \log_2 n + 3n \log_2|\Sigma| + 2n$ bits.

INCREMENTAL EDITING

After the string S has been transformed into the tree T, it may become necessary to change S. This circumstance would be quite common, for example, if Algorithm M were underlying a text editor. Any string S may be mapped into any other string S' by a sequence of incremental changes (replacement of substrings). Algorithms which minimize the length of such changes have been recently studied <Wag 74>. We shall now see that it is possible to make an incremental change to T in response to an incremental change in S.

Suppose that S is the string $\alpha\beta\gamma$, for some (possibly empty) strings α, β, and γ, and that S is to be changed into the string $\alpha\delta\gamma$, where δ is some (possibly empty) string which is different from β.

In order to make it possible to update T incrementally we adopt a string element position numbering scheme, like the Dewey-Decimal library access code, in which a position number need never change after it has been assigned, and such that the sequence of position numbers assigned to the characters of S is strictly monotonic. In particular, this means that the position numbers assigned to the characters of γ (and, of course, those of α) will not change during the replacement of β by δ. It also means that each position number assigned to a character of γ is greater than any position number assigned to a character of α. Of course, all this implies the availability of a large pool of position numbers, few of which are simultaneously in use. (This is in contrast to the small pool of position numbers in the last section; the goals of minimum-space representation and updateability seem mutually incompatible).

We begin by considering what paths of the tree T corresponding to the string

$\alpha\beta\gamma$ are particularly affected by the replacement of β by δ. We define as *β-splitters* (with respect to the change $\alpha\beta\gamma$ -> $\alpha\delta\gamma$) those strings (or their paths) of the form $\varepsilon\gamma$, where ε is a nonempty suffix of $\alpha^*\beta$, and where α^* is defined as the longest suffix of α which occurs in at least two different places in $\alpha\beta\gamma$. Equivalently, *β-splitters* are paths in T which properly contain the suffix γ, but whose terminal arcs do not properly contain $\beta\gamma$.

Informally, *β-splitters* are the only paths whose structure might be directly affected by the change from β to δ. By virtue of our "Dewey-Decimal" positional notation and our way of representing substrings of S in T, all paths in T except *β-splitters* reflect the change in S by default, either because they are too short to contain any character of β, or because they are so long that β is buried in a terminal arc and the change from β to δ cannot affect the structure of the path. The structure of these paths will change only as required by interaction with *β-splitters* and their replacements (which we might call *δ-splitters*). The updating algorithm will remove all *β-splitters* from the tree and then insert all *δ-splitters*, preserving tree properties T1-T3 and the validity of suffix links.

The updating algorithm must first find the longest *β-splitter*, $\alpha^*\beta\gamma$. Here the space-efficiency of Algorithm M works against us. The trees constructed by Algorithm M do not permit the efficient leftward extension of the *β-splitter* $\beta\gamma$ one character at a time until it is no longer a *β-splitter*. To overcome this difficulty the algorithm carries out this search in two phases. In the first phase it examines the paths $\alpha^{(1)}\beta\gamma$, $\alpha^{(2)}\beta\gamma$, $\alpha^{(4)}\beta\gamma$, $\alpha^{(8)}\beta\gamma$, ... , where $\alpha^{(j)}$ is the suffix of α which has length min(j, length(α)). These paths are examined in sequence to see whether they are *β-splitters*. This first phase of the search terminates when a non-*β-splitter* is discovered or α itself is discovered to be a

β-splitter. The reader is invited to convince himself that deciding whether the path $\alpha^{(j)}\beta\gamma$ is a *β-splitter* can be done in time at most linear in j.

Suppose $\alpha^{(k)}\beta\gamma$ was the path under consideration by the first phase of the search when it decided to terminate. The second phase of the search now examines the paths $\alpha^{(k)}\beta\gamma$, $\alpha^{(k-1)}\beta\gamma$, ... , in sequence until a *β-splitter* is discovered. This is $\alpha^{*}\beta\gamma$, the longest *β-splitter*. This phase of the search, since it encounters suffixes of S in precisely the same sequence as Algorithm M, can take full advantage of the suffix links and can be done in time linear in k. We know that length(α^{*}) > k/2 (otherwise the first phase would have stopped earlier), and we have just seen that the time to find $\alpha^{*}\beta\gamma$ is linear in 1+2+4+...+k (for the first phase) + k (for the second). Thus $\alpha^{*}\beta\gamma$ can be found in time linear in the length of α^{*}.

The next order of business is to delete all paths of the form $\varepsilon\gamma$, where ε is a suffix of $\alpha^{*}\beta$. These deletions are done in sequence, from longest string to shortest. Suppose that for all suffixes σ of $\alpha^{*}\beta$ longer than ε the deletion of $\sigma\gamma$ has been done. We now consider how to delete the path $\varepsilon\gamma$. The general case for deletion is shown in figure 6:

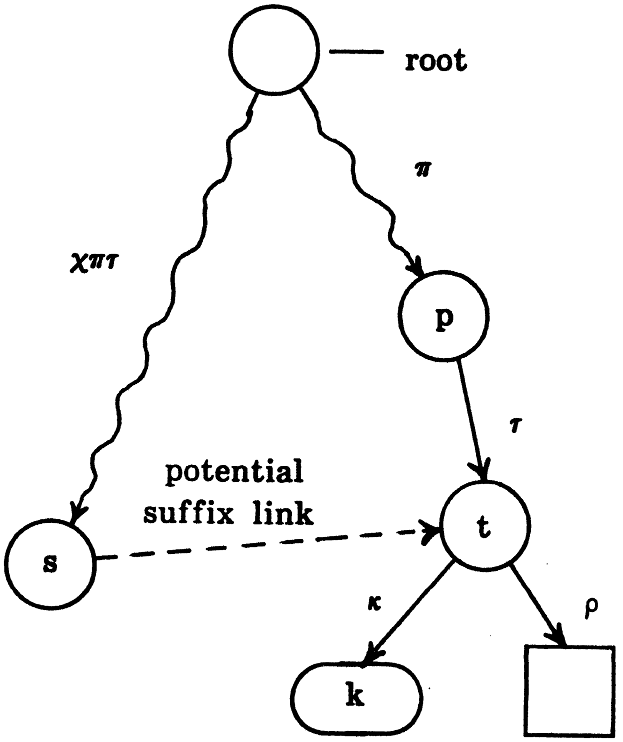


Figure 6

In figure 6, $\epsilon\gamma$ is shown broken up into substrings π, τ, and ρ. If the node t has more than two sons, then the arc ρ (along with attached terminal node) is simply excised. However if node t has exactly two sons, then special action must be taken to avoid violating constraint T2 that every nonterminal node must have at least two son arcs. Node t and its son arc ρ (and attached terminal node) are deleted, and the arc τ from node p and the arc κ from node t are joined together into a new arc $\tau\kappa$ from nonterminal node p to node k.

The only potential flaw is that some nonterminal node s might be suffix-linked to node t when t is deleted. We now argue that this is impossible except for the last nonterminal node in the path $x\alpha^*\delta\gamma$, where $x\alpha^*$ is the shortest suffix of α

which properly contains α^*. (If α^* is α, then the path $x\alpha^*\delta\gamma$ does not exist.) The node having this property may change during the course of the deletions, but at any given time there will be at most one node with this property. Call it s^*.

Lemma 2: Whenever a node t is deleted, no suffix links except perhaps that of node s^* point to it. Furthermore every path in T has a suffix path except perhaps for the path $x\alpha^*\delta\gamma$.

Proof: The lemma is clearly true before any deletions have been done. Suppose we are deleting the path $\varepsilon\gamma$, shown in figure 6 as $\pi\tau\rho$. Further suppose that node t will be deleted.

First we prove by contradiction that there is no nonterminal node s suffix-linked to node t, except perhaps s^*. Suppose there were such a node. In figure 6 we denote the partial path to s by $x\pi\tau$. Because deletions have been done in order of decreasing string length, $\pi\tau\rho$ is assumed to be the longest *β-splitter* in T. If there were a path $x\pi\tau\rho$ in T, a contradiction would arise because that path would be a *β-splitter* and longer than $\pi\tau\rho$.

By inductive hypothesis the only path in T whose suffix path might not be contained in T is $x\alpha^*\delta\gamma$. By constraint T2, node s must have at least two son arcs. Thus, since node t has only one son arc which can have a prefix in T, node s has exactly two son arcs and the only path ($x\alpha^*\delta\gamma$) for which no suffix exists in T must also pass through node s. Further, that son arc of s must lead to a terminal

> node (since otherwise by constraint T2 there would be at least two paths passing through that arc, one of which would have a suffix path in T). Therefore node s is in fact node s^*, the exceptional node.
>
> Now we must show that whether or not node t is deleted, after the path $\varepsilon\gamma$ is deleted, the only path in T which might not have a suffix path is $x\alpha^*\delta\gamma$. At the end of the previous step, the only path which might not have had a suffix path was $x\alpha^*\gamma$. The only action in this step was the removal of the path $\varepsilon\gamma$. If $\varepsilon\gamma$ is $\alpha^*\beta\gamma$, then its prefix path ($x\alpha^*\beta\gamma$) has already been deleted from T (and replaced by $x\alpha^*\delta\gamma$). If $\varepsilon\gamma$ is a proper suffix of $\alpha^*\beta\gamma$, then by inductive hypothesis its prefix path has already been deleted from T. In either case the deletion of the path $\varepsilon\gamma$ does not deprive any path in T of a suffix.

The final order of business is to insert into T all paths of the form $\omega\gamma$, where ω is a nonempty suffix of $\alpha^*\delta$. Inserting these paths is the same as executing a sub-sequence of steps of Algorithm M on a pre-initialized tree. Suppose Algorithm M is invoked with S = $\alpha\delta\gamma$ on an initial tree T_0 which is not empty, but instead contains paths (and suffix links) for all suffixes of γ. We will call this modification Algorithm M(γ). Clearly Algorithm M(γ) will get into trouble as it tries to insert a path for γ into the tree, but we won't let things go that far. Let j = length(α) - length(α^*) + 1 and let k = length($\alpha\delta$). Algorithm M(γ)'s steps j through k are exactly what the updating algorithm must do to insert the paths $\alpha^*\delta\gamma$,...,$d\gamma$ (where d is the last character of $\alpha\delta$). We define $head(\gamma)_i$, *tail*$(\gamma)_i$, *res*$(\gamma)_i$, and $T(\gamma)_i$ in the obvious way. The path-deleting phase of the updating algorithm arranges things so that the path-inserting phase initially has pointers to node s^* and its father. Observe that node s^* is *head*$(\gamma)_{j-1}$, that either it or its father is the contracted locus in $T(\gamma)_{j-1}$ of *head*$(\gamma)_{j-1}$, and that node s^* is the only nonterminal node in $T(\gamma)_{j-1}$ which might fail to have a valid suffix link. These assumptions are precisely the ones necessary to get Algorithm $M(\gamma)$ started at step j. We let it run through the rescanning substep of step k+1, and then stop it. The reader should convince himself that this will result in a tree which satisfies T1-T3 and all of whose suffix links are correct.

Now that the updating algorithm has been informally presented, we can analyze its running time. We have already remarked that finding α^* requires time linear in the length of α^*.

Deleting all paths of the form $\varepsilon\gamma$, where ε is a nonempty suffix of $\alpha^*\beta$, requires finding the terminal arc of each such path, deleting it and its terminal node, and perhaps deleting the nonterminal node from which it emanated. Each of these operations except finding the terminal arc can obviously be done in constant time. Finding the terminal arcs can be done in the manner of Algorithm M, except that pre-existing suffix links eliminate the need for rescanning. Each character of $\alpha^*\beta$ is scanned exactly once in the course of finding all the terminal arcs. Therefore deleting all paths of the form $\varepsilon\gamma$, where ε is a nonempty suffix of $\alpha^*\beta$, can be done in time linear in the length of $\alpha^*\beta$.

How long does it take to run Algorithm $M(\gamma)$ from step j through the rescanning substep of step k+1? Everything but scanning and rescanning may be done in constant time per inserted path. Let d be defined as the last character of $\alpha\delta$, and γ^* as the longest prefix of $d\gamma$ which appears in at least two places in $\alpha\delta\gamma$.

(Note the near-symmetry between γ^* and α^*.) Generalizing the analysis of Algorithm M we see that the number of intermediate nodes encountered during rescanning is $_{i=j}$ int$(\gamma)_i$, which is at most length(*res*$(\gamma)_j$) - length(*res*$(\gamma)_{k+1}$) + int$(\gamma)_{k+1}$. Clearly length(*res*$(\gamma)_j$) is at most length(*suf*$_j$), which is length($\alpha^*\delta\gamma$). In general, int$(\gamma)_i$ is at most the length of β in step i, which in turn is contained in *head*$(\gamma)_{i-1}$. In particular, int$(\gamma)_{k+1}$ is at most length(*head*$(\gamma)_k$). Also in general, length(*res*$(\gamma)_{i+1}$) is at least length(*suf*$(\gamma)_i$) - length(*head*$(\gamma)_i$), so in particular length(*res*$(\gamma)_{k+1}$) is at least length(*suf*$(\gamma)_k$) - length(*head*$(\gamma)_k$). Hence $_{i=j}$ int$(\gamma)_i$ is at most length($\alpha^*\delta\gamma$) - length(dγ) + 2 length(γ^*), so rescanning time is linear in length($\alpha^*\delta$) + length(γ^*).

The same collapsing series used in the analysis of Algorithm M shows that the number of characters scanned during steps j through k is exactly (k-j+1) + length(*head*$(\gamma)_k$) - length(*head*$(\gamma)_{j-1}$). The time spent in scanning is at most linear in length($\alpha^*\delta$) + length(γ^*).

We have thus shown that changing T to reflect a change in S from $\alpha\beta\gamma$ to $\alpha\delta\gamma$ can be performed in time at most linear in the sum of the lengths of α^*, β, δ, and γ^*.

CONCLUSION

The first algorithm to generate suffix trees in linear time was discovered by Weiner <Wei 73>. A lucid description of Weiner's algorithm, with additional insights, appears in unpublished lecture notes by Knuth <Knu 73b>. The state of the art of pattern matching, including Weiner's algorithm, is well presented in a book by Aho, Hopcroft, and Ullman <Aho 74>. Pratt, following Weiner's work, has devised an unpublished algorithm to solve this problem in a slightly different way <Pra 73>. All of these algorithms solve the problem in time (and of course space) linear in the length of the input string.

The difference between Algorithm M and the other algorithms above is that Algorithm M can use less data space. The number of nodes generated by each algorithm is approximately the same, although Weiner's original algorithm generates slightly more than the others. Further, the information content per node is approximately the same among the algorithms, with one significant exception. The exception is that by processing left-to-right and never extending any substring to the left, Algorithm M avoids the leftward pointer per node per alphabet symbol which is required by the other algorithms. This represents a savings of about 25% in data space between the hash-coded version of Algorithm M and similarly coded versions of the other algorithms. One would expect roughly similar savings for other tree representations.

ACKNOWLEDGEMENTS

I am indebted to Peter Weiner and Vaughan Pratt for early discussions of these ideas and to Ben Wegbreit, Ralph Kimball, and Jim Morris for constructive criticism of the paper itself.

REFERENCES

<Kim 74> Kimball, R. B. "A Rapid Substring Searching Algorithm in Speech Recognition", abstracted in Conference Record, IEEE Symposium on Speech Recognition (Pittsburgh, Penna.: April, 1974).

<Knu 73> Knuth, D. E. Sorting and Searching, Vol. 3 of The Art of Computer Programming (Reading, Mass.: Addison-Wesley, 1973), chapter 6.3, pp. 490-493.

<Aho 74> Aho, A. V.; Hopcroft, J. E.; and Ullman, J. D. The design and analysis of computer algorithms (Reading, Mass.: Addison-Wesley, 1974) chapter 9, pp. 317-361.

<Knu 74> Knuth, D. E.; Morris, J. H., Jr.; and Pratt, V. R. Fast Pattern Matching in Strings (Stanford, Calif: Stanford University Computer Science Report STAN-CS-74-440, August, 1974).

<Knu 73b> Knuth, D. E. "Pattern Matching in Strings" (Trondheim, Norway: unpublished lecture notes, May, 1973).

<Wei 73> Weiner, P. "Linear Pattern Matching Algorithms" in Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory, pp. 1-11.

<Pra 73> Pratt, V. R. "Applications of the Weiner Repetition Finder" (Cambridge, Mass.: unpublished paper, May, 1973; revised October, 1973).

<Wag 74> Wagner, R. A. "Order-n Correction for Regular Languages" Comm. ACM, 17 (May, 1974), pp. 265-268.