Efficient and Elegant Subword-Tree Construction

M.T. Chen and J.I. Seiferas
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR129
December, 1983

Abstract

A clean version of Weiner's linear-time compact-subword-tree construction simultaneously constructs the smallest deterministic finite automaton recognizing the *reverse* subwords.

Part of this work was done while the first author visited the University of Rochester. The work of the second author was supported in part by the National Science Foundation under grant MCS-8110430. Authors' addresses: M.T. Chen, Department of Computer Science, University of Nanjing, People's Republic of China; J.I. Seiferas, Department of the Computer Science, University of Rochester, Rochester, NY 14627.

Introduction

Any finite set S of words which is prefix-closed (i.e., $xy \in S \Rightarrow x \in S$) has a *prefix tree* with node set S, ancestor relation "is a prefix of", and father relation "is obtained by dropping the last letter of." The set S of all subwords of a text string is a prefix-closed set whose prefix tree, the text's *subword tree*, is particularly useful. For example, it lets us test arbitrary words for membership in S in time proportional to their own lengths, regardless of how long the entire text is. Even more useful is the subword tree for ¢w\$, where ¢ and \$ are delimiting symbols not occurring in w. This, for example, lets us test easily whether a word is a suffix of w. In one appropriate walk through the tree, we can easily augment each node with such information as the count of its leaf descendents. Then it becomes convenient to tell *how many* times a word appears, *where* a word first or last appears, what is the longest repeated subword, and more. As an example, the subword tree for ¢*aabab*\$ is shown in Figure 1.

The number of distinct subwords of a text string of large length n can be very large (proportional to n^2 for $a^{n/2}b^{n/2}$, for example), so subword trees can have prohibitively many nodes. Fortunately, however, there are compact but functionally equivalent data structures which can even be *built* in time proportional to just n. Weiner [11], McCreight [3], Pratt [4,5,6], and Slisenko [10, Section 2] have described such data structures and algorithms. (See also [1, Section 9.5], [2].) Each of their algorithms is complicated by the maintenance of additional auxiliary structure along with the developing compact subword tree. (In Slisenko's case, more ambitious applications account for an extra measure of additional structure [7-10].) In this report, we describe a version with auxiliary structure which is unusually clean and clearly desirable in its own right.

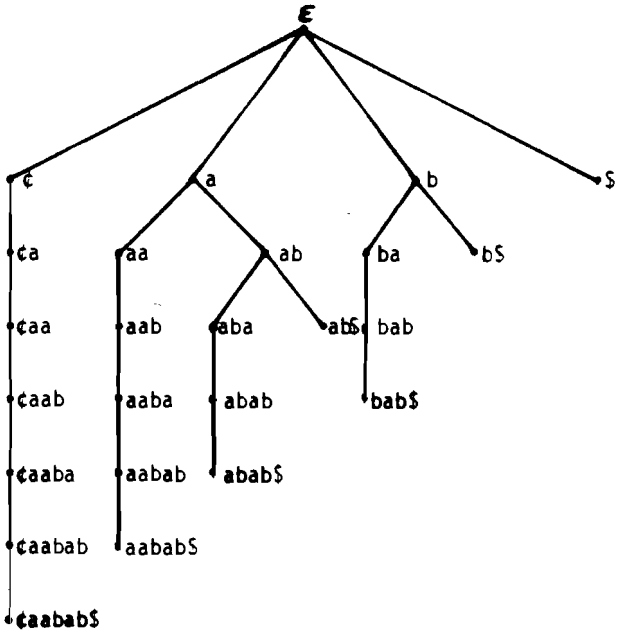

Figure 1: Node-labeled subword tree for ¢aabab$

Compact Representations

The most obvious way to represent a subword tree compactly is to omit interior nodes of degree 1, replacing them by through edges. The string corresponding to each remaining node can be represented by a (not necessarily unique) pair of pointers into the text string; or, alternatively, the incremental substring corresponding to each edge can be represented by such a pair. Resulting representations for ¢*aabab*$ are shown in Figure 2. Either way, no information is lost, and the degree of each remaining node continues to be bounded by the alphabet size. The size of this representation is thus proportional to the number of nodes. And the number of nodes is proportional to the length of the text, because the number of leaves is so bounded (one for each suffix of the text) and because the number of interior nodes is bounded by the number of leaves in a tree without interior nodes of degree 1.

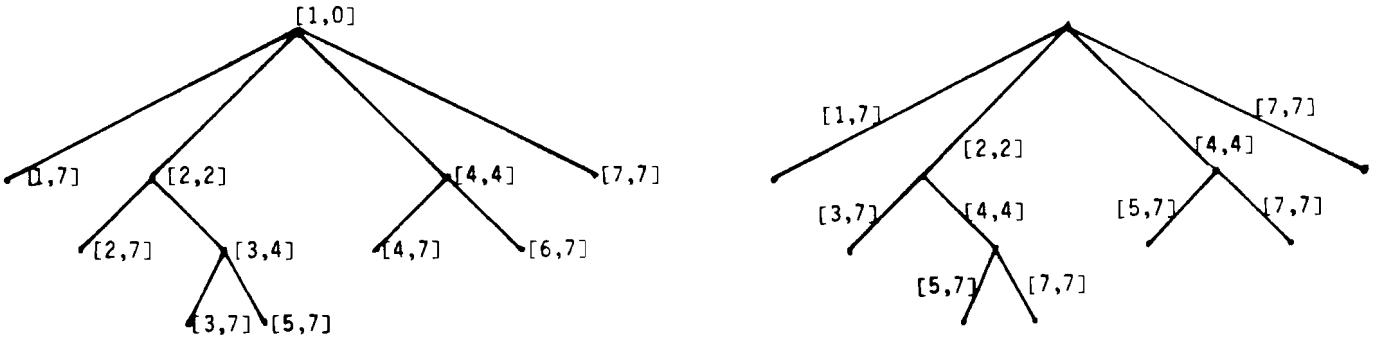

Figure 2: Compacted subword tree for ¢aabab$, node-labeled and edge-labeled

We obtain a quite different compact representation by identifying (edge-)isomorphic subtrees. (The edge labels to be preserved by each such isomorphism are the *strings*, not the indices used to represent them.) The edge-labeled version of the subword tree for ¢*aabab*$ (shown in Figure 3), for example, has isomorphic subtrees below the subwords *b* and *ab*. The result of making all such identifications is shown in Figure 4. (For expositional clarity in our figures, we revert to explicit edge labels. For later reference, however, parenthesized capital letters have been arbitrarily assigned as names for the nodes in Figure 4.) Except for omission of the one nonaccepting state, which is "dead," this directed acyclic graph is just the smallest deterministic finite automaton recognizing the set of subwords of the text. We will see that the size of this representation is again only proportional to the length of the text, *and for essentially the same reason.*

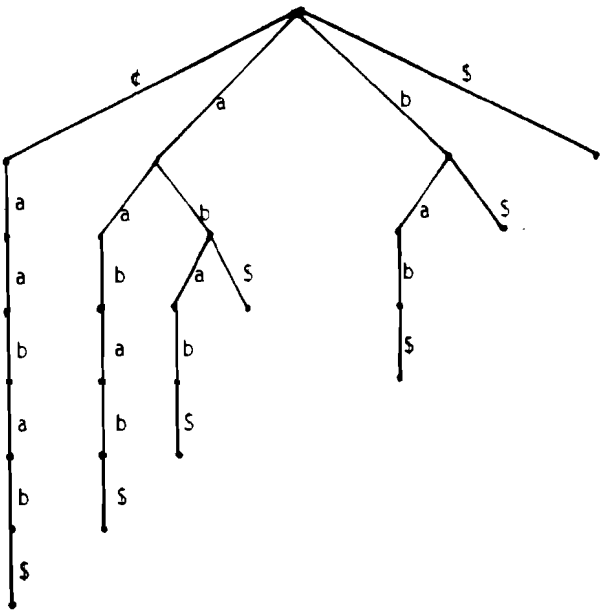

Figure 3: Edge-labeled version of Figure 1

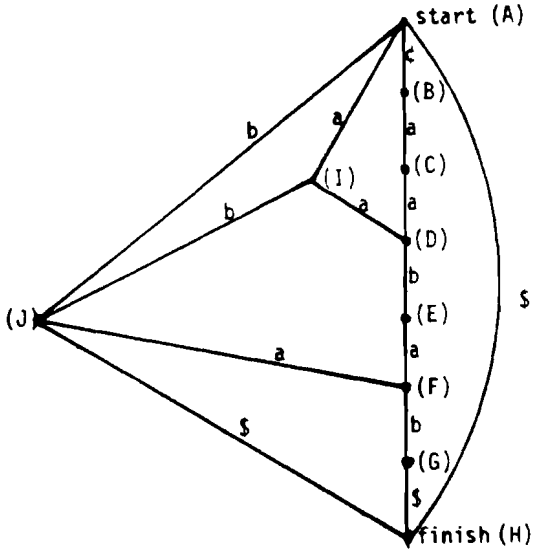

Figure 4: Smallest subword DFA for ¢aabab$

Approaches to Construction

The main objective of Weiner's algorithm is to build the first compact representation of the subword tree. To do this, it maintains auxiliary information which happens to include explicitly some (but not all) of the son links from the *uncompacted, reverse* subword tree. (Such terminology is unambiguous, because the tree of reverse subwords of the text is identical to the tree of ordinary subwords of the reverse text.) Similarly, McCreight's algorithm happens to maintain *father* links from the uncompacted reverse subword tree.

Pratt's and Slisenko's algorithms are essentially the same as Weiner's algorithm, but with a different viewpoint. The main objective is to build that portion of the uncompacted subword tree whose nodes are the text's "longest repetitions." With a few additional patches, this turns out to be as useful as the entire subword tree.

The concept of a "longest repetition" is a dynamic or historical one, based on successive consideration of longer and longer text prefixes. In each prefix, the longest suffix which also occurs elsewhere in the prefix is a "longest repetition." Fortunately, a simple *static* definition is easily seen to be equivalent: A string x is a longest repetition if and only if there are distinct letters a and b for which both ax and bx occur as subwords. From this redefinition, it is clear that the longest repetitions correspond precisely to the interior nodes of degree greater than 2 in the reverse subword tree. The auxiliary structure from Weiner's algorithm turns out to include enough of the uncompacted reverse subword tree so that Pratt and Slisenko can simply run that algorithm on the reverse of their texts, regarding Weiner's bathwater as their baby (and his baby as their bathwater).

Our new observation is that there is a much more natural choice of the "few additional patches" maintained above. Instead of information of some new kind, we can add additional edges (but *no* additional nodes) to the fragment of the uncompacted reverse subword tree to get a directed acyclic graph which, for the reverse subword tree, is the *second* compact representation described above. To see how, note that, whenever one subword y is a suffix of another subword xy *and occurs only in that context*, the subtrees below the two strings must be isomorphic; i.e., for every z, yz is a subword if and only if xyz is. This is the reason for the isomorphism below b and ab in Figure 3, for example; and it is clearly the *only* possible reason for subtree isomorphism in a subword tree. In the *reverse* subword tree, similarly, the subtrees below the reverses of two text subwords are isomorphic if and only if one of the subwords is a *prefix* of the other and occurs only in *that* context. This occurs if and only if all the nodes from the shorter subword through its longer extension have degree 1 in the ordinary subword tree. Thus the nodes of the first comapct representation of the subword tree correspond to a set of distinct representatives of the isomorphism classes of the strings in the uncompacted reverse subword tree, and we will have the second compact representation for the latter is the a-edge from each such node x is directed to the shortest extension axy of ax which is also such a node. The compact representation of the subword tree for ¢*aabab*$ shown in Figure 4 was obtained from the subword tree for $*babaa*¢ (Figure 5) by just this rule. The named nodes in Figure 5 correspond to the similarly named nodes in Figure 4.

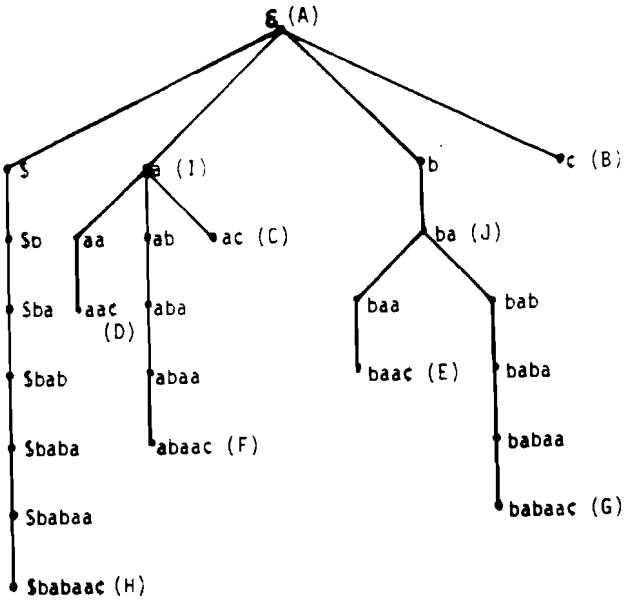

Figure 5: Node-labeled subword tree for $babaa¢

Our Algorithm

Before we spell out our variant of the algorithm, let us summarize the specifications for the cleaned up data structure the algorithm is supposed to construct. The text *w* is a character string, and $ is a right endmarker not occurring in *w*. There is a node for each suffix of *w*$ and a node for each subword *x* occurring in two distinct immediate right contexts (i.e., with distinct letters *a* and *b* for which both *xa* and *xb* are subwords of *w*$). From each node *x*, for each letter *a*, there is an *a-extension* link to the shortest node (if any) with prefix *xa* and an *a-shortcut* link to the shortest node (if any) with prefix *ax*. From each node, there is a *prefix* link to its longest proper prefix node (if any). (Noting that prefix links are just the reverses of extension links, we will leave setting them implicit whenever we create or change extension links.) Finally, for each node, there is a pair of indices into *w*$ to identify (one instance of) the coresponding subword. (We will also leave implicit the setting of these indices. Each new node added will be a prefix of the entire text so far considered; it will be either that entire text or a prefix 1 longer than the known length of some older node.)

As our terminology suggests, we will describe the algorithm from Weiner's viewpoint: The main objective is to build the extension and prefix links, and the other links provide time-saving "shortcuts." Even so, the algorithm works from *right to left* in the text, with no need to look "ahead" (to the left), provided we index the letters of *w*$ from right to left.

The structure for *w*$ = $ is trivial (two nodes) and can obviously be built in constant initial time. To build the structure for *aw*$, we assume inductively that we have the structure for *w*$ and that we have pointers to the root node and to the node *w*$, where we finished the previous step. The new subwords will be the "sufficiently long" prefixes of *aw*$; we will have to install *aw*$ as a new extension from the longest prefix y of *aw*$ which is already a

subword of $w\$$. If the root (corresponding to the null subword) does not yet have an a-extension, then it serves as y. If the root does already have an a-extension, we could still find y by "following $aw\$$ along extension links" down from the root until continuation would leave the tree; but for a string like $a^n\$$, this would accumulate to time proportional to n^2. Instead, noting that y's least-proper suffix $x (y = ax)$ must already be a node in the structure for $w\$$ and that it must be a prefix of $w\$$, we can trace up along *prefix* links from $w\$$, watching for the node x; it will be the first node with an a-shortcut. Following that shortcut will lead to $ax = y$ if it is already a node, or to its shortest extension axz which is a node otherwise. In the latter case, we will have to install y as a new node between axz and its prefix parent, initially with the same shortcut links axz has. The shortcuts *to* the new node will be directed from the nodes lying on the prefix path from x up through the last node x' not already having an a-shortcut link to a *proper prefix* of ax.

With y found and properly installed, we can install $aw\$$ as an extension below it as required, initially without any extension or shortcut links. Shortcut links should be directed *to* this new node from the nodes lying on the prefix path from $w\$$ up through the last node not already having an a-shortcut link. Both these and the shortcut links redirected to y in the case that y had to be installed can be set in a traversal of the prefix path from $w\$$ up through x'; so the time to obtain the structure for $aw\$$ from the one for $w\$$ is proportional to some constant plus the number of nodes on the prefix path from $w\$$ to x'.

To see that this time bound accumulates only to linear time, we look at the node depth of each successive text suffix in the extension tree. The key observation is that, except for some small additive constant, the depth of $aw\$$ within its structure is *reduced* from the depth of $w\$$ within its structure at least enough to compensate for the time-indicative number of nodes on the prefix path from $w\$$ to x' above. (See Figure 6.) To see this, first note that the depth of x' is certainly so reduced. Then note that, if ax'' is any node on the prefix path above y, x'' must be a node on the prefix path above x'. The consequence of the observation is that to spend more than linear total time would require more than linear total depth reduction, which is impossible since the greatest possible *increase* in depth is constant for each iteration.

Conclusion

A clean version of Weiner's algorithm provides natural but very different indices of the subwords and the reverse subwords of a text. It is remarkable that such functional symmetry is efficiently achieved by such an integrated asymmetric construction.

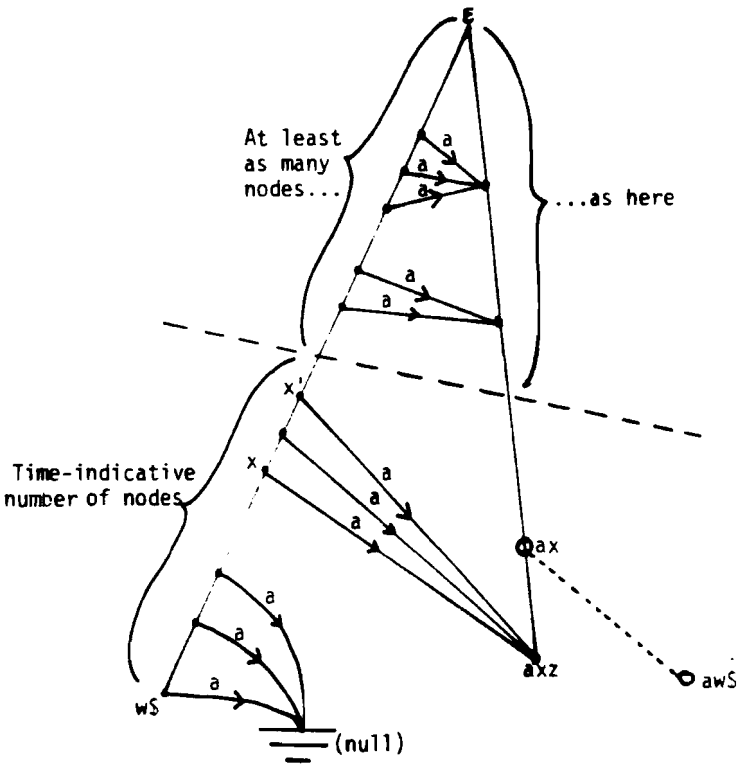

Figure 6: How depth reduction compensates for time

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Massachsuetts, 1974.

2. M.E. Majster and A. Reiser, "Efficient on-line construction and correction of position trees," *SIAM Journal on Computing*, 9, 4 (November 1980), 785-807.

3. E.M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the Association for Computing Machinery, 23*, 2 (April 1976), 262-272.

4. V.R. Pratt, "Improvements and applications for the Weiner repetition finder," unpublished manuscript (May 1973, October 1973, and March 1975).

5. M. Rodeh, V.R. Pratt, and S. Even, "A linear algorithm for finding repetitions and its applications in data compression," Technical report no. 72, Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel (April 1976).

6. M. Rodeh, V.R. Pratt, and S. Even, "Linear algorithm for data compression via string matching," *Journal of the Association for Computing Machinery, 28*, 1 (January 1981), 16-24.

7. A.O. Slisenko, "String-matching in real time," Preprint P-7-77, The Steklov Institute of Mathematics, Leningrad Branch (September 1977) (Russian).

8. A.O. Slisenko, "String-matching in real time: Some properties of the data structure," *Mathematical Foundations of Computer Science 1978* (Proceedings, 7th Symposium,

Zakopane, Poland, 1978) (Lecture Notes in Computer Science 64), Spring-Verlag, Berlin, 1978, 493-496.

9. A.O. Slisenko, "Determination in real time of all the periodicities in a word," *Soviet Mathematics - Doklady*, 21, 2 (March-April 1980), 392-395.

10. A.O. Slisenko, "Detection of periodicities and string-matching in real time," *Journal of Soviet Mathematics*, 22, 3 (June 11, 1983), 1316-1387; translated from Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V.A. Steklova AN SSSR, **105** (1980), 62-173.

11. P. Weiner, "Linear pattern matching algorithms," 14th Annual Symposium on Switching & Automata Theory (Iowa City, Iowa), IEEE Computer Society, 1973, 1-11.