Theoretical Computer Science 40 (1985) 31-55
North-Holland

THE SMALLEST AUTOMATON RECOGNIZING THE SUBWORDS OF A TEXT*

A. BLUMER, J. BLUMER and D. HAUSSLER
Department of Mathematics and Computer Science, University of Denver, Denver, CO 80208, U.S.A.

A. EHRENFEUCHT
Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80302, U.S.A.

M.T. CHEN
Department of Computer Science, University of Nanjing, Nanjing, Jiangsu, People's Republic of China

J. SEIFERAS
Department of Computer Science, University of Rochester, Rochester, NY 14627, U.S.A.

Abstract. Let a partial deterministic finite automaton be a DFA in which each state need not have a transition edge for each letter of the alphabet. We demonstrate that the smallest partial DFA for the set of all subwords of a given word w, $|w|>2$, has at most $2|w|-2$ states and $3|w|-4$ transition edges, independently of the alphabet size. We give an algorithm to build this smallest partial DFA from the input w on-line in linear time.

Introduction

In the classic string matching problem for text, we are given a text w and a pattern string x and we want to know if x appears in w, i.e., if x is a subword of w. Standard approaches to this problem involve various methods for preprocessing x so that the text w can be searched rapidly [1, 9, 16]. Since each search still takes time proportional to the length of w, this method is inappropriate when many different patterns are examined against a fixed text, e.g., for repeated lookups in any fixed textual database. In this case, it is desirable to preprocess the text itself, building an auxiliary data structure that allows one to determine whether x is a subword of w in time proportional to the length of x, not w. Data structures with this property (known as 'suffix trees' or 'compact position trees', earlier as 'PATRICIA trees') have been developed [17, 19, 21, 23, 24, 25, 26, 28] and used in a wide variety of pattern matching applications, in addition to the classic string matching problem given above (e.g., [2, 3, 4, 5, 22, 27]).

Clearly, a deterministic finite automaton (DFA) that recognizes the set of all subwords of w would serve as an auxiliary index for w in the above sense. We can

* Part of this work was done while M.T. Chen visited the University of Rochester. A. and J. Blumer and D. Haussler gratefully acknowledge the support of NSF Grant IST-8317918, A. Ehrenfeucht the support of NSF Grant MCS-8305245, and J. Seiferas the support of NSF Grant MCS-8110430.

also allow this automaton to be partial, i.e., such that each state need not have a transition on every letter. We demonstrate the feasibility of this approach by exhibiting a partial DFA that recognizes the set of subwords of w and has less than $2|w|$ states and $3|w|$ transition edges (independently of the size of the alphabet of w). This DFA can be built in linear time for any fixed alphabet size by an algorithm that operates on-line in the strong sense that the DFA is correct after each letter is processed. As all states of this automaton are accepting, it can be viewed as a directed acyclic graph, which we call the Directed Acyclic Word Graph, or DAWG [6]. Crochemore [14] has pointed out that with a different assignment of accepting states, this DFA is the smallest automaton for the set of all suffixes of w. While it is not the smallest automaton for the subwords of w, with some additions and modifications to the DAWG construction algorithm we derive an algorithm that builds the smallest partial DFA for this language. This algorithm also runs in linear time, and is on-line in the strong sense. These automata can be used in place of suffix trees or compact position trees in most of the applications of these data structures cited above, and have additional desirable properties that in some cases make them more useful [7, 13].

The algorithm that builds the DAWG (or, with a linear postprocessing phase added, the smallest DFA for all suffixes of a word) can be viewed as an extension of earlier algorithms used to build the compact position tree [21, 23, 28]. The linear bound on the running time is obtained by employing auxiliary pointers that, if reversed, form a structure equivalent to the compact position tree of Weiner for the reverse of w. In contrast, the goal of Weiner's algorithm is to build the compact position tree for w, which it does by processing w from right to left, maintaining auxiliary pointers that form part of the DAWG for the reverse of w. Pratt's algorithm can be seen as an intermediate step between Weiner's algorithm and ours, since it is closely patterned after Weiner's algorithm, but is designed to build a structure that forms part of the DAWG for w by processing w from left to right, maintaining the compact position tree as an auxiliary structure as in our algorithm. Slisenko's algorithm is essentially the same as Pratt's, but more ambitious applications lead to an extra measure of additional structure. A more detailed comparison of our algorithm with earlier algorithms can be found in [10, 11], where it is given from Weiner's point of view. The approach taken in this paper follows that used in [27], where a graph closely related to that of [21] is constructed. Modifications to the DAWG algorithm needed to construct the smallest DFA for the subwords of a word have been given independently by Crochemore [12], who also gives precise bounds for this latter DFA.

Notation

Throughout this paper, Σ denotes an arbitrary nonempty finite alphabet and Σ^* denotes the set of all strings (words) over Σ. The empty word is denoted by λ. w

will always denote an (arbitrary) word in Σ^*, and a a letter in Σ. $|w|$ denotes the length of w. If $w = xyz$ for words $x, y, z \in \Sigma^*$, then y is a *subword* of w, x is a *prefix* of w, and z is a *suffix* of w. In addition to the standard terminology for finite automata, we use the term *partial* DFA (for the alphabet Σ) for a deterministic finite automaton in which each state need not have a transition for every letter of Σ. (Lack of any transition needed for a word signifies rejection of that word.) The *smallest* partial DFA for a given language is the partial DFA that recognizes the language and has the smallest number of states. (Uniqueness follows from Nerode's theorem [15, 20].) As usual, an equivalence relation $\equiv$ on Σ^* is *right invariant* if, for any $x, y, z \in \Sigma^*$, $x \equiv y$ implies that $xz \equiv yz$.

1. The directed acyclic word graph

We begin with a brief look at some aspects of the subword structure of a fixed, arbitrary word w. In particular, for each subword y of w we will be interested in the set of positions in w at the ends of occurrences of y. This is essentially the same approach as that taken in [27].

Definition. Let $w = a_1 \ldots a_n$ $(a_1, \ldots, a_n \in \Sigma)$ be a word in Σ^*. For any nonempty y in Σ^*, the *end-set* of y in w is given by $end\text{-}set_w(y) = \{i\colon y = a_{i-|y|+1} \ldots a_i\}$. In particular, $end\text{-}set_w(\lambda) = \{0, 1, 2, \ldots, n\}$. We say that x and y in Σ^* are *end-equivalent* (on w) if $end\text{-}set_w(x) = end\text{-}set_w(y)$, and we denote this by $x \equiv_w y$. We denote by $[x]_w$ the equivalence class of x with respect to $\equiv_w$. The *degenerate class* is the equivalence class of words that are *not* subwords of w (i.e., words with empty end-set).

For illustrations of these definitions, see Fig. 1.

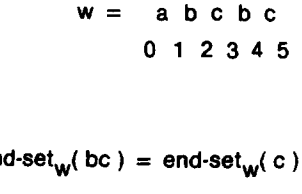

Fig. 1.

The following lemma summarizes some obvious properties of end-equivalence.

Lemma 1.1. (i) *End-equivalence is a right-invariant equivalence relation on* Σ^*.
(ii) *If two words are end-equivalent, then one is a suffix of the other.*

(iii) *Two words xy and y are end-equivalent if and only if every occurrence of y is immediately preceded by an occurrence of x.*

(iv) *A subword x of w is the longest member of* $[x]$ *if and only if either it is a prefix of w, or it occurs in two distinct immediate left contexts (i.e., both ax and bx occur, for some distinct* $a, b \in \Sigma$*).*

We see from Lemma 1.1 that there are really three ways to look at the nondegenerate equivalence classes of $\equiv_w$. We can look at them as the set of distinct end-sets for the subwords of w, as a partition of the subwords of w, or as a set of canonical members of this partition, formed by taking the longest word in each equivalence class. For the latter viewpoint, we will say that the longest member of $[x]_w$ (*canonically*) *represents* the equivalence class $[x]_w$.

Taking advantage of right invariance, we can consistently define a (partial) deterministic finite automaton as in Nerode's well-known construction [15, 20].

Definition. The *Directed Acyclic Word Graph* (DAWG) for w is the (partial) deterministic finite automaton D_w with input alphabet Σ, state set $\{[x]_w^* | x \text{ is a subword of } w\}$, start state $[\lambda]_w$, all states accepting, and transitions $\{[x]_w \to^a [xa]_w | x \text{ and } xa \text{ are subwords of } w\}$.

D_w is illustrated for the word $w = abcbc$ in Fig. 2(a), (b). Since the states of D_w are exactly the nondegenerate classes of $\equiv_w$, we shall use the terms 'state' and 'class' interchangeably throughout the remainder of this paper.

Lemma 1.2. *D_w recognizes the set of all subwords of w.*

Proof. This follows directly from Nerode's theorem, since the union of the equivalence classes that form the accepting states of D_w is exactly the set of subwords of w. □

It is easily verified that D_w is not always the smallest partial DFA for the subwords of w (see Section 3 and Fig. 4). However, as we shall show below, worst-case size bounds for D_w are not significantly higher than those for the smallest DFA. In addition, the correspondence between the states of D_w and the end-sets of the subwords of w is useful in some applications [7], and makes the on-line algorithm for D_w somewhat simpler than that for the smallest automaton. As a result, we will approach the construction of the smallest automaton for the subwords of w by first developing an on-line algorithm for D_w, and then making the necessary modifications to this algorithm needed to build the smallest DFA.

A close relative of D_w is also of some interest from a theoretical point of view.

Definition. Let S_w denote the partial DFA defined as D_w except that the only

accepting states are those equivalence classes that include suffixes of w (i.e., whose end-sets include the position $|w|$).

Proposition 1.3 (Crochemore). *For any $w \in \Sigma^*$, S_w is the smallest partial DFA that recognizes the set of all suffixes of w.*

Proof. By design, the set of suffixes of w is the union of the equivalence classes that are accepting states in S_w. Hence, S_w recognizes the set of suffixes of w, by Nerode's theorem [15, 20]. To prove minimality (again by Nerode's theorem), we need only note that, for any $x, y \in \Sigma^*$, $x \equiv_w y$ if, for all $z \in \Sigma^*$, xz is a suffix of w exactly when yz is a suffix of w. □

We now derive bounds on the maximum number of states and edges in the automaton D_w, in terms of the length of w. For this and for the development in Section 2 of an algorithm to construct D_w it will help to look at the (nondegenerate) states from the end-set point of view. If two strings' end-sets meet, then one of the strings must be a suffix of the other, so that one of the end-sets must be a superset of the other. Therefore, the (nonempty) subsets of $\{0, 1, 2, \ldots, |w|\}$ that are end-sets form a subset tree $T(w)$ (see Fig. 3(a)).

Lemma 1.4. *If x canonically represents an equivalence class modulo $\equiv_w$, then the children of $[x]_w$ in $T(w)$ are those classes $[ax]_w$ for which $a \in \Sigma$ and ax is a subword of w.*

Proof. The children of $[x]_w$ correspond to the maximal proper subsets of *end-set*$_w(x)$ that are end-sets. Any such end-set must be *end-set*$_w(vx)$ for some nonnull word v. As a canonical representative, x is the longest subword with its end-set; so, we already get the children for $|v| = 1$. □

It follows from Lemma 1.4 that when w begins with a unique letter, $T(w)$ is isomorphic to the compact position tree of Weiner for the reverse of w [28], except that its edges are unlabeled. This is illustrated in Fig. 3(a), (b).

Lemma 1.5. *Let $|w| > 2$, D_w has at most $2|w| - 1$ states, and this upper bound is achieved if and only if $w = ab^n$ for some $a, b \in \Sigma$, $a \neq b$.*

Proof. In the special case that w is of the form a^n for $n > 2$, $T(w)$ is a simple chain of $n + 1 < 2n - 1$ nodes. In the remaining case, we show that $T(w)$ has at most $|w|$ nonbranching nodes (nodes of degree less than 2), and hence at most $|w| - 1$ branching nodes, for a total of at most $2|w| - 1$.

By Lemma 1.4, any branching node in $T(w)$ occurs in at least two distinct left contexts. By Lemma 1.1(iv), the only possible other (nonbranching) nodes are the $|w| + 1$ prefixes of w. Since w is not of the form a^n, however, one of these prefixes,

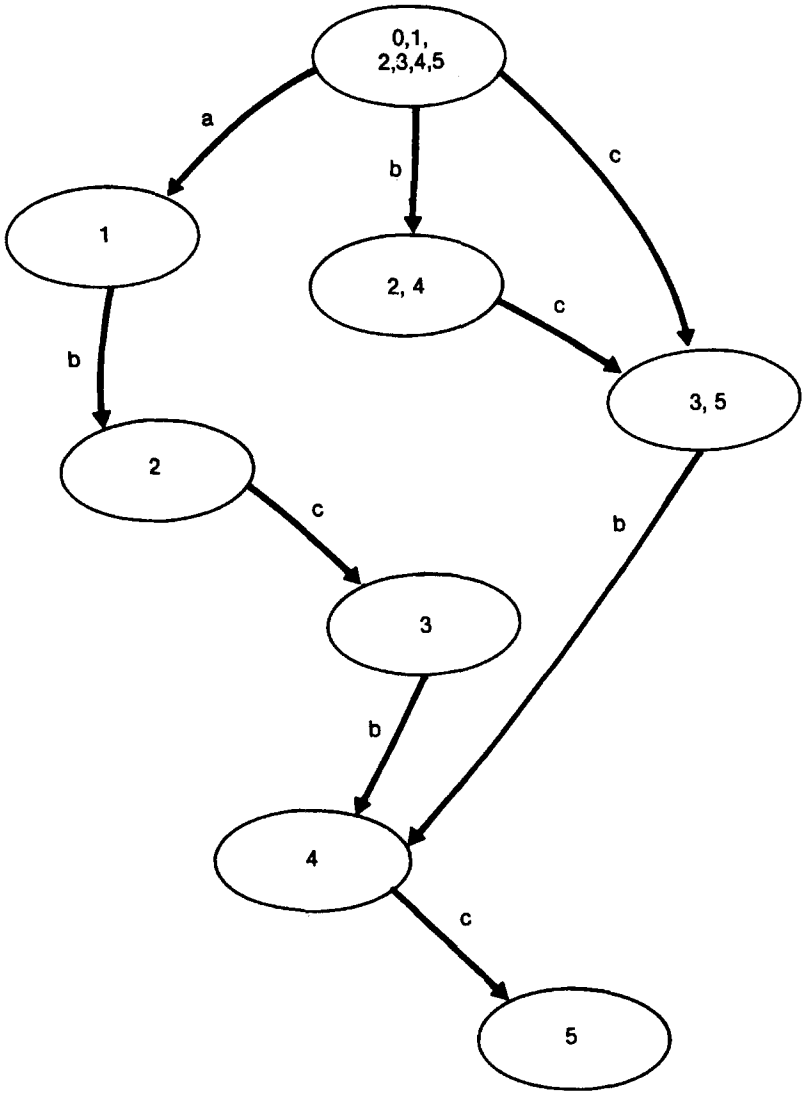

Fig. 2(a). D_w with classes denoted by end-sets.

the null one, appears in two distinct left contexts and so, by Lemma 1.4, is *not* a nonbranching node. This completes the proof of the upper bound.

To reach the upper bound, we need $|w|-1$ branching nodes. This generates at least $|w|$ leaves. Since only the $|w|$ nonnull prefixes of w can be nonbranching nodes, they must all be leaves, and every internal node must have exactly two children. In particular, the one-letter prefix of w must be a leaf; so, by Lemma 1.4, that first letter cannot occur elsewhere in w. Since the null string (the root) can have only one other child beside the one-letter prefix, only one other letter can occur, and w must be of the form ab^n. Conversely, it is easy to verify that the subwords of ab^n do in fact generate $2(n+1)-1$ distinct end-sets for any $n \geq 1$, to complete the proof. □

Lemma 1.6. *For $|w| \geq 2$, there are at most $|w|-2$ more edges than nodes in the transition graph of D_w.*

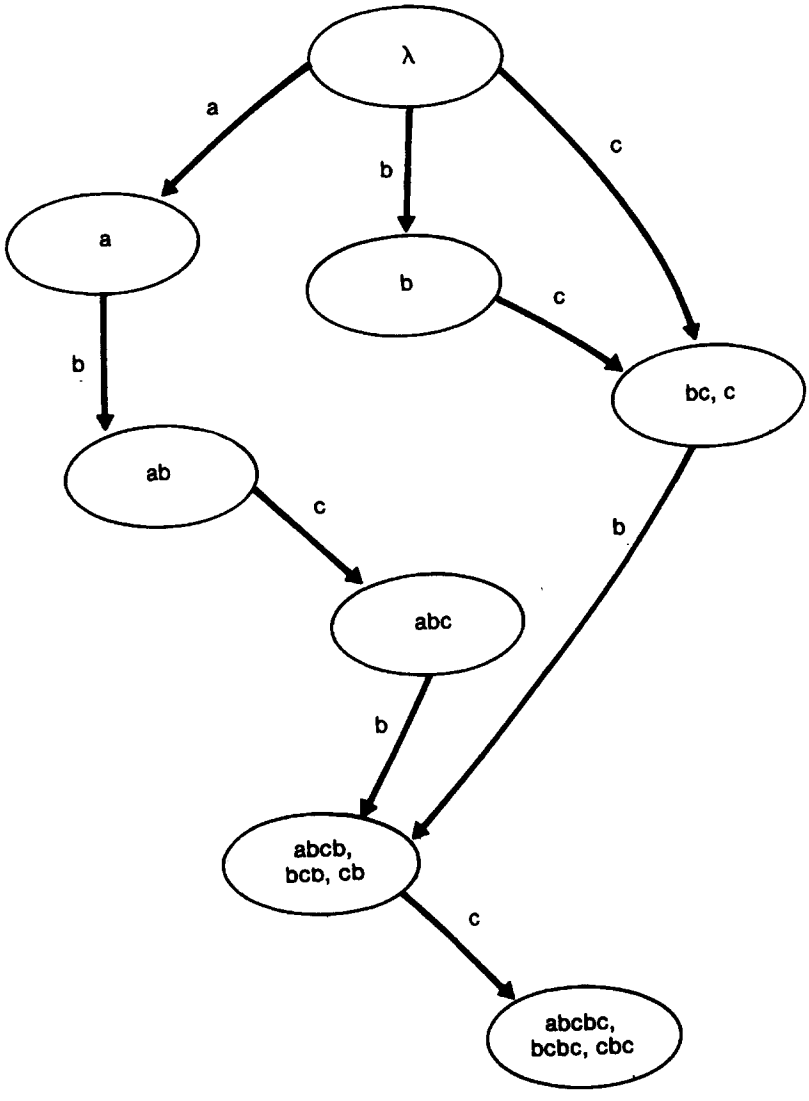

Fig. 2(b). D_w with classes explicitly given.

Proof. Note that the transitions of D_w form a directed acyclic graph with one source ($[\lambda]_w$) and one sink ($[w]_w$). Every state of D_w lies on a path from the source to the sink, and the sequence of labels on each such distinct path forms a distinct nonempty suffix of w.

Any such directed acyclic graph has a directed spanning tree rooted at its source, so focus on one. Being a tree, it will have one fewer edges than nodes; so it only remains to show that at most $|w|-1$ edges of D_w are left out of the spanning tree.

With each edge of D_w not in the spanning tree, we associate one of the $|w|$ nonempty suffixes of w. We obtain that suffix from the labels on a directed path going from the source, through the spanning tree to the tail of the omitted edge, across the omitted edge, and finally on to the sink in any convenient way. Distinct omitted edges are associated with distinct nonempty suffixes, because they are associated with distinct source-to-sink paths. (The paths differ in the first edge traversed outside the spanning tree.) One source-to-sink path lies entirely within the spanning tree, so its nonempty suffix is not assigned; therefore, the number of

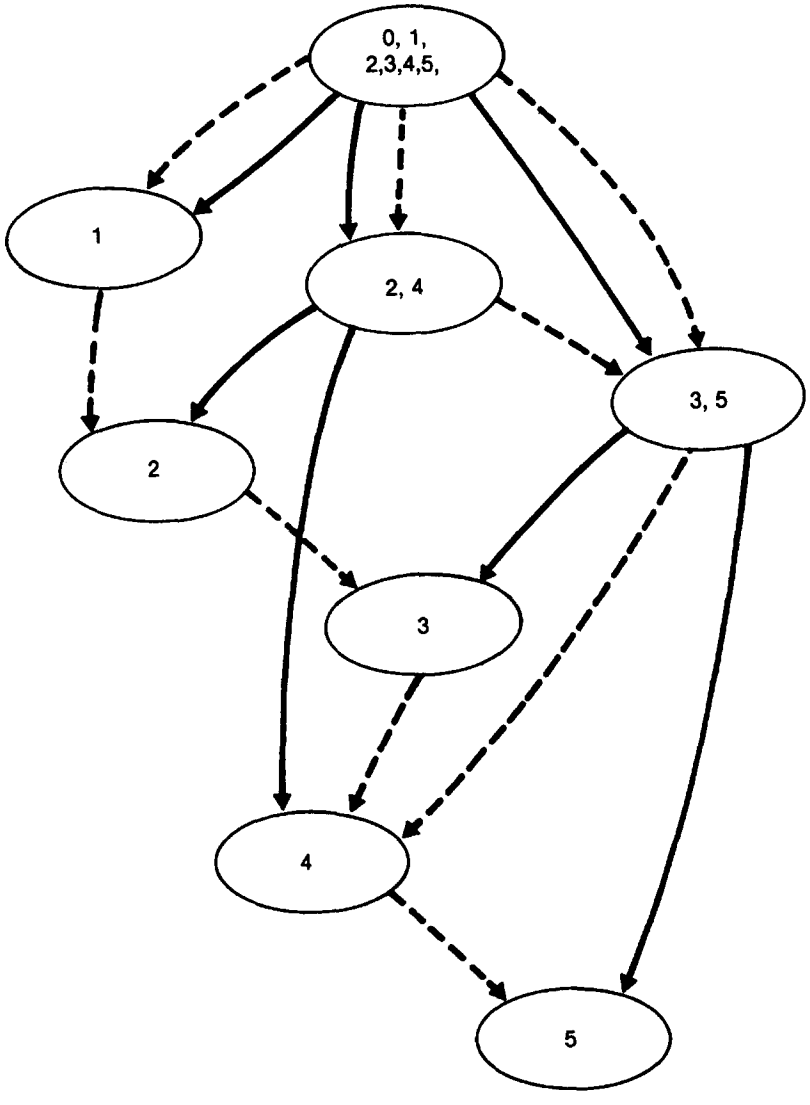

Fig. 3(a). $T(w)$ superimposed on D_w.

assigned nonempty suffixes is bounded by $|w|-1$, and so is the number of edges not in the spanning tree. □

Combining Lemmas 1.5 and 1.6, we obtain the following result.

Theorem 1.7. *For $|w|>2$, the Directed Acyclic Word Graph for w (and hence the smallest partial DFA that recognizes the set of suffixes of w) has at most $2|w|-1$ states and $3|w|-4$ transition edges.*

Proof. The bound on the number of states directly follows from Lemma 1.5. Straightforward combination of Lemmas 1.5 and 1.6 yields a slightly too weak bound of $3|w|-3$ on the number of transition edges. We noted, however, that the bound in Lemma 1.5 can be improved by at least 1 except when w is of the form ab^n, in

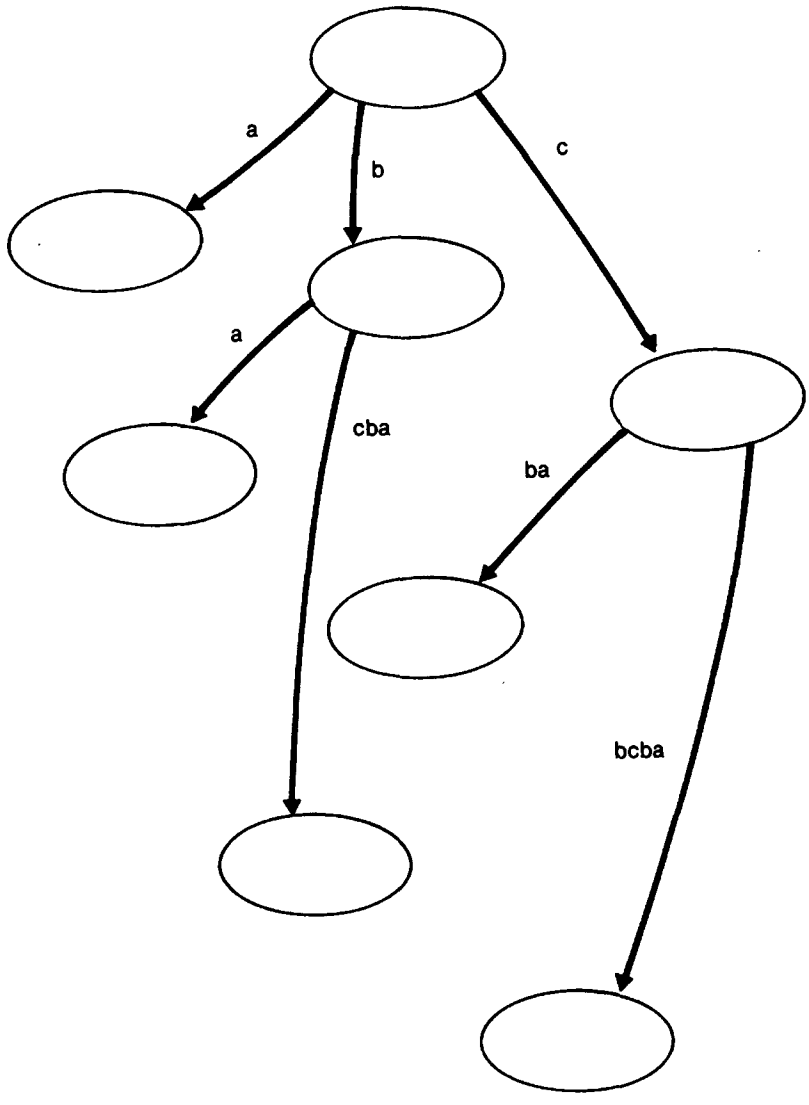

Fig. 3(b). Compact position tree for the reverse of w.

which case D_w has only $2|w|-1$ transition edges. In either case, therefore, the bound can be improved by at least 1. □

It is readily verified that our upper bound on the number of transition edges is achieved when $w = ab^nc$ for $n \geq 1$ and distinct letters a, b, and c.

2. On-line algorithm for D_w

We now consider the problem of constructing D_w in an on-line fashion, processing the letters of w from left to right. At each stage of our construction, the automaton will be correct for the prefix of w that has been processed. This also gives an algorithm to construct S_w, since the accepting states of S_w can be marked in a final step, when all of the letters of w have been processed.

The work that needs to be done for each new letter that is processed can be described by analyzing the difference between the set of nondegenerate equivalence

classes of $\equiv_w$, representing the states of D_w, and those of $\equiv_{wa}$, representing the states of D_{wa}, for an arbitrary word w and letter a. The following definitions will be needed.

Definition. *tail*(w) is the longest suffix of w that occurs more than once in w.

For example, *tail*$(abcbc) = bc$, *tail*$(aaa) = aa$, and *tail*$(aab) = \lambda$.

Definition. Let $w = w_1 y w_2$ with $w_1, w_2, y \in \Sigma^*$, $y \neq \lambda$. This occurrence of y in w is the *first occurrence of y in a new left context* if y occurs at least twice in $w_1 y$ and there exists an $a \in \Sigma$ such that every occurrence of y in $w_1 y$ except the last one is preceded by a. By convention, λ never occurs in a new left context.

For example, if $w = abcbc$, then the second occurrence of bc is the first occurrence of bc in a new left context. This is not true if $w = bcbc$, since we must have all previous occurrence of bc preceded by some letter (which must also be the same letter in all cases).

The following lemma summarizes the modifications that must be made to update the nondegenerate classes of $\equiv_w$ to those of $\equiv_{wa}$.

Lemma 2.1. (i) *wa always represents an equivalence class in* $\equiv_{wa}$, *consisting of all subwords of wa that are not subwords of w.*

(ii) *For any subword x of w, if x represents an equivalence class in* $\equiv_w$, *then x represents an equivalence class in* $\equiv_{wa}$. *The members of this class are the same in both cases, unless* $x \equiv_w$ *tail*(wa) *and tail*(wa) *appears for the first time in a new left context. In this case,* $[x]_w$ *is split into two classes in* $\equiv_{wa}$, *with words longer than tail*(wa) *remaining in* $[x]_{wa}$ *and others going into a new class* $[tail(wa)]_{wa}$, *represented by tail*(wa).

(iii) *There are no equivalence classes in* $\equiv_{wa}$ *beyond those given in* (i) *and* (ii).

Proof. (i) wa, being a prefix of itself, will always represent an equivalence class in $\equiv_{wa}$. The members of this class will be subwords of wa whose end-sets include only the last position in wa, which are exactly the new subwords of wa, not already occurring in w.

(ii) Any word that is either a prefix of w or occurs in two distinct left contexts in w will also do so in wa. Hence, if x represents an equivalence class in $\equiv_w$, then it represents an equivalence class in $\equiv_{wa}$.

We now consider the circumstances under which $[x]_w \neq [x]_{wa}$. It is clear that every word in $[x]_{wa}$ must also be in $[x]_w$, since the positions of w are a subset of those of wa. Hence, we need only consider the case when there is a $y \in [x]_w$ that fails to be in $[x]_{wa}$. Let y be the longest such word in $[x]_w$. By Lemma 1.1(ii), $x = uby$ for some $u \in \Sigma^*$ and $b \in \Sigma$. Since $y \in [x]_w$, y occurs in w and every occurrence of y in w is preceded by ub. Since we also have $y \notin [x]_{wa}$, y must occur as a suffix of wa, not preceded by ub. If by is a suffix of wa, then $by \equiv_w x$ and $by \equiv_{wa} y$, contradicting the maximality of y. Hence, cy is a suffix of wa for some letter $c \neq b$. Since cy

cannot occur in w, it follows that $y = tail(wa)$ and $tail(wa)$ appears for the first time in a new left context. Furthermore, y and all its suffixes in $[x]_w$ will occur as suffixes of wa, while words in $[x]_w$ longer than y will not. Hence, $[x]_w$ will be split into two classes in $\equiv_{wa}$, one represented by $y = tail(wa)$ containing itself and the shorter words, and the other represented by x containing the remaining words. The result follows.

(iii) By parts (i) and (ii), all of the subwords of wa have been accounted for. Hence, there can be no other equivalence classes. □

To allow efficient update of D_w to D_{wa} we annotate D_w with two additional types of information, which are maintained throughout the construction. First, each transition edge is designated as either *primary* or *secondary*. A transition edge labeled a from the class represented by x to the class represented by y is primary if $xa = y$, otherwise it is secondary. Second, each state except the source has a pointer called a *suffix pointer* that points to the parent of the state in the tree $T(w)$, introduced in the previous section. For any word x that represents an equivalence class in $\equiv_w$, the *suffix chain* starting at x, denoted $SC(x)$, is the sequence of classes that form the path from x to the root of $T(w)$. $|SC(x)|$ denotes the length of this sequence.

The following consequences of these definitions are easily verified.

Lemma 2.2. (i) *For any word x that represents an equivalence class in $\equiv_w$, $SC(x)$ partitions the suffixes of x into $|SC(x)|$ classes. In particular, if $x = w$, then the equivalence classes of all suffixes of w can be located (in order of decreasing length of suffix) by traversing the chain of suffix pointers from the sink of D_w back to the source of D_w.*

(ii) *If $w \neq \lambda$, the suffix pointer of the sink of D_w points to $[tail(w)]_w$.*

(iii) *The first class encountered by traversing the chain of suffix pointers from the sink of D_w back to the source of D_w that has an a-transition (if any) must have an a-transition to $[tail(wa)]_w$. If no a-transition is encountered, then a occurs only once in wa, and hence $tail(wa) = \lambda$.*

Thus the addition of suffix pointers allows us to locate the one class in D_w that may need to be split when updating D_w to D_{wa} (Lemma 2.1(ii)). The primary versus secondary designation of transition edges allows us to tell whether or not this class needs to be split, as demonstrated in the following.

Lemma 2.3. *Let $tail(wa) = xa$. Then x represents an equivalence class in $\equiv_w$ and $tail(wa)$ appears for the first time in a new left context if and only if there is a secondary transition edge from $[x]_w$ to $[xa]_w$ in D_w.*

Proof. Since $xa = tail(wa)$, x is a suffix of w and xa occurs in w, implying that x occurs at least twice in w. If every occurrence of x in w is preceded by the same letter b, then bxa occurs twice in wa, contradicting the maximality of $tail(wa)$. Hence, x represents an equivalence class in $\equiv_w$ by Lemma 1.1(iv). Since xa occurs

in w, there must be an a-transition edge from $[x]_w$ to $[xa]_w$. This edge is secondary if and only if xa does not represent $[xa]_w$. By Lemma 1.1(iv) this happens only when every occurrence of xa in w is preceded by the same letter, i.e., if and only if xa is occurring for the first time in a new left context as $tail(wa)$. □

The strategy of the algorithm for updating D_w to D_{wa}, along with its annotations, is to create a new state for the class represented by wa and then traverse the suffix chain from the sink of D_w putting in the necessary a-transitions to the new state until a state on the chain is found that already has an a-transition. This transition will lead to $[tail(wa)]_w$. If the transition is primary, then no more work needs to be done save the addition of a new suffix pointer from the new state to $[tail(wa)]_w$ $(=[tail(wa)]_{wa})$. If it is secondary, then $[tail(wa)]_w$ must be split into two states, and all of the transitions and suffix pointers that need to be modified can be located by examining the old transitions and suffix pointer from $[tail(wa)]_w$, and by continuing to traverse the suffix chain toward the source of D_w. A special case arises when no state with an a-transition is encountered on the suffix chain from the sink of D_w. However, in this case no split needs to be performed, and the processing is analogous to the case where there is a primary edge to $[tail(wa)]_w$.

The linear time bound for the algorithm depends critically on the fact that the suffix chain is traversed from the sink toward the source, so that states on this chain near the source which are not involved in the update are not visited. Any method which visits all states on this chain in every update (e.g., a method like that in [17] for updating position trees) would accumulate $O(n^2)$ time in the worst case (e.g., on the string a^n).

To illustrate some stages of construction, a sequence of D_w for $w = abc$, $abcb$, $abcbc$, $abcbcd$ is illustrated in Fig. 4. These are compared to the corresponding smallest automata for the same languages, discussed in the next section.

We now give a detailed description of the algorithm to build D_w, and a proof of its linear time bound. The algorithm is given below as three procedures, *builddawg*, *update*, and *split*. Procedure *builddawg* is the main procedure, which takes as input a word w, builds D_w by processing w on-line letter by letter, and returns the source. After each letter is processed, D_w and all its annotations are correct for the prefix of w up to this point. With each new letter, *builddawg* modifies the current D_w by calling the procedure *update*, giving *update* the letter to be processed and the current sink state.

Procedure *update* takes this information and in step 1 (see Algorithm A below) creates a new state, the new sink for D_{wa}, which forms the new equivalence class represented by wa (Lemma 2.1(i)). Transition edges labeled a and pointing to this state must come from all states containing suffixes of w that do not have a-transitions in D_w since this new state represents the class of strings that occur only as a suffix of wa. The primary edge from $[w]_w$ (the sink of D_w) is added in step 1, before the algorithm enters the while loop in step 3. The while loop sets *currentstate* to states containing successively shorter suffixes of w at each iteration (Lemma 2.2(i)). Thus,

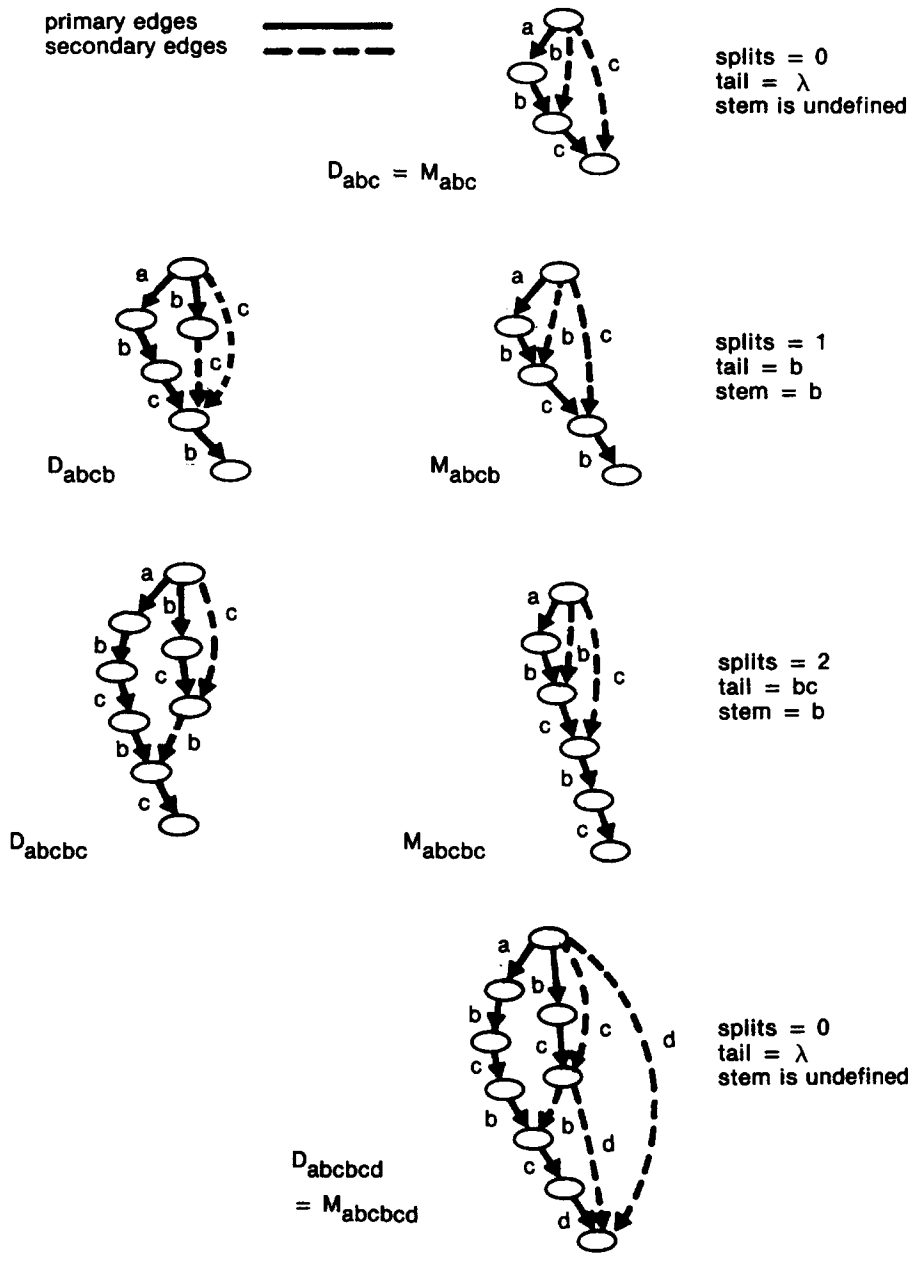

Fig. 4. Illustration of Algorithms A and B for $w = abcbcd$.

in step 3(b), case (1), the lack of an a-transition edge leads to the addition of the remaining transition edges to the new sink. Clearly, these must be secondary edges.

When the first transition edge labeled a is found from a suffix of w, this edge leads to $[tail(wa)]_w$ (Lemma 2.2(iii)). If it is primary, then no further changes need to be made to the equivalence classes of $\equiv_w$ and hence to their transition edges and suffix pointers (Lemmas 2.3 and 2.1(ii)). Step 3(b), case (2) handles this case by simply setting *suffixnode* to $[tail(wa)]_w$ $(=[tail(wa)]_{wa})$, causing a break from the while loop. If it is secondary, then $[tail(wa)]_w$ must be split. The call to the function *split* in step 3(b), case (3) handles all of the updates to the equivalence classes of $\equiv_w$ necessitated by this split (as given in Lemma 2.1(ii)), including those changes to the associated transition edges and suffix pointers, and returns a pointer to the

new equivalence class for $tail(wa)$. Thus, in either case, the variable *suffixstate* becomes set to $[tail(wa)]_{wa}$ and we break out of the while loop. A special case occurs when no state on the suffix chain has an a-transition edge. In this case the loop stops because it has reached the source, which is the equivalence class of $\lambda = tail(wa)$ (Lemma 2.2(iii)). Since λ can never occur for the first time in a new left context, no further classes or transition edges need to be modified. In step 4, *suffixstate* is set to the source, which is $[tail(wa)]_{wa}$ in this case. Finally, the suffix pointer of the new sink is set to point to $[tail(wa)]_{wa}$ (Lemma 2.2(ii)) and the new sink for the fully updated structure is returned.

Procedure *split* takes the class that contains the subword x such that $xa = tail(wa)$ (*parentstate*) and the class that contains $tail(wa)$ (*childstate*), and 'splits' *childstate*, adjusting all affected transition edges and suffix pointers. It begins in step 1 by creating *newchildstate*, which is $[tail(wa)]_{wa}$. Since *newchildstate* is represented by $tail(wa)$ and *parentstate* is represented by x, there must be a primary a-transition from *parentstate* to *newchildstate* (Lemma 2.3). This is installed in step 2. Step 3 adds the edges that come out of *newchildstate*, which clearly must be copies of the edges out of *childstate*, with the exception that they are all secondary, since *newchildstate* contains only the shorter words from $[tail(wa)]_w$ (Lemma 2.1(ii)). Steps 4 and 5 make appropriate adjustments to the suffix pointers, as is easily verified. Finally, the edges coming into *childstate* must be partitioned, so that those coming from classes whose elements are shorter than $tail(wa)$ now point to *newchildstate*. These classes clearly contain only suffixes of x, and are therefore in the suffix chain of *parentstate*. The redirection of these edges is handled in step 7, before $[tail(wa)]_{wa}$ is returned in step 8.

Algorithm A

builddawg(w)

1. Create a state named *source* and let *currentsink* be *source*.
2. For each letter a of w do:
   Let *currentsink* be *update*(*currentsink*, a).
3. Return *source*.

update(*currentsink*, a)

1. Create a state named *newsink* and a primary edge labeled a from *currentsink* to *newsink*.
2. Let *currentstate* be *currentsink* and let *suffixstate* be undefined.
3. While *currentstate* is not *source* and *suffixstate* is undefined do:
   (a) Let *currentstate* be the state pointed to by the suffix pointer of *currentstate*.
   (b) Check whether *currentstate* has an outgoing edge labeled a.
      (1) If *currentstate* does not have an outgoing edge labeled a, then create a secondary edge from *currentstate* to *newsink* labeled a.
      (2) Else, if *currentstate* has a primary outgoing edge labeled a, then let *suffixstate* be the state to which this edge leads.

(3) Else (*currentstate* has a secondary outgoing edge labeled a):
   (a) Let *childstate* be the state that the outgoing edge labeled a leads to.
   (b) Let *suffixstate* be *split*(*currentstate*, *childstate*).

4. If *suffixstate* is still undefined, let *suffixstate* be *source*.
5. Set the suffix pointer of *newsink* to point to *suffixstate* and return *newsink*.

split(*parentstate*, *childstate*)

1. Create a state called *newchildstate*.
2. Make the secondary edge from *parentstate* to *childstate* into a primary edge from *parentstate* to *newchildstate* (with the same label).
3. For every primary and secondary outgoing edge of *childstate*, create a secondary outgoing edge of *newchildstate* with the same label and leading to the same state.
4. Set the suffix pointer of *newchildstate* equal to that of *childstate*.
5. Reset the suffix pointer of *childstate* to point to *newchildstate*.
6. Let *currentstate* be *parentstate*.
7. While *currentstate* is not *source* do:
   (a) Let *currentstate* be the state pointed to by the suffix pointer of *currentstate*.
   (b) If *currentstate* has a secondary edge to *childstate*, make it a secondary edge to *newchildstate* (with the same label).
   (c) Else, break out of the while loop.
8. Return *newchildstate*.

We now establish an upper bound on the time required for Algorithm A.

Lemma 2.4. *If x represents a class in D_w with a primary a-transition edge leading to a class represented by y, then $|\mathrm{SC}(y)| = |\mathrm{SC}(x)| - k + 1$, where k is the number of secondary edges from states in $\mathrm{SC}(x)$ to states in $\mathrm{SC}(y)$.*

Proof. Since the edge from x to y is primary, $y = xa$. Since y ends in a, every class in $\mathrm{SC}(y)$ must have an incoming a-transition except the source of D_w, which lies at the end of $\mathrm{SC}(y)$. Further, for any such class on $\mathrm{SC}(y)$ all incoming a-transitions must be from classes containing suffixes of x, which in turn must lie on $\mathrm{SC}(x)$. Exactly one incoming a-transition will be primary for each class in $\mathrm{SC}(y)$, the others will be secondary. Since each class in $\mathrm{SC}(x)$ can have only one a-transition, the result follows. □

Lemma 2.5. *The execution time for Algorithm* A *is linear in the length of w for any w over a fixed finite alphabet Σ.*

Proof. We will assume that appropriate data structures are employed for states, transitions, and suffix pointers of D_w such that all of the basic operations on these structures, including creating new states and transitions, redirecting transitions, and finding transitions from given states on given letters can be accomplished in constant

time. Since the size of the alphabet is constant and the automaton is deterministic (i.e., there is at most one transition from a state for each distinct letter), this is trivial: we can use a simple linked list of transitions for each state. Thus, each of the individual steps in *update* and *split* take constant time, with the exception of the while loops.

Consider a single call to *update*(*currentsink*, a), returning *newsink*, where *currentsink* $= [w]_w$ and *newsink* $= [wa]_{wa}$. Let the total number of times the bodies of these loops are executed during this call be k, not counting the final pass that causes the exit from each loop. For each such iteration of either of these loops, a secondary edge is installed in D_{wa} from a state on the suffix chain of $[w]_{wa}$ to either $[wa]_{wa}$ or $[tail(wa)]_{wa}$, both on the suffix chain of $[wa]_{wa}$ (Lemma 2.2(i)). Hence, $|SC(wa)| \leq |SC(w)| - k + 1$ in D_{wa} by Lemma 2.4. In the special case when $[tail(wa)]_w$ lies on the suffix chain of $[w]_w$ in D_w and the update of D_w involves splitting this class, $|SC(w)|$ in D_{wa} is equal to $|SC(w)|$ in D_w plus one. Otherwise, $|SC(w)|$ is the same in D_w and D_{wa}. Thus, in any case, $|SC(wa)|$ in D_{wa} is less than or equal to $|SC(w)|$ in D_w minus k plus two.

Each time *update* is called from *builddawg*, *currentsink* $= [w]_w$ in D_w before the call and *currentsink* is set to $[wa]_{wa}$ in D_{wa} after the call, for some w and a. The suffix chain of *currentsink* has length one for the first call to *update* at the beginning of construction, never has zero length, and by the above argument can grow at most two states longer in each cell to *update*. Since the length of this chain decreases by an amount proportional to the number of iterations of the while loops in *update* and *split* on each call, this implies that the total number of iterations of these loops during the entire construction of D_w is linear in the length of w. Since all other steps of the algorithm take constant time for each letter processed, it follows that the algorithm is linear in the length of w. □

Theorem 2.6. *For any w over a fixed finite alphabet Σ, both D_w and S_w can be built in time linear in the length of w.*

Sketch of proof. The above description of Algorithm A indicates how it can be shown that this algorithm correctly constructs D_w on-line. Further details can be found in [8]. Hence, D_w can be built in linear time by Lemma 2.5. To build S_w, we simply need to mark the classes of D_w that contain suffixes of w as accepting states, letting the other states be nonaccepting. Since all of these classes lie on the suffix chain from the sink of D_w to the source (Lemma 2.2(i)), it is a simple matter to mark them, and it clearly requires time at most proportional to the length of w. □

3. The smallest automaton for the set of all subwords

We now have an algorithm that builds D_w on-line in linear time. Next, we turn our attention to the smallest partial DFA that recognizes the set of all subwords of w, which we will denote M_w.

In some cases, M_w can be considerably smaller than D_w. For example, if $w = ab^n$, $a, b \in \Sigma$, and $n \geq 1$, then D_w achieves the previously given upper bound of $2|w|-1$ states (with $2|w|-1$ edges), while M_w has only $|w|+1$ states (and only $|w|+1$ edges). On the other hand, if $w = ab^nc$, $a, b, c \in \Sigma$, and $n \geq 1$, then it is easily verified that $D_w = M_w$, and this automaton has $2|w|-2$ states and $3|w|-4$ edges as mentioned in Section 1. The following theorem asserts that this is the worst case.

Theorem 3.1. *If* $|w|>2$, M_w *has at most* $2|w|-2$ *states and* $3|w|-4$ *edges, and at least* $|w|+1$ *states and* $|w|$ *edges.*

Proof. For the upper bound, note that, by Lemma 1.5, for $|w|>2$, D_w has $2|w|-1$ states only when $w = ab^n$ for some $a, b \in \Sigma$. As mentioned above, M_w is small in this case. For all other w of length greater than 2, D_w has at most $2|w|-2$ states and $3|w|-4$ edges by Theorem 1.7, hence M_w is bounded in this manner as well.

The lower bound follows from the fact that M_w accepts a finite language and so must be acyclic. Thus, there must be at least a state for each letter in w and a start state, yielding a total of at least $|w|+1$ states. Similarly, there must be an edge for each letter in w. The string a^n is a case where this bound is tight. □

By examining the differences between D_w and M_w, we derive a way to modify Algorithm A to produce an algorithm that builds M_w again on-line in linear time. To begin, we look at how the states of M_w differ from those of D_w.

Definition. Let $\equiv'_w$ denote the canonical right invariant equivalence relation on the set of all subwords of w, i.e., $x \equiv'_w y$ if and only if, for all $z \in \Sigma^*$, xz is a subword of w if and only if yz is a subword of w. For any word x, $[x]'_w$ is the equivalence class of x with respect to $\equiv'_w$.

By Nerode's theorem [15, 20], M_w has one state corresponding to each equivalence class determined by $\equiv'_w$, with the exception of the degenerate class (which is the same as the degenerate class of $\equiv_w$). Further, since the equivalence classes determined by $\equiv_w$ are right-invariant, each equivalence class $[x]'_w$ (i.e., each state in M_w) is the union of one or more equivalence classes determined by $\equiv_w$ (i.e., the identification of one or more states in D_w). A state corresponding to an equivalence class $[x]_w$ that does not contain the longest member of $[x]'_w$ is called a *redundant state.*

The following definition and lemma give us a more precise characterization of the redundant states of D_w.

Definition. *stem*(w) is the shortest nonempty prefix of *tail*(w) that occurs (as a prefix of *tail*(w)) for the first time in a new left context. If no such prefix exists, then *stem*(w) is undefined.

For example, *stem*$(abcbc) = b$, but *stem*(aba), *stem*(abc), and *stem*$(abcdbcbc)$ are undefined.

Lemma 3.2. (i) *x represents a redundant state in D_w if and only if $stem(w)$ is defined and x is a prefix of $tail(w)$ such that $|x| \geq |stem(w)|$.*

(ii) *If $w = uxy$ where $xy = tail(w)$ and x represents a redundant state in D_w, then $x = tail(ux)$ and x occurs for the first time in a new left context as $tail(ux)$.*

(iii) *Any two distinct redundant states in D_w are contained in two distinct states in M_w. (Hence any state in M_w contains at most two states in D_w.)*

Proof. (i) '*If*' part. Let $stem(w)$ be defined and let x be a prefix of $tail(w)$ such that $|x| \geq |stem(w)|$. Clearly, x occurs as a prefix of $tail(w)$ for the first time in a new left context. Assume that every prior occurrence of x is preceded by the letter a. Since x is not always preceded by a, ax is not in $[x]_w$, and hence x represents $[x]_w$. We will show that $x \equiv'_w ax$. Assume to the contrary that there exists a $z \in \Sigma^*$ such that xz is a subword of w but axz is not. Consider the leftmost occurrence of xz in w. Let $w = u_1xzu_2$ for this occurrence. Let $w = w_1xw_2$, where $tail(w) = xw_2$. If $|u_1| < |w_1|$, then u_1 must end in a, contradicting our assumption. However, if $|u_1| \geq |w_1|$, then xzu_2 is a suffix of $tail(w)$, and thus this cannot be the leftmost occurrence of xz. This contradiction implies $x \equiv'_w ax$. It follows that $[x]_w$ is redundant.

'*Only if*' part. Let y be the longest word in $[x]'_w$. Since $[x]_w$ is redundant, $|y| > |x|$. Since $x \equiv'_w y$, for any $z \in \Sigma^*$, xz is a subword of w if and only if yz is a subword of w. It follows that the leftmost occurrence of y in w ends in the same position as the leftmost occurrence of x in w. Hence, x is a proper suffix of y, i.e., $y = uax$ for some $u \in \Sigma^*$, $a \in \Sigma$, and the leftmost occurrence of x in w is preceded by a. There must be an occurrence of x in w that is not preceded by a, otherwise $x \equiv_w ax$, contradicting the fact that x is the longest word in $[x]_w$. Consider the leftmost occurrence of x in w that is not preceded by a. Let $w = w_1xw_2$ for this occurrence. Let b be the last letter of w_1. Since xw_2 is a subword of w and $x \equiv'_w y$, yw_2 is a subword of w. Hence, axw_2 is a subword of w. It follows that xw_2 occurs at least twice in w. However, since this was the leftmost occurrence of x that was not preceded by a, it cannot be the case that bxw_2 occurs more than once in w. Thus, $xw_2 = tail(w)$ and hence x is a prefix of $tail(w)$. Further, since this was the first occurrence of x not preceded by a, x is appearing for the first time in a new left context, and so $stem(w)$ is defined and $|x| \geq |stem(w)|$.

(ii) This follows easily from (i).

(iii) Let x and y represent two distinct redundant states in D_w and assume $|y| \geq |x|$. Then, by (i), x and y are both prefixes of $tail(w)$, hence $y = xu$ for some nonempty word u. Consider the leftmost occurrence of x in w. Let $w = w_1xw_2$ for this occurrence. It is clear that xw_2 is a subword of w but yw_2 $(= xuw_2)$ cannot be a subword of w. Thus, we cannot have $x \equiv'_w y$ and hence x and y are members of two distinct states in M_w. □

By the above lemma, every redundant state in D_w can be uniquely associated with a nonempty prefix of $tail(w)$ as described above and no two redundant states

are contained in the same state in M_w. Thus, if M is the number of states in M_w and N is the number of states in D_w, then $M = N - (|tail(w)| - |stem(w)| + 1)$ when $stem(w)$ is defined, otherwise $M = N$ and hence $M_w = D_w$. Since $stem(w)$ is defined only when $tail(w)$ occurs for the first time in a new left context, there are many cases when $M_w = D_w$. One simple case is when the last letter of w is unique, since in this case $tail(w) = \lambda$.

Lemma 3.2 allows us to identify when redundant states are created by Algorithm A. Specifically, by part (ii), the conditions that lead to a redundant state in D_w are precisely those conditions that lead to the 'splitting' of an equivalence class in step 3(b), case (3) of procedure *update* in Algorithm A (Lemmas 2.1 and 2.3). Furthermore, the two states formed by the splitting of a state in D_w remain combined as one state in M_w as w grows, as long as the conditions of Lemma 3.2(i) hold, i.e., as long as the corresponding redundant state in D_w remains redundant. Thus, to modify Algorithm A so that it builds M_w we need to postpone the splitting of these states during the construction as long as the conditions of Lemma 3.2(i) hold. To do this, we need to know when the addition of a new letter a to w causes redundant states in D_w to cease to be redundant in D_{wa}. At this point, the states of M_w that represent two states of D_w must be belatedly 'split' for M_{wa} to be correct. The conditions under which redundant states of D_w cease to be redundant in D_{wa} are given by the following lemmas.

Lemma 3.3. *Assume $tail(wa) = tail(w)a$. Then if D_w contains one or more redundant states, $[tail(wa)]_{wa}$ is redundant and, for all other strings x, x represents a redundant state in D_{wa} if and only if x represents a redundant state in D_w. Otherwise, D_{wa} has at most one redundant state, that state being $[tail(wa)]_{wa}$, which is redundant if and only if $tail(wa)$ appears for the first time in a new left context.*

Proof. By Lemma 3.2, when D_w contains one or more redundant states, $stem(w)$ must be defined. In this case, redundance of $[tail(wa)]_{wa}$ and the status of all other states immediately follows from the fact that $stem(w) = stem(wa)$. When D_w does not contain any redundant states, then $stem(w)$ is undefined. Since $tail(wa) = tail(w)a$, $stem(wa)$ will also be undefined, unless $tail(wa)$ occurs for the first time in a new left context, in which case $stem(wa) = tail(wa)$. The last part of the lemma now follows. □

Lemma 3.4. *If $tail(wa) \neq tail(w)a$, then whenever x represents a redundant state in D_w, x no longer represents a redundant state in D_{wa}. In this case, D_{wa} always has at most one redundant state, that state being $[tail(wa)]_{wa}$, which is redundant if and only if $tail(wa)$ appears for the first time in a new left context.*

Proof. Since $tail(wa) \neq tail(w)a$, $tail(wa)$ must be a suffix of $tail(w)a$, not beginning at the first letter of $tail(w)a$. Hence, any prefix of $tail(wa)$ except $tail(wa)$ itself is a subword of $tail(w)$ with an occurrence in $tail(w)$ not beginning at the first letter

of $tail(w)$. Since there is a previous occurrence of $tail(w)$ in w, any proper prefix of $tail(wa)$ has appeared before in the same left context as $tail(wa)$, and cannot be occurring for the first time in a new left context. Hence, by Lemma 3.2, D_{wa} has no redundant states unless $tail(wa)$ appears for the first time in a new left context, in which case $tail(wa)$ represents the only redundant state in D_{wa}. Since $tail(wa)$ starts in a later position than $tail(w)$, $tail(wa)$ cannot appear for the first time in a new left context both as a suffix of wa and as a prefix of $tail(w)$. Thus, $tail(wa)$ cannot represent a redundant state in both D_w and D_{wa}. Hence, whenever x represents a redundant state in D_w, x no longer represents a redundant state in D_{wa}. □

The on-line algorithm for D_w is easily modified to give an on-line algorithm for M_w by waiting to 'split' the redundant states until the point when they cease to be redundant. Lemmas 3.3 and 3.4 tell us that the only time when the redundant states of D_w cease to be redundant is when $tail(w)a \neq tail(wa)$. This condition can be recognized by the fact that $[tail(w)]_w$ has no a-transition, which is checked during the first iteration of the while loop in step 3 of procedure *update* in Algorithm A. At the point when $tail(wa) \neq tail(w)a$, all redundant states of D_w cease to be redundant, and the refinement of the equivalence classes of D_w that was not done for M_w must now be implemented for M_{wa}. This requires a little bookkeeping in order to save the information needed to create new states until when they cease to be redundant.

We introduce several global variables to Algorithm A to perform these bookkeeping functions. The first, *splits*, refers to the number of prefixes of $tail(w)$ that are of length equal to or greater than the length of $stem(w)$; that is, the number of redundant states in the corresponding D_w. The variable *parent* refers to the state that will have a primary transition edge leading to the newly created state $[stem(w)]$ when it ceases to be redundant. Queue *children* consists of the states in M_w that contain redundant states in D_w; that is, the states that will belatedly be 'split' when these states in D_w cease to be redundant. Queue *oldsuffix* contains states that were *newsink* at the time when the corresponding state in *children* became redundant. It is necessary to keep track of these, because when those states 'split', the suffix pointers will need to be readjusted to point to the newly split off states.

We now give a description of Algorithm B, the algorithm to build M_w (see below). Some stages of construction for $w = abcbcd$ are illustrated in Fig. 4 and contrasted with the corresponding stages of Algorithm A. Like Algorithm A, Algorithm B is composed of three procedures, *buildma*, *update*, and *split*. *Buildma* is identical to *builddawg*, except that it has an additional step to initialize the global bookkeeping variables *splits*, *children*, and *oldsuffix*. *Split* is also identical to the corresponding procedure in Algorithm A, except that it sets one additional suffix pointer. This is the suffix pointer from the first state popped off the queue *oldsuffix*. It corresponds to the suffix pointer that is set in step 5 in the procedure *update* in Algorithm A.

The version of *update* in Algorithm B incorporates the most significant changes from Algorithm A. Like the version of *update* in Algorithm A, *update* takes the

current sink state ($[w]'_w$) and the letter a, and creates the new sink state for M_{wa}. As in Algorithm A, the states on the suffix chain of $[w]'_w$ are traversed and an a-transition to the new sink state is added from each state encountered that does not already have an a-transition. This occurs in step 3(b), case (1).

When the first state on the suffix chain is reached, if case (1) of step 3(b) is encountered, some additional processing takes place. This first state is $[tail(w)]'_w$ so lack of an a-transition from this state indicates that $tail(w)a$ is not a subword of w, and hence $tail(w)a \neq tail(wa)$. Thus, by Lemma 3.4, all redundant states in D_w cease to be redundant and the corresponding states in M_w must be belatedly split. In part (a), all the states containing redundant states from D_w are refined using the subroutine *split*. The variable *splits* is used to count them, the queue *children* is used to locate the states that must be refined, and the queue *oldsuffix* is used to locate states whose suffix pointers must be readjusted. After this operation, the variable *splits* is set to zero, indicating that there are no states of M_w that contain redundant states of D_w. At this point, the structure is essentially the same as the partially updated structure for D_w that would be present at the corresponding point in step 3(b), case (1) of Algorithm A. This special processing is inhibited when case (1) arises for subsequent states in the suffix chain of $[w]'_w$ since *splits* is set to zero.

When a state with an a-transition is found, this transition will lead to $[tail(wa)]'_{wa}$. It then becomes necessary to determine whether this state contains a redundant state from the corresponding D_{wa}. This is done in step 3(b), cases (2) and (3), and is also analogous to the processing performed in the corresponding steps in Algorithm A.

In step 3(b), case (2), *update* checks to see if it is the case that $[tail(wa)]'_{wa}$ does not contain a redundant state from D_{wa}. This is indicated by *splits* having a zero value, and the a-transition to $[tail(wa)]'_{wa}$ being primary. When *splits* is zero, this criterion reduces to the same one applied in the corresponding step of Algorithm A to check if $[tail(wa)]_w$ does not need to be split. When *splits* is nonzero at this point in *update*, it must be the case that $tail(wa) = tail(w)a$. Otherwise, the special processing described above would have occurred on the first iteration of the while loop that traverses the suffix chain, setting *splits* to zero. Since a value of one or more for *splits* also implies that there is at least one redundant state in D_w, $[tail(wa)]_{wa}$ is redundant in this case by Lemma 3.3. Hence, $[tail(wa)]'_{wa}$ always contains a redundant state of D_{wa} when *splits* is greater than zero.

In step 3(b), case (3), *update* handles the case when $[tail(wa)]'_w$ does contain a redundant state from D_{wa}. This is indicated by a nonzero value of splits, or by a secondary edge, i.e., the negation of the above condition for case (2). In this case, *splits* is incremented to reflect the addition of a new state in M_{wa} that contains a redundant state in D_{wa}. The appropriate states are added to *oldsuffix* and to *children* for later use when the redundant states may cease to be redundant. If $stem(w)$ is undefined, then $stem(wa)$ is $tail(wa)$ and *parent* is given the value $[tail(wa)]'_{wa}$.

Finally, in step 5, *update* sets the suffix pointer of the new sink for M_{wa} to point to $[tail(wa)]'_{wa}$ and returns the new sink state it creates.

A description of Algorithm B in pseudocode is given below. Note that the variables *source*, *children*, *parent*, *oldsuffix*, and *splits* are global to all three procedures.

Algorithm B

buildma(*w*)

1. Initialize the global queues *children* and *oldsuffix* to be empty, and set the value of *splits* to 0.
2. Create a state named *source* and let *currentsink* be *source*.
3. For each letter *a* of *w* do:
   Let *currentsink* be *update*(*currentsink*, *a*).
4. Return *source*.

update(*currentsink*, *a*)

1. Create a state named *newsink* and a primary edge labeled *a* from *currentsink* to *newsink*.
2. Let *currentstate* be *currentsink* and let *suffixstate* be undefined.
3. While *currentstate* is not *source* and *suffixstate* is undefined do:
   (a) Let *currentstate* be the state pointed to by the suffix pointer of *currentstate*.
   (b) Check whether *currentstate* has an outgoing edge labeled *a*.
      (1) If *currentstate* does not have an outgoing edge labeled *a*, then:
         (a) For $i = 1$ to *splits*, remove *topchild* and *topsuffix* from the front of the queues *children* and *oldsuffix* respectively, and let *parent* be *split*(*parent*, *topchild*, *topsuffix*).
         (b) If the 'for' loop above was executed, let *currentstate* be *parent* and set $splits = 0$.
         (c) Create a secondary edge from *currentstate* to *newsink* labeled *a*.
      (2) Else, if *splits* is 0 and *currentstate* has a primary outgoing edge labeled *a*, then let *suffixstate* be the state to which this edge leads.
      (3) Else ($splits > 0$ or *currentstate* has a secondary outgoing edge labeled *a*).
         (a) Let *suffixstate* be the state that the outgoing edge labeled *a* leads to.
         (b) Increment the value of *splits*.
         (c) If *splits* is 1, let *parent* be *currentstate*.
         (d) Add *suffixstate* to the end of the queue *children* and add *newsink* to the end of the queue *oldsuffix*.
4. If *suffixstate* is still undefined, let *suffixstate* be *source*.
5. Set the suffix pointer of *newsink* to point to *suffixstate* and return *newsink*.

split(*parentstate*, *childstate*, *oldsuffixstate*)

1. Create a state called *newchildstate*.
2. Make the secondary edge from *parentstate* to *childstate* into a primary edge from *parentstate* to *newchildstate* (with the same label).

3. For every primary and secondary outgoing edge of *childstate*, create a secondary outgoing edge of *newchildstate* with the same label and leading to the same state.
4. Set the suffix pointer of *newchildstate* equal to that of *childstate*.
5. Reset the suffix pointer of *childstate* to point to *newchildstate*.
6. Reset the suffix pointer of *oldsuffixstate* to point to *newchildstate*.
7. Let *currentstate* be *parentstate*.
8. While *currentstate* is not *source* do:
   (a) Let *currentstate* be the state pointed to by the suffix pointer of *currentstate*.
   (b) If *currentstate* has a secondary edge to *childstate*, make it a secondary edge to *newchildstate* (with the same label).
   (c) Else, break out of the while loop.
9. Return *newchildstate*.

Theorem 3.5. *The smallest DFA for the set of all subwords of a word can be built on-line in linear time.*

Sketch of proof. The above description of Algorithm B indicates how it can be shown that this algorithm correctly constructs M_w on-line. Except for bookkeeping functions, every operation performed by Algorithm B is an operation performed for the corresponding word by Algorithm A, although these operations are often performed in a different order. The linear time bound follows from the fact that the only new operations performed by Algorithm B are those involved with the extra bookkeeping. This bookkeeping is bounded by the number of splits that are delayed in Algorithm B, which must be linear since Algorithm A is linear (Lemma 2.5). □

4. Further research

One possible direction for further research is to try to find an on-line linear-time algorithm that builds the smallest partial DFA for the subwords of a finite set of words. A natural extension of the DAWG construction algorithm to finite sets of words is given in [7]. This gives a partial DFA for the subwords of the finite set, with size bounds similar to those given in Theorem 1.7. However, the relationship between the DAWG and the smallest partial DFA is more problematic in this case, and there are no obvious modifications that would allow it to construct the smallest DFA on-line in linear time.

Continuing further along this line, we might consider constructing the smallest DFA for the subwords of an arbitrary regular language, perhaps given as a partial DFA itself. However, here we run into a roadblock, because it can be the case that the regular language has a DFA of size n, but the smallest DFA for the subwords of the language has $O(2^n)$ states. A simple example is the language $(a+b)^n a(a+b)^n c$, where a, b, and c are distinct letters. Here, every word of length n over the letters a and b falls into a different equivalence class with respect to the canonical right

invariant equivalence relation for the subwords of the language, so the smallest DFA for the subwords has at least 2^n states, while the DFA for the language itself has only $2n+3$ states.

Other lines of research relate to implementations of the algorithms given here. Problems arise when building DAWGs or smallest DFAs for very large texts due to the size of the data structure for the automaton, which may be many times larger than will fit in the main memory of the machine. Here we would like two things. The first is a good estimate of how large the automaton is expected to be for a given size text. We have some results of this type for random texts in the case when all letters are independent and equiprobable, which we hope to present in a future paper. The second thing is a good method for dealing with the disk 'thrashing' problem when the automaton is too large to fit in main memory and must be built using secondary storage. This is a problem that plagues algorithms for compact position trees and their relatives as well (see [17]). As yet we have made no progress in this direction.

Finally, one might consider applications of these automata to other text processing problems beyond the simple string search mentioned in the introduction. Several possibilities along these lines are briefly discussed in [7].

Acknowledgment

We would like to thank Ross McConnell for much help with the version of this paper presented at ICALP-84 and for helpful discussions and programming efforts in the early stages of this investigation. We would also like to thank Hermann Maurer for his comments on [6], which led us to look at smallest automata for the subwords of a word.

References

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

[2] A. Apostolico, Some linear time algorithms for string statistics problems, *Publication Series III*, **176** (IAC, Rome, 1979).

[3] A. Apostolico, Fast applications of suffix trees, in: D.G. Lainiotis and N.S. Tzannes, eds., *Advances in Control* (Reidel, Hingham, MA, 1980) 558–567.

[4] A. Apostolico and F.P. Preparata, Optimal off-line detection of repetitions in a string, *Theoret. Comput. Sci.* **22** (1983) 297–315.

[5] A. Apostolico, The myriad virtues of suffix trees, *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, Maratea, Italy, 1984.

[6] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler and R. McConnell, Linear size finite automata for the set of all subwords of a word: An outline of results, *Bull. Europ. Assoc. Theoret. Comput. Sci.* **21** (1983) 12–20.

[7] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler and R. McConnell, Building a complete inverted file for a set of text files in linear time, *Proc. 16th ACM Symp. on Theory of Computing* (1984) 349–358.

[8] J. Blumer, Correctness and linearity of the on-line directed acyclic word graph algorithm, Tech. Rept. MS-R-8410, Univ. of Denver, Dept. of Mathematics and Computer Science, 1984.
[9] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (10) (1977) 762–772.
[10] M.T. Chen and J. Seiferas, Efficient and elegant subword-tree construction, C.S. and C.E. Res. Rev., Univ. of Rochester (1983/84) 10–14.
[11] M.T. Chen and J. Seiferas, Efficient and elegant subword-tree construction, *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, Maratea, Italy, 1984.
[12] M. Crochemore, Optimal factor transducers, *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, Maratea, Italy, 1984.
[13] M. Crochemore, Linear searching for a square in a word, Presented at: *11th Internat. Colloq. on Automata, Languages, and Programming*, Antwerp, Belgium, 1984.
[14] M. Crochemore, Personal communication, 1984.
[15] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).
[16] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.
[17] M.E. Majster and A. Reiser, Efficient on-line construction and correction of position trees, *SIAM J. Comput.* **9** (4) (1980) 785–807.
[18] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM* **23** (2) (1976) 262–272.
[19] D.R. Morrison, PATRICIA—practical algorithm to retrieve information coded in alphanumeric, *J. ACM* **15** (4) (1968) 514–534.
[20] A. Nerode, Linear automaton transformations, *Proc. Amer. Math. Soc.* **9** (1958) 541–544.
[21] V.R. Pratt, Improvements and applications for the Weiner repetition finder, Unpublished manuscript, 1975.
[22] M. Rodeh, V.R. Pratt and S. Even, Linear algorithm for data compression via string matching, *J. ACM* **28** (1) (1981) 16–24.
[23] A.O. Slisenko, String-matching in real time, Preprint P-7-77, The Steklov Institute of Mathematics, Leningrad Branch, 1977 (in Russian).
[24] A.O. Slisenko, String matching in real time: Some properties of the data structure, *Proc. 7th Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **64** (Springer, Berlin, 1978) 493–496.
[25] A.O. Slisenko, Determination in real time of all the periodicities in a word, *Sov. Math. Dokl.* **21** (2) (1980) 392–395.
[26] A.O. Slisenko, Detection of periodicities and string matching in real time, *J. Sov. Math.* **22** (3) (1983) 1316–1387; translated from Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. *V.A. Steklov AN SSSR* **105** (1980) 62–173.
[27] S.L. Tanimoto, A method for detecting structure in polygons, *Pattern Recognition* **13** (6) (1981) 389–394.
[28] P. Weiner, Linear pattern matching algorithms, *IEEE 14th Ann. Symp. on Switching and Automata Theory* (1973) 1–11.