# Complete Inverted Files for Efficient Text Retrieval and Analysis

A. BLUMER, J. BLUMER, D. HAUSSLER, AND R. McCONNELL

*University of Denver, Denver, Colorado*

AND

A. EHRENFEUCHT

*University of Colorado at Boulder, Boulder, Colorado*

Abstract. Given a finite set of texts $S = \{w_1, \ldots, w_k\}$ over some fixed finite alphabet $\Sigma$, a complete inverted file for $S$ is an abstract data type that provides the functions *find*($w$), which returns the longest prefix of $w$ that occurs (as a subword of a word) in $S$; *freq*($w$), which returns the number of times $w$ occurs in $S$; and *locations*($w$), which returns the set of positions where $w$ occurs in $S$. A data structure that implements a complete inverted file for $S$ that occupies linear space and can be built in linear time, using the uniform-cost RAM model, is given. Using this data structure, the time for each of the above query functions is optimal. To accomplish this, techniques from the theory of finite automata and the work on suffix trees are used to build a deterministic finite automaton that recognizes the set of all subwords of the set $S$. This automaton is then annotated with additional information and compacted to facilitate the desired query functions. The result is a data structure that is smaller and more flexible than the suffix tree.

Categories and Subject Descriptors: E.1 **[Data Structures]**: Graphs; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*pattern matching*; H.3.1 **[Information Storage and Retrieval]**: Content Analysis and Indexing—*indexing methods*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: DAWG, inverted file, string matching, suffix tree, text retrieval

## 1. *Introduction*

The notion of an inverted file for a textual database is common in the literature on information retrieval, but precise definitions of this concept vary [9, 13, 23]. We propose the following definition: Given a finite alphabet $\Sigma$, a set of keywords $K \subseteq \Sigma^+$, and a finite set of text words $S \subseteq \Sigma^+$, an *inverted file* for $(\Sigma, K, S)$ is an

The research of A. Blumer and D. Haussler was supported by National Science Foundation grant IST 83-17918 and the research of A. Ehrenfeucht was supported by National Science Foundation grant MCS 83-05245.

Authors' present addresses: A. Blumer, Department of Computer Science, Tufts University, Medford, MA 02155; J. Blumer and D. Haussler, Department of Computer and Information Science, University of California at Santa Cruz, Santa Cruz, CA 95064; A. Ehrenfeucht, Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80302; R. McConnell, Department of Mathematics and Computer Science, University of Denver, Denver, CO 80208.

abstract data type that implements the following functions:

(1) *find*: $\Sigma^+ \rightarrow K \cup \{\lambda\}$, where *find*$(w)$ is the longest prefix $x$ of $w$ such that $x \in K \cup \{\lambda\}$ and $x$ occurs in $S$, that is, $x$ is a subword of a text in $S$.
(2) *freq*: $K \rightarrow \mathbf{N}$, where *freq*$(w)$ is the number of times $w$ occurs as a subword of the texts in $S$.
(3) *locations*: $K \rightarrow 2^{\mathbf{N}\times\mathbf{N}}$, where *locations*$(w)$ is a set of ordered pairs giving the text numbers and positions within the texts in which $w$ occurs.

This definition is simple, and includes many of the essential features of the notions of inverted files in the literature, as well as those of other more general "content addressable" data structures [14, 16].

In this paper, we consider the problem of constructing a *complete inverted file* for a finite set $S$. This is an inverted file for $S$ in which the set of keywords $K$ is *sub*$(S)$, that is, every substring of $S$ is a keyword. We describe a data structure that implements a complete inverted file for $S$ that occupies space linear in the size of $S$, can be built in linear time, and is structured such that each of the query functions takes optimal time. Specifically, the time for *find*$(w)$ is proportional to the length of the output string $x$, the time for *freq*$(w)$ is proportional to the length of $w$, and the time for *locations*$(w)$ is proportional to the length of $w$ plus the number of locations retrieved. Time and space bounds are given using the uniform cost RAM model of computation [1].

The existence of a data structure with these properties is not hard to demonstrate. It is essentially implied in the early work of Morrison on PATRICIA trees [18], which are further refined to the compact position trees of Weiner [24] and the suffix trees of McCreight [2, 17]; see also [21]. Our primary contribution is a new set of basic data structures that are functionally superior to suffix and position trees, and are smaller. Using methods based on the theory of finite automata, we replace the trees used in earlier work with more compact acyclic graphs derived from the directed acyclic word graph (DAWG) [8] (see Figures 1 and 4). These data structures are currently used in programs for analyzing and finding patterns in DNA sequences [11], calculating distances between strings and clustering using the string metric defined in [12], and as part of the user interface and "name generating" software for a set of programs that provide computer-aided instruction in assembly tasks [3].

In Section 2 we describe the basic data structure that we build to implement a complete inverted file, give space bounds for this structure, and indicate how the retrieval functions are implemented using it. Sections 3–5 are devoted to demonstrating that this data structure can be built in linear time, assuming that the size of the alphabet is constant. In Sections 3 and 4 we extend the definition and on-line construction algorithm of the DAWG to apply to a finite set of texts and give linear size bounds on this structure. Section 5 shows how to add a linear post-processing phase to this algorithm to create the final data structure that implements the inverted file. This graph is essentially a labeled suffix tree with isomorphic subtrees collapsed. Section 6 introduces a symmetric version of this data structure with edges that allow one to extend a query by adding letters either before the first letter or after the last letter.

Throughout this paper $\Sigma$ denotes an arbitrary nonempty finite alphabet and $\Sigma^*$ denotes the set of all strings over $\Sigma$. The empty word is denoted by $\lambda$. $\Sigma^+$ denotes $\Sigma^* - \{\lambda\}$. $S$ always denotes a finite subset of $\Sigma^+$. For any $w \in \Sigma^*$, $|w|$ denotes the length of $w$. $\|S\| = \Sigma_{w\in S} |w|$. 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 $w$. $sub(S)$ denotes the union of all subwords of the members of $S$. For $w \in \Sigma^*$, $|w| = n$, $w$ has $n + 1$ *positions*, numbered from 0 to $n$. Position 0 is at the beginning of $w$, before the first letter, and subsequent positions follow the corresponding letters of $w$. For $S = \{w_1, \ldots, w_k\}$, $S$ has $\|S\| + k$ positions, each denoted by a pair $\langle i, j \rangle$ where $1 \leq i \leq k$ and $j$ is a position in $w_i$. For any $x \in sub(S)$, $endpos_S(x)$ denotes the set of all positions $\langle i, j \rangle$ in $S$ immediately following occurrences of $x$ and $beginpos_S(x)$ denotes the set of all positions immediately preceding occurrences of $x$. For $x \notin sub(S)$, $beginpos_S(x) = endpos_S(x) = \varnothing$.

Given a directed acyclic graph $G$ with edges labeled from $\Sigma^+$, a *path* $p$ in $G$ is a sequence of nodes connected by edges, or just a single node. For any edge $e$ in $G$, $label(e)$ denotes the label of $e$. $label(p)$ is the word obtained by concatenating the labels of the edges along $p$. $label(p) = \lambda$ if $p$ is a single node.

## 2. *Implementing a Complete Inverted File*

We begin by describing the basic data structure used to implement a complete inverted file for a set of texts $S$.

*Definition.* A *rule* of $S$ is a "production" $x \rightarrow_S \gamma x \beta$ where $x \in sub(S)$, $\gamma$, $\beta \in \Sigma^*$, and every time $x$ occurs in $S$, it is preceded by $\gamma$ and followed by $\beta$. $x \rightarrow_S \gamma x \beta$ is a *prime rule* of $S$ if it is a rule of $S$ and $\gamma$ and $\beta$ are as long as possible (i.e., for no $\delta$, $\tau \in \Sigma^*$, where $\delta\tau \neq \lambda$ is $x \rightarrow_S \delta\gamma x\beta\tau$ a rule of $S$). If $x \rightarrow_S \gamma x \beta$ is a prime rule of $S$, then $\gamma x \beta$ is called the *implication* of $x$ in $S$, denoted $imp_S(x)$. $P(S) = \{imp_S(x): x \in sub(S)\}$. The members of $P(S)$ are called the *prime* subwords of $S$.

For example, if $S = \{ababc, abcab\}$, then $imp_S(\lambda) = \lambda$, $imp_S(a) = imp_S(b) = ab$, $imp_S(ca) = abcab$ and $P(S) = \{\lambda, ab, abc, ababc, abcab\}$. (It is not true in general that the words of $S$ can be formed by concatenating shorter prime subwords of $S$.)

*Definition.* The *compact DAWG* of $S$ is the directed graph $C_S = (V, E)$ with edges labeled with words in $\Sigma^+$, where $V = P(S)$ and $E = \{(x, imp_S(xa)): a \in \Sigma, xa \in sub(S)\}$. $label((x, \gamma x a \beta)) = a\beta$. The node $\lambda$ is called the *source* of $C_S$.

The compact DAWG for $S = \{ababc, abcab\}$ is given in Figure 1. For comparison, the suffix tree [17] for $S' = \{ababc\$_1, abcab\$_2\}$ is given in Figure 2. Here we assume the natural extension of McCreight's data structure from individual words to sets of words, using unique endmarkers. In the proof of Theorem 1 below, we show that the compact DAWG will always be smaller than the corresponding suffix tree. In fact, the compact DAWG for $S$ can be obtained from the suffix tree for $S'$ by merging isomorphic (edge labeled) subtrees in this structure and deleting the structure associated with the endmarkers.[1] The relevant nodes of the suffix tree are indicated by the solid circles. We elaborate further on the connection between the suffix tree and the compact DAWG in Section 3.

The fundamental properties of $C_S$ are given in the following lemma:

LEMMA 1. *For any $x \in sub(S)$ there is a single path $p$ in $C_S$ from the source to $imp_S(x)$ such that $x$ is a prefix of $label(p)$ and $x$ is not a prefix of the label of any proper initial segment of $p$. Conversely, for any path $p$ from the source to $imp_S(x)$, $label(p)$ is a suffix of $imp_S(x)$ and for any prefix $y$ of $label(p)$ that is not a prefix of the label of a proper initial segment of $p$, $imp_S(y) = imp_S(x)$.*

[1] A similar observation was made for the DAWG (see Section 3) in [10].

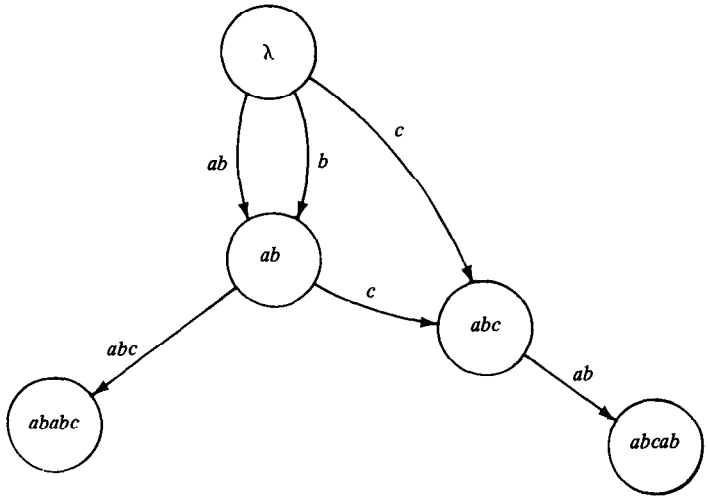


FIG. 1. Compact DAWG $C_S$, for $S = \{ababc, abcab\}$.

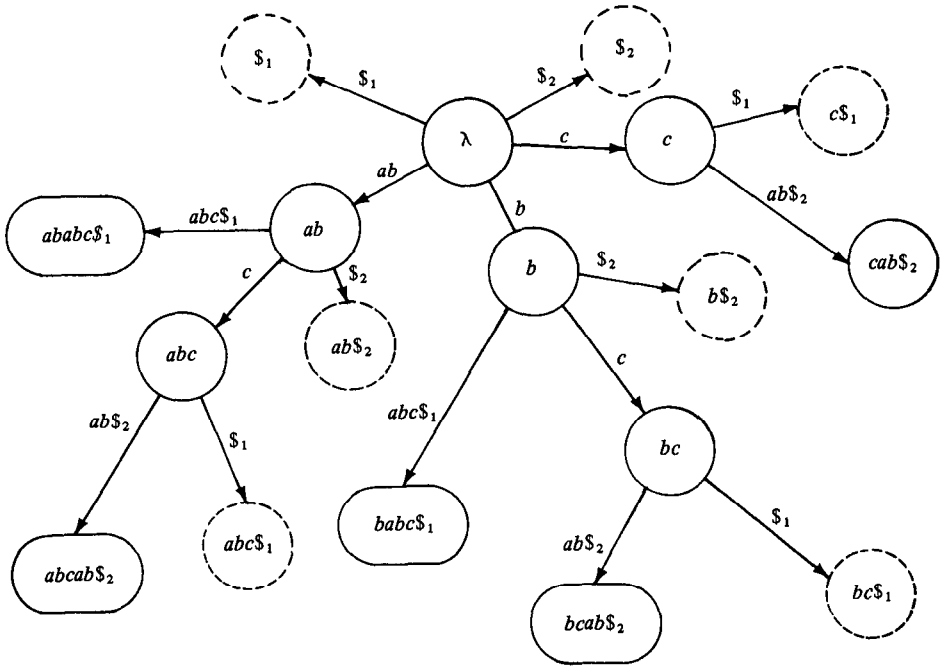


FIG. 2. Suffix tree for $S' = \{ababc\$_1, abcab\$_2\}$.

PROOF. These properties follow easily from the definition of $C_S$. □

Informally, this lemma says that for any $x, y \in sub(S)$, $x$ and $y$ "lead to the same node" in $C_S$ if and only if $imp_S(x) = imp_S(y)$. Thus the nodes of $C_S$ represent the equivalence class of sub($S$) with respect to the following equivalence relation on $\Sigma^*$.

*Definition.* For $x, y \in sub(S)$, $x \equiv_S y$ if and only if $imp_S(x) = imp_S(y)$. If $x \notin sub(S)$ there are two cases: if $y \notin sub(S)$, then $x \equiv_S y$; otherwise, $x \not\equiv_S y$.

Each equivalence class in $sub(S)$ is uniquely represented by a prime subword of $S$ that is the implication of all other words in the class. Thus all the words in any one of these equivalence classes occur with the same frequency and in roughly the same locations in $S$.

A complete inverted file for a finite set of texts $S$ is implemented using the compact DAWG, $C_S$, annotated with certain additional information to facilitate the retrieval functions.

*Definition.* The *labeled* compact DAWG is the graph $I_S$, obtained from $C_S$ by adding the following labeling to each node $x$.

(1) A *frequency label*, that is, an integer indicating the number of times that $x$ occurs in $S$.
(2) A (possibly empty) list of *identification pointers* indicating all texts of $S$ of which $x$ is a suffix.

The labeled compact DAWG for $S = \{ababc, abcab\}$ is given in Figure 3. Note that the identification pointers play a role analogous to the endmarkers in the suffix tree. It is not generally the case that each node has a nonempty list of identification pointers.

In the actual implementation of $I_S$, we assume that the texts of $S$ are stored in RAM, and that the strings labeling the edges of $I_S$ are each given by a pointer to an occurrence of the string in $S$ and a length. Optionally, the nodes of $I_S$ may also be labeled with the prime subwords that represent them in the same manner. This allows some additional query functions (discussed below) to be implemented on $I_S$. Using the uniform-cost RAM model, the space required for each of these labels is constant, as is the size of each frequency label and identification pointer. The overall size bounds for $I_S$ are given in the following theorem:

THEOREM 1. *Let $I_S = (V, E)$. Let $k$ be the number of words in $S$ and let $m$ be the number of identification pointers in $I_S$. Then $|V| \leq \|S\| + k$ and $|E| + m \leq 2(\|S\| + k) - 1$.*

PROOF. This bound is related to the well-known bounds for suffix trees [17]. Assume $S = \{w_1, \ldots, w_k\}$ and let $T$ be the suffix tree for $S' = \{w_1\$_1, \ldots, w_k\$_k\}$. $T$ will have one leaf for each nonempty suffix in $S'$, for a total of $\|S\| + k$ leaves. Let $n(T)$ denote the set of all nodes of $T$ and let $I$ be the number of internal nodes. Then the total number of nodes in $T$ can be written as

$$I + \|S\| + k = \left( \sum_{n \in n(T)} deg(n) \right) + 1, \tag{1}$$

where $deg(n)$ denotes the outdegree of the node $n$. Every internal node of $T$ has outdegree at least 2. Furthermore, since the words in $S$ are nonempty, there are at least $k + 1$ distinct letters occurring in the words of $S'$; hence the outdegree of the root of $T$ is at least $k + 1$. Thus,

$$\left( \sum_{n \in n(T)} deg(n) \right) + 1 \geq k + 1 + 2(I - 1) + 1 = 2I + k. \tag{2}$$

It follows from (1) and (2) that $I \leq \|S\|$.

We use these bounds on the number of internal and leaf nodes of $T$ to obtain bounds on the size of $I_S$. Suppose that $x$ is a prime subword of $S$. If $x$ occurs two or more times in $S$, then $x$ occurs two or more times in $S'$, and since it is a prime

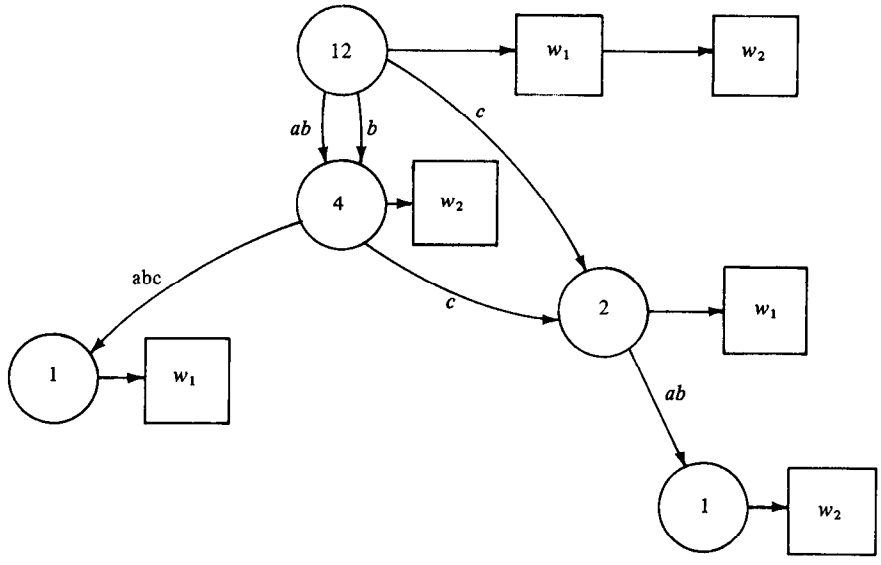


FIG. 3. $I_S$ for $S = \{w_1, w_2\}$, where $w_1 = ababc$, $w_2 = abcab$.

subword of $S$, there are at least two distinct letters that follow occurrences of $x$ in $S'$. This implies that there is a path in $T$ leading from the root to an internal node that is labeled with the letters of $x$. Since each internal node of $T$ is uniquely associated with a subword of $S$ in this manner, this implies that the number of prime subwords of $S$ that occur more than once in $S$ is bounded by the number of internal nodes of $T$, which is at most $\| S \|$. If $x$ is a prime subword of $S$ that occurs only once in $S$, then $x$ must actually be a word in $S$. Thus since $S$ has only $k$ words, the total number of prime subwords of $S$, and hence the number of nodes in $I_S$, is bounded by $\| S \| + k$.

Now assume that $(x, \gamma x a \beta)$ is an edge in $I_S$. Again, this implies that $x$ is a prime subword in $S$ that occurs followed by at least two distinct letters in $S'$ (one of them the letter $a$). Thus there will be an internal node in $T$ corresponding to the word $x$ and this node will have an outgoing edge with a label beginning with $a$. Now suppose instead (or in addition) that $x$ has an identification pointer that points to the word $w_i \in S$. Thus $x$ is a suffix of $w_i$. If $x$ occurs more than once in $S$ then $x$ occurs followed by $\$_i$ and at least one other letter $b \neq \$_i$ in $S'$. Hence again there is an outgoing edge (this time labeled $\$_i$) in $T$ from an internal node corresponding to $x$. As above, each edge in $T$ can correspond to at most one edge or identification pointer in $I_S$. This accounts for all but at most $k$ identification pointers, which are the identification pointers for prime subwords of $S$ that occur only once in $S$. There can be at most $k$ of these. Since the total number of nodes in $T$ is at most $2\| S \| + k$, the number of edges in $T$ is at most $2\| S \| + k - 1$. Thus the number of edges and identification pointers in $I_S$ is bounded by $2(\| S \| + k) - 1$. □

A simple example of a sequence of text sets $S_n$ for which $I_S$ reaches these upper bounds is $S_n = \{a^n\}$ where $a$ is a single letter. Preliminary experimental evidence indicates expected sizes of 0.26 to 0.29 $\| S \|$ nodes and 0.9 to 1.0 $\| S \|$ edges and identification pointers for $I_S$ when $S$ is a single English text. These data are based on the analysis of 12 fairly tales (2000 to 20,000 characters), converted to use only lowercase letters and blank. Each text was taken as one long word. For comparison,

the suffix trees for these texts had approximately 1.5 $\| S \|$ nodes and the same number of edges. For DNA sequences (four-letter alphabet) the size of $I_S$ is higher at 0.53 $\| S \|$ nodes and 1.4 $\| S \|$ edges and identification pointers when $S$ is a single "strand." These data are based on the analysis of the DNA of the viruses ms2, fd, and t7 (approximately 3500, 6500, and 40,000 characters, respectively). Suffix trees for these DNA samples had approximately 1.6 $\| S \|$ nodes and edges. A more detailed average case analysis of these structures, including approximate formulas for the average size of the compact DAWG for a single randomly generated text (equiprobable, independent characters) is given in [5].

We now turn our attention to implementing the functions *find*, *freq*, and *locations*.

LEMMA 2. *Using $I_S$, for any word $w \in \Sigma^*$, $x = find(w)$ can be determined in time $O(|x|)$. For any $x \in sub(S)$, $freq(x)$ can be determined in time $O(|x|)$ and if the nodes of $I_S$ are labeled with the prime subwords that represent them, $imp_S(x)$ can also be determined in time $O(|x|)$.*

PROOF. To implement *find*, we begin at the source of $I_S$ and trace a path corresponding to the letters of $w$ as long as possible. By Lemma 1, this "search path" is determined and continues until the longest prefix $x$ of $w$ in $sub(S)$ has been found. To implement *freq*, we note that $freq(x) = freq(imp_S(x))$ for any $x \in sub(S)$. Thus $freq(x)$ can be obtained by following the procedure of *find* and then returning the frequency label of the node that the final edge of this search path leads to, which will be $imp_S(x)$. If this node is labeled, a pointer to the string for $imp_S(x)$ can be recovered as well. Clearly all queries are $O(|x|)$. □

In implementing *locations*, let us assume that $S = \{w_1, \ldots, w_k\}$ and that for each $i$, $1 \le i \le k$, the length of $w_i$ is available. Assume further that $locations(x)$ returns $beginpos_S(x)$, as described above.

LEMMA 3. *Let $x \in sub(S)$ and $imp_S(x) = \gamma x \beta$. Let $L = \bigcup_{a \in \Sigma} locations(x\beta a)$. Let $T = \{\langle i, j \rangle : x\beta$ is a suffix of $w_i$ and $j = |w_i| - |x\beta|\}$. Then $locations(x) = L \cup T$.*

PROOF. Since $imp_S(x) = \gamma x \beta$, every occurrence of $x$ is followed by $\beta$. Thus occurrences of $x$ can be classified as those that are occurrences of $x\beta a$ for some $a \in \Sigma$, that is, those followed by still more letters, and those that are occurrences of $x\beta$ at the end of a word. These sets correspond to the sets $L$ and $T$, respectively. □

From the above lemma, it is clear that there is a simple recursive procedure for computing $locations(x)$ for any word $x \in sub(S)$, given that we are at the node $\gamma x \beta = imp_S(x)$ and we have the length of $\beta$. We simply compute the list $L$ described above recursively by examining each of the nodes reached by outgoing edges of $imp_S(x)$ and then concatenate this list with the list $T$, obtained from the list of identification pointers associated with the node $\gamma x \beta$. Since all sublists involved in this computation are obviously disjoint and each recursive call produces a nonempty list, the time required is clearly linear in the size $m$ of the final list. Since the time for the initial step of finding the node $imp_S(x)$ and the length of $\beta$ is $O(|x|)$, the total time for $locations(x)$ is $O(|x| + m)$. Thus we have

THEOREM 2. *Using $I_S$, the functions find, freq, and locations can be implemented in optimal time.*

### 3. *The DAWG*

As a preliminary step in building $I_S$, we construct a directed graph called the directed acyclic word graph (DAWG) for $S$. This graph is essentially a deterministic finite automaton that recognizes the set of subwords of $S$. The DAWG for a single word is defined in [8]. Here we extend that definition to a set of words $S$.

*Definition.* Let $x$ and $y$ in $\Sigma^*$ be *right equivalent* on $S$ if $endpos_S(x) = endpos_S(y)$. This relation is denoted by $x \equiv_S^R y$. For any word $x$, the equivalence class of $x$ with respect to $\equiv_S^R$ is denoted $[x]_{\equiv_S^R}$. The equivalence class of all words that are not subwords of $S$ is called the *degenerate* class. All other classes are *nondegenerate.*

It follows from the definition of $\equiv_S^R$ that if $x$ and $y$ are strings in the same nondegenerate equivalence class, then either $x$ is a suffix of $y$, or vice versa. Therefore, each nondegenerate equivalence class has a unique longest member, and all other members of the equivalence classes are suffixes of this longest member.

*Definition.* The unique longest member of a nondegenerate equivalence class under $\equiv_S^R$ is called the *representative* of that class. If $x$ is the representative of $[x]_{\equiv_S^R}$, then we say that $x$ *represents this class.*

Obviously $\equiv_S^R$ is a right invariant equivalence relation on $\Sigma^*$ of finite index. Thus, since $sub(S)$ is the union of all nondegenerate classes of $\equiv_S^R$, using standard methods [19], we can define from $\equiv_S^R$ a deterministic finite automaton that accepts $sub(S)$. Removing the one nonaccepting state, which corresponds to the degenerate class of $\equiv_S^R$, we obtain the following graph [6, 8].

*Definition.* The DAWG for $S$, denoted $D_S$, is the directed graph $(V, E)$ with edges labeled with letters in $\Sigma$, where $V$ is the set of all nondegenerate equivalence classes in $\equiv_S^R$ and $E = \{([x]_{\equiv_S^R}, [xa]_{\equiv_S^R}): x \in \Sigma^*, a \in \Sigma, \text{and } xa \in sub(S)\}$. The edge $([x]_{\equiv_S^R}, [xa]_{\equiv_S^R})$ is labeled $a$. $[\lambda]_{\equiv_S^R}$ is called the *source* of $D_S$.

The DAWG for the set $S = \{ababc, abcab\}$ is given in Figure 4.

It is enlightening to compare the DAWG for $S$ to the compact DAWG $C_S$ (Figure 1) and to the suffix tree for $S'$ (Figure 2), imagining the endmarkers removed. The remaining nodes of the suffix tree (shown with solid circles) naturally correspond to left invariant equivalence classes of $S$ because of equivalence of sets of beginning positions in $S$; that is, $x \equiv_S^L y$ if and only if $beginpos_S(x) = beginpos_S(y)$. This correspondence is analogous to that described for the equivalence classes of $C_S$ in Lemma 1. In fact, the nodes of $C_S$ represent the classes of the union of these two equivalence relations.

LEMMA 4. *$\equiv_S$ is the transitive closure of $\equiv_S^L \cup \equiv_S^R$.*

PROOF. For any $x, y \in \Sigma^*$, if $x \equiv_S^L y$ or $x \equiv_S^R y$, then it is clear that $imp_S(x) = imp_S(y)$, hence $x \equiv_S y$. On the other hand, if $imp_S(x) = \alpha x\beta = \alpha' y\beta' = imp_S(y)$ for some $\alpha, \beta, \alpha', \beta' \in \Sigma^*$, then $x \equiv_S^R \alpha x \equiv_S^L \alpha x\beta = \alpha' y\beta' \equiv_S^L \alpha' y \equiv_S^R y$. □

From Lemma 4, it follows that the compact DAWG can be obtained either by identifying nodes of the suffix tree that are equivalent under $\equiv_S^R$, or by identifying nodes of the DAWG that are equivalent under $\equiv_S^L$. This can be reduced to either identifying isomorphic subtrees of the suffix tree or "compacting" the edges of the DAWG. It is easier and more efficient computationally to use the DAWG. As is the case for the suffix tree, it can be shown that the size of the DAWG for $S$ is linear in $\| S \|$. In order to do so, it is useful to examine another relationship between the DAWG and the suffix tree.

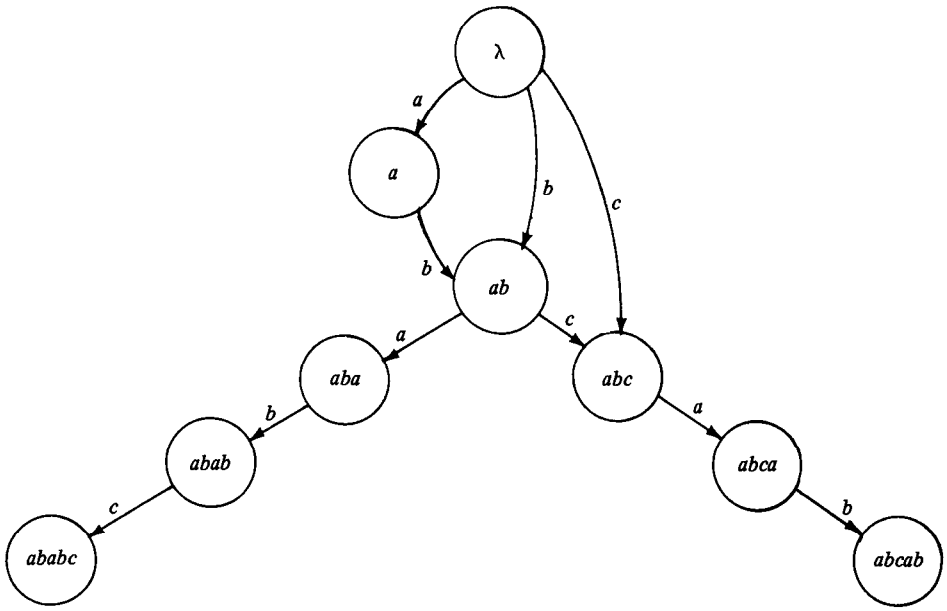


FIG. 4. DAWG for $S = \{ababc, abcab\}$.

Given two strings $x, y \in sub(S)$, if $endpos_S(x) \cap endpos_S(y) \neq \emptyset$, then either $x$ must be a suffix of $y$, or vice versa. This in turn implies that $endpos_S(x) \subseteq endpos_S(y)$, or vice versa. Therefore, the sets of the form $endpos_S(x)$ for $x \in sub(S)$ form a subset tree, which we call $T(S)$. This tree is illustrated in Figure 5 for $S = \{ababc, abcab\}$, where each node of the tree is labeled with the word that represents the equivalence class for the corresponding set of positions.

LEMMA 5. *If $x$ is the representative of $[x]_{\equiv_S^R}$, then any class is a child of $[x]_{\equiv_S^R}$ in $T(S)$ if and only if that class can be expressed as $[ax]_{\equiv_S^R}$ where $a \in \Sigma$ and $ax \in sub(S)$.*

PROOF. The children of $[x]_{\equiv_S^R}$ correspond to the maximal proper subsets of $endpos_S(x)$ that are equivalence classes under $\equiv_S^R$. Any such equivalence class must be $endpos_S(vx)$ for some nonnull word $v$. Since $x$ is the longest string in its equivalence class, the equivalence class of any string $ax$, $a \in \Sigma$, must be a proper subset of $[x]_{\equiv_S^R}$ and these will obviously be maximal. □

It follows from Lemma 5 that when every word in $S$ begins with a unique letter, $T(S)$ is isomorphic to the suffix tree for the set consisting of the reverses of all the words in $S$, except that the edges are unlabeled [8]. We make use of this correspondence in Section 6.

LEMMA 6. *Assume $\| S \| > 1$. Then the DAWG for $S$ has at most $2\| S \| - 1$ nodes.*

PROOF. In the special case where all of the words in $S$ are strings of the form $a^n$ for a single letter $a \in \Sigma$, $T(S)$ is a simple chain of $k + 1$ nodes where $k$ is the length of the longest word in $S$. Since $\| S \| > 1$, $k$ must be at least two and so the bound holds. In the remaining case, we show that $T(S)$ has at most $\| S \|$

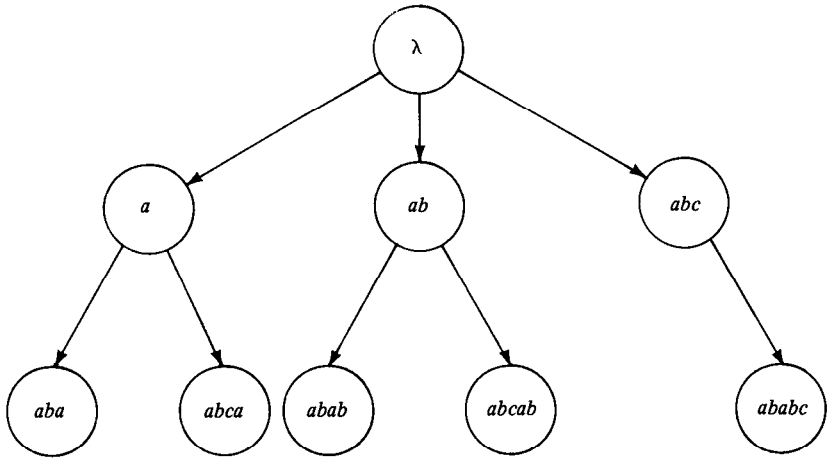


FIG. 5. $T(S)$ for $S = \{ababc, abcab\}$.

nonbranching nodes (nodes of outdegree less than 2), and hence at most $\| S \| - 1$ branching nodes, for a total of at most $2\| S \| - 1$ nodes.

Let $x$ represent the equivalence class of a node in $T(S)$. By Lemma 5, this node is a branching node if and only if there exist distinct letters $a, b \in \Sigma$ such that $ax$ and $bx$ are both in $sub(S)$. If every occurrence of $x$ is preceded by a single letter $a \in \Sigma$, then $ax \equiv_S^R x$, which contradicts the maximality of $x$. Thus, if $x$ represents a nonbranching node in $T(S)$, it must be the prefix of some word in $S$. However, since $S$ contains at least two distinct letters, $[\lambda]_{\equiv_S}$ must have at least two children. Thus $T(S)$ has at most $\| S \|$ branching nodes, since there are at most $\| S \|$ nonnull prefixes of the words in $S$. □

LEMMA 7. *Assume* $\| S \| > 1$. *If the DAWG for S has N nodes and E edges, then* $E \leq N + \| S \| - 2$.

PROOF. Note that the DAWG is a directed acyclic graph with a single source corresponding to $[\lambda]_{\equiv_S}$ and one or more sinks, each of which corresponds to $[w]_{\equiv_S}$ where $w$ is a word in $S$. Every node in the DAWG lies on a path from the source to at least one sink, and the sequence of labels on each such distinct path from source to sink spells out a distinct nonempty suffix of the word $w$ to which the sink corresponds.

Any such directed acyclic graph has a directed spanning tree rooted at its source, so we may choose one such spanning tree. This tree will have $N - 1$ edges. It remains to show that there are at most $\| S \| - 1$ additional edges that are not in the spanning tree.

With each of the edges of the DAWG that is not in the spanning tree, we can associate a distinct sufix of one of the words in $S$ as follows: For each edge there is a path along the spanning tree to the tail of the edge, along the edge, and finally to some sink in any convenient way. Each of these paths is distinct, and traces out a distinct suffix of some word in $S$. (The paths must differ in the first edge that is not in the spanning tree.) There are at most $\| S \|$ distinct suffixes of the words in $S$. For each sink, there must be a path from the source to that sink that lies entirely within the spanning tree. As there is at least one sink, there are at most $\| S \| - 1$ distinct paths in the DAWG that may be associated with edges not in the spanning tree. □

Worst case examples that asymptotically achieve the bounds given in Lemmas 6 and 7 are given in [8] for the special case where $S$ contains only one word.

Using Lemmas 6 and 7, we can give bounds on the total size of the DAWG for $S$ as follows:

THEOREM 3. *Assume* $\| S \| > 1$. *Then the DAWG for S has at most* $2\| S \| - 1$ *nodes and* $3\| S \| - 3$ *edges.*

The size of the DAWG in the average case is discussed in [5].

## 4. *Constructing the DAWG*

We now turn to the problem of constructing the DAWG for $S$. The algorithm we have developed builds the DAWG in a simple on-line fashion, reading each word from $S$ and updating the current DAWG to reflect the addition of the new word. Individual words are also processed on-line in one left-to-right scan.

The algorithm is a simple extension of the algorithm given in [8] to build the DAWG for a single word. Additional steps required are indicated by starred lines. The heart of the algorithm is given in the function *update*, and its auxiliary function, *split*. Given a DAWG for the set $S = \{w_1, \ldots, w_i\}$ (annotated with certain additional information described below), a pointer to the node represented by $w_i$ (called *activenode*) and a letter $a$, *update* modifies the annotated DAWG to create the annotated DAWG for $S' = \{w_1, \ldots, w_{i-1}, w_i a\}$. When processing on a new word begins, *activenode* is set to the source. *Split* is called by *update* when an equivalence class from $\equiv_S^R$ must be partitioned into two classes in $\equiv_{S'}^R$.

Two types of annotation are required. First, each of the transition edges of the DAWG is labeled either as a primary or as a secondary edge. The edge labeled $a$ from the class represented by $x$ to the class represented by $y$ is primary if $xa = y$. Otherwise, it is secondary. The primary edges form a directed spanning tree of the DAWG defined by taking only the longest path from the source to each node. The second kind of annotation is the addition of a *suffix pointer* to each node. The suffix pointer of a node in the DAWG points to the parent of the corresponding node in the tree $T(S)$. Equivalently, there is a suffix from the node represented by $x$ to the node represented by $y$ whenever $y$ is the largest suffix of $x$ that is not equivalent to $x$ under $\equiv_S^R$ (Lemma 5). The source is the only node that does not have a suffix pointer. Suffix pointers are analogous to those used by McCreight [17] and correspond to pointers used by Pratt, who gives an algorithm related to ours in [20].

The algorithm to build the DAWG is given in the Appendix. In consists of the main procedure *builddawg* and auxiliary functions *update* and *split*.

The key to the linear time bound for this construction algorithm is that using suffix pointers and primary or secondary marked edges, all of the structures that must be modified by *update* can be located rapidly. Here it is important that suffix pointers allow us to work from the longest suffixes backward to successively shorter suffixes, stopping when no more work needs to be done. Simpler methods that involve keeping track of all "active suffixes" will be potentially $O(n^2)$ (e.g., [15]). In addition, it is important that the states do not need to be marked with structural information about the equivalence classes they represent. This is in contrast to the $O(n^2)$ methods of [22], that build similar structures by directly partitioning the equivalence classes in an iterative manner.

In [8] we prove that the version of the algorithm that builds the DAWG for a single word is linear in the length of the word for an alphabet of fixed size. In

[4], a formal proof of correctness for this algorithm is given. It is readily verified that the modifications to that algorithm introduced here to build the DAWG for several words do not substantially affect any of the details of these proofs. Thus we have the following result.

THEOREM 4. *The DAWG for S can be built on-line in time linear in* $\| S \|$.

## 5. *Creating an Inverted File from the DAWG*

It remains to demonstrate how the DAWG can be compacted and labeled to create the data structure $I_S$ described in Section 2.

First, let us assume that the DAWG construction algorithm given in the previous section has been extended so that each node of the DAWG is labeled with the word that represents it. This will be accomplished by installing a pointer to the end position of the first occurrence of this word in $S$, along with the length of the word. The functions *update* and *split* are easily extended so that they maintain this information during the construction of the DAWG. Installing the pointers requires only that *update* have the current location in $S$ at the time it adds a new node represented by $w_i a$, as described above. New nodes created by *split* can simply take these pointers from the original nodes being split. To keep track of lengths, every time a new node is created, when the primary edge leading to this node is installed, the length of the representative of the new node is set to one greater than the length of the representative at the source of the incoming primary edge. These additions will not affect the linear time bound for construction.

We also assume that each node is augmented with a list of identification pointers, that indicate all texts in $S$, if any, of which the representative of the equivalence class of this node is a suffix. After the DAWG is built, these pointers can be added by a procedure that traces the suffix pointers from the node represented by $w_i$ to the source for each $w_i \in S$, adding identification pointers for $w_i$ to each node visited. The additional time and space for this procedure is bounded by the number of suffixes of the words in $S$, and hence is $O(\| S \|)$.

Given these extensions to the DAWG construction algorithm, it remains to show how the DAWG can be compacted to form $I_S$, and how the frequency labels are added.

In view of Lemma 4, each of the equivalence classes represented by the nodes of $I_S$ are composed of one or more right equivalence classes represented by nodes in the DAWG. The process of "compacting" the DAWG essentially consists of removing all but one of the nodes for each equivalence class, replacing the chains of classes removed by multiletter edges (see Figures 1 and 4). As a preliminary step to compaction, we make a recursive depth-first search of the DAWG adding the following pointers to the nodes.

*Definition.* If $x$ represents a node $A$ in the DAWG for $S$, the *implication pointer* of $A$ is a pointer to the node $B$ represented by $\alpha x \beta$, where $\alpha x \beta = imp_S(x)$. The *length* of this implication pointer is $| \beta |$.

LEMMA 8. *Let A be a node of the DAWG for S. If A has a single outgoing edge labeled a leading to a node B and an empty set of identification pointers, then the implication pointer of A is equal to the implication pointer of B, except that its length is one longer. Otherwise, the implication pointer of A points to A itself and has length zero.*

PROOF. Assume $A$ is represented by $x$. If $A$ has an identification pointer or two or more outgoing edges, then there is no one letter that always follows $x$. Since $x$

represents a node in the DAWG, there is no one letter that always precedes $x$ either. Hence $imp_S(x) = x$. If $A$ has a single outgoing edge labeled $a$ and no identification pointers, then $x$ is always followed by $a$. Hence, $imp_S(x) = imp_S(xa) = \gamma xa\beta$ for some $\gamma, \beta \in \Sigma^*$. Thus the implication pointers of $x$ and $xa$ are the same, but the former has length $|a\beta|$ and the latter length $|\beta|$. □

From the above lemma, it is clear that we can install implication pointers in the DAWG using a simple recursive depth-first search. Since the size of the DAWG for $S$ is linear in $\|S\|$ (Theorem 3), this procedure is linear in $\|S\|$.

By installing implication pointers, we have identified the nodes of the DAWG represented by prime subwords of $S$, which will be the only survivors of the "compaction" process that creates $I_S$. These are the nodes whose implication pointers point to themselves. We can also derive the edges of $I_S$ from these implication pointers. If $x$ is a prime subword of $S$ (representing a node $A$ in the DAWG), then $x$ will have an outgoing edge in $I_S$ leading to $imp_S(xa)$ for each $a \in \Sigma$ such that $xa \in sub(S)$. A node $B$ for any such $xa$ can be found by following the DAWG edge labeled $a$ from $A$. The node for $imp_S(xa)$ can then be found using the implication pointer in this node. If $imp_S(xa) = \gamma xa\beta$, then the label of this edge will be $a\beta$. Since the length of $\beta$ is given by the length of the implication pointer in $B$, and the node represented by $\gamma xa\beta$ contains a pointer to an occurrence of $\gamma xa\beta$ and its length, we can compute this label directly. It too will be represented using a pointer and a length. Thus with one more traversal of the DAWG, the nodes of $I_S$ can be identified and edges between them installed. This traversal also takes time proportional to the size of the DAWG and hence is $O(\|S\|)$.

A final pass can remove remaining nodes of the DAWG and all DAWG edges. Since lists of identification pointers have already been added to the nodes of the DAWG, the resulting graph will be $I_S$, without the frequency labels. To finish the construction, we can add the frequency labels using another simple recursive procedure, analogous to that used to compute *locations* (see Lemma 3).

Since each step of the construction of $I_S$ from the DAWG for $S$ is linear in $S$, we have the following theorem:

THEOREM 5. *$I_S$ can be built in time linear in $\|S\|$.*

## 6. *A Symmetric Version of the Compact DAWG*

The suffix tree for a set of words $S$ and the suffix tree for the reversed words of $S$ are quite different objects. There is in general no one-to-one correspondence between the nodes of these trees. The same holds for the DAWG. However, by the definition of prime subwords, it is clear that the prime subwords of the reversed words of $S$ are simply the reverses of the prime subwords of $S$. Indeed the partition of $sub(S)$ induced by $\equiv_S$ is invariant under reversal in this manner. Thus the structure of equivalence classes represented by the nodes of the compact DAWG does not depend on whether the words are read left to right or right to left.

The edge structure on the compact DAWG is dependent on reading direction, however. We have defined what might be called *right extension* edges of the form $(x, imp_S(xa))$ where $x$ is a prime subword and $a \in \Sigma$ is appended to the right of $x$. By adding the corresponding *left extension* edges, we obtain a structure that is fully invariant under reversal (up to the labeling of the edges and/or nodes).

*Definition.* The symmetric compact DAWG of $S$ is the graph $C2_S = (V, E_R, E_L)$ with two sets of edges $E_R$ and $E_L$, called *right* and *left*

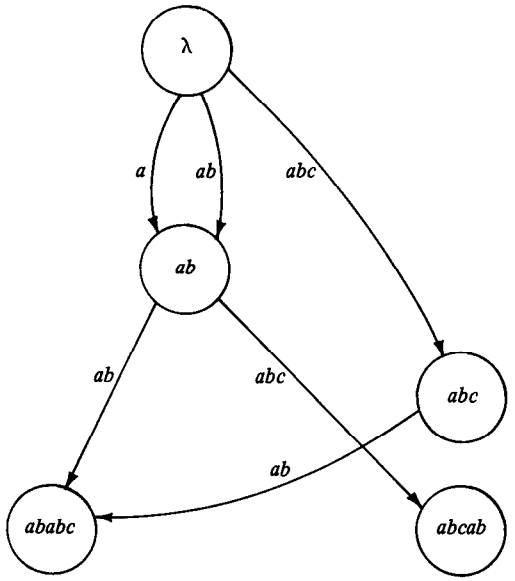


FIG. 6. Left extension edges for $C2_S$, $S = \{ababc, abcab\}$.

*extension* edges, respectively, where $V$ and $E_R$ are the nodes and edges of the compact DAWG of $S$, and $E_L = \{(x, imp_S(ax)) : x \in P(S), a \in \Sigma, ax \in sub(S)\}$. $label((x, \gamma ax\beta)) = \gamma a$.

$C2_S$ is illustrated in Figure 6 for $S = \{ababc, abcab\}$. To avoid confusion, only the left extension edges are drawn. The right extension edges are given in Figure 1.

The symmetric compact DAWG for $S$ is a very flexible tool for exploring the subword structure of $S$. Let us assume that the nodes of $C2_S$ are labeled with the prime subwords of $S$ that identify them (using the usual location/length representation) and their frequencies. Using the procedure *find* in the same manner it is used in $I_S$, given any word $w$, we can determine if $w$ is a subword of $S$ in optimal time $O(|w|)$ and if it is, we can go on to retrieve its frequency in constant time. With the additional node labels, we can also go on to retrieve a pointer to $imp_S(w)$ in constant time. We can then explore any other subwords that contain $w$ by appending various letters to the left or right of $imp_S(w)$ and following appropriate edges in $C2_S$ to other prime subwords. Each time we make an extension, the prime subword that is implied and its frequency can be recovered in constant time.

We conclude this section by demonstrating that $C2_S$ can be built in linear time. To do this, we add one more pass to the process of compacting the DAWG for $S$ described in the previous section. In this pass, we exploit the suffix pointers left behind by the construction of the DAWG.

Recall that there is a suffix pointer in DAWG from the node represented by $x$ to the node represented by $y$ whenever $y$ is the longest suffix of $x$ that is not equivalent to $x$ (under $\equiv_S^R$). Since $y$ is a proper suffix of $x$, there is a nonempty word $\gamma$ such that $x = \gamma y$. We call $\gamma$ the *label* of the suffix pointer from $x$ to $y$. We assume that each node of the DAWG has a pointer and length indicating the word that represents it, so $\gamma$ is easily computed in constant time and need not be explicitly stored. We show that the left extension edges of $C2_S$ correspond to the reversed suffix pointers of the DAWG for $S$.

LEMMA 9

(*i*) *If x and y represent nodes in the DAWG for S and there is a suffix pointer from x to y labeled* $\gamma$, *then there is a left extension edge from* $imp_S(y)$ *to* $imp_S(x)$ *labeled* $\gamma$ *in* $C2_S$.

(*ii*) *For every left extension edge in* $C2_S$ *there exist x and y as described in* (*i*).

PROOF

(i) Since $y$ represents a node in the DAWG for $S$, $y$ is either a prefix of a word in $S$ or occurs preceded by two distinct letters. Hence $imp_S(y) = y\beta$ for some $\beta \in \Sigma^*$. Since every occurrence of $y$ is followed by $\beta$, every occurrence of $x = \gamma y$ is followed by $\beta$. Since, in addition, $x$ represents a node in the DAWG for $S$, $imp_S(x) = x\beta\delta$ for some $\delta \in \Sigma^*$. Let $\gamma = \gamma' a$ where $a \in \Sigma$. By the definition of the suffix pointer, $ay \equiv_S^R \gamma y$, hence $ay \equiv_S \gamma y$. Thus $imp_S(ay\beta) = imp_S(\gamma y\beta) = x\beta\delta$. Hence there is a left extension edge labeled $\gamma$ from $imp_S(y)$ to $imp_S(x)$.

(ii) Assume there is a left extension edge labeled $\gamma$ from $u$ to $v$ in $C2_S$. Then $\gamma = \gamma' a$ for some $a \in \Sigma$ and $imp_S(au) = \gamma u\delta = v$ for some $\delta \in \Sigma^*$. This implies that $\gamma u$ represents a node in the DAWG for $S$ and $au \equiv_S^R \gamma u$. Since $u$ also represents a node in the DAWG for $S$ and $u$ is not equivalent to $\gamma u$ under $\equiv_S^R$, there must be a suffix pointer from the node represented by $\gamma u$ to the node represented by $u$ labeled $\gamma$. Obviously $imp_S(u) = u$ and $imp_S(\gamma u) = v$. □

We can install the left extension edges of $C2_S$ on nodes of the DAWG for $S$ represented by prime subwords as we did the right extension edges in the previous section. We simply traverse the DAWG, and for each node $x$ with suffix pointer to $y$ labeled $\gamma$, we install a left edge from $imp_S(y)$ to $imp_S(x)$ labeled $\gamma$. Assuming that implication pointers have been put into place in a previous pass, nodes for $imp_S(y)$ and $imp_S(x)$ can be located in constant time. Care must be taken to avoid installing duplicate edges. However, the labels of the outgoing left extension edges of a node all have distinct final letters, so we can check for duplicates in constant time, since we are assuming a fixed alphabet size.

The remainder of the construction of $C2_S$, which installs the right extension edges and deletes the superfluous nodes and edges from the DAWG, can be accomplished as described in the previous section. Thus we have the following result.

THEOREM 6. *$C2_S$ can be built in time linear in $\| S \|$.*

## 7. *Conclusions*

Retrieval structures based on the compact DAWG or suffix tree are considerably more powerful than conventional keyword-based retrieval structures [23] in that they allow queries for arbitrary strings, rather than restricting the user to a preselected keyword set. This feature is essential in situations in which there is no obvious notion of a reasonable keyword set, for example, in a library of DNA sequences [11]. It is also helpful in certain textual databases (e.g., in chemical abstracts) where one might want to search for various stems, suffixes, or prefixes of chemical compounds that would not be included even in a "full text" keyword set.

The drawback of the suffix tree is the large space that it occupies. Although significantly smaller than the suffix tree in many cases, the compact DAWG for a large set of texts, say 1 megabyte, still requires considerable storage space, depending on the implementation perhaps an order of magnitude more space than the text

itself. This is compounded by the fact that it is difficult to build a large linked graph such as the compact DAWG or suffix tree on conventional computer systems with limited random access memory, without running into the problem of "thrashing," in which most of the time is spent moving the data to and from the disk [15]. Unless these problems can be overcome, the applications of the compact DAWG will be limited to smaller, intensively searched or analyzed text collections for which conventional keyword-based systems are inadequate, for example, the DNA example cited above.

In addition to the problems of space requirements and thrashing, the research presented here also brings up other interesting questions, which are outlined below.

(1) Theorem 1 shows that the maximal number of prime subwords of a set $S$ of $k$ words is $\| S \| + k$. The worst case that we have found is $\| S \| + 1$, one of which is always the empty word. Is it true that for any finite set of words $S$, $S$ has at most $\| S \|$ nonempty prime subwords? (This stronger result was claimed in [7], but the sketch of proof given there was incorrect.)

(2) Can a complete theoretical analysis of the expected size of $I_S$ for a random text be given? In particular, how many prime subwords are expected for various notions of random text? How many in other types of text? Some results along these lines are given in [5].

(3) What are the most efficient methods of updating $I_S$ when new texts are added to $S$? Can two complete inverted files be merged in linear time?

(4) Do $I_S$ or $C2_S$ have applications in other areas of text processing, for example, spelling correction, inexact match searching, data compression or pattern recognition?

## *Appendix*

The following is a detailed algorithm to build the DAWG for a set of texts $S$.

*builddawg*($S$)
1. Create a node named *source.*
2. Let *activenode* be *source.*
3. For each word $w$ of $S$ **do:***
   A. For each letter $a$ of $w$ **do:**
      Let *activenode* be *update* (*activenode*, $a$).
   B. Let *activenode* be *source.*
4. **Return** *source.*

*update* (*activenode*, $a$)
1. **If** *activenode* has an outgoing edge labeled $a$, **then***
   A. Let *newactivenode* be the node that this edge leads to.*
   B. **If** this edge is primary, **return** *newactivenode.**
   C. **Else, return** *split* (*activenode*, *newactivenode*).*
2. **Else**
   A. Create a node named *newactivenode.*
   B. Create a primary edge labeled $a$ from *activenode* to *newactivenode.*
   C. Let *currentnode* be *activenode.*
   D. Let *suffixnode* be undefined.
   E. **While** *currentnode* isn't *source* and *suffixnode* is undefined **do:**
      i. Let *currentnode* be the node pointed to by the suffix pointer of *currentnode.*
      ii. **If** *currentnode* has a primary outgoing edge labeled $a$, **then** let *suffixnode* be the node that this edge leads to.
      iii. **Else, if** *currentnode* has a secondary outgoing edge labeled $a$ **then**
         a. Let *childnode* be the node that this edge leads to.
         b. Let *suffixnode* be *split* (*currentnode*, *childnode*).
      iv. **Else,** create a secondary edge from *currentnode* to *newactivenode* labeled $a$.

F. If *suffixnode* is still undefined, let *suffixnode* be *source.*
G. Set the suffix pointer of *newactivenode* to point to *suffixnode.*
H. **Return** *newactivenode.*

*split* (*parentnode, childnode*)
1. Create a node called *newchildnode.*
2. Make the secondary edge from *parentnode* to *childnode* into a primary edge from *parentnode* to *newchildnode* (with the same label).
3. For every primary and secondary outgoing edge of *childnode,* create a secondary outgoing edge of *newchildnode* with the same label and leading to the same node.
4. Set the suffix pointer of *newchildnode* equal to that of *childnode.*
5. Reset the suffix pointer of *childnode* to point to *newchildnode.*
6. Let *currentnode* be *parentnode.*
7. **While** *currentnode* isn't *source* **do**:
   A. Let *currentnode* be the node pointed to by the suffix pointer of *currentnode.*
   B. **If** *currentnode* has a secondary edge to *childnode,* **then** make it a secondary edge to *newchildnode* (with the same label).
   C. **Else,** break out of the **while** loop.
8. **Return** *newchildnode.*

ACKNOWLEDGMENTS. D. Haussler would like to thank Prof. Jan Mycielski for several enlightening discussions on these and related topics. We would also like to thank Joel Seiferas for pointing out his recent work in this area [10], and for sending us this work and several related papers.

REFERENCES

1. AHO, V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass., 1974.
2. APOSTOLICO, A., AND PREPARATA, F. P. The myriad virtues of suffix trees. In *Proceedings of the NATO Advanced Research Workshop on Combinatorial Algorithms on Words* (Maratea, Italy, June 18–22). Springer-Verlag, New York, 1985.
3. BAGGETT, P., EHRENFEUCHT, A., AND PERRY, M. A technique for designing computer access and selecting good terminology. In *Proceedings of the 1st Annual Rocky Mountain Conference on Artificial Intelligence.* Breit International Inc., Boulder, Colo., 1986.
4. BLUMER, J. Correctness and linearity of the on-line directed acyclic word graph algorithm. Tech. Rep. MS-8410, Univ. of Denver, Denver, Colo., 1984.
5. BLUMER, A., HAUSSLER, D., AND EHRENFEUCHT, A. Average sizes of suffix trees and DAWGs. Presented at the 1st Montreal Conference on Combinatorics and Computer Science, Univ. of Montreal, Canada, May 1987.
6. BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., AND MCCONNELL, R. Linear size finite automata for the set of all subwords of a word: An outline of results. *Bull. Eur. Assoc. Theoret. Comput. Sci. 21* (1983), 12–20.
7. BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., AND MCCONNELL, R. Building a complete inverted file for a set of text files in linear time. In *Proceedings of the 16th ACM Symposium on the Theory of Computing.* ACM, New York, 1984, pp. 349–358.
8. BLUMER, A., BLUMER J., EHRENFEUCHT, A., HAUSSLER, D., CHEN, M. T., AND SEIFERAS, J. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci., 40* (1985), 31–55.
9. CARDENAS, A. F. Analysis and performance of inverted data base structures. *Commun. ACM 18,* 5 (May 1975), 253–263.
10. CHEN, M. T., AND SEIFERAS, J. Efficient and elegant subword-tree construction. *Univ. of Rochester 1983-84 C.S. and C.E. Res. Rev.,* Univ. of Rochester, N.Y., 1984, pp. 10–14.
11. CLIFT, B., HAUSSLER, D., MCCONNELL, R., SCHNEIDER, T. D., AND STORMO, G. D. Sequence Landscapes. *Nucleic Acids Res., 14,* 1 (1986), 141–158.
12. EHRENFEUCHT, A., AND HAUSSLER, D. A new distance metric on strings computable in linear time. Tech. Rep. UCSC-CRL-86-27, Dept. of Computer and Information Sciences, Univ. of California at Santa Cruz, Oct. 1986.
13. GOLDSMITH, N. An appraisal of factors affecting the performance of text retrieval systems. *Inf. Tech.: Res. Dev. 1* (1982), 41–53.
14. KOHONEN, T. *Content-Addressable Memories.* Springer-Verlag, New York, 1980.

15. MAJSTER, M. E., AND REISER, A. Efficient on-line construction and correction of position trees. *SIAM J. Comput 9*, 4 (Nov. 1980), 785–807.
16. MALLER, V. The content addressable file store—A technical overview. *Angwte. Inf. 3* (1981), 100–106.
17. MCCREIGHT, E. M. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2 (Apr. 1976), 262–272.
18. MORRISON, D. R. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4 (Oct. 1968), 514–534.
19. NERODE, A. Linear automaton transformations. *Proc. AMS 9* (1958), 541–544.
20. PRATT, V. R. Improvements and applications for the Weiner repetition finder. Unpublished manuscript, Mar. 1975.
21. SLISENKO, A. O. Detection of periodicities and string matching in real time (English translation). *J. Sov. Math. 22*, 3 (1983), 1316–1387. (Originally published 1980).
22. TANIMOTO, S. L. A method for detecting structure in polygons. *Pattern Rec. 13*, 6 (1981), 389–394.
23. VAN RIJSBERGEN, C. J. File organization in library automation and information retrieval. *J. Doc. 32*, 4 (Dec. 1976), 294–317.
24. WEINER, P. Linear pattern matching algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*. IEEE, New York, 1973, pp. 1–11.

RECEIVED FEBRUARY 1985; REVISED MARCH 1986; ACCEPTED JUNE 1986