# Building a Complete Inverted File for a Set of Text Files in Linear Time

by
A. Blumer, J. Blumer, A. Ehrenfeucht*,
D. Haussler and R. McConnell

Department of Mathematics and Computer Science
University of Denver
Denver, Colorado 80208

*Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80302

All correspondence to D. Haussler.

Authors A. Blumer and D.Haussler gratefully acknowledge the support of NSF grant IST-83-17918 and author A. Ehrenfeucht the support of NSF grant MCS-83-05245.

## 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 $w$ which occurs in $S$; *freq*$(w)$, which returns the number of times $w$ occurs in $S$; and *locations*$(w)$ which returns the set of positions at which $w$ occurs. We give a data structure to implement a complete inverted file for $S$ which occupies linear space and can be built in linear time, using the uniform cost RAM model. Using this data structure, the time for each of the above query functions is optimal. To accomplish this, we use techniques from the theory of finite automata to build a deterministic finite automaton which 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.

## 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 [Gol 82], [Rij 76], [Car 75]. 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 abstract data type that implements the following functions.

1. *find*: $\Sigma^+ \to K \cup \{\lambda\}$, where *find*$(w)$ is the longest prefix $x$ of $w$ such that $x \in K \cup \{\lambda\}$ and $x$ occurs in $S$, *i.e.* $x$ is a subword of a text in $S$.

2. *freq* : $K \to \mathbf{N}$, where *freq*$(w)$ is the number of times $w$ occurs as a subword of the texts in $S$.

3. *locations* : $K \to 2^{\mathbf{N} \times \mathbf{N}}$, where *locations*$(w)$ is a set of ordered pairs giving the text numbers and positions within the texts where $w$ 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 [Koh 80], [Mal 81].

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*$(S)$, *i.e.*

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-133-4/84/004/0349 $00.75

every substring of $S$ is a keyword. We describe a data structure which implements a complete inverted file for $S$ which occupies linear space 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 $frequency(w)$ 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 (see [Aho 74]).

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 [Mor 68], which are further refined to the compact position trees of Weiner [Wei 73] and the suffix trees of McCreight [McC 76]. Our primary contribution is a new set of basic data structures which are functionally equivalent to suffix and position trees, but are smaller and easier to build. Using methods based on the theory of finite automata, we replace the trees used in earlier work with more compact directed acyclic graphs [1] (see Figure 1). These data structures will be superior for other applications of suffix trees as well (e.g. [Apo 83], [Rod 81], [Apo 79]).

In Section 1 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 2, 3 and 4 are devoted to demonstrating that this data structure can be built in linear time, assuming that the size of the alphabet is constant.

Section 2 introduces the Directed Acyclic Word Graph (DAWG) for a set of texts $S$ (see [Blu 83]), which is essentially a deterministic finite automaton that recognizes the set of all subwords of $S$. [2] In Section 3 we describe an on-line linear time algorithm to construct the DAWG for $S$. In Section 4 we give a collection of linear time procedures to "compact" the edges of the DAWG and label its nodes, creating the final data structure as described in Section 1.

## Notation

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 $\Sigma^* - \{\lambda\}$. $S$ will always denote a finite subset of $\Sigma^*$. For any $w \in \Sigma^*$, $|w|$ denotes the length of $w$. $\|S\| = \sum_{w \in S} |w|$. 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 $<i, j>$ 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 $<i, j>$ in $S$ immediately following occurrences of $x$ and $beginpos_S(x)$ denotes the set of all positions immediately preceding occurrences of $x$. For $x \notin sub(S)$, $beginpos_S(x) = endpos_S(x) = \phi$.

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.

## Section 1. 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 $x \underset{S}{\rightarrow} \alpha x \beta$ where $x \in sub(S)$, $\alpha, \beta \in \Sigma^*$, and

[1] Similar work in [Sei 83] has recently been brought to our attention.

[2] An equivalent data structure is also described in [Sei 83], along with a similar construction algorithm. Insights into the relationship between this and earlier data structures, especially those of [Pra 73], [Sli 80] and [Wei 73], are also given in this paper.

every time $x$ occurs in $S$, it is preceded by $\alpha$ and followed by $\beta$. $x \underset{S}{\rightarrow} \alpha x \beta$ is a *maximal rule* of $S$ if it is a rule of $S$ and $\alpha$ and $\beta$ are as long as possible (i.e. for no $\delta$, $\gamma \in \Sigma^*$, where $\delta\gamma \neq \lambda$ is $x \underset{S}{\rightarrow} \delta\alpha x \beta\gamma$ a rule of $S$). If $x \underset{S}{\rightarrow} \alpha x \beta$ is a maximal rule of $S$, then $\alpha x \beta$ is called the *implication* of $x$ in $S$, denoted $imp_S(x)$. $imp_S(sub(S)) = \{imp_S(x) : x \in sub(S)\}$. ▪

For example, if $S = \{ababc, abcab\}$ then $imp_S(\lambda) = \lambda$, $imp_S(a) = imp_S(b) = ab$, $imp_S(ca) = abcab$ and $imp_S(sub(S)) = \{\lambda, ab, abc, ababc, abcab\}$.

*Definition.* For any $x \in imp_S(sub(S))$ and $a \in \Sigma$, the *a−successor* of $x$ is $imp_S(xa)$ if $xa \in sub(S)$, otherwise it is undefined. The *compact DAWG* of $S$ is the directed graph $G_S = (V, E)$ with edges labeled with words in $\Sigma^+$, where $V = imp_S(sub(S))$ and $E = \{(x, imp_S(xa)) \; : \; a \in \Sigma \; , \; xa \in sub(S)\}$. $label((x, \alpha x a \beta)) = a\beta$. The node $\lambda$ is called the *source* of $G_S$. ▪

The compact DAWG for $S = \{ababc, abcab\}$ is given in in Figure 1a. For comparison, the suffix tree ([McC 76]) for $S' = \{ababc\,\$_1, abcab\,\$_2\}$ is given in Figure 1b. Here we assume the natural extension of McCreight's data structure from individual words to sets of words, using unique endmarkers. Notice that 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. [3] We elaborate further on this connection in Section 2.

The fundamental properties of $G_S$ are given in the following

*Lemma 1.* For any $x \in sub(S)$ there is a single path $p$ in $G_S$ 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 $label(p)$ which is not a prefix of the label of a proper initial segment of $p$, $imp_S(y) = imp_S(x)$. ▪

Informally, this lemma says that for any $x, y \in sub(S)$, $x$ and $y$ "lead to the same node" in $G_S$ if and only if $imp_S(x) = imp_S(y)$. Thus the nodes of $G_S$ represent the equivalence classes 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 $imp_S(x) = imp_S(y)$. For all $x, y \notin sub(S)$, $x \equiv_S y$.

Note that the words in any one of these equivalence classes all 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, $G_S$, annotated with certain additional information to facilitate the retrieval functions.

*Definition.* The *labeled* compact DAWG is the graph $I_S$, obtained from $G_S$ 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 in Figure 1d. Note that the identification pointers play a role analogous to the endmarkers in the suffix tree.

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. Thus 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 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 $|V| \leq \|S\| + 1$ and $|E| + m \leq 2\|S\| + k$.

*Sketch of proof.* This bound is related to the well-known bounds for suffix trees ([McC 76]). In particular, for $w \in \Sigma^*$, the suffix tree for $w\$_1$ has $|w| + 1$ leaves, at most $|w|$

[3] A similar observation was made for the DAWG (see Section 2) in [Sei 82].

internal nodes and thus at most $2|w|$ edges. For a set of texts $S = \{w_1, \ldots, w_k\}$ and $S' = \{w_1\$_1, \ldots, w_k\$_k\}$, the suffix tree for $S'$ will have $\|S\| + k$ leaves and at most $\|S\| - k + 1$ internal nodes, since the root nodes of the trees for the individual words are shared in the suffix tree for $S'$. Hence the number of edges in this tree is at most $2\|S\|$. It can be shown that this gives an upper bound of $2\|S\| + k$ on the number of edges and identification pointers in $I_S$, where the additional $k$ comes from the identification pointers in the nodes for $imp_S(w_i)$, $1 \le i \le k$. The upper bound of $\|S\| + 1$ nodes in $I_S$ comes from the $\|S\| - k + 1$ internal nodes of the suffix tree for $S'$ and at most $k$ additional nodes, again for $imp_S(w_i)$, $1 \le i \le k$. ▪

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 $.29\|S\|$ nodes and $1.0\|S\|$ edges and identification pointers for $I_S$ when $S$ is a single English text. For DNA sequences (four letter alphabet) the values are higher at $.53\|S\|$ and $1.4\|S\|$ for a single "strand".

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 $O(|x|)$, and for any $x \in sub(S)$, $freq(x)$ can be determined in time $O(|x|)$.

*Sketch of 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. Clearly both operations 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 $imp_S(x) = \alpha x \beta$. Let $L = \bigcup_{a \in \Sigma} locations(x\beta a)$. Let $T = \{<i, j> \;:\; x\beta$ is a suffix of $w_i$ and $j = |w_i| - |x\beta|\}$. Then $locations(x) = L \cup T$.

*Sketch of proof.* Since $imp_S(x) = \alpha x \beta$, 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 $a \in \Sigma$, i.e. those followed by still more letters, and those that are occurrences of $x\beta$ at the end of a word. ▪

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 $\alpha x \beta = imp_S(x)$ and we have the length of $\beta$. We simply compute the list $L$ described above recursively by examining each of the successors of $imp_S(x)$ and then concatenate this list with the list $T$, obtained from the list of identification pointers associated with the node $\alpha x \beta$. Since all sublists involved in this computation are obviously disjoint, 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. ▪

The remainder of the paper is devoted to the task of showing that $I_S$ can be built in time linear in the size of $S$.

## Section 2. The DAWG

As a preliminary step in building $I_S$, we construct a directed graph called the DAWG (Directed Acyclic Word Graph) for $S$. This graph is essentially a deterministic finite automaton which recognizes the set of subwords of $S$.

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

Obviously $\equiv^R_S$ is a right invariant equivalence relation on $\Sigma^*$ of finite index. Thus since $sub(S)$ is the union of all nondegenerate classes of $\equiv^R_S$, using standard methods ([Ner 58]), we can define from $\equiv^R_S$ a deterministic finite automaton which accepts $sub(S)$. Removing the one non-accepting state, which corresponds to the degenerate class of $\equiv^R_S$, we obtain the following graph (see [Blu 83]).

*Definition.* The Directed Acyclic Word Graph (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 $\equiv^R_S$ and $E = \{([x]_{\equiv^R_S}, [xa]_{\equiv^R_S}) : x \in \Sigma^*, a \in \Sigma$ and $xa \in sub(S)\}$. The edge $([x]_{\equiv^R_S}, [xa]_{\equiv^R_S})$ is labeled $a$. $[\lambda]_{\equiv^R_S}$ is called the *source* of $D_S$. ■

The DAWG for the set $S = \{ababc, abcab\}$ is given in Figure 1c.

It is enlightening to compare the DAWG for $S$ to the compact DAWG $G_S$ (Figure 1a) and to the suffix tree for $S'$ (Figure 1b), imagining the endmarkers removed. The remaining nodes of the suffix tree naturally correspond to left invariant equivalence classes of $S$ based on equivalence of sets of beginning positions in $S$, i.e. $x \equiv^L_S y$ if and only if $beginpos_S(x) = beginpos_S(y)$. This correspondence is analogous to that described for the equivalence classes of $G_S$ in Lemma 1. In fact, the nodes of $G_S$ represent the classes of the union of these two equivalence relations.

*Lemma 4.* $\equiv_S$ is the transitive closure of $\equiv^L_S \cup \equiv^R_S$.

*Proof.* For any $x, y \in \Sigma^*$, if $x \equiv^L_S y$ or $x \equiv^R_S y$ 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 $x \equiv^R_S \alpha x \equiv^L_S \alpha x \beta = \alpha' y \beta' \equiv^L_S \alpha' y \equiv^R_S y$. ■

While in principle, the compact DAWG can be obtained from either the suffix tree or 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 $\|S\|$. More precisely, we have the following.

*Lemma 5.* Assume $\|S\| > 1$. Then the DAWG for $S$ has at most $2\|S\| - 1$ nodes. ■

*Lemma 6.* Assume $\|S\| > 1$. If the DAWG for $S$ has $N$ nodes and $E$ edges, then $E \leq N + \|S\| - 2$. ■

The proofs of these lemmas are simple extensions of those given for Theorems 1 and 2 of [Blu 83] for the special case where $S$ has cardinality one. Worst case examples which achieve these bounds asymptotically are also given in that paper for this special case.

Using Lemmas 5 and 6, 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. ■

Preliminary experimental results indicate that the expected values are approximately $1.5\|S\|$ nodes and $2.2\|S\|$ edges when $S$ is a single English text. For DNA sequences (four letter alphabet), it appears that the expected values are approximately $1.6\|S\|$ nodes and $2.5\|S\|$ edges.

## Section 3. 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 (see [Maj 80] for a discussion of the advantages of an on-line algorithm).

The heart of the algorithm is given in the function *update*, and its auxiliary function, *split*. Given a DAWG for the set $S = \{w_1, \cdots, w_i\}$ (annotated with certain additional information described below), a pointer to the node representing $[w_i]_{\equiv^R_S}$ (called *activenode*) and a letter $a$, update modifies the annotated DAWG to create the annotated DAWG for $S' = \{w_1, \cdots, w_{i-1}, w_i a\}$.

When processing on a new word begins, *activenode* is set to the source. *Split* is called by *update* when an equivalence class from $\equiv^R{}_S$ must be partitioned into two classes in $\equiv^R{}_{S'}$.

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. 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, analogous to the pointers used in [McC 76] [4]. If the equivalence class of node $A$ consists of the words $\{a_1 \cdots a_r,\ a_2 \cdots a_r, \ldots,\ a_i \cdots a_r\}$ where $a_j \in \Sigma$, $1 \le j \le r$, then the suffix pointer of $A$ points to the node $B$ representing the equivalence class of $a_{i+1} \cdots a_r$ Since the source represents the equivalence class of $\lambda$, its suffix pointer is null.

The algorithm to build the DAWG is given in the Appendix. It 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 which must be modified by *update* can be located rapidly. Here it is important that suffix pointers allow us to work from the longest suffixes backwards to successively shorter suffixes, stopping when no more work needs to be done (see [McC 76]). Simpler methods, which involve keeping track of all "active suffixes" will be potentially $O(n^2)$ (e.g. [Maj 80]). 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 [Tan 81], which build similar structures by directly partitioning the equivalence classes in an iterative manner.

The essentials of the timing analysis of the DAWG algorithm can be given as follows. Since we are assuming that the alphabet size is constant, we can assume that the DAWG is implemented using linked lists of edges in the standard fashion (see e.g. [Aho 82]). Under these assumptions, it is clear that apart from the while loop in *update*, the total processing time is linear in $\|S\|$. Furthermore, since each pass through the body of this while loop except the last installs an edge, and no edges are ever deleted, the total number of times this loop is executed is bounded by the number of edges in the final DAWG, which is linear in $\|S\|$ (Theorem 3). Finally, the total processing time for each execution of the loop is constant, apart from the while loop in *split*. Thus the time bound is reduced to the problem of bounding the total time spent in the while loop of *split* by a linear function of $\|S\|$. This can be done as follows.

*Definition.* Let $S = \{w_1, \ldots, w_k\}$. $tail_S$ is the longest suffix of $w_k$ which occurs more than once in $S$. If $tail_S = \lambda$ then $shorttail_S = \lambda$, otherwise $shorttail_S$ is the longest suffix $w$ of $tail_S$ such that $w \not\equiv^R{}_S\ tail_S$. ▪

*Lemma 7.* Let $S = \{w_1, \ldots, w_i\}$ and $S' = \{w_1, \ldots, w_i a\}$. The number of times that the while loop of *split* is executed when $D_S$ is updated to $D_{S'}$ is bounded by $|shorttail_S| - |shorttail_{S'}| + 2$. ▪

To complete the proof of the linear time bound, we observe that when we sum the bounds given by Lemma 7 over the entire course of the computation, most of the terms cancel, leaving an $O(\|S\|)$ remainder. (See similar techniques in [McC 76]).

## Section 4. 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 1.

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 a pointer to the position in $S$ immediately following the first occurrence of longest member of the equivalence class it represents. The functions *update* and *split* are easily extended so that they maintain this information during the construction of the DAWG. This requires only that

[4] These pointers are originally due to Pratt, who gives an algorithm related to ours in [Pra 73].

*update* have the current location in $S$ at the time it adds a new node for the equivalence class of $w_i a$, as described above. New nodes created by *split* can simply take these pointers from the original nodes being split. 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, which indicate all texts in $S$, if any, of which a representative of the equivalence class of this node is a suffix. (Of course, all members of the equivalence class will be suffixes of these texts.) After the DAWG is built, these pointers can be added by a procedure that traces the suffix pointers from the node representing $[w_i]_{\equiv^R_S}$ 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 of Section 2, 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 multi-letter edges. As a preliminary step to compaction, we make a recursive depth-first search of the DAWG adding the following pointers to the nodes.

*Definition.* For each node $A$ of the DAWG for $S$ representing the equivalence class $[x]_{\equiv^R_S}$, the ***implication pointer*** of $A$ is a pointer to the node $B$ representing $[\alpha x \beta]_{\equiv^R_S}$, where $\alpha x \beta = imp_S(x)$. The ***label*** of this implication pointer is $\beta$. ▪

*Lemma 8.* a) The label of an implication pointer is well-defined, i.e. if $x, y \in sub(S)$ and $x \equiv^R{}_S\, y$ then $imp_S(x) = \alpha x \beta = imp_S(y) = \alpha' y \beta$ for some $\alpha, \alpha', \beta \in \Sigma^*$.
b) Let $A$ be a node of the DAWG for $S$ representing $[x]_{\equiv^R_S}$. 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 label is preceded by the letter $a$. Otherwise, the implication pointer of $A$ points to $A$ itself and has label $\lambda$.

*Sketch of proof.* Part (a) follows easily from the definitions of $imp_S(x)$ and $\equiv^R{}_S$, as in Lemma 4, Section 2. For part (b), if $A$ has a single outgoing edge labeled $a$ and no identification pointers, then words in the equivalence class represented by $A$ are always followed by the letter $a$. Hence $imp_S(x) = imp_S(xa)$ for any $x$ in this class. Otherwise, no letter is "predicted to the right" by the words in this class, so the longest member of the class is $imp_S(x)$ for all $x$ in this class. ▪

From Part (b) of the above Lemma, it is clear that we can install implication pointers in the DAWG using a simple recursive depth-first search. By using the pointers to occurrences of the longest string of each right equivalence class that are also present in the nodes, we can create the labels on the implication pointers using pointers to occurrences of the strings they represent and lengths, each in constant time. Since the size of the DAWG for $S$ is linear in $||S||$ (Theorem 3), the entire labeling procedure is linear in $||S||$. Following this labeling, another traversal of the DAWG can be made in which the nodes whose implication pointers do not point to themselves are removed, and edges leading to them from the remaining nodes are replaced with the appropriate successor edges of $I_S$, derived from the implication pointers of these nodes. 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 4.* $I_S$ can be built in time linear in $||S||$. ▪

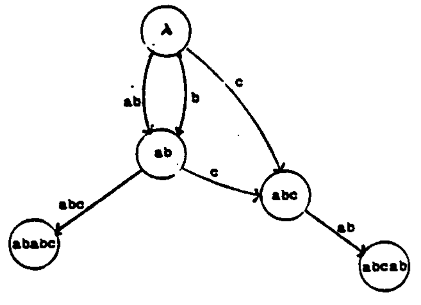
## Further Research

Numerous directions for further research remain. We list only a few.
1. Can a theoretical analysis of the expected size of $I_S$ for a random text be given?
2. Can $I_S$, without frequency labels, be built directly on-line in linear time?
3. What are the most efficient methods of updating $I_S$ when new texts are added or removed from $S$?
4. Notice that the definition of the nodes of $I_S$ is symmetric with regard to the direction in which the texts of $S$ are read. A symmetric set of edges can be added to the existing edges of $I_S$ which give the "left" a-successors of the nodes of $S$, in a manner analogous to the (right) a-successors defined here. An analogous set of "prefix identification pointers" can also be added. What might be the further applications of this "symmetric inverted file"? Can it be built in linear time?
5. By a simple extension of our algorithm, the nodes of $I_S$ can be labeled with a pointer to the word they represent and a length. With this extension, $imp_S(x)$ can be computed for any $x \in sub(S)$ in optimal time. Does this have further applications in other areas of text processing, e.g. spelling correction?

## Acknowledgement

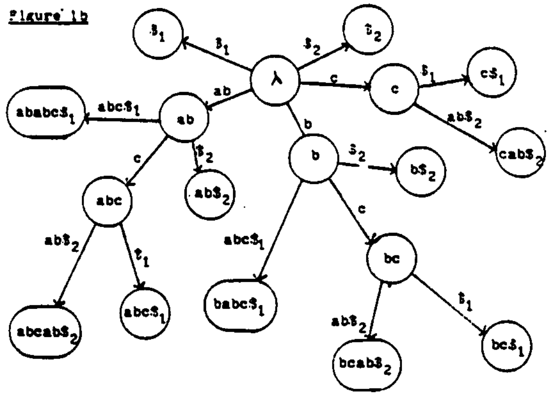
Author D. Haussler would like to thank Prof. Jan Mycielski for several enlightening discussions on these and related topics. We would also like to thank Joel Seiferas for pointing out his recent work in this area, and for sending us this work and several related papers.

Figure 1a



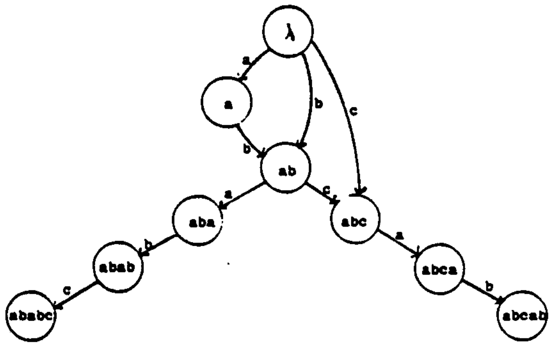
Compact DAWG $G_s$, for S = {ababc, abcab}

Figure 1b



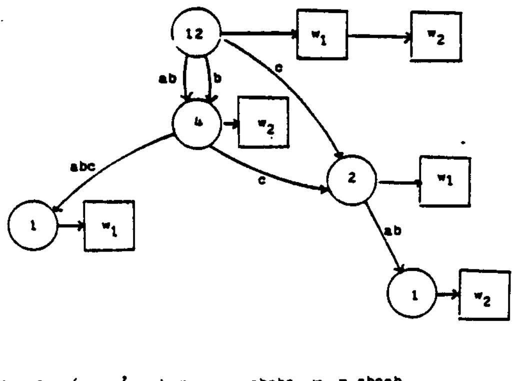
Suffix tree for $S' = \{ababc\$_1, abcab\$_2\}$

Figure 1c



DAWG for S = {ababc, abcab}

Figure 1d



$I_s$ for $S = \{w_1\ w_2\}$, where $w_1$ = ababc, $w_2$ = abcab

## 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)
A. If activenode has an outgoing edge labeled a then
        1. Let newactivenode be the node that this edge leads to.
        2. If this edge is primary, return newactivenode.
        3. Else, return split (activenode, newactivenode).
B. Else
        1. Create a node named newactivenode
        2. Create a primary edge labeled a from activenode to newactivenode.
        3. Let currentnode be activenode.
        4. Let suffixnode be undefined.
        5. While currentnode isn't source and suffixnode is undefined do:
                a. Let currentnode be the node pointed to by the suffix pointer of currentnode.
                b. If currentnode has a primary outgoing edge labeled a then let suffixnode be
                the node that this edge leads to.
                c. Else, if currentnode has a secondary outgoing edge labeled a then:
                        1. Let childnode be the node that this edge leads to.
                        2. Let suffixnode be split (currentnode, childnode).
                d. Else, create a secondary edge from currentnode to newactivenode labeled a.
        6. If suffixnode is still undefined, let suffixnode be source.
        7. Set the suffix pointer of newactivenode to point to suffixnode.
        8. 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, make it a secondary edge to
        newchildnode (with the same label).
        c. Else, break out of the while loop.
8. Return newchildnode.
```

## References

[Aho 82] Aho, Alfred V., John E. Hopcroft and Jeffrey D. Ullman; *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Massachusetts (1974).

[Apo 79] Apostolico, A.; "Some linear time algorithms for string statistics problems," Publication Series III, 176, Instituto per le Applicazioni del Calcolo "Mauro Picone" (IAC), Rome, 1979, 28pp.

[Apo 83] Apostolico, A. and F. P. Preparata; "Optimal off-line detection of repetitions in a string," *Theoretical Computer Science*, v. 22, 1983, 297-315.

[Blu 83] Blumer, A., J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell; "Linear size finite automata for the set of all subwords of a word: an outline of results," *Bul. Eur. Assoc. Theor. Comp. Sci.*, 1983, no. 21, 12-20.

[Car 75] Cardenas, A. F.; "Analysis and performance of inverted data base structures," *Comm ACM*, 1975, v. 18, no. 5., 253.

[Gol 82] Goldsmith, N.; "An appraisal of factors affecting the performance of text retrieval systems," *Information Technology: Research and Development*, 1982, 1, 41-53.

[Koh 80] Kohonen, T.; *Content-Addressable Memories*, Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[Maj 80] Majster, M. E. and Angelika Reiser; "Efficient on-line construction and correction of position trees," *SIAM J. Comput.*, v. 9, no. 4, Nov. 1980, 785-807.

[Mal 81] Maller, V.; "The content addressable file store - a technical overview," *Angwte. Infor.* (3) (1981), 100-106.

[McC 76] McCreight, Edward M.; "A space-economical suffix tree construction algorithm," *JACM*, v. 23, no. 2, April 1976, 262-272.

[Mor 68] Morrison, Donald R.; "PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric," *JACM*, v. 15, no. 4, October 1968, 514-534.

[Ner 58] Nerode, Anil; "Linear automaton transformations," *Proc. AMS*, v. 9, 1958, 541-544.

[Pra 73] Pratt, V. R., "Improvements and applications for the Weiner repetition finder," unpublished manuscript, May 1973 (revised Oct. 1973, March 1975).

[Rij 76] Van Rijsbergen, C. J.; "File organization in library automation and information retrieval," *Journal of Documentation*, v. 32, no. 4, December 1976, 294-317.

[Rod 81] Rodeh, Michael, Vaughan R. Pratt, and Shimon Even; "Linear algorithm for data compression via string matching," *JACM*, v. 28, no. 1, Jan. 1981, 16-24.

[Sei 83] Seiferas, J. and Chen, M. T., "Efficient and Elegant Subword-Tree Construction," *Univ. of Rochester 1983-84 C.S. and C.E. Research Review*, 10-14.

[Sli 80] 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).

[Tan 81] Tanimoto, S. L., "A Method for Detecting Structure in Polygons," *Pattern Recognition*, 1981, v. 13, no. 6, pp. 389-394.

[Wei 73] Weiner, P.; "Linear pattern matching algorithms," *IEEE 14th Annual Symposium on Switching and Automata Theory*, 1973, 1-11.