Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®
Theoretical Computer Science 301 (2003) 103–117
Theoretical Computer Science
www.elsevier.com/locate/tcs

Optimal insertion in deterministic DAWGs

Kyriakos N. Sgarbas*, Nikos D. Fakotakis, George K. Kokkinakis

Department of Electrical and Computer Engineering, Wire Communications Laboratory, University of Patras, GR-26500, Rio, Greece

Received 8 December 2000; received in revised form 24 April 2002; accepted 8 July 2002
Communicated by M. Crochemore

Abstract

In this paper, we present an on-line algorithm for adding words (strings) in deterministic directed acyclic word graphs (DAWGs) i.e. acyclic deterministic finite-state automata (DFAs). The proposed algorithm performs optimal insertion, meaning that if applied to a minimal DAWG, the DAWG after the insertion will also be minimal. The time required to add a new word is $O(n)$ with respect to the size of the DAWG. Repetitive application of the proposed insertion algorithm can be used to construct minimal deterministic DAWGs incrementally, although the algorithm is not time-efficient for building minimal DAWGs from a set of words: to build a DAWG of n words this way, $O(n^2)$ time is required. However, the algorithm is quite useful in cases where existing minimal DAWGs have to be updated rapidly (e.g. speller dictionaries), since each word insertion traverses only a limited portion of the graph and no additional minimization operation is required. This makes the process very efficient to be used on-line. This paper provides a proof of correctness for the algorithm, a calculation of its time-complexity and experimental results.

Keywords: DAWG; DFA; Minimization; Incremental algorithms; Spelling checker

1. Introduction

The directed acyclic word graph (DAWG) is a very efficient data structure for lexicon representation [2,18] and fast string matching [1,5,9], with a great variety of applications ranging from Speech Processing [13] to DNA analysis [9].

* Corresponding author.
E-mail addresses: sgarbas@wcl.ee.upatras.gr (K.N. Sgarbas), fakotaki@wcl.ee.upatras.gr (N.D. Fakotakis), gkokkin@wcl.ee.upatras.gr (G.K. Kokkinakis).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00571-6

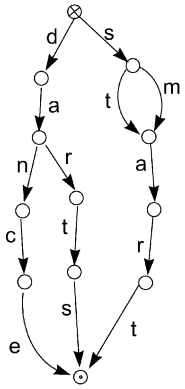

Fig. 1. Example of a deterministic DAWG.

An example of DAWG[1] is shown in Fig. 1. It consists of *nodes* (states) and *links* (transitions) between nodes. Each link has a *label*. The words are stored as directed paths on the graph. They can be retrieved by traversing the graph from an initial node (*source*) to a terminal node (*sink*), collecting the labels of the links encountered. In this way, traversing the DAWG of Fig. 1 from the source ($\otimes$) to the sink ($\odot$) we retrieve the words *dance*, *darts*, *start* and *smart*. DAWGs constitute very compact representations of lexicons: common word prefixes and suffixes are represented by the same links. The DAWG representation facilitates content-addressable pattern matching (i.e. approximate and incomplete search).

The DAWG of Fig. 1 is called *deterministic* because no links exist that have the same labels and leave the same node. This property results in a very efficient search function (linear to the length of the word), since the number of links leaving any node of a deterministic DAWG is bound to the size of the alphabet.

There are many possible deterministic DAWGs expressing the same lexicon. Of them, the one with the least number of nodes is called the *minimal* DAWG. Several algorithms [1,2,12,17] are known for the construction of the minimal DAWG, the most efficient being the algorithm by Revuz [18] working in linear time. However, all minimization algorithms are quite time-consuming because they have to check every node and link of the graph in order to perform the minimization. For this reason they are usually used off-line. The main practical drawback of the minimization approach is that if a change is made in the DAWG (even a slight one, such as the addition of a new word) the minimization process has to run again for the whole graph. For applications like spellers, which need to frequently update their lexicon, this is time-consuming and cumbersome.

[1] Some authors (e.g. [9]) use the term DAWG to denote the suffix automaton of a string (i.e. the graph containing all substrings of a string). In this paper, we do not have this constraint. We use the definition of Perrin [17] and Aoe [3] considering that a DAWG is an acyclic deterministic automaton and it may contain any finite-number of finite-length strings.

Realizing the practical value of the solution to the above problem we have devised an on-line algorithm for adding words in a deterministic DAWG [20]. A new word is inserted by slightly altering the topology of the original DAWG, without performing a total traversal of it. However, the insertion is optimal, i.e. this process guarantees that if the original DAWG is minimal, the updated DAWG will also be minimal. This eliminates the need of minimization after the insertion is completed. The time complexity for the word insertion is $O(n)$, in respect to the size of the DAWG. We can also use the algorithm to construct minimal DAWGs incrementally, by successive insertion of new words to an empty DAWG. This task requires $O(n^2)$ time for n words. If a list of words is given beforehand, there are more time-efficient algorithms to produce a minimal DAWG from them (e.g. [18]). But if the words are not all known beforehand, or if an initial DAWG is already available, or if the size of the intermediate structure is crucial (i.e. if we do not want to create a huge trie that will be later minimized to a DAWG), then the proposed algorithm can be very useful. In fact, the strong point of the algorithm is its usefulness in cases where existing minimal DAWGs have to be updated rapidly (like in speller dictionaries). Since each word insertion traverses only a limited portion of the graph and no additional minimization operation is used, this makes the process very efficient to be used on-line (faster than using minimization algorithms that have to access every node in the graph).

We first published this algorithm in [20] as a heuristic process, together with an analogous algorithm for non-deterministic DAWGs. In this paper, we provide a theoretical analysis of the algorithm, we provide some interesting lemmas concerning its function, we prove its optimality (i.e. that it actually produces minimal deterministic DAWGs), we calculate its time complexity and present experimental results for 230,000 words.

Several variations of the presented algorithm have been developed independently by different researchers. The first one by Aoe et al. [3], although it does not mention the minimality of the resulted DAWGs, it also shows the process to delete words from an existing DAWG. Later, Park et al. [16] proved that the aforementioned algorithm produces minimal DAWGs. The same research team has also presented a method [4] for compacting tries in a structure that, unlike a DAWG, is able to associate specific information to each word. More recent implementations of incremental algorithms have been reported by Mihov [14,15] for sorted input lexicons, Daciuk et al. [10,11] both for sorted and unsorted lexicons, Ciura and Deorowicz [7,8] and Revuz [19]. [2]

In Section 2 of this paper some basic definitions are given that will be used throughout the paper. We have tried to define appropriate concepts that simplify the proofs of the lemmas. The presentation of the algorithm follows in Section 3, where a step-by-step example is presented. In Section 4, a theoretical analysis is provided together with a set of interesting lemmas, resulting to a proof that the algorithm produces minimal DAWGs. The time complexity is calculated in Section 5, while implementation issues and experimental results are presented in Section 6. The paper conclusions follow in Section 7.

[2] See http://odur.let.rug.nl/alfa/fsa_stuff/ for more information on incremental algorithms for building automata.

2. Definitions

Let Q be a set of *nodes* (vertices) and Σ be a set of *symbols* (*alphabet*). A *labeled directed link* is then defined as a triple (n_1, n_2, s) from node n_1 to n_2 with label s, where $n_1, n_2 \in Q$ and $s \in \Sigma$.

Let $L \subseteq Q \times Q \times \Sigma$ be a set of labeled directed links. Then an ordered series $[(n_0, n_1, s_1), (n_1, n_2, s_2), (n_2, n_3, s_3), \ldots, (n_{k-2}, n_{k-1}, s_{k-1}), (n_{k-1}, n_k, s_k)]$ of successive links of L is called a *succession* from node n_0 to node n_k and is denoted by $\mathrm{succ}(n_0, n_k)$. We say that node n_k is a *successor* of node n_0 and that node n_0 is a *predecessor* of node n_k. In the special case where $|\mathrm{succ}(n_0, n_k)| = 1$, node n_k is an *immediate successor* of n_0 and node n_0 is an *immediate predecessor* of n_k.

There may be more than one successions between two nodes. We define as $\mathrm{SUCC}(n_0, n_k)$ the set of all successions from n_0 to n_k.

For a succession $G = \mathrm{succ}(n_0, n_k) = [(n_0, n_1, s_1), (n_1, n_2, s_2), (n_2, n_3, s_3), \ldots, (n_{k-2}, n_{k-1}, s_{k-1}), (n_{k-1}, n_k, s_k)]$, we define as $\mathrm{label}(G)$ the ordered series $[s_1, s_2, \ldots, s_{k-1}, s_k]$ of symbols as derived by the labels of the links in G.

For a set of successions $H = \mathrm{SUCC}(n_0, s_k)$, we define as $\mathrm{LABEL}(H)$ the set of all $\mathrm{label}(G)$, $\forall G \in H$.

Based on the above, a *Directed Word Graph* is defined as the quintuple $(Q, L, \Sigma, \mathrm{source}, \mathrm{sink})$, where $\mathrm{source}, \mathrm{sink} \in Q$ and $\forall n \in Q\text{-}\{\mathrm{source}\}$, $\mathrm{SUCC}(\mathrm{source}, n) \neq \emptyset$ and $\forall n \in Q\text{-}\{\mathrm{sink}\}$, $\mathrm{SUCC}(n, \mathrm{sink}) \neq \emptyset$. In other words, *source* is a predecessor of every other node in Q and *sink* is a successor of every other node in Q.

The graph is *acyclic* (i.e. *DAWG*) iff $\forall n \in Q$, $\mathrm{SUCC}(n, n) = \emptyset$.

The set of *strings* or *words* contained in a DAWG is finite and equals to LABEL(SUCC(source,sink)). All words in a DAWG have finite-length.

If two nodes $n_1, n_2 \in Q$ (with $n_1 \neq n_2$) satisfy the property $\mathrm{LABEL}(\mathrm{SUCC}(n_1, \mathrm{sink})) = \mathrm{LABEL}(\mathrm{SUCC}(n_2, \mathrm{sink}))$, then we say that n_1 and n_2 are *equivalent* and we note $n_1 \equiv n_2$. Two nodes that are not equivalent are called *distinct*. A single node is called *distinct* if it is not equivalent with any other node in Q. A DAWG that contains no equivalent nodes is called *minimal* (i.e. it has the least nodes possible).

For each node $n \in Q$, we consider the links entering n and the links leaving n and we define two sets: the *fan-in* set of n, $F_{\mathrm{IN}}(n) = \{(n', s)\colon (n', n, s) \in L\}$ and the *fan-out* set of n, $F_{\mathrm{OUT}}(n) = \{(n', s)\colon (n, n', s) \in L\}$. Although the links contained in these two sets are not full links (node n is missing from the triples) they are considered as links, since node n is always known and they can be restored into triples at any time. However, the above representation facilitates the comparison of fan-in and fan-out sets of different nodes.

A DAWG is *deterministic* iff $\forall n \in Q$, $\forall s \in \Sigma$ $|\{n'\colon (n', s) \in F_{\mathrm{OUT}}(n)\}| \leqslant 1$. Thus, in deterministic DAWGs $\forall n \in Q, |F_{\mathrm{OUT}}(n)| \leqslant |\Sigma|$.

A node $n \in Q$ with $|F_{\mathrm{IN}}(n)| > 1$ is called a *receiver*.

If two nodes $n_1, n_2 \in Q$ (with $n_1 \neq n_2$) satisfy the property $F_{\mathrm{OUT}}(n_1) = F_{\mathrm{OUT}}(n_2)$, then we say that n_1 and n_2 are *similar* and we note $n_1 \approx n_2$. In other words, two nodes are similar if the output links of the one match the output links of the other in labels and destinations. Similar nodes are always equivalent, but equivalent nodes are not necessarily similar (see Lemma 1, below).

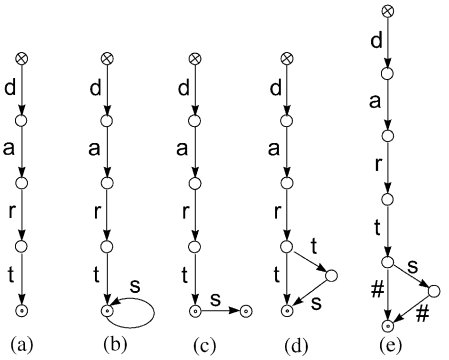

Fig. 2. Use of null links.

In a deterministic DAWG with one initial (source) and one terminal (sink) node as defined above, all links entering the sink must be *null*, i.e. must have null labels. Fig. 2 displays the reason. In Fig. 2a, a deterministic DAWG is shown, containing only the word *dart*. Figs. 2b–e display several attempts to add the word *darts* in the DAWG. The graph of Fig. 2b is not acyclic. The graph of Fig. 2c is not single-sink. The graph of Fig. 2d is not deterministic. Only the DAWG of Fig. 2e, which uses null links, is valid. We denote the null link label by the symbol #, which although we consider it to be a member of Σ, it can only appear at the end of a word. Thus, we can regard a null link as a stop link and # as an end-of-word indicator.

We have used the above definition of a DAWG because it best suits the implementation presented in this paper. However, another popular way to define a DAWG is to regard it as a special case of finite-state automaton (FSA) [12,17,18], i.e. a quintuple (Q,Σ,F,T,q_0), where Q is a set of states, Σ is a finite alphabet, q_0 is the initial state, T is the set of terminal states ($T\subseteq Q$) and F is a function $Q\times\Sigma\to Q$ defining the transitions between the states. Despite their apparent differences, the two definitions are equivalent. We have used the set L of labeled directed links to express the function F because in our algorithm we need to explicitly identify and store individual transitions. The use of only one sink instead of the set T of terminal states facilitates the detection of similar nodes. However, we can always transform a DAWG with multiple terminal states to a single-sink DAWG by adding a sink node and creating links from every terminal state to the sink. Similarly, we can always transform a single-sink DAWG to a multi-terminal one, by marking as terminal every node that has a null link leaving from it. In this paper, we assume single-source single-sink deterministic DAWGs.

Let $D=(Q,L,\Sigma,\text{source},\text{sink})$ be a deterministic DAWG as defined above. We present the following Lemmas:

Lemma 1. *Two equivalent nodes of D are either similar or their immediate successors are also equivalent.*

Proof. Let $p,q\in Q$ with $p\equiv q$. Consider two links (p,p',s) and (q,q',s) such that $p'\neq q'$. If for no $s\in\Sigma$ two such links exist, that implies $F_{\text{OUT}}(p)=F_{\text{OUT}}(q)$ and $p\approx q$.

Otherwise, if the links exist, consider the nodes p' and q'. Suppose that they are not equivalent. Then $\mathrm{LABEL}(\mathrm{SUCC}(p', \mathrm{sink})) \neq \mathrm{LABEL}(\mathrm{SUCC}(q', \mathrm{sink}))$. But this implies that $\mathrm{LABEL}([(p, p', s)] \cup \mathrm{SUCC}(p', \mathrm{sink})) \neq \mathrm{LABEL}([(q, q', s)] \cup \mathrm{SUCC}(q', \mathrm{sink}))$ and since D is deterministic, then $\mathrm{LABEL}(\mathrm{SUCC}(p, \mathrm{sink})) \neq \mathrm{LABEL}(\mathrm{SUCC}(q, \mathrm{sink}))$, which contradicts to $p \equiv q$. Therefore $p' \equiv q'$. □

Lemma 2. *D is not minimal iff similar nodes exist in Q.*

Proof. First we show that if $p, q \in Q$ and $p \approx q$, then D is not minimal: Similar nodes are always equivalent. Therefore $p \equiv q$ and D is not minimal. Next, we show that if D is not minimal then Q contains similar nodes: If D is not minimal then we can find $p, q \in Q$ with $p \equiv q$. By Lemma 1, it is either $p \approx q$, or their immediate successors p' and q' are equivalent. Supposing the latter, we can apply Lemma 1 to p' and q'. Since there is only one sink and $|\mathrm{succ}(p, \mathrm{sink})|$ is finite, we eventually arrive in two equivalent nodes, which are also similar. □

Lemma 2 constitutes a criterion for checking whether a deterministic DAWG is minimal or not. We call it the *Minimality Criterion*. Checking a DAWG using the Minimality Criterion is more efficient than searching for equivalent nodes, since given two nodes, it is much faster to decide if they are similar than it is to check their equivalence.

3. Presentation of the algorithm

The proposed algorithm adds a new word to an existing deterministic DAWG. Fig. 3 displays an example of word insertion. The original DAWG of Fig. 3a contains the words *pair*, *part*, *dart* and *start*. We wish to add the word *stair*. The insertion is performed in three stages:

Stage 1: Starting from the source (Fig. 3b), we follow existing links to form the maximum possible prefix of the new word. These links are marked. In Fig. 3 they are shown dashed. Each time a marked link j reaches a receiver node n (the gray nodes in Figs. 3b,c), a new node n' similar to n is created and j is redirected to n': $F_{\mathrm{IN}}(n) = F_{\mathrm{IN}}(n) - \{j\}, F_{\mathrm{IN}}(n') = \{j\}$. Nodes n and n' are shown as marked pairs in Figs. 3c,d. Stage 1 stops when no further traversal is possible (in Fig. 3d the prefix *sta-* of the word *stair* has been formed and no link with label *i* continues).

Stage 2: Starting from the node at which Stage 1 has stopped, we create new links and nodes for the remaining suffix of the word (see Fig. 3e). All new links are also marked. The last link is connected to the sink.

Stage 3: We consider the last marked link (n', p, c) of Stage 2 and we search in $F_{\mathrm{IN}}(p)$ for a node n similar to n' (see Fig. 3e). Node n' is deleted (and so all its outgoing links) after redirecting its incoming link to n: $F_{\mathrm{IN}}(n) = F_{\mathrm{IN}}(n) \cup F_{\mathrm{IN}}(n')$ (the gray node in Fig. 3f). The process is repeated, considering the new last-marked link (Fig. 3f) with a new pair of similar nodes, until no similar nodes can be found this way (see Figs. 3g–i).

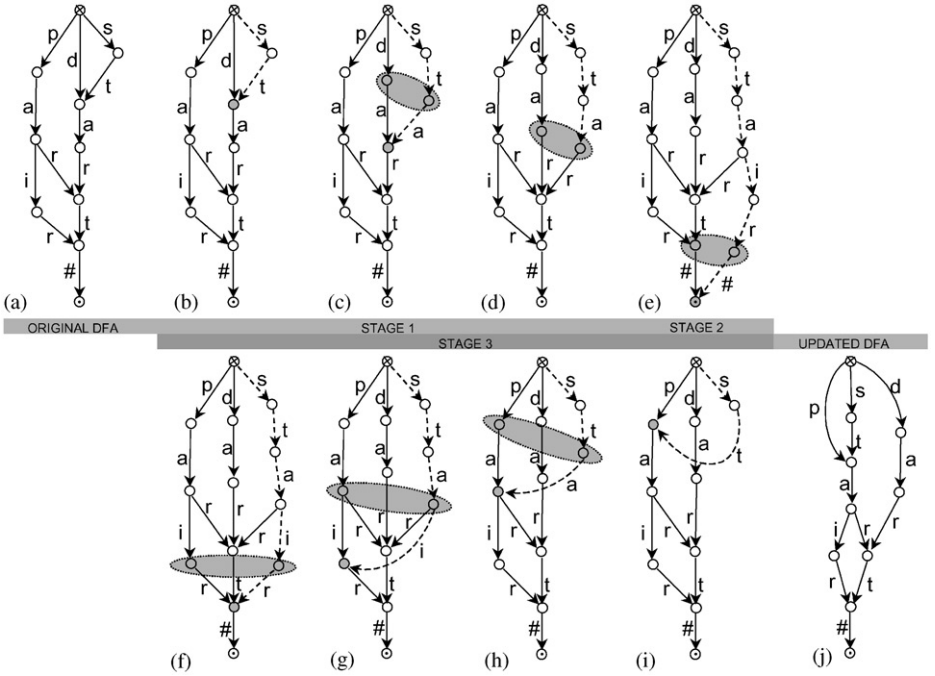

Fig. 3. The word "stair" is inserted to a minimal deterministic DAWG (acyclic DFA).

The updated DAWG (Fig. 3j) contains all the words of the original DAWG, plus the new word. Note that the algorithm does not traverse every node and link of the original DAWG in order to add the new word. Despite that, the insertion is optimal, meaning that if the original DAWG is minimal, the updated one will also be minimal. A proof of this is given in the next section.

The algorithm is shown in Table 1.

4. Algorithm analysis

For the analysis of this section, consider again the example of Fig. 3. The original DAWG of Fig. 3a is minimal.

The algorithm creates new nodes and links in the first two stages and it deletes nodes and links in Stage 3. However, note that none of the algorithm stages alters the F_{OUT} set of any existing node.

At Stage 1, every time a receiver n is encountered, a new node n' is created, similar to n. It is obvious that every immediate successor of n' will also be a receiver: since $F_{\mathrm{OUT}}(n) = F_{\mathrm{OUT}}(n')$, every immediate successor of n' will have both n and n' as immediate predecessors. Therefore, when the first receiver is encountered, a sequence of node creations starts, which stops only at the end of Stage 1 (see Fig. 3d).

Table 1
The optimal insertion algorithm

```
   // STAGE 1: Follow existing links
1        n0 ← source; i ← 0;                                  // array w[] contains the new word
2        do {
3                 if (n, w[i]) ∈ FOUT(n0) then {              // only one such link may exist
4                         j ← (n0, w[i]);
5                         if |FIN(n)| > 1 then {              // n is a receiver
6                                 create new node n′;
7                                 FOUT(n′) ← FOUT(n);         // n′ ≈ n
8                                 FIN(n) ← FIN(n) − {j};      // redirect j
9                                 FIN(n′) ← {j};
10                                n ← n′;
                          }
11                        n0 ← n; i++;
12                } else break do;
         }
   // STAGE 2: Attach remaining suffix
13       while (i < M) {                                      // M = |w[]|
14                create new node n;
15                create new link (n0, n, w[i]);
16                n0 ← n;
         }
17       create new link (n0, sink, '#'); j ← (n0, '#');
   // STAGE 3: Search for similar nodes
18       p ← sink;
19       (n′, c) ← j;                                         // the last-marked link
20       while exists another (n, c) ∈ FIN(p) with n ≠ n′ {   // search only links in FIN(p)
21                if FOUT(n) = FOUT(n′) then {                // similarity check
22                        {j} ← FIN(n′);                      // |FIN(n′)| = 1
23                        FIN(n) ← FIN(n) ∪ {j};              // redirect j
24                        delete n′, FIN(n′), FOUT(n′);
25                        p ← n; go to 19;                    // j is the next last-marked link
                  }
   }
```

Then, at Stage 2, a series of new links and nodes is created from the node at which Stage 1 has stopped, to the sink (Fig. 3e). The DAWG at the end of Stage 2 (Fig. 2e) is deterministic and contains all the words of the original DAWG plus the new word, but it is not minimal.

The minimization is performed in Stage 3, based on the Minimality Criterion: Since a non-minimal DAWG always contains similar nodes (Lemma 2), Stage 3 finds similar nodes and merges them, until no more similar nodes can be found in the DAWG. Then by the Minimality Criterion the DAWG will be minimal. Stage 3 can be considered as the counterpart of Stage 1. New nodes created at the previous two stages are now eliminated if they are found to be similar to other nodes of the DAWG. Note that the algorithm does not search the whole DAWG to find similar nodes. For every link (n', p, c) in the path of the newly inserted word, starting from the last and continuing

upwards, it considers only the nodes n such that $(n,c)\in F_{\mathrm{IN}}(p)$ and $n\neq n'$, and it checks only them for similarity with n'.

Now consider the dashed (marked) links of Fig. 3e. The algorithm keeps track of them. They form a succession that corresponds to the new word. Let Z be the set of nodes contained in that succession, excluding source and sink. Then Z contains all the new nodes created by the process. Z also contains any non-receiver nodes encountered in Stage 1 before the first receiver. If any node n' in Z is found similar to some other node, node n' is deleted from Z. Note that the process causes only new nodes (nodes from set Z) to be deleted during Stage 3. After a node is deleted, its redirected input link (see step 23 in Table 1) becomes the last-marked link for the next iteration (step 19).

The following Lemmas prove the correctness and the efficiency of the proposed insertion algorithm. Lemmas 3 and 4 address properties of set Z. Lemmas 5 and 6 ensure that for every pair of equivalent nodes in Q, one member of the pair will always be in Z. Lemma 7 ensures that the process of Stage 3 can be stopped safely without checking any more nodes, if a node in Z is encountered that fails the similarity check. Finally, Lemma 8 proves the optimality of the algorithm.

Lemma 3. *For every* $n\in Z, |F_{\mathrm{IN}}(n)|=1$.

Proof. Set Z contains two kinds of nodes: non-receivers and new nodes. Consider a node $n\in Z$. If n is non-receiver, then $|F_{\mathrm{IN}}(n)|=1$. If n is a new node created in Stage 1, then $|F_{\mathrm{IN}}(n)|=1$ because only one link is redirected to each new node at that stage. If n is a new node created in Stage 2, then again $|F_{\mathrm{IN}}(n)|=1$ since a single succession of links is created during that stage. □

Lemma 4. *There are no equivalent nodes in* Z.

Proof. Let $p,q\in Z$. Since they belong in the same succession, one must be successor of the other. Let q be the successor and p the predecessor. Then, since D is acyclic, $\exists s\in \mathrm{LABEL}(\mathrm{SUCC}(p,\mathrm{sink}))$ such that $s\notin \mathrm{LABEL}(\mathrm{SUCC}(q,\mathrm{sink}))$, and p cannot be equivalent to q. □

Lemma 5. *There are no equivalent nodes in the set* $Q-Z$.

Proof. Suppose there are two equivalent nodes in $Q-Z$. Then, by Lemma 1 we can find two similar nodes $p,q\in Q-Z$. Let p' and q' be the nodes in the original (minimal) DAWG that correspond to p and q. Since none of the algorithm stages alters the F_{OUT} set of any node, it would be $F_{\mathrm{OUT}}(p')=F_{\mathrm{OUT}}(p)$ and $F_{\mathrm{OUT}}(q')=F_{\mathrm{OUT}}(q)$. Therefore $F_{\mathrm{OUT}}(p')=F_{\mathrm{OUT}}(q')$ and $p'\approx q'$ which (by Lemma 2) contradicts our assumption for a minimal original DAWG. □

Lemma 6. *Every node in* Z *has at most one equivalent node.*

Proof. Let $n \in Z$, $p, q \in Q$ such that $n \equiv p$ and $n \equiv q$. By Lemma 4, neither p nor q can belong to Z, since $n \in Z$. Thus they must both belong to $Q - Z$. But since $n \equiv p$ and $n \equiv q$ imply $p \equiv q$, this contradicts Lemma 5. □

Lemma 7. *If a node $n \in Z$ is distinct, then every predecessor of n is also distinct.*

Proof. By Lemma 3 all predecessors of n belong to Z. Let $m \in Z$ be the immediate predecessor of n. If exists $m' \in Q$ equivalent to m, then since n is distinct Lemma 1 implies that $m \approx m'$. Therefore n is an immediate successor of both m and m', which contradicts Lemma 3. Thus, node m must be distinct. Extending the syllogism to the distinct node m, we derive that every predecessor of n is also distinct. □

Lemma 8. *If we use the described algorithm to add a new word to a minimal DAWG, then the updated DAWG is also minimal.*

Proof. Suppose that after the end of Stage 3 the updated DAWG is not minimal. Then there should be two equivalent nodes in Q. Let us examine in what sets these two nodes can belong to: By Lemma 4 they cannot both belong to Z. By Lemma 5 they cannot both belong to $Q - Z$. Therefore, one must belong to Z and the other to $Q - Z$. But since Stage 3 has been completed, the last node examined in Z was found distinct and by Lemma 7 all nodes in Z are distinct. Therefore it is not possible to find two equivalent nodes in Q. Thus the DAWG is minimal. □

5. Complexity calculation

For the time-complexity calculation of the algorithm, consider the listing in Table 1. We define $M = |w[\,]|$, the length of the word, $S = |\Sigma|$, the size of the alphabet and $\Lambda = |\mathrm{SUCC}(\mathrm{source}, \mathrm{sink})|$ the number of words in the DAWG. We shall also need the following Lemma:

Lemma 9. *For any node n of a deterministic* DAWG, $|F_{\mathrm{IN}}(n)| \leqslant \Lambda$.

Proof. Let $n \in Q$ be a node with $|F_{\mathrm{IN}}(n)| = K > \Lambda$. Then $|\mathrm{SUCC}(\mathrm{source}, n)| \geqslant K$ and $|\mathrm{SUCC}(\mathrm{source}, \mathrm{sink})| \geqslant K > \Lambda$, which contradicts to the definition of Λ. □

In our implementation we have used linked lists to represent the F_{IN} and F_{OUT} sets for every node of the DAWG, as detailed in Section 6. In the following we give the time complexity calculated for the implementation with linked lists and between angle brackets $\langle\,\rangle$ we give an analogous calculation for a hypothetical ideal data structure that has optimal access time in all operations [23].

Stage 1: The search of step 3 (see Table 1) of the algorithm is an $\mathrm{O}(S)$ $\langle \mathrm{O}(\log S) \rangle$ operation. The check of step 5 is performed in constant time. If true, line 6 creates a second node (constant time), line 7 copies the F_{OUT} of the initial node to the new one (an $\mathrm{O}(S)$ operation), line 8 deletes the input link from the F_{IN} of the original node

(an $O(\Lambda)\langle\text{constant}\rangle$ operation) and line 9 inserts it as the single input link to the second node (constant time). Since the steps 2–12 of the algorithm are executed at most M times, we have a total of $O(M(\Lambda+2S))\langle O(M(S+\log S)\rangle$ time for Stage 1.

Stage 2: Stage 2 is straightforward. It needs $O(S)\langle O(\log S)\rangle$ time for the first link, $O(\Lambda)\langle O(\log \Lambda)\rangle$ for the last link in step 17 and $O(M)$ time for the rest ones in steps 13–16. The combined time for Stage 2 is $O(S+\Lambda+M)\langle O(\log S+\log \Lambda+M)\rangle$.

Stage 3: The check in step 21 can be done in $O(S)$ time. If the nodes are found to be similar, steps 22–25 are executed. Since $|F_{IN}(n')|=1$ (Lemma 3), step 23 requires $O(\Lambda)\langle O(\log \Lambda)\rangle$ time. The while-loop (steps 20–25) is repeated at most Λ times, but since a successful similarity check (at step 21) exits the loop at step 25, the loop 20–25 requires $O(\Lambda S+\Lambda)\langle O(\Lambda S+\log \Lambda)\rangle$ time. The outer loop 19–25 is repeated at most M times. Thus, the overall time requirement for Stage 3 is $O(M\Lambda(S+1))\langle O(M(\Lambda S+\log \Lambda))\rangle$.

Combining the time complexities of the three stages we derive: $O(M(\Lambda+2S))+O(S+\Lambda+M)+O(M\Lambda(S+1))=O(\Lambda)$ for the implemented version of the algorithm, and $\langle O(M(S+\log S))\rangle+\langle O(\log S+\log \Lambda+M)\rangle+\langle O(M(\Lambda S+\log \Lambda))\rangle=\langle O(\Lambda)\rangle$ also for the ideal implementation.

Considering that the number of nodes and the number of links in a DAWG is linear to the number of words that the DAWG contains [6,9], we deduce that the proposed algorithm requires linear time ($O(\Lambda)$) in respect to the size of the DAWG to insert a new word. Building from scratch a DAWG using successive insertions of $O(\Lambda)$ time each, is an $O(\Lambda^2)$ operation. These calculations are verified by the experimental results of the next section (see Fig. 6).

6. Implementation and experimental verification

The presented algorithm has been implemented in C++. The DAWG has been represented using two arrays, one for the nodes and one for the links, as shown in Figs. 4 and 5, respectively. Nodes and links were numbered. Each node was assigned a node-ID and each link was assigned a link-ID. Fig. 4 shows a node together with its two associated sets of links: the fan-in set (the set of input links) and the fan-out set (the set of output links). We kept these sets in two different linked lists for every node. The fan-in list was sorted primarily by label and secondarily by the node-ID of the departure node. The fan-out list was sorted primarily by label and secondarily by the node-ID of the destination node.

The array of nodes (Fig. 4) mapped each node-ID to a pair of link-IDs. The first member of the pair was the link-ID of the first fan-out link (ffl[0] in Fig. 4) for the corresponding node. Similarly, the second member of the pair was the link-ID of the first fan-in link (ffl[1]) for the node. Thus, given a node-ID we had immediate access to its first input and its first output links.

The elements of the link array (Fig. 5) were more complex: Every link connected two nodes (the departure node and the destination node) and it participated in two linked lists: the fan-out list of the departure node and the fan-in list of the destination node. Thus, for any given link-ID the array provided the node-ID of its departure node

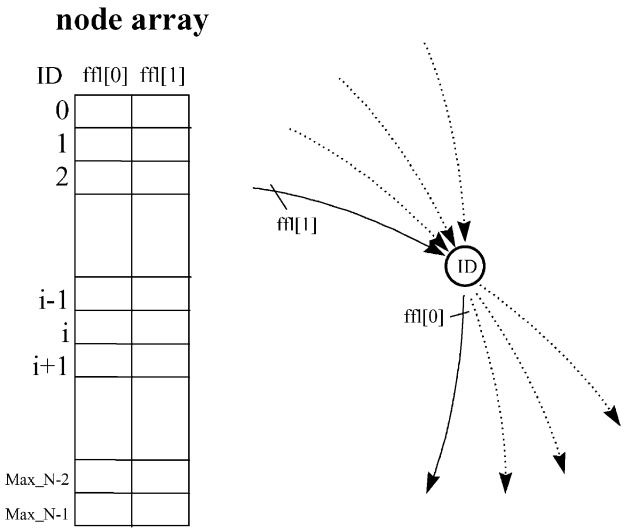

Fig. 4. Representation of the node structure.

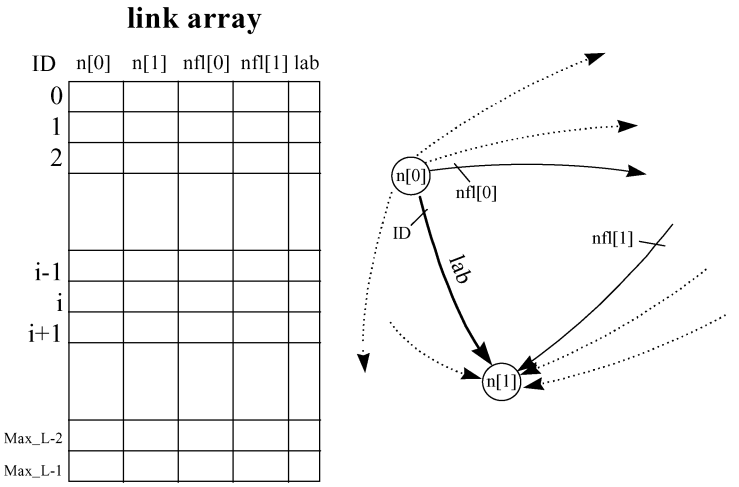

Fig. 5. Representation of the link structure.

(n[0] in Fig. 5), the node-ID of its destination node (n[1]), the link-ID of the next fan-out link (nfl[0]), the link-ID of the next fan-in link (nfl[1]) and the label of the link (lab). With this representation all the fan-in and fan-out lists could be easily traced through the link array.

The algorithm has been tested for correctness and minimization efficiency using a lexicon of 230,000 Greek words in random order. The average word length in the

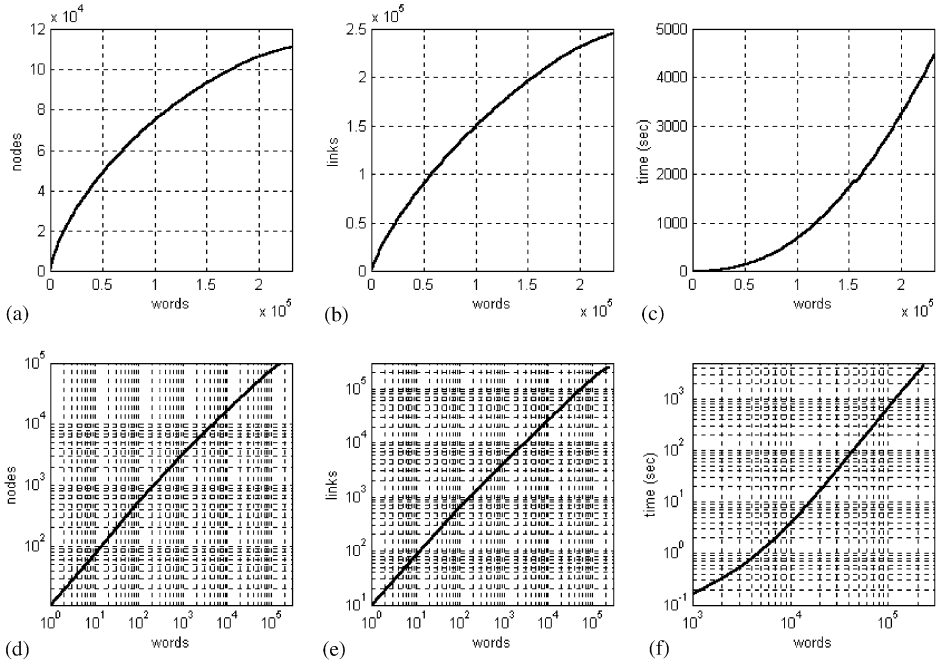

Fig. 6. Measurement of DAWG size and incremental construction time for 320,000 Greek words.

lexicon was 9.5 characters; the size of the alphabet was 36. Measurements for DAWG size and incremental construction time were performed. The experiment was performed on a Pentium-200MHz PC. The results are shown in Fig. 6: Figs. 6a, 6b, and 6c display, respectively, the number of nodes, the number of links and the time (in seconds) required to build the corresponding minimal DAWG, in respect to the size of the lexicon (number of words). The same results are also shown in logarithmic scales in Figs. 6d–f. The slope of the time line in Fig. 6f indicates an $O(n^2)$ time complexity, a result in agreement with the complexity calculation of Section 5. The slopes of the node (Fig. 6d) and link (Fig. 6e) lines indicate an $O(n)$ DAWG size, which also agrees with theory [6,9].

7. Conclusion

We have presented an insertion algorithm for adding words (strings) in deterministic DAWGs and provided some interesting lemmas resulting to a proof for its optimal behavior, i.e. if this algorithm is applied to a minimal deterministic DAWG it will not damage its minimality. This optimal insertion algorithm provides an efficient and elegant way to update minimal deterministic DAWGs without the need to perform a minimization operation afterwards. However, the proposed algorithm does not require a minimal original DAWG in order to work properly. It can add words to non-minimal

deterministic DAWGs as well, but the updated DAWGs are not guaranteed to be minimal.

The optimal insertion algorithm does not traverse every node in the DAWG but only a limited number of nodes, in the neighborhood of the path of the inserted word. Thus, adding a word with the proposed algorithm is faster than adding it in the classic way (i.e. put the word in the graph and perform a minimization process afterwards), because every minimization process needs to access every node in the graph. The time required for the insertion of a new word was shown to be $O(n)$, in respect to the size of the DAWG. Moreover, if we use the algorithm repeatedly starting from an empty DAWG we can incrementally build minimal deterministic DAWGs. This process requires $O(n^2)$ time for n words and (although it is not time-efficient) it could be handy if the size of the intermediate structure (the one to be minimized) is crucial or if part of the structure has been already constructed. The behavior of the algorithm has also been tested experimentally using a lexicon of 230,000 words. The results have verified the theoretical expectations.

Apart from its theoretic interest, this algorithm has direct practical uses. On-line word insertion is highly desirable in every application where the data need to be updated regularly. Thus the proposed algorithm can be used to update the lexicon of a spell-checker on-line, or it can be used to add records in a DAWG-structured database. We have already used this algorithm to build morphological lexicons [22] and we are currently working on its incorporation to an experimental Web-Agent who will be able to automatically create lexical resources from the Internet [21].

References

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

[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1988.

[3] J. Aoe, K. Morimoto, M. Hase, An algorithm for compressing common suffixes used in trie structures, Systems and Computers in Japan, 24(12) (1993) 31–42 (translated from Trans. IEICE J75-D-II(4) (1992) 770–779).

[4] J. Aoe, K. Morimoto, M. Shishibori, K. Park, A trie compaction algorithm for a large set of keys, IEEE Trans. Knowledge Data Eng. 8 (3) (1996) 476–491.

[5] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell, Building the minimal DFA for the set of all subwords of a word online in linear time, in: J. Paredaens (Ed.), Lecture Notes in Computer Science, Vol. 172, Springer, Belgium, 1984, pp. 109–118.

[6] A. Blumer, D. Haussler, A. Ehrenfeucht, Average sizes of suffix trees and DAWGs, Discrete Appl. Math. 24 (1989) 37–45.

[7] M. Ciura, S. Deorowicz, Experimental study of finite automata storing static lexicons, Report BW-453/RAu-2/99, 1999, also at http://www-zo.iinf.polsl.gliwice.pl/~sdeor/pub.htm

[8] M. Ciura, S. Deorowicz, How to squeeze a lexicon, Software - Practice & Experience 31 (11) (2001) 1077–1090.

[9] M. Crochemore, R. Verin, Direct construction of compact directed acyclic word graphs, Proc. Eighth Annual Symp. CPM 97, Aarhus, Denmark, June 30–July 2, 1997, pp. 116–129.

[10] J. Daciuk, S. Mihov, B. Watson, R. Watson, Incremental construction of minimal acyclic finite state automata, Computational Linguistics 26 (1) (2000) 3–16.

[11] J. Daciuk, R.E. Watson, B.W. Watson, Incremental construction of acyclic finite-state automata and transducers, Proc. Finite State Methods in Natural Language Processing, Bilkent University, Ankara, Turkey, June 29–July 1, 1998.
[12] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[13] R. Lacouture, R. De Mori, Lexical tree compression, EuroSpeech '91, Proc. 2nd European Conf. on Speech Communications and Techniques, Genova, Italy, September 24–26, 1991, pp. 581–584.
[14] S. Mihov, Direct Building of Minimal Automaton for a Given List, Annuaire de l' Universite de Sofia 'St. Kl. Ohridski', Vol. 91, livre 1, Faculte de Mathematique et Informatique, Sofia, Bulgaria, 1998.
[15] S. Mihov, Direct Construction of Minimal Acyclic Finite States Automata, Annuaire de l' Universite de Sofia 'St. Kl. Ohridski', Vol. 92, livre 2, Faculte de Mathematique et Informatique, Sofia, Bulgaria, 1998.
[16] K. Park, J. Aoe, K. Morimoto, M. Shishibori, An algorithm for dynamic processing of DAWG's, International J. Comput. Math. 54 (1994) 155–173.
[17] D. Perrin, Finite automata, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A, Elsevier, Amsterdam, 1990, pp. 3–57.
[18] D. Revuz, Minimization of acyclic deterministic automata in linear time, Theoret. Comput. Sci. 92 (1992) 181–189.
[19] D. Revuz, Dynamic acyclic minimal automaton, CIAA 2000, Proc. 5th Internat. Conf. on Implementation and Application of Automata, London, Canada, 2000, pp. 226–232.
[20] K. Sgarbas, N. Fakotakis, G. Kokkinakis, Two algorithms for incremental construction of directed acyclic word graphs, Internat. J. Artificial Intelligence Tools 4 (3) (1995) 369–381.
[21] K. Sgarbas, N. Fakotakis, G. Kokkinakis, WATCHER: an intelligent agent for automatic extraction of language resources from the Internet, Proc. ELSNET, in Wonderland. The Netherlands, March 25–27, 1998, pp. 100–104.
[22] K. Sgarbas, N. Fakotakis, G. Kokkinakis, A straightforward approach to morphological analysis and synthesis, Proc. COMLEX 2000, workshop on Computational Lexicography and Multimedia Dictionaries, Kato Achaia, Greece, September 22–23, 2000, pp. 31–34, also at http://slt.wcl.ee.upatras.gr/Sgarbas/PublAbsEN.htm
[23] D. Wood, Data Structures, Algorithms, and Performance, Addison-Wesley, Reading, MA, 1993.