Recognizing Leveled-Planar Dags in Linear Time*

Lenwood S. Heath** and Sriram V. Pemmaraju***

1 Introduction

Let $G = (V, E)$ be a directed acyclic graph (dag). A *leveling* of G is a function $lev : V \to \mathbf{Z}$ mapping the nodes of G to integers such that $lev(v) = lev(u) + 1$ for all $(u, v) \in E$. G is a *leveled dag* if it has a leveling. If $lev(v) = j$, then v is a *level-j node.* Let E_j denote the set of arcs in E from level-j nodes to level-$(j + 1)$ nodes. Without loss of generality, we may assume that the image of lev is $\{1, 2, \ldots, m\}$ for some m. Let $V_j = lev^{-1}(j)$ denote the set of level-j nodes. Each V_j is a *level* of G. The leveling partitions V into the levels $V_1, V_2, \ldots, V_m$, and according we denote G as $G = (V_1, V_2, \ldots, V_m; E)$.

Let ℓ_j denote the vertical line in the Cartesian plane $\ell_j = \{(j, y) \mid y \in \mathbf{R}\}$, where $\mathbf{R}$ is the set of reals. Suppose G has a planar embedding in which all nodes in V_j are placed on ℓ_j and each arc in E_j, where $1 \leq j < m$, is drawn as a straight line segment between lines ℓ_j and ℓ_{j+1}. Then this planar embedding is called a *directed leveled-planar embedding* of G. Figure 1 shows a directed leveled-planar embedding of a dag. A dag is called a *leveled-planar dag* if it has a directed leveled-planar embedding.

In this paper we present a linear time algorithm for the problem of determining if a given dag has a directed leveled-planar embedding. Our algorithm uses a variation of the PQ-tree data structure introduced by Booth and Lueker [2]. One motivation for our algorithm is that it can be extended to recognize 1-queue dags, thus answering an open question in [6]. Combinatorial and algorithmic results related to queue layouts of dags and posets can be found in [4, 7, 5]. Our algorithms also contrasts leveled-planar undirected graphs and leveled-planar dags, since the problem of recognizing leveled-planar graphs has been shown to be NP-complete by Heath and Rosenberg [8]. Another motivation comes from the importance of the above problem in the area of graph drawing. Our result extends the work of Di Battista and Nardelli [1], Chandramouli and Diwan [3], and Hutton and Lubiw [9]. These authors assume solve the problem assuming certain restrictions on the given dag and leave the general problem open.

The organization of the rest of the paper is as follows. Section 2 discusses the nature of the problem and outlines our approach. Section 3 defines the data

* This research was partially supported by National Science Foundation Grant CCR-9009953.

** Department of Computer Science, Virginia Tech, Blacksburg, VA 24061-0106, heath@cs.vt.edu.

*** Department of Computer Science, University of Iowa, Iowa City, IA 52242-1316, sriram@cs.uiowa.edu.

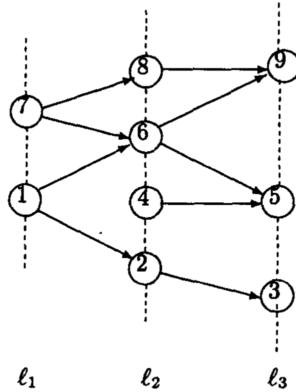

Fig. 1. A leveled-planar dag.

structures (PQ-trees and collections) that we need to represent sets of permutations of nodes in a particular level. Section 4 defines the operations we use to restrict or combine sets of permutations. Section 5 presents our linear time algorithm for recognizing leveled-planar dags.

2 The Problem

It is easy to check whether a dag is leveled in linear time. Therefore, without loss of generality, we may assume that $G = (V_1, V_2, \ldots, V_m; E)$ is a connected, leveled dag, and we wish to determine whether G has a directed leveled-planar embedding.

Suppose G has a directed leveled-planar embedding $\mathcal{E}$. For each j, where $1 \leq j \leq m$, $\mathcal{E}$ determines a total order $\leq_j$ on V_j given by the bottom to top order of the nodes on ℓ_j. Conversely, if a total order $\leq_j$ on V_j is given for each j, then it is easy to check whether those total orders witness a directed leveled-planar embedding of G. It suffices to check that there are no two arcs (u, v) and (x, y) such that $lev(u) = lev(x) = j$, $u <_j x$, and $y <_{j+1} v$. In Figure 1, the total orders are given by $1 <_1 7$, $2 <_2 4 <_2 6 <_2 8$, and $3 <_3 5 <_3 9$.

The problem of recognizing whether a connected leveled dag G is a leveled-planar dag is then equivalent to determining whether there are total orders on all the levels that are witness to a leveled-planar embedding of G. Let G_j denote the subgraph of G induced by $V_1 \cup V_2 \cup \cdots \cup V_j$. (Note that, unlike G, G_j is not necessarily connected.) Each total order on V_j can be thought of as a permutation on V_j. Moreover, for each j, there is a set of permutations Π_j that contains exactly the total orders on V_j that occur in witnesses to directed leveled-planar embeddings of G_j. So to recognize whether G is a leveled-planar dag, we need only compute Π_m and check that it is nonempty. Our basic approach to doing this efficiently is to perform a left-to-right sweep processing the levels in

the order $V_1, V_2, \ldots, V_m$. For each level V_j, we say that a permutation π of the nodes in V_j is a *witness* to a directed leveled-planar embedding of G_j if the nodes in V_j appear in a bottom to top order on line ℓ_j according to π in some directed leveled-planar embedding of G_j. For each level V_j, the algorithm constructs a representation of all permutations on V_j that are witness to some directed leveled-planar embedding of G_j. The data structure that we use for maintaining sets of permutations is called a *collection.* So after processing $V_1, V_2, \ldots, V_j$, we have a collection C_j. The algorithm then processes V_{j+1} and uses C_j to construct the next collection C_{j+1}.

3 PQ-trees and Collections

In order to define the collection data structure precisely, we need the PQ-tree data structure of Booth and Lueker [2] to represent sets of permutations. A PQ-tree T for a set S is a rooted tree that contains three types of nodes: leaves, P-nodes, and Q-nodes. The leaves in T are in one-one correspondence with the elements of S. The set S is called the *yield* of T, denoted YIELD(T). The PQ-tree T represents permutations of YIELD(T) according to the following rules: (a) The children of a P-node may be permuted arbitrarily, (b) The children of a Q-node must occur in the given order or in the reverse order. As a special case, the empty PQ-tree ϵ represents the empty set of permutations. The set of permutations represented by T is denoted by PERM(T). The *yield* YIELD(r) of a node r in T is the yield of the subtree rooted at r. Without loss of generality, we may assume that every P-node has 3 or more children and that every Q-node has 2 or more children. A *collection* is a finite set of PQ-trees with pairwise disjoint yields.

For any PQ-tree T and connected leveled-planar dag F with k levels, for some $k \geq 0$, we say that T *represents* F if and only if PERM(T) is the set of all permutations of the level-k nodes in F that witness some leveled-planar embedding of F. For each level V_j, our algorithm maintains a collection C_j satisfying the property stated in the following theorem.

Theorem 1. *For each j, $1 \leq j \leq m$, and for each connected component F in G_j, there is a corresponding PQ-tree $T[F]$ in C_j that represents F.*

Since $G = G_m$ is connected, the above theorem implies that C_m contains a single PQ-tree $T[G]$, that represents G. So C_m contains a non-empty PQ-tree if and only if G has a directed leveled-planar embedding. Thus the goal of our algorithm is to compute C_m. The proof of Theorem 1 is inductively established in the following description of the algorithm.

The algorithm initializes $C_1 = \{\text{leaf } v \mid v \in V_1\}$. Thus for $j = 1$ (the base case), the correspondence claimed in Theorem 1 is trivially true. The algorithm then proceeds to inductively construct $C_2, C_3, \ldots, C_m$ in that order. As an inductive hypothesis, we assume that Theorem 1 holds for some $j \geq 1$.

In order to construct C_{j+1} from C_j, we assume that some information is maintained in each non-leaf node of a PQ-tree in C_j and one additional piece of

information is maintained at the root of a PQ-tree in C_j. Let F be any connected component of G_j. By the inductive hypothesis, $T[F]$ is the PQ-tree in C_j that represents F. For any subset S of the set of nodes in V_j that belong to F, define MEETLEVEL(S) to be the greatest $d \leq j$ such that $V_d, \ldots, V_j$ induces a dag in which all nodes of S occur in the same connected component. For example, in Figure 1, MEETLEVEL($\{3,5\}$) = 1 and MEETLEVEL($\{5,9\}$) = 2. Note that if $|S| > 1$, then MEETLEVEL(S) $< j$. For a Q-node q in $T[F]$ with ordered children $r_1, r_2, \ldots, r_t$, maintain in node q integers denoted ML(r_i, r_{i+1}), where $1 \leq i < t$, that satisfy ML(r_i, r_{i+1}) = MEETLEVEL(YIELD(r_i)$\cup$YIELD(r_{i+1})). For a P-node p in $T[F]$, maintain in node p a single integer denoted ML(p) that satisfies

$$\text{ML}(p) = \text{MEETLEVEL}(yield(p)).$$

Let S be any subset of the set of nodes in V_j that belong to F. Now define LEFTLEVEL(S) to be the smallest d such that F contains a node in V_d. We always have LEFTLEVEL(S) $\leq$ MEETLEVEL(S) and inequality is possible. At the root of $T[F]$, maintain a single integer denoted LL($T[F]$) satisfying

$$\text{LL}(T[F]) = \text{LEFTLEVEL}(\text{YIELD}(T[F])).$$

When our algorithm computes the collection C_{j+1} from C_j, it also maintains the values of ML and LL in the PQ-trees in C_{j+1}. Note that since every PQ-tree in C_1 is a leaf, ML values are not defined, while LL(T) = 1 for each tree $T \in C_1$.

4 Operations

We have described our data structure and now describe two simple operations on PQ-trees that serve as building blocks of the algorithm that constructs C_{j+1} from C_j.

1. ISOLATE(T, x), where T is a PQ-tree and $x \in$ YIELD(T). This operation returns a PQ-tree T' such that (a) The root of T' is a Q-node with x as its first or last child. (b) PERM(T') is the subset of permutations in PERM(T) in which x is either the first or the last element. If YIELD(T) = $\{x\}$, then ISOLATE(T, x) returns a PQ-tree that is just the single leaf x. If there is no permutation in PERM(T) that has x as its first or last element, then ISOLATE(T, x) returns ϵ.
2. IDENTIFY(T, x, y, z), where T is a PQ-tree, $x, y \in$ YIELD(T), $x \neq y$, and $z \notin$ YIELD(T). Let P be the subset of permutations in PERM(T) in which x and y appear consecutively. Let P' be obtained from P as follows: If P contains the permutation $a, \ldots, b, x, y, c, \ldots, d$, then put in P' the permutation $a, \ldots, b, z, c, \ldots, d$, obtained by replacing x, y by z. The operation IDENTIFY(T, x, y, z) returns a PQ-tree T' such that PERM(T') = P'. Note that P' may be empty, in which case $T' = \epsilon$.

The operation ISOLATE(T, x) is a special case of Booth and Lueker's REDUCE operation and can be implemented as follows. Let r be the root of T. If

$x = r$, then ISOLATE(T, x) simply returns T. Otherwise, there are two cases based on whether x is a child of r or not.

Base Case: x is a child of r. If r is a Q-node and x is not its first or last child, then there are no permutations in PERM(T) with x at the end or at the beginning, so the operation returns ϵ. If r is a Q-node and x is either the first or the last child of r, then nothing needs to be done and the operation simply returns T. If r is a P-node, then T is transformed as shown in Figure 2.

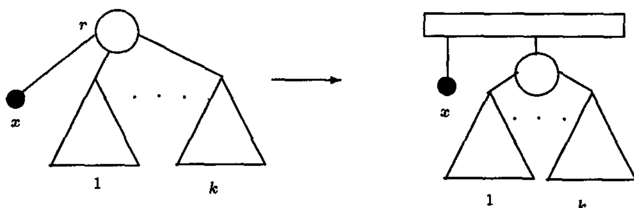

Fig. 2. The transformation of T in the base case of ISOLATE(T, x).

Inductive Case: x is not a child of r. Let T' be the subtree rooted at a child of r whose yield contains x. Let $T'' =$ ISOLATE(T', x). If $T'' = \epsilon$, then ISOLATE(T, x) also returns ϵ. Otherwise, replace T' by T''. The root of T'' is a Q-node with x as either its first or its last child. If r is a P-node, perform the transformation on T shown in Figure 3. If r is a Q-node and T'' is not the

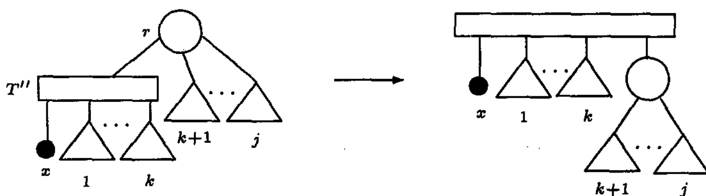

Fig. 3. The transformation of T in the inductive case of ISOLATE(T, x) when r is a P-node.

first or the last child of r, then the algorithm returns ϵ; otherwise, perform the transformation on T shown in Figure 4. The running time of ISOLATE(T, x) is proportional to the depth of x in T.

The operation IDENTIFY(T, x, y, z) can be implemented in the following four steps.

Step 1. Locate r, the node in T that is the least common ancestor of x and y.

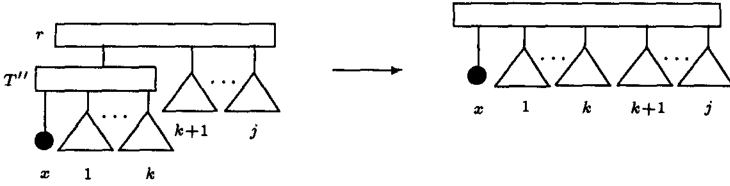

Fig. 4. The transformation of T in the inductive case of ISOLATE(T, x) when r is a Q-node.

Step 2. Let T_1 and T_2 be subtrees of T rooted at a children of r such that $x \in$ YIELD(T_1) and $y \in$ YIELD(T_2). Let $T_1' =$ ISOLATE(T_1, x) and $T_2' =$ ISOLATE(T_2, y). If either $T_1' = \epsilon$ or $T_2' = \epsilon$, then IDENTIFY(T, x, y, z) returns ϵ. Otherwise, replace T_1 by T_1' and T_2 by T_2'. The root of T_1' (respectively, T_2') is a Q-node with x (respectively, y) being the first or the last child of the root.

Step 3. This step depends on whether r is a P-node or a Q-node.

(a) r is a P-node. T is transformed as shown in Figure 5.

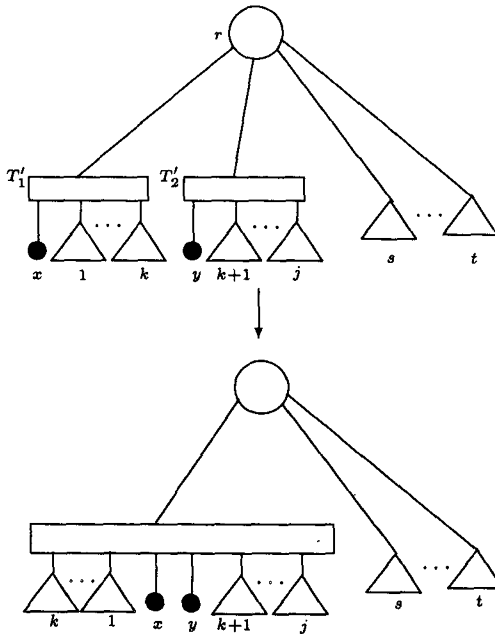

Fig. 5. The transformation of T in IDENTIFY(T, x, y, z) when r is a P-node.

(b) r is a Q-node. If the subtrees T_1 and T_2 are not adjacent children of r, then the operation returns ϵ. Otherwise, T is transformed as shown in Figure 6.

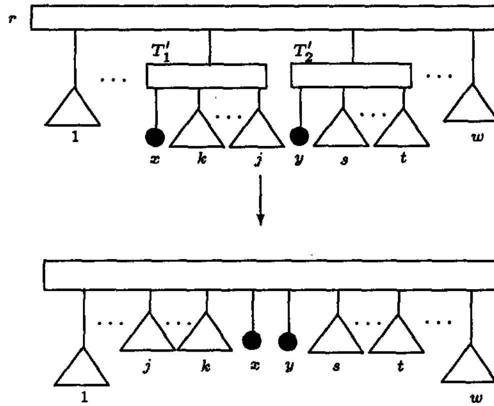

Fig. 6. The transformation of T in IDENTIFY(T, x, y, z) when r is a Q-node.

Step 4. Leaf z replaces leaves x and y.
The running time of IDENTIFY(T, x, y, z) is proportional to the sum of the depths of x and y in T. In the transformations described above we have ignored several special cases caused by the fact that a transformation might lead to the birth of a P-node with two children. Instead of dealing with these cases separately, we simply note that whenever this happens, the P-node is replaced by a Q-node. The operations ISOLATE and IDENTIFY also update ML and LL values. Details are omitted due to lack of space.

5 Recognizing Leveled-Planar Dags

We are now ready to describe how our algorithm constructs C_{j+1} from C_j. To understand the intuition behind the construction, imagine that through a sequence of simple operations (to be described later) dag G_j is transformed into dag G_{j+1}. This yields a sequence of dags $H_1, H_2, \ldots, H_k$ with $G_j = H_1$ and $G_{j+1} = H_k$. Correspondingly, collection C_j is transformed into collection C_{j+1} via a sequence of operations that mimic those applied to H_s, $1 \leq s < k$. This yields a sequence of collections $D_1, D_2, \ldots, D_k$, where $C_j = D_1$ and $C_{j+1} = D_k$. In what follows, we show that the operation applied to D_s, for each s, $1 \leq s < k$, mimics the operation applied to H_s in such a way that the correspondence between G_j and C_j, claimed in the induction hypothesis, also exists between G_{j+1} and C_{j+1}.

We use the following three operations to transform G_j into G_{j+1}:

```
/* Algorithm for transforming dag G_j into dag G_{j+1} */
H:=G_j;

/* GROWTH PHASE */
for all connected components F in H do
        for all level-j nodes u in F do
                N_u := {v[u] | (u,v) ∈ E_j};
                Replace F in H by GROW(F, u, N_u);

/* MERGE PHASE */
for all pairs of level-(j+1) nodes (v[X], v[Y]) in H do
        if v[X] and v[Y] belong to the same connected component F then
                Replace F in H by MERGE1(F, v[X], v[Y], v[X,Y])
        else if v[X] and v[Y] belong to components F1 and F2 then
                Replace F1 and F2 in H by MERGE2(F1, F2, v[X], v[Y], v[X,Y]);

/* CLEANUP PHASE */
Relabel each node v[X] in H as v;
Add all the level-(j+1) sources in G to H;

G_{j+1}:=H;
```

Fig. 7. Transforming G_j into G_{j+1}

1. GROW(F, u, S), where F is a connected, leveled-planar dag with k or $k+1$ levels, for some $k \geq 0$, u is a level-j node in F, and S is a set of nodes not in F. The operation returns the dag obtained by adding arcs (u, v) for all $v \in S$, to F.
2. MERGE1(F, u, v, w), where F is a connected, leveled-planar dag with k levels, for some $k \geq 0$, u and v are distinct level-$(j+1)$ nodes in F, and w is a node not in F. The operation returns the dag obtained by identifying nodes u and v in F and replacing the resulting node by w.
3. MERGE2(F_1, F_2, u, v, w), where F_1 and F_2 are connected, leveled-planar dags, each with k levels, for some $k \geq 0$, u is a level-$(j+1)$ node in F_1 and v is a level-$(j+1)$ node in F_2, and w is a node not in F_1 or F_2. The operation returns the dag obtained by identifying nodes u and v in F and replacing the resulting node by w.

Figure 7 shows the algorithm that uses the above operations to transform G_j into G_{j+1}. In this algorithm, H is initialized to G_j and then transformed into G_{j+1} by repeatedly applying the three operations described above in three *phases.* In the GROWTH PHASE, to each level-j node in H with outdegree equal to d, d out-neighbors are attached. Thus after the GROWTH PHASE, each level-$(j+1)$ node v in G with p in-neighbors is represented by p copies in H, each copy having in-degree equal to 1. Note that these p copies have labels $v[u_1], v[u_2], \ldots v[u_p]$,

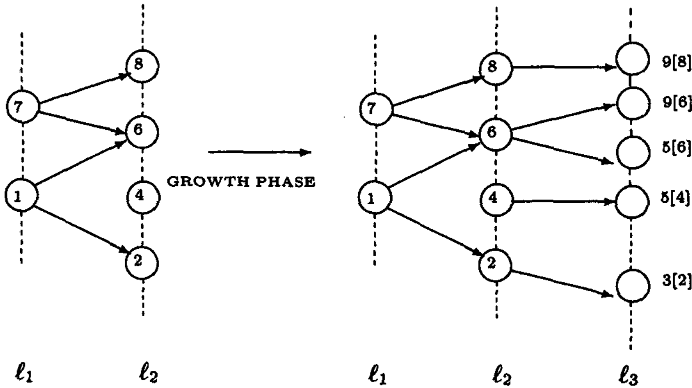

Fig. 8. Illustration of the GROWTH PHASE.

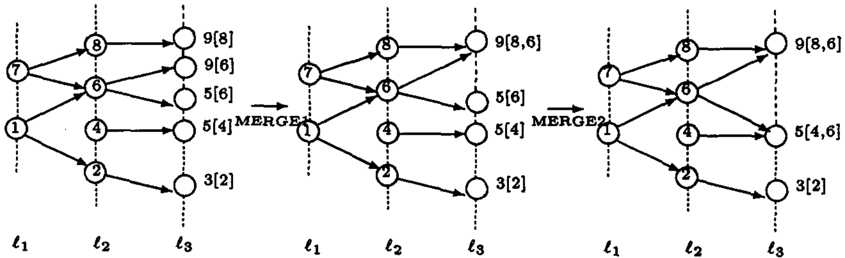

Fig. 9. Illustration of the MERGE PHASE.

where $u_1, u_2, \ldots, u_p$ are the p in-neighbors of v in G. Figure 8 illustrates the GROWTH PHASE applied to G_2, the subgraph of the dag in Figure 1, induced by the first two levels.

We extend the $v[u]$ notation to include $v[X]$, where X is a sequence of distinct level-j nodes that are all adjacent to v; we think of X as a set of nodes, while $v[X]$ is a single level-$(j+1)$ node that is adjacent to each $u \in X$. In the MERGE PHASE, each pair of level-$(j+1)$ nodes $(v[X], v[Y])$, where $X \cap Y = \emptyset$, is merged into a single node with label $v[X, Y]$. Note that $v[X]$ and $v[Y]$ may belong to the same connected component or to different connected components and accordingly the operations MERGE1 or MERGE2 are used. Figure 9 illustrates the MERGE PHASE applied to the dag resulting from the GROWTH PHASE shown in Figure 8. In the CLEANUP PHASE, each level-$(j+1)$ node in H with label $v[X]$ is relabeled v so as to match its name in G_{j+1}. For example, the dag obtained after the MERGE PHASE in Figure 9 contains nodes 9[8, 6], 5[4, 6], and 3[2] which are relabeled 9, 5, and 3 in the CLEANUP PHASE. Finally level-$(j+1)$ sources in G are added to H. This completes the transformation of H into G_{j+1}.

Having described the sequence of operations that transforms G_j into G_{j+1}, we now describe the parallel sequence of operations that transforms C_j into C_{j+1}. As mentioned earlier, the operations on collections closely mimic the operations on dags and so corresponding to each of the operations GROW, MERGE1, and MERGE2 that operate on connected leveled-planar dags, we define the following three operations that operate on PQ-trees:

1. GROW(T, u, S), where T is a PQ-tree, $u \in$ YIELD(T), and $S \cap$YIELD$(T) = \emptyset$.
2. MERGE1(T, u, v, w), where T is a PQ-tree, $u, v \in$ YIELD(T), and $w \notin$ YIELD(T).
3. MERGE2(T_1, T_2, u, v, w), where T_1 and T_2 are PQ-trees, $u \in$ YIELD(T_1), $v \in$ YIELD(T_2), and $w \notin$ YIELD$(T_1) \cup$ YIELD(T_2).

Each of these operation returns a PQ-tree. To see how these operations are applied to collections, let us suppose that a collection D is initialized to C_j. Imagine that when F is replaced by GROW(F, u, N_u) in H in the GROWTH PHASE, the corresponding PQ-tree $T[F]$ is replaced by GROW$(T[F], u, N_u)$ in D. Similarly, in the MERGE PHASE, when F is replaced by MERGE1$(F, v[X], v[Y], v[X, Y])$, then the corresponding PQ-tree $T[F]$ is replaced by the PQ-tree MERGE1$(T[F], v[X], v[Y], v[X, Y])$. Finally, when F_1 and F_2 are replaced by the dag

$$\text{MERGE2}(F_1, F_2, v[X], v[Y], v[X, Y]),$$

then the corresponding PQ-trees $T[F_1]$ and $T[F_2]$ are replaced by

$$\text{MERGE2}(T[F_1], T[F_2], v[X], v[Y], v[X, Y]).$$

Corresponding to the relabeling of the level-$(j+1)$ nodes in H in the CLEANUP PHASE, all the leaves of trees in D are similarly relabeled and corresponding to the addition of the level-$(j+1)$ sources to H, PQ-trees that just contain a leaf are added to D. This completes the construction of C_{j+1}. Thus the transformation of C_j into C_{j+1} can also be viewed as proceeding in three distinct phases: GROWTH PHASE, MERGE PHASE, and CLEANUP PHASE. Our task now is to describe the three operations on PQ-trees mentioned above and to prove their correctness.

1. **GROW**(T, u, S). This operation returns a PQ-tree T' obtained from T as follows. If $S = \emptyset$, then T' is obtained by deleting u from T. Otherwise, let $S = \{v_1, v_2, \ldots, v_k\}$, for some $k \geq 1$. If $k = 1$, then T' is obtained by replacing u by the leaf v_1. If $k = 2$, then T' is obtained by replacing u by a Q-node q whose children are the leaves v_1 and v_2. Then the information ML$(v_1, v_2) = j$ is inserted at q. If $k > 2$, then T' is obtained by replacing u by a P-node p whose children are the leaves $v_1, v_2, \ldots, v_k$. Then the information ML$(p) = j$ is inserted at the new P-node. The operation leaves the LL value of T unchanged, that is, LL$(T') =$ LL(T). The correctness of GROW(T, u, S) is embodied in the following lemma, which follows from the discussion.

Lemma 2. *After the* GROWTH PHASE, *for each connected component F in H, there is a PQ-tree $T[F]$ in D that represents F.*

2. **MERGE1**(T, u, v, w). This operation simply calls IDENTIFY(T, u, v, w) and returns the PQ-tree obtained. The correctness of MERGE1(T, u, v, w) is embodied in the following lemma.

Lemma 3. *Suppose that F is a leveled-planar dag with j levels. Further suppose that T is a PQ-tree that represents F. If F' = MERGE1(F, u, v, w) and T' = MERGE1(T, u, v, w), then T' represents F.*

3. **MERGE2**(T_1, T_2, u, v, w). Without loss of generality, suppose that LL$(T_1) \leq$ LL(T_2). The trees T_1 and T_2 are merged in two steps. In the first step, the PQ-tree T_2 is attached to T_1 at an appropriate location. The resulting tree, T_3, contains the leaves u and v. In the second step, these two leaves in T_3 are identified into one leaf w using the operation IDENTIFY. The two steps are discussed in detail below.

Step 1. Attaching T_2 to T_1. Start with the leaf u in T_1 and proceed upward in T_1 until a node r' and its parent r are encountered such that:

1. r is a P-node with ML$(r) <$ LL(T_2). T_3 is obtained by attaching T_2 as a child of r in T_1.
2. r is a Q-node with ordered children $r_1, r_2, \ldots, r_t$, $r' = r_1$, and ML$(r_1, r_2) <$ LL(T_2). T_3 is obtained by replacing r_1 in T_1 with a Q-node q having two children, r_1 and the root of T_2. The case where $r' = r_t$ and ML$(r_{t-1}, r_t) <$ LL(T_2) is symmetric.
3. r is a Q-node with ordered children $r_1, r_2, \ldots, r_t$, $r' = r_i$, for some i satisfying $1 < i < t$, and both ML$(r_{i-1}, r_i) <$ LL(T_2) and ML$(r_i, r_{i+1}) <$ LL(T_2). T_3 is obtained by replacing r_i in T_1 with a Q-node q having two children, r_i and and root of T_2.
4. r is a Q-node with children $r_1, r_2, \ldots, r_t$, $r' = r_i$, $1 < i < t$, and

   $$\mathrm{ML}(r_{i-1}, r_i) < \mathrm{LL}(T_2) \leq \mathrm{ML}(r_i, r_{i+1}).$$

   T_3 is obtained by attaching T_2 as a child of r between r_{i-1} and r_i. The case where

   $$\mathrm{ML}(r_i, r_{i+1}) < \mathrm{LL}(T_2) \leq \mathrm{ML}(r_{i-1}, r_i)$$

   is symmetric.
5. r' is the root of T_1. In this case, construct T_3 by making its root a Q-node q with two children, r' and the root of T_2.

Step 2. Identifying u and v in T_3. Return IDENTIFY(T_3, u, v, w).
Steps 1 and 2 described above also update ML and LL values. Details are not presented due to lack of space. The following lemma establishes the correctness of MERGE2(T_1, T_2, u, v, w).

Lemma 4. *Suppose that F_1 and F_2 are connected leveled-planar dags and T_1 and T_2 are PQ-trees that represent F_1 and F_2 respectively. If $F =$ MERGE2(F_1, F_2, u, v, w) and $T =$ MERGE2(T_1, T_2, u, v, w), then T represents F.*

This completes the discussion of our algorithm and establishes this theorem.

Theorem 5. *A leveled-planar dag can be recognized in linear time.*

The time complexity follows from an amortized analysis that we sketch here. It suffices to show that the time complexity of all the MERGE1 and MERGE2 operations together is linear. Each arc in the dag G is allocated three credits; since G is planar, the total number of credits is linear. In MERGE1, there are two paths in the dag involved in forming a new face; since each arc is in two faces, two credits from each arc in the face pays for the MERGE1. In MERGE2, the work is proportional to the height of the shorter dag; there is always a path in the dag that has a credit on each arc. We conclude that the allocated credits are sufficient and that the time complexity is linear.

References

1. Giuseppe Di Battista and Enrico Nardelli. Hierarchies and planarity theory. *IEEE Transactions on Systems, Man, and Cybernetics*, 18:1035–1046, 1988.
2. Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.
3. Mahadevan Chandramouli and A. A. Diwan. Upward numbering testing for triconnected graphs (an extended abstract). Accepted at Graph Drawing 95.
4. Lenwood S. Heath and Sriram V. Pemmaraju. Stack and queue layouts of posets. Technical Report 93-06, University of Iowa, 1993. Submitted.
5. Lenwood S. Heath and Sriram V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. Technical Report 95-06, University of Iowa, 1995. Submitted.
6. Lenwood S. Heath, Sriram V. Pemmaraju, and Ann Trenk. Stack and queue layouts of directed acyclic graphs. In William T. Trotter, editor, *Planar Graphs*, pages 5–11, Providence, RI, 1993. American Mathematical Society.
7. Lenwood S. Heath, Sriram V. Pemmaraju, and Ann Trenk. Stack and queue layouts of directed acyclic graphs: Part I. Technical Report 95-03, University of Iowa, 1995. Submitted.
8. Lenwood S. Heath and Arnold L. Rosenberg. Laying out graphs using queues. *SIAM Journal on Computing*, 21(5):927–958, 1992.
9. Michael D. Hutton and Anna Lubiw. Upward planar drawing of single source acyclic digraphs. In *Proceedings of the 2nd Symposium on Discrete Algorithms*, pages 203–211, 1991.