Discrete Applied Mathematics 43 (1993) 261–295
North-Holland

A faster algorithm to recognize undirected path graphs

Alejandro A. Schäffer*

Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251, USA

Received 18 July 1988
Revised 7 June 1991

Abstract

Schäffer, A.A., A faster algorithm to recognize undirected path graphs, Discrete Applied Mathematics 43 (1993) 261–295.

Let $\mathcal{F}$ be a finite family of nonempty sets. The undirected graph G is called the *intersection graph* of $\mathcal{F}$ if there is a bijection between the members of $\mathcal{F}$ and the vertices of G such that any two sets F_i and F_j (for $i \neq j$) have a nonempty intersection if and only if the corresponding vertices are adjacent. We study intersection graphs where $\mathcal{F}$ is a family of undirected paths in an unrooted, undirected tree; these graphs are called (*undirected*) *path graphs*. They constitute a proper subclass of the chordal graphs. Gavril [Discrete Math. 23 (1978) 211–227] gave the first polynomial time algorithm to recognize undirected path graphs; his algorithm runs in time $O(n^4)$, where n is the number of vertices. The topic of this paper is a new recognition algorithm that runs in time $O(mn)$, where m is the number of edges.

1. Introduction

A simple undirected graph is *chordal* if every cycle contains an edge between two vertices that are not consecutive around the cycle. Chordal graphs arise in the study of sparse matrix computations [16, 17] and acyclic database schemes [1, 5] among other places. Some NP-complete problems can be solved in polynomial time if the input graph is chordal [6, 12, 20].

Correspondence to: Professor A.A. Schäffer, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251, USA.

* Most of this research was done while the author was a student in the Stanford University Computer Science Department and a Research Student Associate at IBM Almaden Research Center. At Stanford, the work of the author was primarily supported by a Fannie and John Hertz Foundation Fellowship and was supported in part by NSF grants DCR 83-20000 and CCR 87-04170. Revisions done at Rice were supported by NSF grant CCR 90-10534.

We present a new recognition algorithm for *undirected path graphs*, which comprise a proper subclass of chordal graphs. We denote the set of vertices of a graph G by V or $V(G)$ and its cardinality by n or n_G. We denote the set of edges by E or $E(G)$ and its cardinality by m or m_G. A *clique* is a *maximal* completely connected subgraph. We denote the set of cliques of G by $\mathscr{C}(G)$ or $\mathscr{C}$ and its cardinality by p or p_G. If C is a clique, the symbol C is shorthand for $V(C)$.

The class of undirected path graphs, which we define below, is motivated by an intersection graph characterization of chordal graphs due to Buneman and Gavril:

Theorem 1.1 [3, 7]. *A graph G is chordal if and only if there is an unrooted and undirected tree T and a family of subtrees $\mathscr{S}$ indexed by the vertices of G, such that subtrees S_v and S_w share a node of T if and only if the vertices v and w are adjacent. One can construct T so that there is a bijection between the nodes of T and $\mathscr{C}(G)$, where the subtree S_v is comprised of all nodes that correspond to cliques containing v.*

We say that T is a *clique tree* for G. We refer to points of G as *vertices* and to points of a clique tree as *nodes*. The intersection graph characterization in Theorem 1.1 was motivated by a subclass of chordal graphs called *interval graphs*. A graph G is an interval graph if there exists a set I of intervals on the real line and a bijection from $V(G)$ to I such that two vertices are adjacent if and only if the corresponding two intervals overlap. Gilmore and Hoffman give another characterization of interval graphs:

Theorem 1.2 [10]. *A graph is an interval graph if the set of cliques can be linearly ordered so that for any vertex v, the set of cliques containing v occur consecutively in the linear ordering.*

Thus a chordal graph is an interval graph if the clique tree described in Theorem 1.1 can be chosen as a path (all the subtrees will be subpaths). In view of the previous two theorems, some researchers have sought to identify classes of graphs with intersection graph characterizations that lie between chordal and interval graphs. Others have studied classes where the intersecting paths of subtrees share an edge and not just a node. In [13], Monma and Wei survey the previous work on such classes and characterize each of them in terms of separating cliques.

Two classes between chordal graphs and interval graphs are rooted directed path graphs, which Monma and Wei call RDV graphs, and undirected path graphs, which Monma and Wei call UV graphs. A chordal graph is an RDV graph if the tree model in Theorem 1.1 can be chosen so that: one node is chosen as the root, all tree edges are directed away from the root, and all subtrees are directed paths. As in the case of chordal graphs one can always construct the tree so that there is a bijection between the nodes of the tree and the cliques of the graph [8, 13].

A chordal graph is an *undirected path graph* or UV (for Undirected tree where

paths intersect in a Vertex) graph if the tree in Theorem 1.1 can be chosen so that all the subtrees are paths. Again, it is always possible to construct the tree so that nodes in the tree are in one-one correspondence with cliques of the graph [9, 13]. Interval graphs are RDV graphs, RDV graphs are UV graphs, and UV graphs are chordal graphs; all the containments are proper [13].

To study algorithms on chordal graphs, two bounds are important:

Remark 1.3 [11]. If G is chordal, then $p_G \leq n_G$.

Remark 1.4 [11]. If G is chordal, then $\sum_{C \in \mathscr{C}(G)} n_C \leq m_G + n_G$. Furthermore, the recognition algorithms for chordal graphs cited below can list the vertices in each clique in time $O(m+n)$.

Interval graphs can be recognized in $O(m+n)$ time by an algorithm of Booth and Lueker [2]. The recognition algorithm yields an interval model. Each clique can be made to correspond to a point on the real line; the ordering prescribed in Theorem 1.2 can be found by sorting the cliques from left to right in $O(m+n)$ time from the output of the recognition algorithm.

Chordal graphs can be recognized in time $O(m+n)$ using either an algorithm due to Rose, Tarjan and Lueker [18] or an algorithm due to Tarjan and Yannakakis [19]. Either recognition algorithm can be extended to an algorithm that also produces a clique tree in $O(m+n)$ time [14, pp. 59–60]. Dietz gives a very complicated $O(m+n)$ time algorithm to recognize RDV graphs and build a clique tree [4].

The problem of characterizing UV graphs is first raised by Renz [15], who attributes the problem to Klee and gives a combinatorial, nonalgorithmic characterization. Gavril shows how to recognize UV graphs and build a clique tree in $O(pn^3)$ steps (in his paper the bound is stated as $O(n^4)$, but the more precise bound follows from the analysis therein) [9]. In this paper we describe a more efficient and very different algorithm that runs in $O(p(m+n))$ steps.

2. Definitions and facts about clique trees

We initially test that the input graph G is chordal and connected in $O(m+n)$ time. If G is disconnected, each component is processed separately. If all components are UV graphs, one constructs a clique tree by joining the forest of trees (one per component) arbitrarily.

The foundation of the work of Monma and Wei [13] is the existence of cliques whose removal separates G. A clique C *separates* G if the removal of $V(C)$ and all incident edges leaves a nonempty and disconnected graph. If some upper-case letter, possibly with subscripts, for example C_j, denotes a clique, then the corresponding lower-case letter, c_j in this case, denotes the node representing C_j. Figure 1 shows a UV graph, G, we use to illustrate some definitions and parts of the algorithm. The

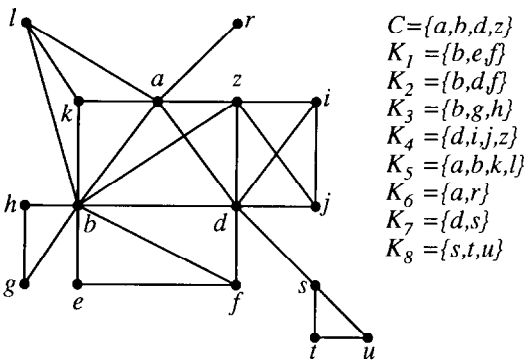

Fig. 1. An undirected path graph G.

graph G has nine cliques; one is the clique C that we use to separate the graph, and the others are $K_1, \ldots, K_8$. Figure 3 shows a clique tree for G.

Any chordal graph with more than two cliques has a separating clique [13]. We follow the notational convention of [13] that the removal of a separating clique C leaves s components induced by the vertex sets $V_1, V_2, \ldots, V_s$. Let G_i be the graph induced by $C \cup V_i$ for $1 \leq i \leq s$. In Fig. 1, the removal of $C = \{a, b, d, z\}$ leaves six components. Our numbering choice is $V_1 = \{e, f\}$, $V_2 = \{g, h\}$, $V_3 = \{i, j\}$, $V_4 = \{k, l\}$, $V_5 = \{r\}$, $V_6 = \{s, t, u\}$. We state some simple remarks about clique trees of chordal graphs. They are all used explicity or implicitly in [13].

Remark 2.1. Every clique of G_i is a clique of G. Every clique of G, except C, is contained in exactly one G_i and is a clique of that G_i. C is a clique of every G_i.

Remark 2.2. Let T be a clique tree for G. The set of nodes that represent cliques of G_i other than c induce a subtree in T.

Remark 2.3. The removal of clique C from a subgraph G_i does not separate G_i. In any clique tree for G_i or for a subgraph of G_i that contains $V(C)$, the node c is a leaf.

Remarks 2.2 and 2.3 suggest a recursive clique tree construction algorithm for chordal graphs. If G has one or two cliques, then its tree has one or two nodes. Otherwise, find a separating clique C, and recursively build a clique tree T_i for each separated subgraph G_i. In each T_i, the node c is a leaf, so join together the T_i trees by coalescing all copies of c into one node. If each T_i satisfies the subtree property of Theorem 1.1, then the merged tree satisfies it also because any vertex that occurs in more than one G_i occurs in C.

If the input graph is an undirected path graph, the above algorithm may build a clique tree in which the subtree of cliques containing vertex v is not a path. Our

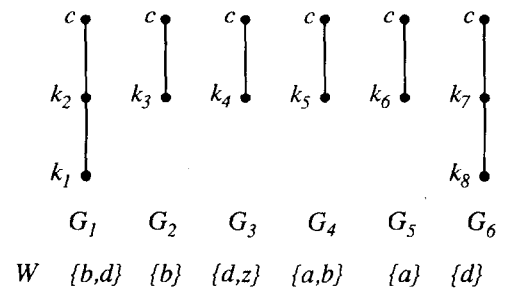

Fig. 2. Clique trees for the separated subgraphs corresponding to Fig. 1.

goal is to modify the algorithm to obtain trees that satisfy the path property. Instead of coalescing all copies of c, we look for other ways to connect the T_i trees to one another. By Remark 2.2, all nodes of T_i, except c, should stay together as a subtree, but we may be able to delete the copy of c in T_i and add an edge between one node of T_i and one node of another tree T_j.

From now on, we require anything we call a clique tree to satisfy the path property in the definition of undirected path graphs. A clique tree may be partially built and contain nodes for only some cliques of G. Figure 2 shows clique trees for each G_i graph corresponding to the graph G in Fig. 1. Figure 3, shows a clique tree satisfying the path property for the entire graph G. We will explain how our algorithm would construct the tree of Fig. 3 later.

We repeat a series of definitions from [13] that help us develop the new recognition algorithm. A clique is *relevant* if it has a vertex in common with C, but is not itself C. All the cliques in the graph G of Fig. 1 are relevant, except C and K_8. Two relevant cliques C_1 and C_2 are unattached, denoted $C_1 \mid C_2$, if there is no vertex that belongs to C, C_1, and C_2; otherwise, C_1 and C_2 are *attached* and we write $C_1 \bowtie C_2$. In G, for example, $K_2 \bowtie K_3$, while $K_2 \mid K_6$. Relevant clique C_1 dominates relevant clique C_2, written $C_1 \geq C_2$, if every vertex in $V(C) \cap V(C_2)$ is in C_1; relevant clique C_1 *properly dominates* relevant clique C_2, written $C_1 > C_2$, if C_1

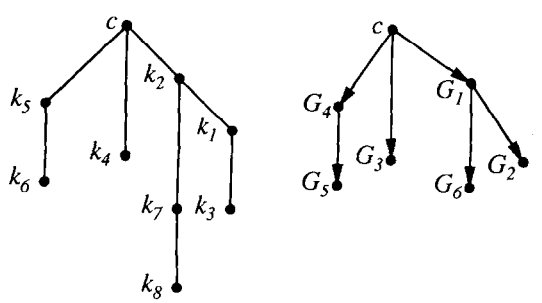

Fig. 3. A clique tree and its valid parent relation for the graph of Fig. 1.

dominates C_2 and there is a vertex that is in both C and C_1, but is not in C_2. In G, for example, $K_1 \geq K_3$ and $K_5 > K_3$. Both $\geq$ and $>$ are transitive.

Two relevant cliques C_1 and C_2 are *antipodal*, denoted $C_1 \leftrightarrow C_2$, if $C_1 \bowtie C_2$ but neither dominates the other. In G, for example, $K_5 \leftrightarrow K_2$. Monma and Wei proved that if $C_1 \leftrightarrow C_2$, then in any clique tree, the path between c_1 and c_2 passes through c.

These definitions are generalized for the subgraphs G_i. Two of these graphs, G_1 and G_2, are *unattached*, which we write $G_1 \mid G_2$, if $C_1 \mid C_2$ for every relevant clique $C_1 \in G_1$ and every relevant clique $C_2 \in G_2$. Otherwise, the two graphs are *attached* and we write $G_1 \bowtie G_2$. The graph G_1 *dominates* the graph G_2, written $G_1 \geq G_2$, if $G_1 \bowtie G_2$, and for each relevant clique $C_1 \in G_1$, one of the following two conditions holds:

(1) for each relevant clique $C_2 \in G_2$, the clique C_1 dominates C_2, or

(2) for each relevant clique $C_2 \in G_2$, the cliques C_1 and C_2 are unattached.

We say that G_1 properly dominates G_2, written $G_1 > G_2$, if G_1 dominates G_2, but G_2 does not dominate G_1. Finally, G_1 and G_2 are *antipodal*, written $G_1 \leftrightarrow G_2$, if $G_1 \bowtie G_2$, but neither dominates the other. Note that $G_1 \geq G_2$, $G_2 \geq G_1$, and $G_1 \leftrightarrow G_2$ each imply $G_1 \bowtie G_2$.

For any separated subgraph G_i, let $W(G_i)$ be the set of $v \in C$ such that there is a vertex $w \in V_i$ for which the edge $v - w \in E(G)$. Since $w \notin C$, there is a clique in G_i distinct from C containing $v - w$. Therefore, for distinct separated subgraphs G_1 and G_2, $G_1 \bowtie G_2$ if and only if $W(G_1) \cap W(G_2) \neq \emptyset$. Also, $G_1 \geq G_2$ if and only if $W(G_2) \subset W(G_1)$ and for every clique $C_1 \in G_1$, $W(G_2) \subset V(C_1) \cap W(G_1)$ or $V(C_1) \cap W(G_2) = \emptyset$. We use $\subset$ to denote proper containment or equality. Figure 2 shows the sets $W(G_i)$ for the graphs G_i corresponding to the graph G in Fig. 1; we defined the sets V_i above.

Lemma 2.4. *Let the clique C separate G as above. Assume that each separated graph G_i has a clique tree T_i. Let N_i be the set of nodes of $T_i \setminus \{c\}$. Let c_i be the node in N_i closest to c in T_i. Then $W(G_i) \subset V(C_i)$.*

Proof. Let $v \in W(G_i)$ be given. By definition of W, v is in C and in another clique of G_i. The path for v in T_i includes c and a node in N_i. The path for v must pass through c_i; thus $v \in V(C_i)$. Since v is arbitrary, $W(G_i) \subset V(C_i)$. □

Given a clique tree T_i for G_i and $C_1, C_2 \in \mathscr{C}(G_i)$, let $\pi(T_i, c_1, c_2)$ be the path from c_1 to c_2. We may drop the argument T_i. We write that $C_3 \in \mathscr{C}(G_i)$ is in $\pi(c_1, c_2)$ to mean $c_3 \in \pi(c_1, c_2)$.

3. Representing a clique tree

Let G be a UV graph with more than two cliques. Choose a separating clique C

and define the separated subgraphs G_i as in Section 2. Let T be any clique tree for G. By Remark 2.2, we can define a rooted tree structure on c and the separated graphs. Let c be the root. If a node $c_j \neq c$ representing a clique in G_j is adjacent to c, then c is the *tree parent* of G_j and G_j is a *tree child* of c. If a node $c_k \neq c$ that represents a clique in G_k is adjacent to a node $c_l \neq c$ that represents a clique in G_l and of the two nodes, c_k is closer to c, then G_k is the *tree parent* of G_l and G_l is a *tree child* of G_k. For example, in the tree in Fig. 3, G_4 is the parent of G_5 because k_5 representing a clique in G_4 is adjacent to k_6 representing a clique in G_5, and k_5 is closer to c.

We now relax the assumptions on G slightly. We still suppose that G has a separating clique C and the G_i subgraphs are known to be UV graphs, *but* G itself is not necessarily a UV graph. A binary relation $\mathscr{P}$ on the set $\bigcup_i \{G_i\} \cup \{c\}$ is a *valid parent relation* if it satisfies four conditions:

(1) The directed graph having an arc $X \rightarrow Y$ if $\mathscr{P}(X, Y)$ is a rooted, directed tree with root c.

(2) Whenever $\mathscr{P}(G_j, G_k)$, $G_j \geq G_k$.

(3) For any vertex $v \in V(G)$, there exist at most two distinct separated subgraphs, call one of them G_j, such that $\mathscr{P}(c, G_j)$ and $v \in W(G_j)$ (and similarly for the other subgraph).

(4) If for any i, j, k, we have $\mathscr{P}(G_i, G_j)$ and $\mathscr{P}(G_i, G_k)$, then $G_j \mid G_k$.

Given a valid parent relation $\mathscr{P}$, the *ancestor* relation is the reflexive-transitive closure of $\mathscr{P}$. The *descendant* relation contains exactly the same pairs as the ancestor relation, but the order is reversed.

Theorem 3.1 shows that the valid parent relations defined formally by the four conditions are the same as the tree parent relations we informally associated with specific clique trees. We prove the theorem after two preliminary lemmas. The proof of sufficiency shows how to combine *any* clique trees for the G_i graphs to form a clique tree for G. Figure 3 shows a valid parent relation for the graph of Fig. 1; it is also the tree parent relation for the tree in Fig. 3.

Theorem 3.1. *G is an undirected path graph if and only if it has a valid parent relation (with respect to any choice of separating clique C) or has at most two cliques.*

Lemma 3.2 [13]. *If $G_1 \leftrightarrow G_2$, then neither can be a tree ancestor of the other in any clique tree for G.*

Lemma 3.3. *If G_1 is a tree ancestor of G_2 in clique tree T for G, then $G_1 \geq G_2$, and $W(G_2) \subset W(G_1)$.*

Proof. As in the proof of Lemma 2.4, define c_1 to be the node representing a relevant clique in G_1 that is closest to c in T, and similarly define c_2 for G_2. By Lemma 2.4, $V(C_1) \cap V(C) = W(G_1)$ and $V(C_2) \cap V(C) = W(G_2)$. Let D_1 be any relevant clique in G_1. If $d_1 \in \pi(c_1, c_2)$, then by Remark 2.2, for any $v \in W(G_2)$, d_1 lies be-

tween c and a node c_2 representing a relevant clique C_2 in G_2 that contains v. Hence $v \in V(D_1)$; since v was arbitrary, D_1 dominates every relevant clique in G_2. If $d_1 \notin \pi(c_1, c_2)$, then for any $v \in W(G_2)$, the path for v cannot pass through D_1, so D_1 is unattached to every relevant clique in G_2. Thus $G_1 \geq G_2$. Furthermore, $C_1 \geq C_2$, which implies $W(G_2) \subset W(G_1)$. □

We can now prove Theorem 3.1.

Proof of Theorem 3.1. (only if) Let T be a clique tree, let $\mathscr{P}(T)$ be its tree parent relation. We prove that $\mathscr{P}(T)$ is valid. Every separated subgraph has a unique parent in $\mathscr{P}(T)$ and has c as tree ancestor. Furthermore c has no parent in $\mathscr{P}(T)$, so validity condition (1) is satisfied. By Lemma 3.3, condition (2) is satisfied. If condition (3) fails on vertex v, then cliques containing v will not induce a path. If condition (4) fails, say because $v \in W(G_j) \cap W(G_k)$, then we know that $v \in V(C)$ by definition. Then the path condition fails because it is impossible for a clique in G_j, a clique in G_k, a clique in the lowest common ancestor of G_j and G_k, and c to be simultaneously on a path.

(if) Let $\mathscr{P}$ be a valid parent relation. We show how to build a clique tree T for G that has tree parent relation $\mathscr{P}$ by using an arbitrary set of clique trees for the graphs G_i. By Remark 2.3, c is a leaf in each clique tree T_i for G_i. Start by setting $T := c$. Then, for any G_j that is a child of c, add $T_j \setminus c$ to T by inserting an edge in T between the node that is the neighbor of c in T_j and c. Now for any tree $T_j \setminus c$ that has already been connected, we can connect the trees for the children of G_j. Let G_k be a child of such a G_j. Let c_k be the neighbor of c in T_k; by Lemma 2.4, we know that $W(G_k) \subset V(C_k)$. By condition (2), we know that $T_j \setminus c$ has a node whose clique contains every vertex in $W(G_j)$. Let c_j' be such a node that is furthest away from c in T; this node is unique because of the path property. Connect $T_k \setminus c$ by adding an edge between c_j' and c_k.

By Remark 2.1, $V(T)$ is in one-one correspondence with $\mathscr{C}(G)$. The graph T is connected since one can show by induction that there is a path from every node to c. We started with one component for each T_i and one for c, and added one fewer edges than components, so T is a tree. Any vertex $w \notin V(C)$ is in only once G_i, so the path property for w is satisfied in T because it is satisfied in T_i. Now suppose $v \in C$. Let $A \in \mathscr{C}(G_i)$ be another clique containing v. When we connected $T_i \setminus c$ to T we checked that there was a path from a to a node not in G_i that is closer to c and whose clique contains v. Thus the set of nodes representing cliques that contain v induces a subtree. To prove it is a path, we observe that each node has degree at most 2 within the subtree. The node c has degree at most 2, by condition (3). Any node that is chosen as c_j' when $v \in C_k$ has degree at most 2 by condition (4). Any other node has degree at most 2 because the trees we started with satisfy the path property. □

In view of the important role played by c_j' above, we make the following definition. *If* $v \in W(G_i)$ and T_i is a clique tree for G_i, let $f(T_i, v)$ be the node of T_i

representing a clique containing v that is *furthest* from c. For example, in Figs. 2 and 3, $f(T_1, d) = k_2$ and $f(T_1, b) = k_1$.

In Section 4, we show how to find a valid parent relation, if one exists. That algorithm considers the graphs G_i in a special order $\mathcal{Q}$ produced by the routine SORT-G_i below. $\mathcal{Q}$ has the property that if G is a UV graph, there must be a clique tree T' such that c followed by $\mathcal{Q}$ is a topological order of the tree parent relation of T' if we choose C as the separating clique. We derive an algorithm to compute such an ordering from the following lemma.

For any separated subgraph G_i, let $X(G_i)$ be the set of vertices, $v \in C$, such that v belongs to a relevant clique C_i of G_i with the additional property that $C_i \cap C \neq W(G_i)$.

Lemma 3.4. *Suppose $W(G_1) = W(G_2)$, where G_1 and G_2 are separated subgraphs. Let $\mathcal{P}$ be a valid parent relation.*

(1) *If $X(G_1) \neq \emptyset$ and $X(G_2) \neq \emptyset$, then $G_1 \leftrightarrow G_2$ and neither is an ancestor of the other in $\mathcal{P}$.*

(2) *If $X(G_1) = \emptyset$, but $X(G_2) \neq \emptyset$, then G_2 does not dominate G_1,* and *G_2 is not an ancestor of G_1 in $\mathcal{P}$.*

(3) *If $X(G_1) = X(G_2) = \emptyset$, and G_2 is not a child of G_1, we can change $\mathcal{P}$, so that G_2 becomes a child of G_1, without destroying validity.*

Proof. (1) By Lemma 2.4, we can choose a clique $C_{11} \neq C$ in G_1 and a clique $C_{21} \neq C$ in G_2 such that $C_{11} \cap C = W(G_1) = W(G_2) = C_{21} \cap C$. Since $X(G_1)$ and $X(G_2)$ are nonempty, we can choose cliques C_{12} in G_1 and C_{22} in G_2 such that $C_{12} \cap C$ and $C_{22} \cap C$ are proper nonempty subsets of $W(G_1) = W(G_2)$. By construction, $C_{11} \bowtie C_{22}$, and $C_{11} > C_{22}$. Similarly, $C_{21} \bowtie C_{12}$, and $C_{21} > C_{12}$. Therefore $G_1 \leftrightarrow G_2$. By validity rule (2) and the transitivity of domination, any graph must dominate its descendants in $\mathcal{P}$, and cannot be antipodal to them.

(2) Since $X(G_2) \neq \emptyset$, we can choose a clique D_2 such that $D_2 \cap C$ is a proper nonempty subset of $W(G_2) = W(G_1)$. By Lemma 2.4, there is a relevant clique C_1 in G_1 that is attached to D_2 such that D_2 does not dominate C_1. Hence G_2 does not dominate G_1. By the same reasoning as in case (1), G_2 cannot be an ancestor of G_1.

(3) We construct a new valid parent relation $\mathcal{P}'$ in which G_2 is a child of G_1. Let G_j (which may be c) be the parent of G_2. In $\mathcal{P}'$, the new parent of G_2 is G_1. Every child of G_2 in $\mathcal{P}$ becomes a child of G_j in $\mathcal{P}'$. Every child of G_1 becomes a child of G_2 instead.

Let $\mathcal{T}$ and $\mathcal{T}'$ be the directed graphs induced by $\mathcal{P}$ and $\mathcal{P}'$ respectively. In $\mathcal{P}'$ each separated graph still has exactly one parent, so $\mathcal{T}$ and $\mathcal{T}'$ have exactly the same number of edges. Since $\mathcal{T}$ is a tree, for each node G_i other than c there is a unique path from c to G_i. A similar path still exists in $\mathcal{T}'$: if the old path goes through G_2, then shorten it by omitting G_2, while if the old path goes through G_1 lengthen it by inserting G_2 just after G_1 (some paths may have G_2 omitted and then reinserted). Thus $\mathcal{T}'$ is connected and must be a tree.

Every change of parent, child, or sibling from $\mathscr{P}$ to $\mathscr{P}'$ involves G_1 or G_2. Since both X sets are empty, any graph $G_3 \notin \{G_1, G_2\}$ is dominated by G_1 (G_2) if and only if for every relevant $C_3 \in \mathscr{C}(G_3)$, $C_3 \cap C \subset W(G_1)$ ($W(G_2)$). Any graph $G_3 \notin \{G_1, G_2\}$ dominates G_1 (G_2) if and only if $G_3 \bowtie G_1$ (G_2) and for every relevant $C_3 \in \mathscr{C}(G_3)$, $C_3 \cap C \supset W(G_1)$ ($W(G_2)$) or $C_3 \cap W(G_1)$ ($W(G_2)$) $= \emptyset$. Since $W(G_1) = W(G_2)$, validity condition (2) for $\mathscr{P}'$ is equivalent to validity condition (2) for $\mathscr{P}$. Similarly, any graph $G_3 \notin \{G_1, G_2\}$ is attached to G_1 (G_2) if and only if $W(G_1) \cap W(G_3)$ ($W(G_2) \cap W(G_3)$) is nonempty; since $W(G_1) = W(G_2)$, $\mathscr{P}'$ satisfies validity condition (4). If G_1 becomes a child of c in $\mathscr{P}'$, then it replaces G_2; since $W(G_1) = W(G_2)$, $\mathscr{P}'$ satisfies validity condition (3). □

The above proof shows that a set of separated subgraphs that have identical W sets and an empty X set are equivalent and interchangeable in a strong sense:

Corollary 3.5. *Let $\mathscr{D}$ be a set of separated subgraphs having identical W sets and an empty X set. We can modify any valid parent relation, so that the elements of $\mathscr{D}$ form a chain, preserving validity. That is, one graph in $\mathscr{D}$ is parent of the next, is the parent of the next, and so on (for any order of the elements of $\mathscr{D}$). Only the last element of the chain may have children not in $\mathscr{D}$. Furthermore, we can add elements to $\mathscr{D}$ and preserve validity just by inserting them in the chain.*

Using Corollary 3.5, we can temporarily eliminate all but one element $G_{\mathscr{D}}$ of the set $\mathscr{D}$ and let $G_{\mathscr{D}}$ represent $\mathscr{D}$. If we successfully construct a valid parent relation for the remaining separated subgraphs, we can extend it to a valid relation on the whole set by replacing $G_{\mathscr{D}}$ with a chain containing exactly the elements of $\mathscr{D}$ in some arbitrary order.

To efficiently eliminate interchangeable graphs we use a fact that follows immediately from Monma and Wei's characterization of UV graphs by separating cliques: if there exist three distinct pairwise antipodal separated subgraphs G_1, G_2, G_3 and a vertex $v \in W(G_1) \cap W(G_2) \cap W(G_3)$, then G is not a UV graph. Suppose $|W(G_i)| = r$, $X(G_i) = \emptyset$, and we want to find which other graphs are interchangeable with G_i. Firstly, we can restrict attention to those graphs that have a W set of size exactly r and an empty X set. Second, we can choose a vertex $v \in W(G_i)$ and further restrict attention to those graphs that have v in their W sets. Let G_j be such a graph. Since $|W(G_j)| = |W(G_i)|$, either G_i and G_j are interchangeable or antipodal depending on whether their W sets are equal or incomparable. Thus we need to test only whether $W(G_i)$ is equal to one of at most two other W sets. If there are already two other distinct W sets of the same size containing v, we can quit, knowing that G is not a UV graph. Note that this quitting provision is solely for the sake of efficiency (so we do not need to do too many tests with $W(G_i)$) and it is highly dependent on the choice of v.

To renumber the G_i graphs and temporarily eliminate interchangeable ones, we use the method below. In Step 3, we implement the test for interchangeable graphs that was just described.

SORT-G_i.

Step 1. Sort the graphs $G_1, G_2, \ldots, G_s$ so that the G_i precedes G_j, if $|W(G_i)| > |W(G_j)|$. Since $|W(G_i)| \leq n$, the sorting can be done in $O(n)$ steps with bucketsort.

Step 2. Among a set of G_i having $|W(G_i)|$ equal, those with $X(G_i) = \emptyset$ go first; others go in any order.

Step 3. Try to reduce any set of interchangeable subgraphs to one representative, storing the full set for later use. We quit if we discover three pairwise antipodal graphs G_1, G_2, G_3 such that there is a vertex $v \in W(G_1) \cap W(G_2) \cap W(G_3)$.

The condensed output order is $\mathcal{Q}$. For the graph G of Fig. 1, with the separated graphs G_i as defined above, $\mathcal{Q}$ might be $G_4, G_1, G_3, G_2, G_5, G_6$. The order of the second and third graphs is arbitrary because they each have W sets of size 2 and empty X sets. Similarly, the order of the last three graphs is arbitrary. However, G_4 must precede G_1 because $X(G_4) = \emptyset$, while $X(G_1) = \{b\}$ (because $K_2 \cap C = \{b\}$).

To compute $W(G_i)$, we use:

COMPUTE-W.

Step 1. Store in a table indexed on $V(G)$ which vertices belong to C.

Step 2. For each $C_j \in \mathcal{C}(G_i)$, compute $C_j \cap C$ and $|C_j \cap C|$ in time $O(|C_j|)$.

Step 3. Find a clique C_k having the largest intersection with C; by Remark 2.3, $C_k \cap C = W(G_i)$.

Step 1 of COMPUTE-W should be done only once. While computing $W(G_i)$, we can test if $X(G_i)$ is empty or not. Suppose the intersection, computed in Step 2, between some relevant clique and C, is of size r. Then $X(G_i)$ is empty if and only if every such intersection is of size r (for relevant cliques) or of size 0 (for irrelevant cliques).

For any clique C_i, the routine COMPUTE-W performs a constant number of operations per member of C_i. Each clique other than C belongs to exactly one G_i. As explained above, when deciding if G_i is equivalent to other separated graphs, we only need to test whether $W(G_i)$ is equal to at most two different W sets. Thus by Remark 1.4, COMPUTE-W and SORT-G_i require at most $O(m+n)$ steps.

Lemma 3.6. *If the subgraphs are renumbered by* SORT-G_i, *then* $G_j > G_k$ *implies* $j < k$.

Proof. Suppose $G_j > G_k$. If $W(G_k)$ is a proper subset of $W(G_j)$, then SORT-G_i makes $j < k$. Otherwise $W(G_j) = W(G_k)$, and we must have $X(G_j) = \emptyset$ by Lemma 3.4. Since the domination is proper, $X(G_k) \neq \emptyset$, and SORT-G_i makes $j < k$. □

Theorem 3.7. *Renumber of subgraphs using* SORT-G_i. *If G is a* UV *graph, there is a valid parent relation $\mathcal{P}$ such that $c, G_1, G_2, \ldots$ is a topological ordering of $\mathcal{P}$.*

Proof. By Theorem 3.1, we may choose a valid parent relation $\mathcal{P}$. Let G_j and G_k

be two separated subgraphs such that G_j is ancestor of G_k in $\mathscr{P}$. We must have $G_j \geq G_k$ by condition (2) and the transitivity of domination. Therefore $W(G_k) \subset W(G_j)$. If $W(G_k) = W(G_j)$, then $X(G_j) = X(G_k) = \emptyset$, by Lemma 3.4, G_j and G_k are interchangeable. However, the control structure of SORT-G_i eliminates interchangeable graphs, so we must have $|W(G_k)| > |W(G_j)|$, and SORT-G_i ensures that $j < k$. □

4. Building a valid parent relation

This section describes the difficult part of our path graph recognition algorithm. The algorithm can be summarized as:

(1) If G has more than two cliques, find a separating clique C.

(2) Recursively test the graphs G_i. If any one is not a UV graph, stop; otherwise, return a clique tree for each one.

(3) If we can renumber the graphs according to SORT-G_i, use the order $\mathscr{Q}$ in Step 4; else, quit.

(4) Try to form a valid parent relation respecting SORT-G_i. If this fails, quit because G is not a UV graph.

(5) For any set $\mathscr{D}$ of interchangeable separated graphs, replace the representative chosen in Step 3 with a chain containing all elements of $\mathscr{D}$ in any order.

(6) Convert the parent relation into a clique tree.

If G has one or two cliques, there is no separating clique, so we directly build a clique tree with one or two nodes. This section describes Step 4.

As above, suppose C is a separating clique for G. Suppose each separated subgraph G_i is a UV graph and T_i is a clique tree for it. We show how to find a valid parent relation for the set $\bigcup_i \{G_i\} \cup c$ or decide that none exists. The correctness proof is in Section 5. The proof of Theorem 3.1 shows how to build a clique tree from a valid parent relation.

Following terminology of [13], in any parent relation, each child of c is identified with a color. Each graph G_j is assigned the color of its unique (not necessarily proper) ancestor that is a child of c. We write "vertex v has color G_h" to mean $v \in W(G_h)$ and G_h is a child of c. In a valid parent relation any vertex has at most two colors because of condition (3).

We process $G_1, \ldots, G_s$ in the order produced by SORT-G_i. Let $\hat{G}_i$ be the graph induced by $V(C) \cup V_1 \cup V_2 \cup \cdots \cup V_i$. Let E_i be $\{c, G_1, G_2, \ldots, G_i\}$. We intend to fulfill the following invariant:

Invariant 4.1. For any t, such that $1 \leq t \leq s$, after we have processed $G_1, G_2, \ldots, G_t$, either:

(i) We have decided correctly that $\hat{G}_t$ is not an undirected path graph; or

(ii) we have computed a valid parent relation for E_t that respects $\mathscr{Q}$.

Let T_j be a clique tree for G_j. Node b representing $B \in \mathscr{C}(G_j)$ is an *end* for $v \in C$ if: $v \in W(G_j)$, $b = f(T_j, v)$ and G_j does not have a child G_k with $v \in W(G_k)$. If we build the tree as in Theorem 3.1, an end for v is an end of the path of nodes for v. Node c is *never* an end, so any vertex has zero, one or two ends. Here is a summary of how we process G_i:

PROCESS(G_i).

Step 1. Compute $W(G_i)$. We already did this to sort the G_i graphs.

Step 2. For each v in $W(G_i)$, find the ends for v in the parent relation computed so far.

Step 3. Either find a parent for G_i, possibly changing existing parent-child pairs, or give up.

We consider six different cases in Step 3; distinguishing the cases is part of Step 2. The six cases are:

(a) All vertices in $W(G_i)$ have zero ends.

(b) All vertices in $W(G_i)$ have one end and it is the same for every vertex.

(c) No vertex in $W(G_i)$ has two ends. Some vertex has one end, but not all vertices in $W(G_i)$ have one.

(d) All vertices in $W(G_i)$ have exactly one end, but they are not the same ends.

(e) A vertex $v \in W(G_i)$ has two ends. At least one of those ends is shared by every member of $W(G_i)$.

(f) A vertex $v \in W(G_i)$ has two ends. For each end, some member of $W(G_i)$ does not share it.

Most pairs of cases can be distinguished by checking the number of ends for each vertex in $W(G_i)$. Counting ends distinguishes four sets of cases: {(a)}, {(b),(d)}, {(c)}, {(e),(f)}. To distinguish case (b) from case (d), we do the following. Let v be any vertex in $W(G_i)$ with end f_v. Compare the end for each vertex in $W(G_i)$ other than v against f_v. If we find an end different from f_v, we have case (d); otherwise, we have case (b).

To distinguish between cases (e) and (f), let the two ends for v be f_{1v} and f_{2v}. For each $w \in W(G_i)$, check if w has f_{1v} and/or f_{2v} as an end. If there exist $x \in W(G_i)$ and $y \in W(G_i)$ (possibly $x = y$), such that f_{1v} is not an end for x and f_{2v} is not an end for y, then we have case (f); otherwise, we have case (e). Once we find the ends, we can decide into which case G_i falls using at most $4(|W(G_i)|)$ comparisons and constant time per comparison.

We now specify Step 3 of PROCESS for the first five cases.

(a) *All vertices in $W(G_i)$ have zero ends. Make G_i* a child of c.

(b) *All vertices in $W(G_i)$ have one end and it is the same for every vertex.* Let f_j be the end shared by all vertices, and suppose F_j is a clique in G_j. Make G_i a child of G_j.

(c) *No vertex in $W(G_i)$ has two ends. Some vertex has one end, but not all vertices in $W(G_i)$ have one.* Make G_i a child of c.

(d) *All vertices in $W(G_i)$ have exactly one end, but they are not the same ends.* Make G_i a child of c.

(e) *A vertex $v \in W(G_i)$ has two ends. At least one of those ends is shared by every member of $W(G_i)$. At least one member of $W(G_i)$ does not share both ends.* Choose one shared end. If that shared end represents a clique in G_h, make G_i a child of G_h.

Recall that for G in Fig. 1, SORT-G_i produces the order $G_4, G_1, G_3, G_2, G_5, G_6$. G_4 falls under case (a). G_1 falls under case (c) because $W(G_1) = \{b, d\}$, vertex b has one end, and vertex d has none. G_3 also falls under case (c). G_2 falls under case (e). G_5 falls under case (b). G_6 falls under case (e).

Let $\mathscr{P}$ be the valid parent relation that we construct for E_{i-1}. Case (f) is difficult because making G_i the child of c or any G_h, $h < i$, makes the parent relation invalid. In case (f), we try to change $\mathscr{P}$ before placing G_i.

The first step is to find candidates for the parent of G_i. One possibility is c. This drastic measure requires that for every $v \in W(G_i)$, we consolidate all G_j with $j < i$ and $v \in W(G_j)$, so they have the same color. We might also try to assign G_i an already existing color, call it G_a. For the assignment to be valid, G_a must dominate G_i. If $G_a \geq G_i$, we choose a descendant of G_a to be the parent of G_i. We choose the descendant of G_a that dominates G_i but has no children that dominate G_i. This graph is unique because G_a and each descendant have at most one child that dominates G_i, by condition (4) of validity. Since c has at most two children that dominate G_i, there are at most two such *possible parent* graphs. If there is only one, we try to change $\mathscr{P}$, so that we can make G_i a child of that graph. If there are two, we try the one with *larger* index first. If there are none, we call NEW-END(G_i), which is described later.

For the next few pages, we assume that either G_h is the only possible parent graph of G_i or that of two possible parents G_h has larger index. If we fail to change $\mathscr{P}$ so that G_i can be made a child of G_h, we consider changing $\mathscr{P}$ so that G_i can be made a child of the other possible parent (if it exists), and if that fails, we try to make G_i a child of c. For the second and third rearrangement attempts, we restore the parent relation to be $\mathscr{P}$. Restoring $\mathscr{P}$ can be done by maintaining a "log" of changes. Undoing changes (in reverse) costs no more than doing them.

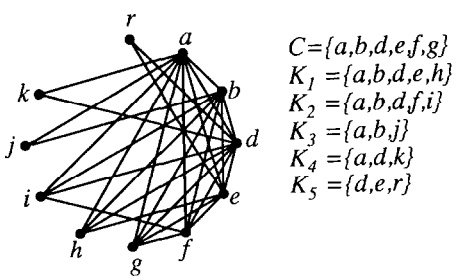

Fig. 4. A graph to illustrate the use of swaps.

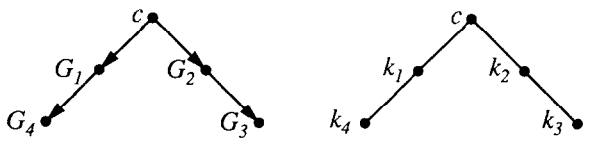

Fig. 5. A valid parent relation and clique tree for part of the graph of Fig. 4.

To make G_i a child of G_h and keep $\mathscr{P}$ valid, we have to move any descendant of G_h that is antipodal to G_i. We call the routine SWAP described below.

The UV graph in Fig. 4 illustrates SWAP. This graph has six maximal cliques. The removal of C leaves five components with a single vertex in each. Let G_i be the separated subgraph containing the two cliques C and K_i. Assume the graphs are numbered as they would be by SORT-G_i. G_1 falls under case (a), G_2 falls under case (c), G_3 and G_4 fall under case (e). After processing G_4, we may have the valid parent relation in Fig. 5.

The graph G_5 falls under case (f). The only possible parent of G_5 is G_1, but G_4 is (in Fig. 5) a child of G_1, and $G_5 \leftrightarrow G_4$. We decide to move G_4. The only possible parent for G_4 other than G_1 is G_2. However, G_2 has a descendant, G_3, that is antipodal to G_4. We remove that descendant, and make G_4 a child of G_2, leaving G_3 without a parent. Now observe that G_1 is a possible parent for G_3, and $G_3 \mid G_5$. Therefore, we make G_3 a child of G_1. Finally, we make G_5 a child of G_1, obtaining the parent relation in Fig. 6.

The SWAP operation uses two queues and one boolean table; when SWAP exits, the queues are emptied and the table entries are reset. SQUEUE-1 contains graphs that start with the color of G_h, but are waiting to get a new color and parent. SQUEUE-2 contains graphs that started with a color other than that of G_h, but are waiting to get a new parent that has the color of G_h. The table is a boolean array of s bits; recall that s is the number of separated subgraphs and $s \leq n$. TABLE[j] = TRUE if and only if G_j has already received a new parent in the current call to SWAP.

SWAP(G_h, G_i).

Step 1. For each child G_j of G_h:

(a) If $G_j \leftrightarrow G_i$, then

(1) G_j no longer has a parent or a color.

(2) Append G_j to SQUEUE-1.

(3) For each child G_q of G_j, repeat the test of Step 1(a) with G_q in place of G_j.

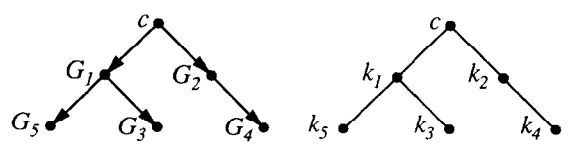

Fig. 6. A valid parent relation and clique tree for the graph of Fig. 4.

Step 2. While SQUEUE-1 is not empty:
- (a) Let G_j be the item at the head.
- (b) If TABLE[j] = TRUE, quit the loop and go to Step 6.
- (c) Try to find a graph G_a with a different color from G_h, such that G_a dominates G_j, but none of its children dominate G_j. (We explained above that G_a is unique because of validity condition (4).) If no such graph exists, let $G_a := c$.
- (d) For any child G_b of G_a such that G_b is not an ancestor of G_h, and $G_b \leftrightarrow G_j$:
  - (1) Append G_b to SQUEUE-2; G_b no longer has a parent or a color.
  - (2) Repeat Step 2(d) for each child G_q of G_b with G_q in place of G_b.
- (e) For each child G_r of G_a such that $G_j \geq G_r$:
  - (1) Change the parent of G_r to be G_j.
- (f) Make G_a the parent of G_j.
- (g) Remove G_j from SQUEUE-1.
- (h) Set TABLE[j] := TRUE.

Step 3. While SQUEUE-2 is not empty:
- (a) Let G_j be the item at the head.
- (b) If TABLE[j] = TRUE, or G_h does not dominate G_j, or $G_j \leftrightarrow G_i$, quit the loop and go to Step 6.
- (c) Let G_d be the lowest (not necessarily proper) descendant of G_h that dominates G_j.
- (d) For any child G_b of G_d such that $G_b \leftrightarrow G_j$:
  - (1) Append G_b to SQUEUE-1; G_b no longer has a parent or a color.
  - (2) Repeat Step 3(d) for each child G_q of G_b with G_q in place of G_b.
- (e) For each child G_r of G_d such that $G_j \geq G_r$:
  - (1) Change the parent of G_r to be G_j.
- (f) Make G_d the parent of G_j.
- (g) Remove G_j from SQUEUE-2.
- (h) Set TABLE[j] := TRUE.

Step 4. If SQUEUE-1 is nonempty, return to the loop at Step 2.

Step 5. Make G_i a child of G_h.

Step 6. Reset all TRUE entries in TABLE to FALSE, and empty the queues.

In those cases where we quit the loop in Step 2 or Step 3, we say that we "quit SWAP", although actually we clean up at Step 6 before exiting. If we quit SWAP(G_h, G_i) and there were two possible parents for G_i, then we restore the parent relation to $\mathscr{P}$ and try SWAP(G_k, G_i). If SWAP is unsuccessful with all possible parents, we again restore the parent relation to be $\mathscr{P}$.

The last option we try to place G_i is the first one we suggested: make G_i a child of c. Figure 7 shows a UV graph that illustrates this operation. This graph has eight cliques, $C, K_1, K_2, \ldots, K_7$. We number the separated graphs G_i so that $V_1 = \{h\}$, $V_2 = \{i\}$, $V_3 = \{j\}$, $V_4 = \{k\}$, $V_5 = \{l, q\}$, $V_6 = \{r\}$ and $\mathscr{C}(G_1) = \{C, K_1\}$, $\mathscr{C}(G_2) =$

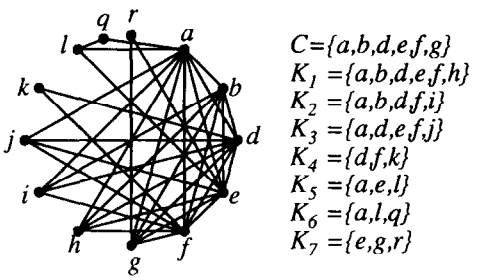

Fig. 7. A graph G to illustrate new end creation.

$\{C, K_2\}$, $\mathscr{C}(G_3) = \{C, K_3\}$, $\mathscr{C}(G_4) = \{C, K_4\}$, $\mathscr{C}(G_5) = \{C, K_5, K_6\}$, $\mathscr{C}(G_6) = \{C, K_7\}$. Assume that this ordering is the output of SORT-G_i. Figure 8 shows the valid parent relation and clique tree after $G_1, \ldots, G_5$ are processed.

To rearrange $\mathscr{P}$ in our attempt to make G_i a child of c, we use one auxilliary data structure—a bitmap called CONSOLIDATE of size s. The graphs G_j for which CONSOLIDATE$[j] = 1$ are the separated graphs containing vertices whose two colors must be consolidated, so that G_i will not be a third color for any vertex. We set CONSOLIDATE$[j] = 1$ if and only if $j < i$ and $W(G_j) \cap W(G_i) \neq \emptyset$.

The overall plan for rearranging the parent relation is:

NEW-END(G_i).

Step 1. Compute the entries in CONSOLIDATE.

Step 2. Let $\mathscr{M} := \{G_j \mid \text{CONSOLIDATE}[j] = 1\}$. For each $G_j \in \mathscr{M}$, G_j no longer has a parent.

Step 3. Make G_i a child of c.

Step 4. Attempt to compute eventual parent-child relationships for $\mathscr{M}$, but do not change the parent relation yet.

Step 5. If Step 4 succeeds, assign parents to members of $\mathscr{M}$ in increasing parent order. Some children not in $\mathscr{M}$ may need new parents.

We need to describe Steps 4 and 5 more precisely. In Step 4, we consider members of $\mathscr{M}$ in increasing order of subscript. There are three cases for each G_h:

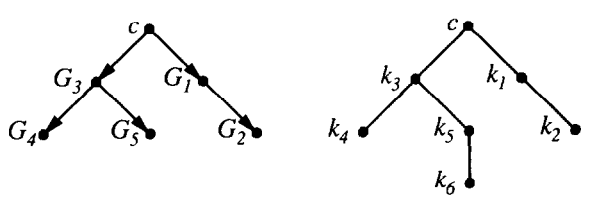

Fig. 8. A valid parent relation and clique tree for part of the graph of Fig. 7.

4(a) For every $v \in W(G_h) \cap W(G_i)$, there is no $G_q \in \mathcal{M}$ such that $v \in W(G_q)$ and $q < h$. Then G_h will get c as a parent. (The definition of this case effectively checks validity condition (3).)

4(b) The set $P_h := \{G_q \in \mathcal{M} \mid q < h,\ G_q \geq G_h\} \neq \emptyset$. Let G_a be the graph in P_h with maximum subscript. If G_h is attached to any graphs in $\mathcal{M}$ assigned G_a as intended parent, then quit. Otherwise, make G_a the intended parent of G_h.

4(c) If neither 4(a) nor 4(b) applies, then quit.

In Step 5, we again process $\mathcal{M}$ in increasing order, but we consider c before the first member of $\mathcal{M}$. To process c, we formally assign c the children chosen for it in case 4(a). To process G_h, attempt to assign G_h the children chosen for it, in case 4(b). Denote the set of children from 4(b) by $\mathcal{M}_h$, and denote the children of G_h not in $\mathcal{M}$, by $\mathcal{N}_h$. Assignment of new children to G_h is conceptually done in a similar manner to the way we make G_i a child of G_h when we never get as far as having to call NEW-END. If no member of $\mathcal{M}_h$ is attached to any member of $\mathcal{N}_h$, we assign G_h as parent of each member of $\mathcal{M}_h$ without violating validity. If, however, some member of $\mathcal{M}_h$ is attached to a member of $\mathcal{N}_h$, we first try to change the parent of any offending member, G_q of $\mathcal{N}_h$. This is done as follows. First we test if $G_q \in \mathcal{N}_h$ is dominated by any member of $\mathcal{M}_h$. If so, we change the parent of G_q to be c. Otherwise if G_q is attached to a member of $\mathcal{M}_h$ it must be antipodal to that graph (G_q cannot dominate that graph because $G_q \mid G_i$, while members of $\mathcal{M}_h$ are attached to G_i), so we use SWAP($G_h, \mathcal{M}_h$) as above making two slight changes since the second argument is a set of pairwise unattached separated graphs dominated by G_h instead of just one separated graph. Instead of testing for antipodal to G_i in 1(a) and 3(b), we test for antipodal to any member of $\mathcal{M}_h$; if we succeed in moving the graphs attached to some member of $\mathcal{M}_h$, all members of $\mathcal{M}_h$ get G_h as their parent.

In our example from Figs. 7 and 8, the graph G_6 has no possible parents, so swaps cannot succeed. The vertex $e \in W(G_6)$ has two ends (k_1 and k_5) in the tree of Fig. 8, so we cannot just make G_6 a child of c. Instead, we consolidate the three separated subgraphs with e in their W sets (G_1, G_3, G_5) and rearrange the parent relation. The final tree and parent relation are shown in Fig. 9. Notice that e now has only two ends, k_6, k_7. When we made G_3 a child of G_1, we moved G_2 to being a child of c because it is antipodal to G_3.

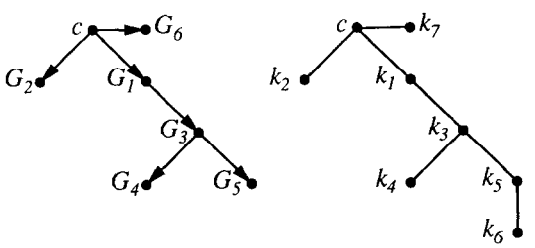

Fig. 9. A valid parent relation and clique tree for the graph of Fig. 7.

5. Correctness

In this section we do the hard parts of the proof that the algorithm described in Sections 3 and 4 recognizes UV graphs. In particular we focus on Step 4, finding a valid parent relation. The relationship between parent relations and clique trees and the uses of SORT-G_i in Steps 3 and 5 were explained in Section 3.

Remark 5.1. Invariant 4.1 holds before G_1 is processed.

Lemma 5.2. *If G_i falls under one of cases* (a) *through* (e) *and Invariant* 4.1 *holds before we process G_i, it also holds afterwards.*

Proof. For Invariant 4.1, we check the four validity conditions and the sorted order condition. We always give G_i one parent, so condition (1) remains satisfied. The parent has an index smaller than i, so $\mathcal{Q}$ is respected. Suppose G_i is made the child of G_j. In selecting cases (a) through (e), we find an end that is $f(G_j, v)$ for every $v \in W(G_i)$. By definition of f, any other relevant clique in G_j that intersects $W(G_i)$ is on $\pi(c, f(G_j, v))$ if we build the tree as in the proof of Theorem 3.1. Such a clique contains $W(G_i)$ and dominates every relevant clique in G_i. Hence G_j dominates G_i and condition (2) is satisfied. Since $f(G_j, v)$ is an end, G_j cannot have another child with v in its W set; thus condition (4) is satisfied. In cases where we make G_j a child of c, there was at most one end for every $v \in W(G_i)$; thus condition (3) is satisfied. □

We now present a sequence of lemmas regarding case (f).

Lemma 5.3. *If G_i falls into case* (f) *and G_h is a possible parent of G_i, then G_i is antipodal to a child of G_h.*

Proof. As in the proof of Lemma 5.2, the possible parent G_h dominates G_i. By definition of possible parent, no child of G_h dominates G_i. As in the proof of Lemma 5.2, $f(G_h, v)$ is the same for every $v \in W(G_i)$. But since we are in case (f) there exists an $x \in W(G_i)$ such that $f(G_h, x)$ is not an end for x. This implies that G_h has a child G_q such that $x \in W(G_q)$; in particular, $G_i \bowtie G_q$. Since G_q is not a possible parent for G_i, we know that G_q does not dominate G_i. Since $i > q$, it cannot be that $G_i > G_q$, by Lemma 3.6. The only relationship they can have is $G_i \leftrightarrow G_q$. □

Lemma 5.4. *Suppose we are in case* (f) *and Invariant* 4.1 *holds. Suppose that we are trying to make G_i a child of G_h and we call* SWAP *(because G_i is antipodal to a child of G_h). If we exit from* SWAP *successfully, then Invariant* 4.1 *still holds. An analogous assertion holds for using* SWAP *to make G_i a child of G_k, if G_k is a second possible parent.*

Proof. The proof is divided into several parts. First we show that every graph gets exactly one parent. Second we check that each parent distinct from c dominates its children (validity condition (2)). Third we show that the new parent relation respects $\mathcal{Q}$ and induces a tree structure (condition (1)). Fourth we check that each vertex has at most two colors (condition (3)). Finally, we check that there are no attached siblings (condition (4)).

A graph may lose its parent in Steps 1(a1), 2(d1), 2(e1) 3(d1), or 3(e1) of SWAP. Those graphs that lose parents in Steps 1(a1) or 3(d1) get a new parent in Step 2(f), or else they never get off of SQUEUE-1, and we quit. Those graphs that lose a parent in Step 2(d1), get a new parent in Step 3(f). In Steps 2(e1) and 3(e1), we immediately assign a new parent. No steps assign parents to graphs that already have a parent. Thus at the (successful) exit of SWAP each graph has exactly one parent. For the assignments in Step 2(e1) and 3(e1), we check explicitly that G_i dominates its new children. For Steps 2(f) and 3(f), we check that the new parent is c or that it dominates G_j. Thus validity condition (2) holds at the end of SWAP.

By Lemma 3.4, if a parent dominates its child, but has a higher index, then they have the same W set and empty X sets and are interchangeable. But SORT-G_i eliminated interchangeable graphs. Thus $\mathcal{Q}$ is respected. Since each parent has a smaller index than its children, the directed graph induced by the new parent relation cannot have cycles, and is a tree (condition (1)).

Suppose, to derive a contradiction, that we create a third color, G_j, for some vertex v. G_j can become a child of c only in Step 2(f), and this happens only if $G_a := c$ in Step 2(c). Let the other two colors be G_q and G_r; one of these, say G_q, must be the color of G_h. $G_r \bowtie G_j$ because $v \in W(G_j) \cap W(G_r)$. In Step 2(c), we must have checked that G_r does not dominate G_j, so either $G_j \leftrightarrow G_r$ or $G_j > G_r$. If $G_j \leftrightarrow G_r$, then G_r loses its parent, c, in Step 2(d1), before we reach Step 2(f), and all the children of G_r are checked too. If $G_j > G_r$, then G_r gets a new parent and ceases being a color in Step 2(e), before we reach Step 2(f). Thus validity condition (3) is preserved.

When G_j gets a new parent $G_a \neq c$, we check in choosing G_a that it has no children that dominate G_j. Any child of G_a antipodal to G_j gets a new color in Steps 2(d) or 3(d). Any descendant of G_a dominated by G_j gets a new parent in Steps 2(e) or 3(e) before we make G_j a child of G_a in Steps 2(f) or 3(f). Hence no two children of $G_a (\neq c)$ or of G_j are attached, and condition (4) is preserved. □

Lemma 5.5. *Suppose we call* NEW-END *in case* (f) *and* $\mathcal{P}$ *satisfies Invariant* 4.1. *Suppose* $G_h \in \mathcal{M}$ *is chosen in Step* 4(b) *of* NEW-END *as the parent for* $G_j \in \mathcal{M}$. *Suppose we have not quit before processing* G_h *in Step* (5) *and* G_h *has a child* $G_k \notin \mathcal{M}$ (*i.e.*, $G_k \in \mathcal{N}_h$) *such that* $G_j \bowtie G_k$. *Then either*:

(a) $G_j \leftrightarrow G_k$ *and* G_k *must be moved to another color, or*

(b) $G_j \geq G_k$, *and we make* G_k *a child of* c, *thereby recoloring* G_k *and all its descendants. The parent change preserves validity.*

Proof. By assumption, $W(G_j) \cap W(G_h) \cap W(G_i) \neq \emptyset$, and $h > j > i$. Therefore, in any valid parent relation respecting $\mathcal{Q}$ and having c as parent of G_i, G_j is a descendant of G_h. By assumption, $G_j \bowtie G_i$, but $G_k \mid G_i$. Hence there exists $w \in W(G_j) \setminus W(G_k)$, and G_k cannot dominate G_j. If $G_j \leftrightarrow G_k$ (case (a)), then G_k must be recolored, because G_j and G_h must have the same color. If G_j and G_k are not antipodal, the one remaining possibility is $G_j \geq G_k$ (so, assume this for the rest of the proof).

$\mathcal{P}$ is the parent relation on E_{i-1} at the start of NEW-END. Since $G_j \geq G_k$ but G_k is not a descendant G_j at Step 4, G_j and G_k have different colors in $\mathcal{P}$. By conditions (3) and (4) of validity any graph that is attached to G_k, is not in $\mathcal{M}$, and has a color different from G_k in $\mathcal{P}$ is a descendant of G_j in $\mathcal{P}$. Hence, when we make G_j the child of G_h, each vertex in $W(G_k)$ has only one color, and making G_k a child of c does not violate condition (3) of validity. The other three conditions are not violated by changing a parent to c. Hence Invariant 4.1 is preserved. □

Lemma 5.6. *If in case* (f), *$\mathcal{P}$ satisfies Invariant* 4.1, *and we successfully exit from* NEW-END, *then we have a valid parent relation for E_i satisfying Invariant* 4.1.

Proof. The parent relation at the start of NEW-END is $\mathcal{P}$. No graph in $\mathcal{M}$ can be made parent of G_i or a child of G_i in NEW-END. Thus the validity restrictions that any vertex (in $W(G_i)$) have at most two colors, that a parent dominate its child, and that siblings be unattached, combined with the sorted order restriction force the parent-child relationships for members of $\mathcal{M}$ to be those computed in Step 4 of NEW-END.

When we make $G_j \in \mathcal{M}$ the child of $G_h \in \mathcal{M}$, we give a new parent to any child G_q of G_h attached to G_j. The new parent is c or is chosen in SWAP. If $G_j \geq G_k$, we proved in Lemma 5.5 that making G_k a child of c preserves validity (4.1) if we exit from NEW-END successfully. The fact that using SWAP preserves Invariant 4.1 can be proved as in the proof of Lemma 5.4. □

Lemma 5.7. *If in case* (f), *$\mathcal{P}$ satisfies Invariant* 4.1 *and we quit in* NEW-END, *then there is no valid parent relation for E_i, respecting $\mathcal{Q}$, in which G_i is a child of c.*

Proof. We may quit in either Step 4 or Step 5 of NEW-END. We treat these separately.

Step 4: Suppose we quit while seeking a parent for $G_j \in \mathcal{M}$. Since we cannot quit in case 4(a), there exists $G_q \in \mathcal{M}$ and a vertex v such that $v \in W(G_q) \cap W(G_i)$ and $q < j$. Vertex v has two colors—one is G_i, and the other is an ancestor of G_q. By assumption, G_j cannot get the color of G_i. We quit in 4(b) if there exists a $G_a \in \mathcal{M}$ with $a < j$ that is attached to G_j but does not dominate G_j. Lemma 3.6 implies that G_j does not dominate G_a; thus $G_a \leftrightarrow G_j$. By Lemma 3.2 and validity condition (4), there is no color for G_j, and quitting is justified. We quit in case 4(c) if no member

of $\mathcal{M}$ with index less than j dominates G_j. Designating c as the parent of G_j would give v a third color. Choosing any member of $\mathcal{M}$ as the parent of G_j would violate validity condition (2).

Step 5: We quit only within a call to SWAP. Suppose we quit while trying to make the members of $\mathcal{M}_h$ children of G_h. We may quit in SWAP for three reasons:

(1) Some G_j has already been swapped once, making TABLE[j] = TRUE, and gets enqueued again (Steps 2(b) and 3(b)),

(2) some G_j put on SQUEUE-2 is antipodal to a member of $\mathcal{M}_h$, or

(3) some G_j put on SQUEUE-2 is not dominated by G_h.

To justify reasons (1) and (2), we consider an auxiliary graph D on the vertex set $E_i \setminus \{c\}$. If two graphs in E_i are antipodal, put an edge between them in D. Any G_k that is enqueued is antipodal to a graph that we would like to color with G_k's former color. Any graph in $\mathcal{M}_h$ must have the color of G_h because G_i is a child of c. Any graph adjacent in D to a member of $\mathcal{M}_h$, must have a color different from that of G_h. Now suppose G_q is at distance 2 from a member of $\mathcal{M}_h$ and that $G_h \geq G_q$. Since any vertex has at most two colors, and for every vertex in $W(G_q)$ one of the colors is that of G_h, we are forced to color G_q with the color of G_h. An analogous argument can be made for vertices at any even distance, provided all intermediate graphs on the path are dominated by G_h. Any graph put on SQUEUE-1 is a descendant of G_h just before being on the queue, and any graph put on SQUEUE-2 becomes a descendant of G_h. Thus in studying D we restrict attention to paths on which all internal vertices are dominated by G_h. One can show by induction on path length that vertices at an odd distance from a member of $\mathcal{M}_h$ cannot receive the color of G_h, while vertices at an even distance must receive the color of G_h.

If G_j is put on both queues during SWAP (equivalent to quitting reason number (1)), it is at odd distance from a member of $\mathcal{M}_h$ and at even distance from a member of $\mathcal{M}_h$. For either possible color for G_j, there is a graph of the same color that is antipodal to G_j. Quitting reason (2) can be analyzed similarly. Here G_j is at even distance from a member of $\mathcal{M}_h$, so it must have the color of G_h. All members of $\mathcal{M}_h$ have this color too, and none of them should be antipodal to G_j.

For quitting reason (3), we observe that any G_j put on SQUEUE-2 must have the same color as G_h. Since G_j is attached to a former descendant of G_h, $G_h \bowtie G_j$. If G_h and G_j must have the same color and are attached, one is the ancestor of the other in any valid parent relation; hence one dominates the other. If G_h does not dominate G_j (quitting reason (3)), then $G_j \geq G_h$. But the control structure of NEW-END implies that $G_j \notin \mathcal{M}$, since members of $\mathcal{M}$ are never enqueued in SWAP. Hence $G_j \mid G_i$, while $G_h \bowtie G_i$. This implies that there exists a vertex in $W(G_h) \setminus W(G_j)$, and G_j cannot dominate G_h. Therefore $G_h \leftrightarrow G_j$, justifying quitting reason (3). □

Lemma 5.8. *Suppose we are in case* (f) *and G_h is the only possible parent for G_i. Suppose $\mathcal{P}$ satisfies Invariant* 4.1, *but we quit in* SWAP. *If there is a valid parent relation for E_i respecting $\mathcal{Q}$, there is such a relation in which c is the parent of G_i.*

Proof. $\mathscr{P}$ is our valid parent relation on E_{i-1}. Let $\mathscr{P}'$ be a valid parent relation on E_i, obeying $\mathscr{Q}$, in which G_i is a child of G_h'. We split into two cases.

Case 1: $G_h = G_h'$. Construct the directed graph D as in the proof of Lemma 5.7, and restrict attention to paths in which all internal vertices are dominated by G_h. Any graph in E_i that is a descendant of G_h in $\mathscr{P}$ and antipodal to G_i is forced to have a color different from G_h by SWAP and has a different color from G_h in $\mathscr{P}'$. Let this set of antipodal graphs be B. As in the proof of Lemma 5.7, we can show the following. If G_r is at odd distance from a member of B then G_r must have the same color as G_h, and has the same color in $\mathscr{P}'$; if G_r is at even distance from a member of B, then G_r must have a color different from G_h, and has a different color from G_h in $\mathscr{P}'$. Thus if SWAP were to quit for reasons (1) or (2), the relation $\mathscr{P}'$ would have two antipodal graphs with the same color and be invalid. Any graph put on SQUEUE-2 cannot dominate G_i (or else there would be two possible parents) and hence cannot dominate G_h. Since graphs on SQUEUE-2 have the color of G_h in $\mathscr{P}'$, they are descendants of G_h in $\mathscr{P}'$. Hence such graphs are dominated by G_h and SWAP cannot quit for reason (3).

Case 2: $G_h' \neq G_h$. Here it is possible that SWAP quits and $\mathscr{P}'$ is valid, but we construct a valid parent relation $\mathscr{P}''$ on E_i that obeys $\mathscr{Q}$ and has c as parent of G_i. Since G_i has one possible parent, any graph that dominates G_i is an ancestor of G_h in $\mathscr{P}$. Let A be the set of G_h and its ancestors in $\mathscr{P}$. Define $\mathscr{P}''$ as follows:

(1) The parent of G_i is c.

(2) If G_j is a proper descendant of G_h in $\mathscr{P}'$, give G_j the parent it has in $\mathscr{P}'$.

(3) If G_j is a proper descendant of some G_l in $\mathscr{P}'$ such that G_l is a child of G_h' in $\mathscr{P}'$, and $G_l \bowtie G_h$, then G_j gets the parent it has in $\mathscr{P}'$.

(4) If $G_j \neq G_i$ is a child of G_h' in $\mathscr{P}'$, and G_j is attached to a child of G_h in $\mathscr{P}'$, then G_j becomes a child of c.

(5) Any G_j that does not get a parent in tests (1), (2), (3), or (4), keeps the parent it has in $\mathscr{P}$.

All elements of A are covered only by test (5), and hence keep their parents. We now check that $\mathscr{P}''$ satisfies the four validity conditions and obeys the order produced by SORT-G_i.

Conditions (1) and (2) and sorted order: Tests (2) and (3), and (2) and (4) are pairwise mutually exclusive because G_h and G_h' have different colors in $\mathscr{P}'$. Tests (3) and (4) are mutually exclusive because test (3) concerns descendants of G_h' that are *not children*, while test (4) concerns only children. Test (5) is explicitly exclusive of the other four and ensures that the tests cover all members of $E_i \setminus \{c\}$. Thus each element of $E_i \setminus \{c\}$ gets exactly one parent. Any G_j gets c as parent or keeps its parent from a relation respecting $\mathscr{Q}$. Hence the parent of G_j in $\mathscr{P}''$ is c or a graph with smaller index. Thus the directed graph on E_i induced by $\mathscr{P}''$ is a tree, parents dominate their children, and $\mathscr{P}''$ respects $\mathscr{Q}$.

Condition (3): Let $v \in V(C)$ be given. Let G_l be the ancestor of G_h in $\mathscr{P}'$ that is a child of c. We split into three cases: $v \in W(G_i)$, $v \in W(G_l) \setminus W(G_i)$ and $v \notin W(G_l)$. First, suppose $v \in W(G_i)$. In $\mathscr{P}''$, v has two colors: G_i and the color of G_h.

Suppose that G_r is a third color. $G_r \notin A$; thus in $\mathscr{P}'$, G_r is a descendant of G'_h or G_h. However $G_r \bowtie G_i$, a child of G'_h, and $r<i$ ruling out G'_h. G_h is ruled out by test (2); if G_r were a descendant of G_h, then G_r would keep its parent and could not be a child of c.

Second, suppose $v \in W(G_l) \setminus W(G_i)$. In $\mathscr{P}$, v has at most two colors. A new color not in $\mathscr{P}$ can be created only by test (4). Thus a third color G_r would be a child of G'_h in $\mathscr{P}'$ and attached to a child of G_h. In all three relations, one color for v is the first graph in A; the second color in $\mathscr{P}''$, call it G_q, could only be a child of c in $\mathscr{P}$ and hence G_q gets its parent from test (5). We derive a contradiction. Since $v \in W(G_q)$ and the two colors for v in $\mathscr{P}'$ are those of G_h and G'_h, G_q has the same color as G_h or as G'_h in $\mathscr{P}'$. Furthermore, $G_q \notin A$, so G_q is a descendant of one of G_h or G'_h in $\mathscr{P}'$. If it is a descendant of G_h, then it gets its parent by test (2) (and not test (5)). Now suppose G_q is a descendant of G'_h; recall that G_r is a child of G'_h in $\mathscr{P}'$. Since $G_q \bowtie G_r$, G_q must be a proper descendant of G_r in $\mathscr{P}'$. Since G_r is attached to a child of G_h in $\mathscr{P}'$, $G_h \bowtie G_r$. Combining the last two facts, we see that if G_q is a descendant of G'_h in $\mathscr{P}'$, it gets its parent by test (3) (and not test (5)). Thus G_q cannot get its parent by test (5), a contradiction.

Finally, suppose $v \notin W(G_l)$. As above, the first color for v is the first graph in A; the second color $G_q \notin A$, is a child of c in $\mathscr{P}$, and gets its parent from test (5); the third color $G_r \notin A$ and gets its parent from test (4). We derive a contradiction. Since $v \notin W(G_l) \supset W(G_h)$, G_q cannot have the color of G_h in $\mathscr{P}'$, and G_r cannot have the color of G_h in $\mathscr{P}$. Since v can have only two colors, G_r has the color of G_q in $\mathscr{P}$, and is a descendant of G_q. Thus $G_q \geq G_r$, $q<r$, and $W(G_q) \supset W(G_r)$. By construction of test (4), and the restriction that G_h dominates its children, there exists $w \in W(G_r) \cap W(G_h)$; hence $w \in W(G_q)$ too, and G_q is attached to a child of G_h. We proved that G_q cannot have the color of G_h in $\mathscr{P}'$, so G_q has the color of G'_h by the two color rule. Since $w \in W(G'_h) \cap W(G_q)$ one graph must be the ancestor of the other. However, G_q cannot be a proper ancestor of G'_h because $G_q \notin A$, and G_q cannot be a proper descendant of G'_h (and hence of G_r) because $q<r$. Thus condition (3) is preserved.

Condition (4): The proof is by contradiction. Suppose that G_q and G_r are both children of some G_j, but $G_q \bowtie G_r$. Since $\mathscr{P}$ and $\mathscr{P}'$ are valid, G_q and G_r cannot keep their parents from the same relation. We assume without loss of generality that G_q keeps its parent from $\mathscr{P}'$ and G_r keeps its parent from $\mathscr{P}$. Thus G_q is assigned its parent in test (2) or test (3).

Test (2): Here G_q is a proper descendant of G_h in $\mathscr{P}'$ and remains a proper descendant. Since G_r and G_q are siblings and condition (2) holds, we know $G_h \bowtie G_r$ and $G_h \bowtie G_q$. Since $G_r \notin A$, it must be a descendant of either G_h or G'_h in $\mathscr{P}'$. If G_r were a descendant of G_h in $\mathscr{P}'$, it would keep its parent by test (2). If G_r were a proper descendant of a child of G'_h in $\mathscr{P}'$, it would keep its parent by test (3). If G_r were a child of G'_h, it would become a child of c in $\mathscr{P}''$ by test (4).

Test (3): Here G_q is a proper descendant of G_l and G_l is a child of G'_h in $\mathscr{P}'$, with $G_l \bowtie G_h$. Since $G_r \bowtie G_q$, G_r is a proper descendant of G_h or G'_h in $\mathscr{P}'$. If G_r were a

descendant of G_h, it would get its parent from $\mathscr{P}'$ by test (2). If G_r were a proper descendant of a child of G'_h, that child would be G_l and G_r would keep its parent from $\mathscr{P}'$ by test (3). Otherwise, $G_r = G_l$ and G_q would keep G_r as an ancestor, contradicting the assumption that they are siblings. □

Lemma 5.9. *Suppose that after processing G_{i-1}, Invariant* 4.1 *holds, and we have a valid parent relation $\mathscr{P}$ on E_{i-1}. Suppose that G_i falls into case* (f) *and has two possible parents G_h and G_k with $k<h$. Suppose that we quit in both calls to* SWAP. *If there is a valid parent relation for E_i respecting $\mathscr{Q}$, it has c as the parent of G_i.*

Proof. The proof is by contradiction. Recall that $\mathscr{P}$ is the parent relation at the start of SWAP(G_h, G_i) and SWAP(G_k, G_i). Suppose that $\mathscr{P}'$ is a valid parent relation on E_i, respecting $\mathscr{Q}$, and that G'_h is the parent of G_i in $\mathscr{P}'$. We split into four cases. In each case we analyze the graph D also used in the proofs of Lemmas 5.6 and 5.7 to prove that at least one of the two calls to SWAP should not fail. The call SWAP(G_h, G_i) can fail for three reasons:

(1) Some G_j has already been swapped once, making TABLE$[j]$ = TRUE, and is enqueued again (Steps 2(b) and 3(b)),

(2) some G_j on SQUEUE-2 is antipodal to G_i, or

(3) some G_j on SQUEUE-2 is not dominated by G_h.

The call SWAP(G_k, G_i) can fail for three analogous reasons (replace G_h with G_k).

Case 1: $G'_h = G_h$. We claim that the call SWAP(G_h, G_i) should have exited successfully. Most of the justification for why SWAP would not quit is given in the first case of the proof of Lemma 5.8. The only place where we use the assumption that G_i has only one possible parent is to show that any graph on SQUEUE-2 does not dominate G_h. Here we need a different proof of that fact. Observe that if G_j gets on SQUEUE-2, it gets there because G_j is antipodal to a graph G_l that is dominated by G_h. If G_j dominated G_h, it would also dominate G_l by transitivity, contradicting $G_j \leftrightarrow G_l$.

Case 2: $G'_h = G_k$. We claim that the call SWAP(G_k, G_i) should have exited successfully. The argument proceeds as in Case 1, except that in the auxiliary graph D, we allow paths in which interval vertices are graphs dominated by G_k.

Case 3: *G'_h is a proper ancestor of G_k in $\mathscr{P}$ (hence $h'<k$).* We claim that the second call SWAP(G_k, G_i) should succeed. Since $G_k \geq G_i$ and $G_h \geq G_i$, $G_h \bowtie G_k$. In this case neither G_k nor G_h can have the color of G'_h in $\mathscr{P}'$. This is because $\mathscr{P}'$ obeys $\mathscr{Q}$, $h'<k<h<i$, and G'_h cannot have descendants with indices less than i attached to its child G_i. Hence in $\mathscr{P}'$, G_k and G_h have the same color; this implies that G_k is an ancestor of G_h, which in turn implies $G_k \geq G_h$.

Let $\mathscr{A}$ be the set of children of G_k in $\mathscr{P}$ that are antipodal to G_i. The members of $\mathscr{A}$ must have the color of G_h in $\mathscr{P}'$ and they are attached to G_h. Since they do not dominate G_i, they cannot dominate G_h, and hence G_h dominates every member of $\mathscr{A}$. Thus any graph put on SQUEUE-1 in Step 1, is dominated by G_h. Hence any graph put on SQUEUE-2 the first time through Step 2 is a descendant of

G_h in $\mathscr{P}$. This argument can be repeated for each alternation between queues to show that every graph enqueued is dominated by G_h. Since $G_k \geq G_h$, every graph enqueued is dominated by G_k, and we cannot quit for reason (3).

For the other two reasons, we consider the graph D. Although it seems natural to restrict attention to paths with all internal vertices dominated by G_k, we use G_h instead, since we just showed that all the graphs moved by SWAP are dominated by G_h. Every graph at odd distance from a member of $\mathscr{A}$ (those graphs on SQUEUE-2) must have the same color as G_i in $\mathscr{P}'$, and hence no graph on SQUEUE-2 could be antipodal to G_i (quitting reason (2)). Every graph at even distance must have the same color as G_k and G_h and a color different from G_i in $\mathscr{P}'$. Thus no graph can be both at even distance and at odd distance (quitting reason (1)).

Case 4: *G_h' is an ancestor of G_h in $\mathscr{P}$.* Since G_h' is an ancestor of G_h, $W(G_h) \subset W(G_h')$. As we showed above $G_k \bowtie G_h$, so $W(G_k) \cap W(G_h')$ cannot be empty and we infer that $W(G_k) \cap W(G_h) \cap W(G_h')$ is nonempty. In $\mathscr{P}'$, G_h' has no proper descendants that dominate G_i, so G_h and G_h' have different colors. Since any vertex can have at most two colors, G_k must have the color of G_h' or the color of G_h in $\mathscr{P}'$. In either case, G_k is an ancestor of G_h' or of G_h, and we infer that $G_k \geq G_h$ directly or by transitivity ($G_k \geq G_h'$ if G_k is an ancestor of G_h' in $\mathscr{P}'$, and $G_h' \geq G_h$ because G_h' is an ancestor of G_h in $\mathscr{P}$). We can complete the argument exactly as in Case 3 (from the point where we showed $G_k \geq G_h$).

Since any graph that dominates G_i must be G_h, G_k or one of their ancestors in $\mathscr{P}$, these four cases exhaust all candidates for G_h'. □

Lemma 5.10. *Suppose that $\mathscr{P}$ is a valid parent relation satisfying Invariant* 4.1. *Suppose that in* PROCESS(G_i) *we give up. Then $\hat{G}_i$ is not an undirected path graph.*

Proof. If we give up, G_i falls in case (f). By Theorems 3.1 and 3.7, if $\hat{G}_i$ is an undirected path graph, then E_i has a valid parent relation satisfying Invariant 4.1(ii), so we restrict attention to such relations. Lemmas 5.6 and 5.7 show that if there is a valid parent relation for E_i satisfying 4.1(ii) that has G_i as a child of c, then we find one in NEW-END. Lemmas 5.8 and 5.7 show that if G_i has only one possible parent and we quit both in SWAP and in NEW-END, then E_i has no valid parent relation satisfying 4.1(ii). Lemmas 5.9, 5.7, and 5.6 show that if G_i has two possible parents, we succeed in placing G_i using SWAP, we find a valid parent relation in which G_i is a child of c, or else E_i has no valid parent relation satisfying 4.1(ii). □

Lemma 5.11. *For each i, such that $1 \leq i \leq s$, Invariant* 4.1 *holds before we process G_i and afterwards.*

Proof. The lemma follows by induction on i. Remark 5.1 covers the base case. Lemmas 5.2, 5.4, 5.6, and 5.10 cover the induction step. □

Corollary 5.12. *The algorithm described in Sections* 2–4 *recognizes undirected path graphs.*

6. Implementation details and running time

The top level of the recognition algorithm described in Section 4, the routine SORT-G_i, and most aspects of PROCESS should be straightforward to implement. In this section, we describe data structures that can be used for implementation, describe some subtle details of PROCESS (particularly concerning SWAP and NEW-END), and prove a bound of $O(p(m+n))$ on running time.

Some data structures are local to a recursive call (Step 2 of the top level algorithm); unless otherwise noted, they are *global*. Let the vertices be numbered $1, 2, \ldots, n$, and let the cliques be numbered $1, \ldots, p$. We assume a list of cliques is computed in $O(m+n)$ time during the preliminary chordality test. Most data structures are described with respect to a separating clique C. If G_i has only two cliques, then either one may be chosen as C for these definitions.

The first data structure is an array CMEMBERS of size p that associates with each j, $1 \leq j \leq p$, the list of vertices in clique number j.

In Step 2, we copy C to each graph G_i. To avoid confusion, we assign each copy a distinct number larger than p. Each G_i contains one copy of the separating clique, so the number of new clique names is at most the number of recursive calls, which we prove below is at most $p-1$. Clique names are between 1 and $2p-1$ inclusive. The second data structure is an array CNAMES of size $2p-1$ that associates with each j, $1 \leq j \leq 2p-1$ the original number of the clique with name j. The members of clique name j are in CMEMBERS[CNAMES[j]], but we denote this set by C_j.

The third data structure is an $n \times n$ array F that stores the value $f(T_j, v)$ in entry $F[j, v]$. If there is no graph G_j or $v \notin W(G_j)$, then $F[j, v] = 0$. To expedite reinitializing after each recursive call, we keep F global; each call sets the necessary entries, and then resets these entries to zero, just before exiting.

The fourth data structure is an $n \times 2$ array VCOLORS that stores the colors for vertex v. If v has one color then VCOLORS$[v, 2] = 0$. If v has no colors or $v \notin V(C)$, then VCOLORS$[v, 1]$ = VCOLORS$[v, 2] = 0$. VCOLORS is local to a recursive call.

The fifth data structure is an array PARENT of length n. PARENT[j] is the index of the parent of G_j. We use 0 as the index of c and -1 for graphs with no parent. PARENT is local.

The sixth data structure is an $n \times n$ array CHILDREN. CHILDREN is local to a resursive call. If there is a graph G_j that has a child G_q with $v \in W(G_j) \cap W(G_q)$, then CHILDREN$[j, v] = q$; otherwise, CHILDREN$[j, v] = 0$. Reinitialization of nonzero entries is handled as for F.

The seventh data structure is an array TREE of at most $2p-1$ adjacency lists. If a graph H at some level of recursion is a UV graph, then after that call completes, the entries of TREE for the clique names of H yield the edges in a clique tree for H. The tree edges are undirected.

The eighth data structure stores $W(G_i)$ for each G_i in the current recursive call. The W sets are stored both as linked lists (to list W in time proportional to its size)

and as bitmaps (to test membership in constant time). Finally, we store the node c_i for each G_i; c_i is the unique neighbor of c in our tree for G_i.

The next lemma covers the running time of all the simple steps and outlines a strategy for bounding the cost of calls to PROCESS.

Lemma 6.1. *The cost of the main algorithm, summed over all recursive calls, is* $O(p(m+n))$ *if*:

(1) *all global data structures can be initialized in time* $O(pn+m)$,

(2) *all local data structures can be initialized in time* $\beta_1(m+n)$ *per call to* PROCESS,

(3) *a valid parent relation can be converted to a tree in* $\beta_2(n)$ *time,*

(4) *the number of operations resulting from each call to* PROCESS *is at most* $\beta_3(m+n)+O(1)$, *where* β_1, β_2, *and* β_3 *are constants that do not depend on* G_i *or the level of recursion.*

Proof. Each recursive call in Step 2 uses a different separating clique. Each separating clique is an original clique of G; thus, there are at most $p-1$ calls.

If clique Q does not separate a graph H, then Q cannot separate any subgraph of H. Therefore, among all calls, each clique can be tested for being a separating clique (in Step 1) at most once. Each test can be done with one connected components computation in $O(m+n)$ time. The cost for Step 1 over all calls is $O(p(m+n))$. As stated above, the time needed for one call to SORT-G_i is $O(m+n)$. Hence the time used by SORT-G_i over all recursive calls is $O(p(m+n))$.

In Step 5, we expand sets of equivalent graphs replacing the representative of such a set with a chain in the parent relation. This can be done in constant time per parent-child pair in the expanded parent relation by simple tree operations.

To complete a recursive call we convert a valid parent relation into a tree. Let T_i be the clique tree for G_i. For each T_i, we delete the edge incident to the node representing a copy of C in T_i and insert one edge between the node c_i and either c or an end in the tree for the parent of G_i. The end in the parent graph can be found by consulting at most n values of f (see Lemma 6.5 below). Hence we have to do at most $\beta_3 n$ operations per G_i to convert from parent relations to trees. We show next that at most $O(p)$ separated subgraphs can exist over all recursive calls.

To analyze PROCESS, associate each separated subgraph having three or more cliques with the clique name C that separates it at the next iteration. Associate each separated subgraph having exactly two cliques with a member clique (name). Each clique name that has some G_i associated with it is the name of a clique of G. Each clique name may have at most two graphs G_i associated with it: one that it separates, and one having one or two cliques. There are at most $2p-1$ clique names, and therefore at most $4p-2$ calls to PROCESS. If we can bound the time of a call by $\beta_2(m+n)+O(1)$, then the total cost over all calls to PROCESS is $O(p(m+n))+O(p)=O(p(m+n))$. Furthermore, we will have proved the same time bound for the entire algorithm. □

Most of this section concerns the implementation and analysis of PROCESS. We showed that over all recursive calls to the main algorithm there are $O(p)$ calls to PROCESS; we will show that each call to PROCESS can be carried out in $O(m+n)$ time. These bounds do not imply that each recursive call takes $O(m+n)$ time, as some recursive calls may make a nonconstant number of calls to PROCESS.

Lemma 6.2. *All global data structures can be initialized in time* $O(pn+m)$.

Proof. We can initialize CMEMBERS in time $O(m+n)$, CNAMES in time $O(n)$, and TREE in time $O(n)$. □

To simplify the notation, we assume for the rest of the section that there are no interchangeable separated subgraphs. Thus SORT-G_i just reorders the list of separated subgraphs, but does not compress it. This assumption is reasonable since any reduction in the number of separated subgraphs just reduces the number of calls to PROCESS and hence the overall worst-case running time of such calls.

Lemma 6.3. *If* $G_1, G_2, \ldots, G_s$ *are the separated subgraphs at a particular recursive call, then*

$$\sum_{1 \le i \le s} (|W(G_i)|) \le m+n.$$

Proof. By Lemma 2.4, the set $W(G_i)$ is contained in $V(C_i)$. Each clique C_i is distinct and is a clique in G by Remark 2.1. The lemma follows from Remark 1.4. □

Lemma 6.4. *All local data structures can be (re)initialized in time* $\beta_1(m+n)$, *per subgraph* G_i, *for some constant* β_1.

Proof. The arrays VCOLORS and PARENT are all of size at most $2n$. To find c_i, we find the unique neighbor of (the copy of) c in the tree for G_i. This can be done by examining each node. Then $W(G_i) = V(C) \cap V(C_i)$. We also need a bitmap representation of W.

To compute $f(T_i, v)$ we do one breath-first traversal of the model for G_i starting at c_i. Each time we visit a new node c_j', for each $v \in V(C_j')$, if $v \in V(C)$, we update $f(T_i, v)$. This costs constant time per pair (C_j, v), such that C_j is a clique in G_i and $v \in V(C_j)$. By Remarks 2.1 and 1.4, there are at most $m+n$ such pairs.

The values of F, W and c_i do not change while graphs in any given recursive call are processed. To reinitialize F and CHILDREN, we set to 0 each entry that was set during this call. By Remark 1.4, there are at most $m+n$ entries set during a recursive call. By keeping a list of such entries, we can reinitialize them in constant time per entry. The queues and tables used in SWAP are of size at most n. The array CONSOLIDATE used in NEW-END, is of size at most n. □

We showed in Section 4 that once we found all the ends for G_i we could distinguish among the six cases in $O(n)$ tests. We now bound the number of operations needed to find the ends.

Lemma 6.5. *Finding the ends for all vertices in $W(G_i)$ takes $\alpha_1(m+n)$ operations for some constant α_1.*

Proof. Any vertex $v \in W(G_i)$ has at most two colors that can be found in constant time in VCOLORS. For a particular color G_q, we find the highest-indexed graph with v in its W set and of color G_q by chasing pointers in CHILDREN. First we examine CHILDREN$[q, v]$. If it is 0, then G_q is the answer; if it is t, then we examine CHILDREN$[t, v]$, and so on. If G_r is the last graph with v in its W set, then $F[q, v]$ is the end we seek. Each probe of CHILDREN can be charged to a distinct pair (G_t, v) such that $v \in W(G_t)$. By Lemma 6.3 there are at most $m+n$ such pairs. □

Lemma 6.6. *To assign a new parent to G_i at most $\alpha_2(|W(G_i)|)$ data structure entries must be changed, for some constant α_2. Each change takes constant time.*

Proof. For each $v \in W(G_i)$ we may change one entry in VCOLORS (if the parent is c). For each $v \in W(G_i)$, we may change one entry in CHILDREN. We change one entry in PARENT. □

Lemma 6.7. *Testing whether $G_j | G_k$, or $G_j \leftrightarrow G_k$, or $G_j \geq G_k$,* or *$G_k \geq G_j$ takes at most $\alpha_3(min(|W(G_j)|, |W(G_k)|))$ operations for some constant α_3. Furthermore, suppose that for a fixed graph G_i, we want to classify all the children of another graph G_h that are antipodal to G_i, that dominate G_i or are dominated by G_i. This classification can be done in $\alpha_4(|W(G_i)|)$ operations, for some constant α_4.*

Proof. (For the first part) suppose without loss of generality that $|W(G_j)| \geq |W(G_k)|$ and $j < k$. First test for each $v \in W(G_k)$ if $v \in W(G_j)$; this can be done in $|W(G_k)|$ steps using the two representations of W. If the two sets are disjoint then $G_j \mid G_k$. If not, then $G_j \bowtie G_k$. If they are not disjoint and $W(G_k) \not\subset W(G_j)$, then $G_j \leftrightarrow G_k$. If $W(G_k) \subset W(G_j)$, we have two possibilities that are distinguished by values of f. If $f(T_j, v)$ is identical for every choice of $v \in W(G_k)$, then $G_j \geq G_k$; otherwise, the graphs are antipodal. We cannot have $G_j \geq G_k$ and $G_k \geq G_j$ because this makes the graphs interchangeable.

(For the second part) find all children that appear as CHILDREN$[G_h, v]$ for some $v \in W(G_i)$. These are the children of G_h attached to G_i. For each child G_k do the tests almost as above with G_i in the role of G_j. The only difference is that as soon as we find a single vertex in $W(G_k) \setminus W(G_i)$, we do no more membership tests in this direction because G_i cannot dominate G_k. Instead we reverse their roles and do the membership tests in the other direction. If neither W set is contained in the other, the graphs are antipodal. To bound the running time, the key observation is

that the W sets for the different children are disjoint. Thus we never do two successful membership tests for any vertex v. There are at most two unsuccessful tests per child G_k; these can be unambiguously charged to the probe of CHILDREN that found a vertex shared by $W(G_i)$ and $W(G_k)$. □

The next lemma gives some insight into how one can implement SWAP efficiently. It applies both to direct calls and to calls from within NEW-END. For any graph G_j such that $j<i$ and G_j is not a child of c in $\mathscr{P}$, let $G_{P(j)}$ denote the parent of G_j in $\mathscr{P}$.

Lemma 6.8. *Suppose we have called* SWAP *with first argument* G_h. *Let* $\mathscr{D}_h$ *be the set of proper descendants of* G_h *in* $\mathscr{P}$. *If* $G_j \in \mathscr{D}_h$ *is put on* SQUEUE-1, *then either* $P(j)=h$ *or* $G_{P(j)}$ *was on* SQUEUE-1 *before* G_j.

Proof. For simplicity of notation, we assume SWAP was called with second argument G_i. The same proof works for calls from NEW-END, where the second argument would be the set of graphs $\mathscr{M}_h$. Suppose G_j gets on SQUEUE-1 in Step 1 of SWAP. It must have been enqueued because it is antipodal to G_i. The graphs tested for antipodality in Step 1 are children of G_h or children of graphs that are already on SQUEUE-1.

Suppose as inductive hypothesis that the lemma holds for $l-1$ iterations of the loop at Steps 2–4, where l may be 1. Consider was happens on iteration l. In Step 2, some graphs are put on SQUEUE-2 because each one is antipodal to a graph formerly on SQUEUE-1 (that satisfies the induction hypothesis). In Step 3, we may put some graphs on SQUEUE-1. Suppose G_j is added to SQUEUE-1 because it is antipodal to G_k from SQUEUE-2. If $P(j)=h$, then there is nothing to prove, so assume $P(j)\neq h$. Since $G_j \leftrightarrow G_k$ and $G_{P(j)} \geq G_j$, we know that $G_{P(j)} \bowtie G_k$ and G_k cannot dominate $G_{P(j)}$. The possible relationships are $G_{P(j)} \leftrightarrow G_k$ and $G_{P(j)} \geq G_k$. If $G_{P(j)} \leftrightarrow G_k$, then $G_{P(j)}$ will be put on SQUEUE-1 in Step 3 before G_j, if it was not on SQUEUE-1 already.

Suppose $G_{P(j)} \geq G_k$. There is a graph G_r put on SQUEUE-1 either in Step 1 or in a previous loop iteration, such that $G_r \leftrightarrow G_k$, and this antipodality caused G_k to be put on SQUEUE-2. The sets $W(G_r) \cap W(G_k) \neq \emptyset$ and $W(G_k) \subset W(G_{P(j)})$. Thus there is a vertex $v \in W(G_r) \cap W(G_k) \cap W(G_{P(j)})$. Vertex v has at most two colors; at the beginning of SWAP neither G_r nor $G_{P(j)}$ had the same color as G_k. Hence G_r and $G_{P(j)}$ had the same color. Since they share v, one was a descendant of the other. Because of their relationships with G_k, it must be that $G_{P(j)} \geq G_r$. But G_r satisfies the induction hypothesis, and all ancestors of G_r that are proper descendants of G_h, including $G_{P(j)}$ were on SQUEUE-1 by the end of iteration $l-1$. □

We can use Lemma 6.8 in implementing SWAP. Lemma 6.8 is useful in Steps 2(c) and 3(c), where we search for the new parent of the graph G_j at the head of the queue. The obvious way to search for a parent is to find the second color, call it G_q for a vertex in $W(G_j)$ (the first color being the one G_j has before it was en-

queued) and to do a depth-first search in the parent relation starting at G_q and following pointers in CHILDREN, applying Lemma 6.7 to find the furthest graph that dominates G_j. Each graph can cause any CHILDREN entry to be probed at most once, but there is no obvious constant bound on the number of graphs that cause an entry to be probed.

For Step 2(c), Lemma 6.8 implies that if $P(j) \neq h$, then $G_{P(j)}$ has already been recolored with a color different than that of G_h. Since a vertex can have only two colors, the new color of G_j must be the color of $G_{P(j)}$ already received. Since $j > P(j)$ and $G_{P(j)} \geq G_j$, the new parent must be a descendant of $G_{P(j)}$. Thus we can start our depth-first search at $G_{P(j)}$ instead of at a child of c.

Step 3(c) is more subtle. We proved in Lemmas 5.7, 5.8, and 5.9, that at each use of SWAP the new parent of G_j is a descendant of G_h, provided that $G_h \geq G_j$. G_j was put on SQUEUE-2 because it is antipodal to a former proper descendant of G_h, call it G_r. By Lemma 6.8, any graphs that started SWAP as ancestors of G_r and proper descendants of G_h already have a new color. Thus the new parent of G_j must be either G_h or a graph that had a different color at the beginning of SWAP. Let the set of such graphs, that now are proper descendants of G_h be B. Because of the two colors per vertex rule, if $G_j \bowtie G_l$, for $G_l \in B$, then G_j and G_l had the same color in $\mathscr{P}$. Since $\mathscr{P}$ is valid, G_j cannot be antipodal to a member of B. Thus we do not actually need the parent-child relationships among members of B until the end of SWAP. For the purposes of antipodality tests against nonmembers of B, we pretend that G_j gets G_h as a parent and test G_j against proper descendants of G_h that started with the same color as G_h and whose $\mathscr{P}$ parents have already been swapped. The proper descendants of G_h in $\mathscr{P}$ may be put on SQUEUE-1. However, we do not actually assign a parent to G_j or change the parents of original descendants of G_h unless they are antipodal to G_j or their parent gets enqueued. The next lemma states that once SWAP is completed, we can find the correct parents and children for all members of B in $O(m+n)$ time.

Lemma 6.9. *Suppose we call* SWAP *with first argument* G_h. *Let* B *be the set of graphs put on* SQUEUE-2 *during the call, and assume we implement Step* 3 *as described above. It is possible to assign a parent to every member of* B *and to change the parents of some descendants of* G_h *in* $\mathscr{P}$, *so that Invariant* 4.1 *is preserved, and the assignments take* $\alpha_5(m+n)$ *operations for some constant* α_5.

Proof. We proved above that no two members of B can be antipodal, and that every member of B either gets a parent in B or gets G_h as parent. We sort the members of B using SORT-G_i again.

We use an array END indexed on $V(G)$, such that END[v] is the highest index of a graph in B that already has a parent and also has v in its W set. Initially all END entries are 0. For each $G_r \in B$ in increasing order of r, followed by all current children of G_h (recall that these may be attached to members of B), find $t = \max_{v \in W(G_r)}(\text{END}[v])$. If $t = 0$, then make G_r a child of G_h. Otherwise, make G_r

a child of G_t. In either case, update the END entries for vertices in $W(G_r)$. The work for each G_r can be done in time proportional to $|W(G_r)|$. The call to SORT-G_i takes $O(m+n)$ time. By Lemma 6.3, the total time is $\alpha_5(m+n)$ for some constant α_5. □

Lemma 6.10. *A call to* SWAP takes $\alpha_6(m+n)$ *operations, for some constant* α_6.

Proof. By Lemma 6.7, each antipodality test for G_j in Step 1 can be done in time proportional to $|W(G_j)|$. By Lemma 6.6, each change of parent in Step 1 takes time proportional to the size of the W set of the child. Thus by Lemma 6.3, Step 1 takes time $O(m+n)$. Each G_j is enqueued at most once during SWAP. Step 2(b) takes constant time. We discuss Step 2(c) below. In Step 2(d), any successful antipodality test can be charged to the W set of the graph enqueued as a result. All antipodality tests against G_j that fail, correspond to unattached siblings; by Lemma 6.7, they take time proportional to $|W(G_j)|$. Steps 2(e) and 2(f) take time proportional to $|W(G_j)|$, by Lemmas 6.6 and 6.7. Steps 2(g) and 2(b) take constant time. The analysis of Step 3 is identical to that of Step 2, except for Step 3(c), analyzed in Lemma 6.9. Step 4 takes constant time. Step 5 takes time proportional to $|W(G_h)|$, by Lemma 6.6. Resetting entries in Step 6 can be charged as part of the cost of setting them.

All that remains is Step 2(c). By Lemma 6.8, the elements on SQUEUE-1 are enqueued in a topological order consistent with $\mathscr{P}$. If we implement Step 2(c) as described above, the search for the parent of $G_{P(j)}$ does not probe any of the same entries in CHILDREN as the search for the new parents of the children of $G_{P(j)}$ (or any proper descendants) in $\mathscr{P}$. In any search, and for any fixed vertex v, each CHILDREN probe under vertex v is successful, except possibly the last probe. Each succesful probe can be charged to a pair (G_q, v) that such $v \in W(G_q)$. The last probe can be charged to (G_j, v), where G_j is the graph whose new parent we seek. Each (graph, vertex) pair can be charged at most once for a successful probe and at most once for an unsuccessful probe. By Lemma 6.3, the total cost of Step 2(c) is proportional to $(m+n)$. □

Lemma 6.11. *One call to* NEW-END *takes at most* $\alpha_7(m+n)$ *operations for some constant* α_7.

Proof. In Step 1, we test for each G_j, $j<i$, whether $W(G_j) \cap W(G_i) \neq \emptyset$ in time proportional to $|W(G_j)|$; by Lemma 6.3, the total worst-case time is proportional to $m+n$. Step 2 takes constant time per graph. Step 3 takes time proportional to $|W(G_i)|$. Step 4 can be implemented efficiently using a similar method to that used in the proof of Lemma 6.9. Keep a table of the end for each vertex in the W set of the first member of $\mathscr{M}$. Using this table of ends one can compute the necessary domination tests achieving the bounds of Lemma 6.6. Updating the table of ends takes time proportional to $|W(G_h)|$ when we find the eventual parent of G_h. Thus

the total time in Step 4 is proportional to the sum of $|W|$ over all consolidated graphs, which is at most $m+n$, by Lemma 6.3.

The analysis of Step 5 is more subtle. It may appear that we can make $O(n)$ calls to SWAP and the time bound for SWAP is too high. In SWAP, define a vertex v to be *moving* if there exists a graph G_j such that $v \in W(G_j)$ and G_j is on one of the queues at some time during the call. The proof of Lemma 6.9 shows that except for a constant amount of overhead, all operarations in SWAP can be charged to pairs (G_j, v), where $v \in W(G_j)$ and v is a moving vertex.

We claim that in any call to NEW-END any vertex can be moving in at most three successful calls to SWAP. At most one SWAP call is unsuccessful, since such a call causes us to quit NEW-END. A vertex v can be moving for two reasons. The first is that $v \in W(G_q)$, $G_q \in \mathcal{N}_h$ and we are about to place the children of G_h that are in $\mathcal{M}_h$. Because $\mathcal{P}$ is valid, this cannot happen for two different choices of the first argument to SWAP that have the same color in $\mathcal{P}$ (G_q would be attached to a proper descendant of G_h that is in $\mathcal{M}$ and hence neither an ancestor nor a descendant of G_q). Since v has at most two colors, v can be moving for this reason at most two times. The first reason accounts for all situations where v is moving by virtue of a graph placed on SQUEUE-1. If $v \in W(G_q)$ for some G_q on SQUEUE-2, but not in the W set for anything on SQUEUE-1, then G_q will be made a descendant of some $G_r \in \mathcal{M}$, and no graphs in $\mathcal{M}_r$ can have v in their W sets. Thus none of them dominates a graph with v in its W set, and none of them can have a descendant with v in its W set in a valid parent relation. Therefore if a graph with v in its W set is enqueued on SQUEUE-2 in a later call to SWAP, the call fails. Thus the running time of calls to SWAP from within NEW-END is at most four times the bound in Lemma 6.10 (which is $O(m+n)$), plus $O(1)$ overhead for each of the at most n calls. □

Theorem 6.12. *Undirected path graphs can be recognized in time* $O(p(m+n))$.

Proof. This follows from Lemmas 6.1, 6.2, 6.4, 6.5, 6.6, 6.10, and 6.11. □

Acknowledgement

One referee made many helpful suggestions that led to substantial simplifications in the algorithm and improvements in the exposition.

References

[1] C. Beeri, R. Fagin, D. Maier and M. Yannakakis, On the desirability of acyclic database schemes, J. ACM 30 (1983) 479–513.

[2] K.S. Booth and G.S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, J. Comput. System Sci. 13 (1976) 335–379.

[3] P. Buneman, A characterization of rigid circuit graphs, Discrete Math. 9 (1974) 205–212.

[4] P.F. Dietz, Intersection graph algorithms, TR 84-628 (Ph.D. Thesis), Computer Science Department, Cornell University, Ithaca, NY (1984).
[5] R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes, J. ACM 30 (1983) 514–550.
[6] F. Gavril, Algorithms for minimum coloring, maximum clique, minimum covering by cliques and maximum independent set of a chordal graph, SIAM J. Comput. 1 (1972) 180–187.
[7] F. Gavril, The intersection graphs of subtrees in trees are exactly the chordal graphs, J. Combin. Theory Ser. B 16 (1974) 47–56.
[8] F. Gavril, A recognition algorithm for the intersection graphs of directed paths in directed trees, Discrete Math. 13 (1975) 237–249.
[9] F. Gavril, A recognition algorithm for the intersection graphs of paths in trees, Discrete Math. 23 (1978) 211–227.
[10] P.C. Gilmore and A.J. Hoffman, A characterization of comparability graphs and interval graphs, Canad. J. Math. 16 (1964) 539–548.
[11] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs (Academic Press, New York, 1980).
[12] D.S. Johnson, The NP-completeness column: an ongoing guide, J. Algorithms 6 (1985) 434–451.
[13] C.L. Monma and V.K. Wei, Intersection graphs of paths in a tree, J. Combin. Theory Ser. B 41 (1986) 141–181.
[14] M.B. Novick, Parallel algorithms for intersection graphs, TR 90-1096 (Ph.D. Thesis), Computer Science Department, Cornell University, Ithaca, NY (1990).
[15] P.L. Renz, Intersection representations of graphs by arcs, Pacific J. Math. 34 (1970) 501–510.
[16] D.J. Rose, Triangulated graphs and the elimination process, J. Math. Anal. Appl. 32 (1970) 597–609.
[17] D.J. Rose, A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in: R.C. Read, ed., Graph Theory and Computing (Academic Press, New York, 1972) 183–217.
[18] D.J. Rose, R.E. Tarjan and G.J. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. 5 (1976) 266–283.
[19] R.E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (1984) 566–579.
[20] M. Yannakakis and F. Gavril, The maximum k-colorable subgraph problem for chordal graphs, Inform. Process. Lett. 24 (1987) 133–137.