ELSEVIER

Discrete Mathematics 201 (1999) 189–241

DISCRETE MATHEMATICS

Modular decomposition and transitive orientation

Ross M. McConnell [a,1], Jeremy P. Spinrad [b,*]

[a] *Department of Computer Science and Engineering, University of Colorado at Denver, P.O. Box 173364, Denver, CO 80217-3364, USA*

[b] *Department of Computer Science, Vanderbilt University, Nashville, TN 37235, USA*

Abstract

A *module* of an undirected graph is a set X of nodes such for each node x not in X, either every member of X is adjacent to x, or no member of X is adjacent to x. There is a canonical linear-space representation for the modules of a graph, called the *modular decomposition*. Closely related to modular decomposition is the *transitive orientation problem*, which is the problem of assigning a direction to each edge of a graph so that the resulting digraph is transitive. A graph is a *comparability graph* if such an assignment is possible. We give $O(n+m)$ algorithms for modular decomposition and transitive orientation, where n and m are the number of vertices and edges of the graph. This gives linear time bounds for recognizing permutation graphs, maximum clique and minimum vertex coloring on comparability graphs, and other combinatorial problems on comparability graphs and their complements.

1. Introduction

A partial order may be viewed as a transitive directed acyclic graph. A *comparability graph* is the graph obtained by ignoring the edge directions of a transitive directed acyclic graph. It is well known that every partial order is the intersection of a set of total (linear) orders [10]. A *two-dimensional partial order* is a partial order that is the intersection of two linear orders, and a *permutation graph* is the corresponding comparability graph. These classes of graphs and partial orders arise in many combinatorial problems and have applications in scheduling theory. For a survey, see [18,24].

Let $V(G)$ denote the vertices of a graph G. A *module* is a set X of vertices such that for any $x \in V(G) - X$, either x is adjacent to every element of X or x is adjacent to no element of X. The *modular decomposition* of G is an $O(n)$-space representation of the modules of G, which may be exponential in number. The decomposition was first

[1] Formerly in the Department of Mathematics and Computer Science, Amherst College. Supported in part by the graduate school 'Algorithmische Diskrete Mathematik', which is supported by the Deutsche Forschungsgemeinschaft, grant WE 1265/2-1.

* Corresponding author. E-mail: spin@vuse.vanderbilt.edu

0012-365X/99/$ – see front matter
PII: S0012-365X(98)00319-7

described in the 1960s by Gallai [16], and is also known as *substitution decomposition* [24], *prime tree decomposition* [13,14,12], and *X-join decomposition* [20], sometimes in a generalized context. For a survey, see [24].

The *transitive orientation problem* is the problem of orienting the edges of a comparability graph so that the result is a partial order. If such an orientation is provided for a comparability graph, a large number of combinatorial problems, such as maximum clique and minimum vertex coloring, are solvable in linear time.

There have been a number of $O(n^4)$, $O(n^3)$, $O(nm)$, and $O(n^2)$ algorithms for finding modular decomposition [2,11,17,20,22,26,33], some of them for special cases or generalizations of the problem. The cotree decomposition of cographs and the series–parallel decomposition of series–parallel partial orders are special cases on graphs and digraphs, respectively, for which linear-time solutions have been given [7,34]. $O(n+m \log n)$ [8] and $O(n+m\alpha(m,n))$ [31] bounds for arbitrary undirected graphs have recently been given. Here, we give a modification of the algorithm of [31] that eliminates the $\alpha(m,n)$ factor in the time bound for modular decomposition, giving a linear time bound for the problem. A summary of the new algorithm was previously given at [23]. Cournier and Habib [9] have since found a linear-time algorithm that is general to directed, as well as undirected, graphs. Their algorithm is thus preferable to the one we describe here if just the modular decomposition is desired. We have found a way to modify our decomposition algorithm to give a linear time bound for the transitive orientation problem, and we have not been able to do this with theirs.

The previous algorithms for transitive orientation took either $O(n^2)$ time [30], or $O(\delta m)$ time [17,19,27], where δ is the maximum degree of any vertex in the graph. A recent linear-time algorithm for the case where the comparability graph is also triangulated (chordal) is given in [21]. (A graph is triangulated or chordal, if every cycle on four or more vertices has a chord.)

Our algorithm produces a *linear extension* (topological sort) of the transitive orientation, that is, a total ordering of the nodes such that whenever (a,b) is an undirected edge, and b is a successor of a in the ordering, then the orientation of (a,b) in the transitive orientation is from a to b. As with the algorithm of [30], our transitive orientation algorithm fails to recognize when its input is not a comparability graph, and instead produces a nontransitive orientation of the graph. Surprisingly, this shortcoming is not an obstacle to many applications.

The following problems can be solved on comparability graphs in $O(n+m)$ time with the algorithm. A key element in many of the results is that if G is a graph whose complement is a comparability graph, our algorithm can produce a linear extension of a transitive orientation of the *complement* of G in time that is linear in the size of G.

1. *Recognition of permutation graphs and two-dimensional partial orders*: Recognition of partial orders of dimension k, where k is greater than two, is NP complete [35]. Recognition of two-dimensional partial orders clearly reduces to recognition of permutation graphs. Previous $O(n^3)$ and $O(n^2)$ algorithms for the problems have been given [4,27,30,29]. A graph G is a permutation graph if and only if both G

and its complement are comparability graphs [18]. If G is a comparability graph, we produce linear extensions of transitive orientations of G and its complement in $O(n+m)$ time. From these two linear extensions, we construct two linear orders whose intersection gives G. If G is not a permutation graph, the algorithm produces two linear orders whose intersection fails to give G. It may be verified in time linear in the size of G whether the linear orders give G.

2. *Recognition of cointerval graphs and interval graphs:* A graph is an interval graph if it is the intersection graph of a set of intervals on the line. A cointerval graph is the complement of an interval graph. A cointerval graph is a comparability graph, since one interval occurring before another is a transitive relation. For cointerval graph recognition, we produce a linear extension of the transitive orientation of the cointerval graph. From this, it is easy to construct a set of intervals that realize the cointerval graph. If the graph is not a cointerval graph, we detect that it is not possible to construct such a set of intervals. Since we have a linear time bound for finding a linear extension of a transitive orientation of the complement of a graph, the same procedure gives linear-time interval graph recognition, though this bound is already known for this problem [1].
3. *Recognition of circular permutation graphs* [28]: A circular permutation graph is a graph where each vertex of G corresponds to a chord connecting two concentric circles, and where two vertices are adjacent in the graph if and only if the corresponding chords intersect each other. Using our bounds for transitive orientation and permutation-graph recognition, R. Sritharan has obtained linear bounds for recognition of circular permutation graphs [32].
4. *Maximum clique and minimum vertex coloring in comparability graphs*: Transitively orient G and, using a depth-first traversal, label each node with the length of the longest path originating at the node in the result. This gives a vertex coloring on G. The size of a clique is a lower bound on the number of colors in any vertex coloring. If the longest path corresponds to a clique, then since the labeling is a vertex coloring of the same size, the longest path and the labeling are a maximum clique and minimum vertex coloring. If the longest path is not a clique, then the orientation of the edges is not transitive, and the input graph is not a comparability graph.
5. *Maximum independent set and minimum clique cover in co-comparability graphs*: A graph is a co-comparability graph if its complement is a comparability graph. Using a linear extension of the transitive orientation of the complement of the graph, we are able to label each node according to the length of the longest path beginning at a node in the transitive orientation of the complement, in $O(n+m)$ time. The result then follows in the same way that it does for maximum clique and minimum vertex coloring on comparability graphs.

2. Preliminaries

In this paper, we will consider only graphs that have no loops or multiple edges. If G is a graph or digraph, the set of nodes of G is denoted $V(G)$. If X and Y are

disjoint subsets of nodes of G, an edge (x, y) such that $x \in X$ and $y \in Y$ is said to *go from X to Y*. An edge *goes between* X and Y if it either goes from X to Y or from Y to X. X has an *outgoing edge* if there is an edge that goes from X to $V(G) - X$. If $x \in V(G)$ then the *neighbors of x*, denoted $N_G(x)$ are the set $\{y : (x, y)$ is an edge of $G\}$. The *nonneighbors* of x are $V(G) - N_G(x)$ and denoted $\overline{N}_G(x)$. If $X \subseteq V(G)$, then the neighbors of X, denoted $N_G(X)$, are given by $\bigcup\{N_G(x) : x \in X\} - X$. The subscript G may be dropped when it is understood.

If G is a graph or digraph, a *2-edge* is any pair (x, y) such that $x, y \in V(G)$. The *color* of a 2-edge (x, y) will be defined to be 1 if (x, y) is an edge of G, and 0 if it is not. This is the coloring of the 2-edges given by the adjacency array representation of G. A node x of G *distinguishes* or *splits* nodes y and z if (x, y) and (x, z) are not the same color or (y, x) and (z, x) are not the same color. Alternatively, when x distinguishes y and z, we may say that y and z *disagree on x*. A *module* is a set $X \subseteq V(G)$ such that no node in $V(G) - X$ distinguishes members of X. The *trivial modules* are $V(G)$ and the singleton subsets. It is easily seen that if two modules are disjoint, all 2-edges that go between the modules are the same color.

If G is a graph or digraph, and $X \subseteq V(G)$, then the subgraph induced in G by X is denoted $G|X$. If $\mathscr{P}$ is a partition of the nodes of G, a *system of representatives* from $\mathscr{P}$ is a set consisting of one node from each member of $\mathscr{P}$. If each member of $\mathscr{P}$ is a module of G, then $\mathscr{P}$ is called a *congruence partition* [25], and all systems of representatives induce isomorphic subgraphs. This subgraph is denoted $G/\mathscr{P}$, and completely specifies the colors of all edges that are not internal to a member of $\mathscr{P}$.

Definition 2.1. Two sets *overlap* if they intersect and neither of them contains the other. A *decomposable set family* $\mathscr{F}$ on a universe U is a set family with the following properties [3,25]:

1. U and its singleton subsets are members of $\mathscr{F}$.
2. Whenever X and Y are overlapping members of $\mathscr{F}$, then $X \cap Y$, $X \cup Y$, $X - Y$, and $X \Delta Y$ are also members of $\mathscr{F}$, where Δ is the symmetric set difference $X - Y \cup Y - X$.

Let $\mathscr{F}$ be a decomposable set family defined on universe U. Suppose no two members of $\mathscr{F}$ overlap. Then the transitive reduction (Hasse diagram) of the subset relation on $\mathscr{F}$ is a rooted tree that has one leaf for each member of U. If we create a data structure with one node x for each $X \in \mathscr{F}$, the leaf descendants of the node give X, and they may be listed in $O(|X|)$ time by traversing the subtree rooted at x. We will call such a tree a *union tree* on G. The union tree represents $\mathscr{F}$ in $O(|U|)$ space. We will refer interchangeably to a node of the union tree and the set it represents.

If T is a union tree on universe U, the family of children of a node X in T will be denoted $children_T(X)$. A subfamily of $children_T(X)$ is *nontrivial* if it consists of at least two, but not all, of its members.

Central to this paper is the observation that a union tree may be used to represent an *arbitrary* decomposable family in $O(|U|)$ space. Let the *strong members* of a decomposable set family be those that overlap no other member.

Theorem 2.2 (Möhring [25]). *The strong members of a decomposable set family $\mathscr{F}$ on universe U define a union tree, T, on U. For each strong member X of $\mathscr{F}$, one of the following cases applies:*
1. *X is* degenerate: *the union of any subfamily of $children_T(X)$ is a member of $\mathscr{F}$;*
2. *X is* prime : *no union of any nontrivial subfamily of $children_T(X)$ is a member of $\mathscr{F}$.*

An arbitrary decomposable set family may be represented by constructing the union tree of Theorem 2.2 and labeling the nodes degenerate or prime. This labeled union tree will be called the *decomposition tree* for the family.

Theorem 2.3 (Möhring [25]). *The family of modules of an undirected graph is a decomposable set family.*

The modular decomposition of an undirected graph G is precisely the decomposition tree for the family of modules of G, and will be denoted $MD(G)$. The strong members of the family of modules are called *strong modules*.

A graph is *prime* if it has no nontrivial modules, and *degenerate* if every subset of its nodes is a module. A graph is degenerate iff all of its 2-edges are the same color, that is, if the graph is either complete or edgeless. Prime and degenerate graphs are the only kinds of graphs that have decomposition trees of height 1; the root of a prime graph's decomposition tree is labeled prime while the root of a degenerate graph's tree is labeled degenerate.

The following two theorems show that modules are preserved under some types of mappings of nodes of one graph to another.

Theorem 2.4 (Möhring [25]). *If $\mathscr{P}$ is a congruence partition on an undirected graph G, then the union of a family $\mathscr{F}$ of classes in $\mathscr{P}$ is a module in G if and only if the corresponding set of nodes of $G/\mathscr{P}$ is a module in $G/\mathscr{P}$.*

Theorem 2.5 (Möhring [25]). *If X is a module of an undirected graph G, then the modules of G that are subsets of X are given by the modules of $G|X$. The strong modules of G that are proper subsets of X are given by the strong modules of $G|X$.*

Let U be a strong module of G, and let T be the modular decomposition of G. By Theorems 2.4 and 2.5, if U is labeled degenerate in T, then $(G|U)/children_T(U)$ is degenerate, and if U is labeled prime in T, then $(G|U)/children_T(U)$ is prime. If U is labeled degenerate, it is either a *1 node* which means $U/children_T(U)$ is complete, or a *0 node*, which means $U/children_T(U)$ is edgeless.

Theorem 2.6. *Let G be a graph and let T be its modular decomposition. If U and W are parent and child in T and both degenerate, then one of them is a 0 node and one is a 1 node.*

Theorem 2.7 (Corneil et al. [6]). *In every prime undirected graph G there exist four nodes $\{a,b,c,d\}$ such that the edges of $G|\{a,b,c,d\} = \{(a,b),(b,c),(c,d)\}$.*

Such an induced subgraph $G|\{a,b,c,d\}$ is called a P_4, and a P_4 is the only prime undirected graph on four or fewer nodes. By Theorems 2.4 and 2.5, the subgraph induced by a system of representatives from the children of a prime node of $MD(G)$ is a prime graph, while a system of representatives from the children of a degenerate node is either a complete or an edgeless graph. Thus, by Theorem 2.7, a graph has a P_4 iff its modular decomposition has prime nodes. If there is no P_4 in the graph, the graph is known as a *cograph*, and the modular decomposition, which consists exclusively of degenerate nodes, is often called its *cotree* [6].

3. The modular decomposition algorithm

A decomposition tree T on the nodes of G is an *M tree* if the modules of G are a subfamily of the decomposable family it represents. Note that the set represented by a node of an M tree does not overlap any module of G. If T_1 and T_2 are decomposition trees, then we will say T_2 is *stronger* than T_1 if the decomposable family it represents is a subfamily of the one that T_1 represents. $MD(G)$ is clearly the strongest possible M tree. The algorithm works by starting with a weak M tree called a P_4 *tree* [31] and computing a sequence of increasingly stronger M trees until $MD(G)$ is obtained. We distinguish three classes of M trees that characterize T during different phases of the refinement.

M1: Internal nodes are labeled prime or degenerate, and for each degenerate node U, there exists system of representatives from $\text{children}_T(U)$ that induces a degenerate subgraph in G.

M2: Internal nodes are labeled prime or degenerate, and for each degenerate node U, the members of $\text{children}_T(U)$ are modules in $G|U$ and $(G|U)/\text{children}_T(U)$ is degenerate.

M3: Same as M2, but with the additional constraint that every node of T is a module in G.

By Theorem 2.3, $MD(G)$ is the unique M3 tree on G. We give a linear algorithm to compute this tree from an M2 tree and linear algorithm to compute an M2 tree from an M1 tree.

Computing an M3 tree from an M2 tree is straightforward; the algorithm is given in Section 3.2. In [31], a linear algorithm is given that computes a P_4 tree, which is an M1 tree. Thus, the only difficult step is computing an M2 tree from an M1 tree, T. To do this, we perform a sequence of operations on T that remove members of the decomposable set family it represents, subject to the invariant that T continues to be an M1 tree. Such a strategy eventually leads to an M2 tree. The basic operations are to change a node's label from degenerate to prime, and to insert a new node that is a union of some of the children of a degenerate node. Each of these operations

must obviously restrict the decomposable set family represented by the tree, and if it is done carefully, all modules of G remain members of the new family. We will call this operation a *refinement of U*. A *refinement of T* is obtained by a sequence of refinement operations on its nodes.

Remark 3.1. Any M tree obtained by refining an M1 tree is also an M1 tree.

3.1. Basic procedures

It is an easy exercise to verify that if there are no degenerate nodes in an M tree, then just having a label on each node of the tree indicating whether it is a module in G allows one to produce the modular decomposition by simply discarding the nonmodules from the family of sets that make up the nodes of the tree. From this observation, we see that we need additional information only about children of degenerate nodes. In this section we show the following:

- How to label nodes of an M tree as to whether they are modules.
- How to label all children of degenerate nodes with a list of graph vertices that split them.

The first step can be computed directly on any M tree (or even any union tree) on $V(G)$. The second step requires us to produce a slightly stronger variant of the given M tree before computing the lists, in order to ensure that the sum of cardinalities of the computed lists is $O(n+m)$.

A directed tree whose internal nodes are labeled prime or degenerate represents a decomposable set family on its leaves; each internal node becomes identified with the set of leaf descendants it has, and its labeling as prime or degenerate tells which unions of its children are members of the family. If the tree has nodes that have only one child, the decomposable set family represented by the tree is still defined, but the nodes with only one child have no effect on the family represented by the tree. In this section, the data structure will occasionally develop an internal node that has only one child. In this case, we will perform a *contraction* to restore the tree to its canonical form. The obvious operation is defined formally as follows:

Definition 3.2. Let u be an internal node of a directed tree that has only one child, c. A *Contraction* on u is defined as follows: If u has a parent p, then move c and the subtree rooted at it to be a child of p, and remove u from the tree. If u has no parent, then remove it from the tree and let c be the new root.

Definition 3.3. Assume that T is a union tree on graph G and that the vertices of G are numbered in the order in which they are encountered in a depth-first traversal of T. It follows that for any $X \in T$, X is given by an interval $[MIN(X)\ldots MAX(X)]$ on the numbering of vertices. Let $low(X)$ and $high(X)$ give the lowest- and highest-numbered vertices of G that distinguish members of X. Thus, X is a module iff $min(X) \leqslant low(X) \leqslant high(X) \leqslant max(X)$. Let $SN(X)$ denote the *strong neighbors* of X,

that is, those vertices that are adjacent to every element of X. These are all given inductively as follows. For each $i \in V(G)$, $SN(\{i\}) = N(i)$, $low(i) = high(i) = MIN(i) = MAX(i) = i$, and $W(i) = \emptyset$. For each internal node X of T:

- $SN(X) = \bigcap\{SN(Y) : Y \in children_T(X)\}$,
- $W(X) = \bigcup\{SN(Y) : Y \in children_T(X)\} - SN(X)\}$,
- $MIN(X) = min\{MIN(Y) : Y \in children_T(X)\}$,
- $MAX(X) = max\{MAX(Y) : Y \in children_T(Y)\}$,
- $low(X) = min\{\{low(Y) : Y \in children_T(X)\} \cup W(X)\}$,
- $high(X) = max\{\{high(Y) : Y \in children_T(X)\} \cup W(X)\}$,

Though we might like to make a list for each X of all nodes not in X that distinguish members of X, the sum of cardinalities of such lists can exceed $O(n + m)$, so this is not possible in linear time. However, observe that the nodes that distinguish members of X are given by $\bigcup\{W(Z) : Z$ is a descendant of $X\} - X$.

Since all of these terms are defined by induction on the height of a node in the tree, we may compute them with a postorder traversal with the following procedure:

Procedure **Modules** (G, T) [31]

Input: An undirected graph G and an arbitrary union tree T on G

Result: A list for each $X \in T$ that gives $SN_G(X)$ and, if X is not a module, a vertex $x \in V(G) - X$ that splits X.

Number the nodes of G in the order in which they are encountered in a depth-first traversal of T. Do a two-pass radix sort the edges of the graph to get the adjacency lists sorted according to this order. This ordering will be maintained on the SN lists. Initialize $MIN(i), MAX(i), low(i), high(i) = i$, $SN(i) = N(i)$ and $W(i) = \emptyset$ for each leaf i of T.

For each internal node X in a postorder traversal of T
  Let $\{Y_1, Y_2, ..., Y_k\}$ be the children of X
  $SN(X) := SN(Y_1); W(X) := \emptyset$
  For $i := 2$ to k do
    $W(X) := W(X) \cup (SN(X) \Delta SN(Y_i))$
    $SN(X) := SN(X) \cap SN(Y_i)$
  Compute $MIN(X), MAX(X), low(X), high(X)$ as described by their definition
  If $MIN(X) > low(X)$ then label X with splitting node $low(X)$
  Else if $MAX(X) < high(X)$ then label X with splitting node $high(X)$
  Label X as a module

There are $O(n)$ nodes in the tree, since each node has at least two children. The degree sum of all SN lists is $O(m)$ since the degree sum of SN lists of leaves is $O(m)$ and $|SN(X)| \leqslant 1/2 \sum\{|SN(Y_i)| : Y_i \in children_T(X)\}$ for each inner node X. The time spent in the inner loop may be charged to elements of SN lists of children of X since

all SN lists are in the same sorted order, giving an $O(n+m)$ bound for the running time and on the sum of cardinalities of all computed lists.

Remark 3.4. When a vertex x is inserted in $W(X)$ in the first line of the inner loop, it is either adjacent to Y_i and nonadjacent to Y_{i-1} or *vice versa*. Thus, for arbitrary $a \in Y_{i-1}$ and $b \in Y_i$, a and b are a certificate that X disagrees on x.

We now show how to compute a refinement of an M tree on which for each node Y whose parent is degenerate, Y is labeled with a full list of vertices of $V(G) - Y$ that it disagrees on.

Procedure **Splitters**(G, T) [31]:

Input: a graph G and an M1 tree T on G

Result: A refinement T and a list of splitting
nodes for each child of a degenerate node of the refinement

Run **Modules** (G, T) using T as the union tree and G as the graph.
For each degenerate node X in T
  Relabel X prime
  Create a new tree node z labeled degenerate
  For each child Y of X such that $low(Y) \geqslant MIN(X)$ and $high(Y) \leqslant MAX(X)$
    Move Y to be a child of z
  If z now has children then
    Make z a child of X
  If z is now the only child of X then
    Contract X out of the tree to avoid development of chains
  If z has only one child then
    Contract z out of the tree
Let T_r denote the state of T at this point

For each degenerate node U in T_r
  For each child Y of U in T_r do
    Let $T(Y)$ be the subtree of T_r rooted at Y
  Let $T'(Y)$ be $T(Y)$ minus subtrees rooted at degenerate nodes of $T(Y)$
  Let $Disagree(Y) := \bigcup\{W(Z) : Z$ is a descendant of Y in $T'(Y)\}$
  For each $i \in Disagree(Y)$
    If $MIN(Y) \leqslant i \leqslant MAX(Y)$ then
      Remove i from $Disagree(Y)$
Recompute $MIN(X)$, $MAX(X)$, $low(X)$, $high(X)$ for each node of T_r

Any union of children of X that did not become children of z in T_r fails to be a module, since each of these children is split by a node lying outside of X. Thus, changing X to be a prime node with child z preserves the property that T_r is still an M tree. It remains an M1 tree by Remark 3.1.

T_r has the important property that for any Y, W such that Y and W are children of degenerate nodes and Y is an ancestor of W, then all nodes that split W reside inside Y. This follows from the fact that W was not split by any node outside its parent in the input tree, otherwise it would have been turned into a child of a prime node in the first `for` loop. Thus, nodes that split W are irrelevant to computation of nodes that split Y. This allows $T'(Y)$, rather than $T(Y)$, to be searched for nodes that split Y. $T'(Y)$ may be discovered and visited by depth-first traversal starting at Y. Each node of T lies in $T'(Y)$ for at most one child Y of a degenerate node, so for any $Z \in T$, $W(Z)$ is examined at most once over all iterations. Initial *Disagree* lists may be generated by concatenating the relevant lists; this results in *Disagree* lists where the same node may appear more than once, which is all that is needed for our purposes. The lists can be sorted and purged of duplicate members in $O(n+m)$ time by using a collective radix sort of the members of all lists, using disagree-list number and vertex number as sort keys.

3.2. Constructing the modular decomposition from an M2 tree

Recall that the three main steps of the algorithm are constructing an M1 tree, finding an M2 tree given an M1 tree, and finding the modular decomposition from an M2 tree. In this section we describe the last of these steps.

Lemma 3.5. *Let T be an M2 tree for an undirected graph G. A set W of nodes of G is a strong module if and only if either:*
1. *W is a member of T that is a module;*
2. *It is a maximal union of children of a degenerate node X that is not distinguished by any node in $V(G)-X$.*

Proof. Every non-trivial module of G is a union of children of two or more children of some node U in T. (If a module of G is a node of T, then it is the union of all of the children of that node.) Suppose W is a maximal union of children of U that is a module of G. If W overlaps a module X of G, then X is also a union of children of U, since T is an M tree. $W \cup X$ is also a module since the modules of G are a decomposable family, contradicting the maximality of W. Thus, W overlaps no module of G and must be a member of $MD(G)$. If U is prime in T, then $W = U$, W is prime in $MD(G)$, and its children in $MD(G)$ are each a subset of a child of U in T. If U is degenerate in T, W is a module in $G|U$, so by Theorem 2.5, W is degenerate in $MD(G)$ and its children in $MD(G)$ are the children of U that it contains. In either case, there is no proper subset of W that is a union of children of U and a member of $MD(G)$. □

Remark 3.6. Given an initialized set of n buckets and a set of sorted adjacency lists whose sum of cardinalities is k, one may produce a grouping of the lists into maximal groups of identical lists in $O(k)$ time. The algorithm is a variant of radix sort. Partition

the lists into buckets according to the elements in their first position, keep a list of which buckets are currently in use, collect them from the buckets in use, reinitialize the buckets, remove the first element from each adjacency list, and then recurse on each of the sets of lists from a common bucket.

The following algorithm gives the modular decomposition, given an M2 tree.

Function **Decomp** (G, T)

Input: An undirected graph G and an M2 tree T corresponding to G

Output: The modular decomposition of G

Execute **Modules** (G, T), to find which members of T are modules in G. These are members of $MD(G)$ since their status as a member of an M tree means that they can overlap no other module of G. Any other member of $MD(G)$ is a nontrivial union of children of a degenerate node of T. For each degenerate node, U, purge the nodes lying in the interval of node-numbers occupied by U from the SN lists for children of U. Partition the children of U that are modules of G according to their remaining SN lists, using the algorithm of Remark 3.6. The union of each set of children that is not distinguished by the sort is a member of $MD(G)$.

Insert a node in T for each member of $MD(G)$ that is not a member of T. Purge T of those members that are not modules. Return the result.

By Lemma 3.5, **Decomp** (G, T) computes the members of $MD(G)$. If a member of $MD(G)$ is a member of the M2 tree, its classification as prime or degenerate is the same as it is in the M2 tree, by Theorem 2.5. Otherwise, it is a union of more than one child of a degenerate node in the M2 tree, and is thus degenerate, by Theorem 2.5. For the time bound, we have already established that the strong adjacency lists of children may be traversed a constant number of times in **Modules** (G, T) without violating the $O(n+m)$ time bound. The purge and the partition steps adds two additional traversals of the lists, so **Decomp** takes $O(n+m)$ time.

3.3. Constructing an M1 tree

In this section, we give a definition of the P_4 *tree*, which may be constructed in linear time and which satisfies the definition of an M1 tree [31].

If a graph G has no induced P_4 (i.e., G is a cograph), the P_4-tree is equal to the cotree for G. If a P_4 is found in G, we use the P_4 to divide the graph into a number of pieces as follows.

Let the vertices a, b, c, d form a P_4 in G. We partition the vertices of G into the following sets.

(1) $A = N(b) - N(c) - N(d)$,
(2) $B = (N(a) \cap N(c)) - N(d)$,
(3) $C = (N(b) \cap N(d)) - N(a)$,
(4) $D = N(c) - N(b) - N(a)$,

(5) $U = (N(a) \cap N(b) \cap N(c) \cap N(d)) \cup (\overline{N}(a) \cap \overline{N}(b) \cap \overline{N}(c) \cap \overline{N}(d))$,
(6) $E = V - A - B - C - D - U$.

The set A gets its name from the fact that, like a, any vertex in A will form a P_4 together with b, c, and d. The sets B, C, and D have a similar relationship to b, c, and d respectively. The set U consists of vertices that do not distinguish a, b, c, and d, while E contains all other vertices.

The following procedure defines a P_4-tree for a graph G. In general, a single graph G can have many nonisomorphic P_4-trees, and a single P_4-tree can represent many nonisomorphic graphs.

Function **Createtree** (G)

Input: An undirected graph G

Output: A P_4 tree corresponding to G

if $G$is acograph then T = `cotree` (G)
else
  let $(a,b),(b,c),(c,d)$ be a P_4 in G;
  $T_A :=$ **Createtree**$(G|A)$;
  $T_B :=$ **Createtree**$(G|B)$;
  $T_C :=$ **Createtree**$(G|C)$;
  $T_D :=$ **Createtree**$(G|D)$;
  $T_E :=$ **Createtree**$(G|E)$;
  select $x \in \{a,b,c,d\}$
  $T :=$ **Createtree**$(G|(U \cup \{x\}))$;
  Let p be the leaf of T that corresponds to $\{x\}$
  Label p a P_4 node
  for $i = A,B,C,D,E$ do
    make T_i a child of p;
return T;

Consider all the nodes not labeled prime to be degenerate (these are nodes created by the Cotree algorithm). By Theorem 4 of [31], every module of G is either a P_4 node or a union of children of a cotree node in the P_4 tree. Thus, when P_4 nodes are considered prime and cotree nodes are considered degenerate, the tree is an M tree. It is also shown in [31] that there exists a system of representatives from the children of any degenerate node that induce either a complete or an edgeless subgraph. We will call such nodes 1 and 0 nodes, respectively. This ensures that the P_4 tree is an M1 tree. However, it is not necessarily true that every system of representatives induces such a subgraph, so the tree is not necessarily an M2 tree.

3.4. Finding an M2 tree, given an M1 tree

In this section, we give an algorithm that computes an M2 tree from an M1 tree in linear time. Since we have given an algorithm above for computing the modular

decomposition from an M2 tree, and the implementation of **Createtree** given in [31] gives an M1 tree in linear time, this step completes the decomposition algorithm. The algorithm proceeds by adding nodes while preserving the invariant that the tree continues to be an M1 tree. Eventually and M2 tree is obtained. Only degenerate nodes of an M1 tree may violate the conditions needed for the tree to be an M2 tree, so once a prime node is produced, it is largely ignored thereafter.

Let G be a directed graph. The *component graph* for G has one node for each strongly connected component [5]. If X and Y are two strongly connected components of G, then (X, Y) is an edge in the component graph if there is an edge of G that goes from X to Y. The strongly connected components and the component graph may be computed in $O(n+m)$ time [6].

Definition 3.7. Let T be an M1 tree on graph G, and let U be a degenerate node of T. The graph $G(G,U,T) = (children_T(U), E)$, where $E = \{(X,Y) : X, Y \in children_T(U)$ and X is split by a vertex of G that is contained in $Y\}$. The graph $G_c(G,U,T)$ is the component graph of $G(G,U,T)$.

Remark 3.8. Suppose X is a union of a set $\mathscr{X}$ of children of a degenerate node U of an M1 tree T. We will say that X has an *outgoing forcing edge* if there exist $Y, Z \in children_T(U)$ such that $Y \in \mathscr{X}$, $Z \notin \mathscr{X}$ and (Y,Z) is an edge of $G(G,U,T)$. In this case, X is not a module of G if it has an outgoing forcing edge, since $Y \subseteq X$ is split by a graph vertex contained in $Z \subseteq V(G) - X$.

We will refer to $G(G,U,T)$ as a *forcing graph*, since the existence of an edge (Y,Z) in it indicates that Z is forced to be contained in any module that contains Y. We now give an algorithm for computing a refined M tree and a set of forcing graphs for its degenerate nodes.

Procedure **ForcingGraphs** (G,T)

Input: A graph G and a corresponding M1 tree T

Result: A refinement T_r of T that is also an M1 tree and a labeling of each degenerate node U of T_r with $G(G,U,T_r)$.

Run **Splitters** (G,T) to find a refinement T_r of T that is labeled with *Disagree* lists

```
For each degenerate node U of T_r in postorder
  For each child Y of U
    For each member z of Disagree(Y)
      Find the sibling Z of Y that contains z
      Install (Y,Z) as a directed edge in G(G,U,T_r) if it is not
        already an edge in G(G,U,T_r)
```

The correctness follows from the definition of $G(G,U,T)$ and the correctness of *Splitters*. For the time bound, note that finding the sibling of Y that contains z is a

FIND operation in a sequence of *UNIONs* and *FINDs* that occur during the postorder traversal. There are $O(m)$ such *FINDs* because the *Disagree* lists collectively have $O(m)$ size. There are $O(n)$ *UNION* operations, since the graph has n nodes and we work only with siblings in the tree during each postorder traversal. Since T_r is predetermined in advance of the *UNIONs* and *FINDs*, the Gabow and Tarjan *UNION − FIND* operations may be applied [15] for an $O(n+m)$ bound.

For any M1 tree T, every module of G is a union of children of some node U in G. From Remark 3.8, it follows that no module of G may overlap any strong component of $G(G,U,T)$, and no proper subset of a strong component may be a union of members of the strong component. Thus, if for each strong component of $G(G,U,T)$, a new node is added to T, the resulting tree is still an M tree. By Remark 3.1, it is an M1 tree. The component graph $G_c(G,U,T)$ now defines a forcing graph on the new children of U in the refined tree, and this graph is again a forcing graph in the sense that no union of the new children of U is a module if it has an outgoing edge in this graph, by Remark 3.8. However, $G_c(G,U,T)$ has the property of being acyclic, which we will use below. The following algorithm takes advantage of this observation to create an M1 tree that has an acyclic forcing graph on the children of each degenerate node.

Procedure **SCC** (G,T)

Input: An M1 tree T

Result: A refinement of T that is an M1 tree, and where each degenerate node is labeled with a forcing graph on its children that is acyclic.

Call **ForcingGraphs** (G,T) to get a refinement T_r of T that is labeled with forcing graphs;

For each degenerate node U in T_r do
  Find the strongly connected components of $G(G,U,T_r)$
  Find the component graph $G_c(G,U,T_r)$
  For each strongly connected component $\mathscr{C}$ of $G(G,U,T_r)$
    If $\mathscr{C}$ contains more than one child of U
      Create a new child r of U in T_r
      For each child X of U in $\mathscr{C}$
        Move X and its subtree from a child of U to a child of r
  {$G_c(G,U,T_r)$, is a graph on U's new children, which are called **SCC nodes**}
  Label U with $G_c(G,U,T_r)$

The algorithm for computing strongly connected components runs in linear time [5]. Thus, the time bound for the operation is linear in the size of the forcing graphs, which is $O(n+m)$, since they were created by **ForcingGraphs**.

Lemma 3.9. *Let T be an M1 tree on graph G. A module X of G is either a node of T or the union of a set $\mathscr{X}$ of children of a degenerate node U such that $\mathscr{X}$ is a module in $G(G,U,T)$ that has no outgoing edges in $G(G,U,T)$.*

Proof. That $\mathscr{X}$ has no outgoing edges in $G(G,U,T)$ follows from Remark 3.8. Otherwise, suppose $\mathscr{X}$ is not a module in $G(G,U,T)$. There exists $Z \in children_T(U) - \mathscr{X}$ such that in $G(G,U,T)$, Z has an edge to some, but not all, members of $\mathscr{X}$. Let $X_1, X_2 \in \mathscr{X}$ such that (Z,X_1) is an edge of $G(G,U,T)$ and (Z,X_2) is not. Since (X_1,Z) and (X_2,Z) are not edges neither X_1 nor X_2 disagrees in G on any member of Z. However, there exist $z_1, z_2 \in Z$ that disagree in G on nodes of X_1 but not on nodes of X_2. It follows that $X_1 \cup X_2$ disagrees either on z_1 or on z_2, so X cannot be a module, a contradiction. □

Note that a module of a directed graph G that has no outgoing edges in G must be a union of strong components of G that form a module in the component graph for G. From this observation, we get the following corollary to Lemma 3.9.

Corollary 3.10. *Let T be an M1 tree on graph G. Any module of G is either a node of T or the set of leaf descendants of a module of $G_c(G,U,T)$ for some degenerate node U of G.*

By Corollary 3.10, we may further refine the tree produced by **SCC** by adding nodes that discard from the tree's decomposable family only sets that either have outgoing edges or fail to be a module in $G(G,U,T)$. The following procedure makes use of this idea to produce a further refinement of the tree produced by **SCC**. Note the comments, which define the terms 'driver' and 'passenger'.

Procedure **PQ** (G,T)

Input: An M1 tree T on graph G

Result: A refinement of T that is an M1 tree.

Call **SCC** (G,T) to get a refinement T_r of T in which every degenerate node of T_r is a degenerate node of T. Give a labeling of each degenerate $U \in T_r$ with $G_c(G,U,T)$.

For each degenerate node U of T_r
  Let $\mathscr{F}_C$ be the current children of U
  For each $X \in \mathscr{F}$ in topological order, from sink to source do
    Mark X as a representative
    Create a new tree node p, label it prime, mark it as a 'P' node, and make it a child of U
    Move X and its subtree to be a child of p
    $\{X$ is the driver of $p\}$
    Create a new tree node q, label it degenerate, mark it as a "Q" node, and make it a child of p
    For each edge (X,Y) of $G_c(G,U,T)$ do
      If Y marked as a "representative"
        Unmark Y as a representative
        Let Z be the current child of U that contains Y

```
        Move Z and its subtree to be a child of p
        {Z is a 'passenger' of p}
    If q has no children then
      Remove q from the tree
      {Consider X to be its own 'driver'}
    Contract p or q in T if it has only one child
```

Finding the topological sort of a directed acyclic graph takes time linear in the size of the graph [5]. The invariant is maintained that the representative of a child W of U is either W or a child of W, so identifying W given its representative takes O(1) time. Thus, all operations on U take time that is linear in the size of $G_c(G,U,T)$. The sum of sizes of the $G_c(G,U,T)$ graphs over all degenerate nodes U is $O(n+m)$, since they are computed with a linear-time algorithm (**SCC**). A linear-time bound for **PQ** thus follows.

Lemma 3.11. *The tree produced by* **PQ** *is an M1 tree whenever its input tree is an M1 tree.*

Proof. By Remark 3.1, it suffices to show that it is an M tree. Adopt as an inductive hypothesis that the tree is an M tree at the beginning of an iteration of the outer loop. The hypothesis is true before the first iteration, since the tree produced by **SCC** is an M tree.

Let P and Q be the final sets represented by p and q after an iteration of the outer loop. To show that the inductive hypothesis holds at the end of the loop, it suffices to show that no module overlaps P or Q, since Q is degenerate, P has only two children, and P and Q are the only new tree nodes added to the M tree by the iteration.

Let $\mathscr{C}$ be the set of children of U at the beginning of the iteration of the loop. Q is the union of members of $\mathscr{C}$ whose representatives are pointed to by X in $G_c(G,U,T)$. Let M be a module of G. If it contains U then it does not overlap P or Q. If M is contained in a member of $\mathscr{C}$ then it does not overlap P or Q. By the inductive hypothesis, in any remaining case M is a union of a subfamily $\mathscr{C}'$ of members of $\mathscr{C}$. By Corollary 3.10 it is the union of a module $\mathscr{M}$ of $G_c(G,U,T)$.

Suppose M does not contain X. It contains the representatives of the members of $\mathscr{C}'$. If some member C of $\mathscr{C}'$ is contained in Q, then the representative of C is pointed to by X in $G_c(G,U,T)$. Since $\mathscr{M}$ is a module of $G_c(G,U,T)$, all nodes of $G_c(G,U,T)$ that are contained in M are pointed to by X. In particular, all representatives of members of $\mathscr{C}'$ are pointed to by X. Thus, every member of $\mathscr{C}'$ is contained in Q. We conclude that if M does not contain X it is contained in Q, hence in P, and it can overlap neither Q nor P.

Suppose M contains X. X has an outgoing edge to the representative of each member of $\mathscr{C}$ that is contained in Q. Thus, M must contain Q, and since it contains X, it contains P. Again, it overlaps neither P nor Q.

In all cases, the inductive hypothesis applies at the end of the iteration. □

We now give the final algorithm for turning an M1 tree into an M2 tree:

Procedure **M1M2** (G, T)

Input: An undirected graph G and an M1 tree T on G

Result: A refinement of T that is an M2 tree

Run **PQ**(G, T) to get a refinement T_1 of T
Run **SCC**(G, T_1) to get a refinement T_2 of T
Return T_2

A linear time bound follows immediately from the bounds for **PQ** and **SCC**. We now show that the tree it computes is an M2 tree

Lemma 3.12. *Suppose U is a degenerate node in M1 tree T. Suppose $\mathscr{A}$ and $\mathscr{B}$ are two sets of children of U such that no member of $\mathscr{A} \times \mathscr{B}$ or $\mathscr{B} \times \mathscr{A}$ is an edge of $G(G, U, T)$. Then every member of $\cup\mathscr{A} \times \cup\mathscr{B}$ is an edge of G or else every member of $\cup\mathscr{A} \times \cup\mathscr{B}$ is a nonedge of G.*

Proof. There is a system of representatives from children of U that forms a complete or empty graph. Without loss of generality, suppose it is a complete graph. For any $A \in \mathscr{A}$ and $B \in \mathscr{B}$, A agrees in G on each member of B and vice versa because neither (A, B) nor (B, A) is an edge of $G(G, U, T)$. Since there exists representatives $a \in A$ and $b \in B$ such that (a, b) is an edge of G, every member of $A \times B$ is an edge of G. Since A and B are arbitrary in $\mathscr{A}$ and $\mathscr{B}$, respectively, every member of $\cup\mathscr{A} \times \cup\mathscr{B}$ is an edge of G. □

Lemma 3.13. *If* **M1M2** *inputs an M1 tree T on an undirected graph G, it returns an M2 tree.*

Proof. The algorithm returns an M1 tree if its parameter is an M1 tree, since we have shown that this holds for each of the procedures it calls. It remains to show that children of degenerate nodes of T_2 satisfy the requirements of an M2 tree.

When **SCC** is called directly from **M1M2**, **SCC** calls **ForcingGraphs**, which produces an M1 tree that we will denote T_r. We prove the lemma by showing that for any degenerate node U in T_r, each connected component of $G(G, U, T_r)$ is strongly connected. Since the strongly connected components become the children of U in the tree T_2 returned by **M1M2**, there are no edges of $G(G, U, T_r)$ connecting children of U in T_2. By Lemma 3.12, the children of U satisfy the requirements of an M2 tree.

To show this result, we show that whenever (A, B) is an edge of $G(G, U, T_r)$ then (B, A) is also an edge of $G(G, U, T_r)$. **PQ** calls **SCC**, which calls **ForcingGraphs**. Let T_f be the refinement of T returned by this call to **ForcingGraphs**. $G_c(G, U, T_f)$ is the forcing graph that **PQ** traverses in reverse topological order when it processes a degenerate node U. Let (X, Y) be an edge of $G_c(G, U, T_f)$, and let Y' be the sibling of X that contained Y at the time when X was reached in the topological traversal of

$G_c(G, U, T_f)$. If there is also an edge of $G(G, U, T_f)$ from X to the representative of Y', then the tree node P created at that point becomes the least common ancestor of (X, Y). Since P is prime and prime nodes are not further refined, the least common ancestor of (X, Y) remains prime in T_2. If there is no edge from X to the representative of Y', then Y' disagrees on X. We conclude that if the least common ancestor in T_1 of (X, Y) is degenerate, then the child A of that ancestor that contains X is split by the child B that contains Y, and vice versa.

Conversely, if two children of a degenerate node in T_1 are not connected by any such edge (X, Y), then each of them agrees on the other.

Summarizing, if one child A of a degenerate node of T_1 disagrees on a vertex of G that is contained in one of its siblings, B, then B also disagrees on a vertex of G that is contained in A. When a forcing graph is computed on any refinement of T_1, its connected components are strongly connected. □

4. The transitive orientation algorithm

An undirected graph may be considered to be a special case of a directed graph, where each undirected edge (a, b) is represented by two directed edges (a, b) and (b, a). Let $G = (V, E)$ be this representation. Finding a transitive orientation consists of selecting $F \subset E$ such that (V, F) is transitive, and such that $(a, b) \in F$ iff $(b, a) \notin F$.

Define the binary relation Γ on E as follows [18]: $(a, b)\Gamma(a', b')$ iff either $a = a'$ and $(b, b') \notin E$ or $b = b'$ and $(a, a') \notin E$. Our algorithm is based on the well-known, obvious observation that if $(a, b)\Gamma(c, b)$, then $(a, b) \in F$ if and only if $(c, b) \in F$. Let Γ^* denote the reflexive transitive closure of Γ. F is an equivalence class in Γ^* [18].

This proof of the following is a straightforward application of the Γ observation; the reader is referred to [18].

Theorem 4.1 (Golumbic [18]). *A prime comparability graph has only two transitive orientations, where one is obtained from the other by reversing the directions of all the edges.*

Let T be the modular decomposition of a comparability graph G that is not prime. A transitive orientation may be obtained by computing for each $U \in T$ a transitive orientation of $(G|U)/children_T(U)$. If A and B are children of U, then the members of $(A \times B) \cap E$ are in the transitive orientation of G if and only if (A, B) is in the transitive orientation of $(G|U)/children_T(U)$. If the modular decomposition is available, then transitively orienting G reduces to the problem of transitively orienting these quotients, all of which are either complete graphs, empty graphs, or prime comparability graphs. The first two cases are trivial. (A transitive orientation of an empty graph is empty. Any total order on the nodes of a complete graph is a valid transitive orientation of it.) Thus, if the modular decomposition of a comparability graph is provided, the problem reduces to finding a transitive orientation of a prime quotient. The proof is again an

application of the Γ observation; the reader is referred to [18]. Given the linear time bound for modular decomposition, the following is immediate:

Theorem 4.2. *Transitive orientation of arbitrary comparability graphs is no harder than transitive orientation of prime comparability graphs.*

Henceforth, we will assume without loss of generality that the graph G that we are to orient is prime.

We now describe an operation called *vertex partitioning*, which is central to our algorithm. The algorithm inputs a partition $\mathscr{P}$ of vertices of G, and repeatedly selects a *pivot vertex* x and refines $\mathscr{P}$ by splitting partition classes that do not contain x into subclasses that are entirely adjacent or entirely nonadjacent to x. Edges from x to any of these classes then become irrelevant to future splitting operations, so they are then removed.

VertexPartition $(G, \mathscr{P})$

Inputs: A prime graph $G=(V,E)$ and a partition $\mathscr{P}$ of vertices of G such that $\mathscr{P} > 1$ and every member of $\mathscr{P}$ is nonempty.

While not every member of $\mathscr{P}$ is a singleton class do
  Select a *pivot vertex* x
  Let X be the member of $\mathscr{P}$ that contains x
  Select a subfamily $\mathscr{Q}$ of classes of $\mathscr{P} - X$
  For each $Y \in \mathscr{Q}$
    Let $Y_a := Y \cap N(x)$
    Let $Y_n := Y - Y \cap N(x)$
    Let $\mathscr{P} = (\mathscr{P} - \{Y\}) \cup \{Y_a, Y_n\}$
    If $Y_a = \emptyset$ then $\mathscr{P} := \mathscr{P} - \{Y_a\}$
    If $Y_n = \emptyset$ then $\mathscr{P} := \mathscr{P} - \{Y_n\}$
    Let $E := E - (\{x\} \times Y_a)$
    { elements of $Y \times \{x\}$ may remain in E}

Let a *pivot sequence* denote a sequence of choices of x and $\mathscr{Q}$. The removal of $\{x\} \times Y_a$ from E is inserted in the code to make it easier to obtain the time bound. It has no effects on how future pivots refine $\mathscr{P}$. To see this, note the removal of $\{x\} \times Y_a$ does not affect whether a future pivot on a node other than x splits a subset of $V(G)$. In any future refinement of $\mathscr{P}$, each partition class is either a subset of Y_a or of $V(G) - Y_a$. The removal of $\{x\} \times Y_a$ does not affect whether a future pivot on x splits an arbitrary subset of $V(G) - Y_a$. Since x cannot split members of Y_a either before or after the removal, it does not affect whether a future pivot on x splits an arbitrary subset of Y_a.

If $\mathscr{P}$ contains a nonsingleton set U, then U is not a module, since G is prime. U disagrees on some $x \in V(G) - U$, so x can be used as a pivot to split U, thus

further refining $\mathscr{P}$. We conclude that there always exists a pivot sequence that causes the algorithm to halt.

Definition 4.3. The *degree* of a pivot on vertex x is $\sum_{Y\in\mathscr{Q}} |N(x)\cap Y|$.

Lemma 4.4. *The degree sum of any pivot sequence is* $O(m)$

Proof. The degree of a pivot is equal to the number of edges deleted from E during the pivot. □

To maintain the partition classes, we keep each partition class as doubly linked lists of vertices, where each vertex in the list has a pointer to the list's header. When class Y is split by a pivot on vertex x into two classes Y_a and Y_n, remove the elements from Y that are adjacent to x, insert them in a new list corresponding to Y_a, and relabel these vertices of Y_a with pointers to the header of Y_a's list. The remainder of Y's list is left alone; it now represents Y_n. Once the members of Y_a have been identified, the cost of modifying the data structures to reflect Y's split is $O(|Y_a|)$. Maintaining these data structures thus takes $O(m)$ time over any pivot sequence by Lemma 4.4, so we may ignore it from here on. The data structures give us the following:

Remark 4.5. At any time during **VertexPartition**, we may find the current class that contains a given vertex in $O(1)$ time.

A pivot operation looks up an element y in x's adjacency list, looks up the partition class Y that contains y, determines whether $Y\in\mathscr{Q}$, and, if so, moves y to the list Y_a that corresponds to Y. It then repeats this lookup on other elements in x's adjacency list. The pivot operation is correct as long as the lookup is performed on every member of x's adjacency list that lies in a member of $\mathscr{Q}$. The simplest pivot, called a *universal pivot*, sets $\mathscr{Q}=\mathscr{P}-\{X\}$ and performs the lookup on every y in x's adjacency list. Clearly, a universal pivot takes $O(|N(x)|)$ time.

We now show that linear-time transitive orientation of a prime comparability graph G reduces to finding a strategy for selecting x and $\mathscr{Q}$ that guarantees that **VertexPartition** halts in $O(n+m)$ time. We will call such a strategy a *pivot selection strategy*.

Procedure **TransitiveOrientation** (G)

Input: a prime comparability graph G

Output: A list of vertices that gives a topological sort of a transitive orientation of G

Select an arbitrary $v\in V(G)$
$\mathscr{L}=$ **OrderedVertexPartition**(G,v)
Let u be the vertex in the rightmost class of $\mathscr{L}$
$\mathscr{L}=$ **OrderedVertexPartition**(G,u)

Procedure **OrderedVertexPartition**(G, v)

Input: A prime graph $G = (V, E)$ and a vertex v

Output: If v is an arbitrarily selected vertex, the output is an ordering of $V(G)$ such that the last node in the ordering is a source or sink in any transitive orientation of G. If v is a source or sink, then the output ordering is a linear extension of the transitive orientation of G.

Let $\mathscr{L}$ be an *ordered* list of partition classes
Initialize $\mathscr{L} = (\{v\}, V - \{v\})$
While not every member of $\mathscr{L}$ is a singleton class do
  Select a *pivot vertex* x
  Let X be the partition class containing x
  Select a subfamily $\mathscr{Q}$ of classes of $\mathscr{L} - X$
  For each $Y \in \mathscr{Q}$
    Let $Y_a := Y \cap N(x)$
    Let $Y := Y - Y \cap N(x)$
    If Y appears after X in $\mathscr{L}$ then
      Insert Y_a immediately following Y in $\mathscr{L}$
    Else
      Insert Y_a immediately preceding Y in $\mathscr{L}$
    Remove Y from $\mathscr{L}$ if Y is empty
    Remove Y_a from $\mathscr{L}$ if Y_a is empty
    Let $E := E - (\{x\} \times Y)$

Lemma 4.6. *The time bound for* **OrderedVertexPartition** *is no greater than that for* **VertexPartition**.

Proof. The initial partition assumed by **OrderedVertexPartition** is a special case of that assumed by **VertexPartition**. Thus, the only significant difference between the two procedures is that **OrderedVertexPartition** requires us to determine which of Y_n or Y_a should go before the other in $\mathscr{L}$. Maintain the invariant that a subinterval of $(1, 2, \ldots, n)$ is associated with each partition class by labeling it with the first and last elements in the subinterval. Initially, $(2, 3, \ldots, n)$ is associated with $V - \{v\}$, and (1) is associated with $\{v\}$. By looking up the interval corresponding to x's current class, it can be determined in O(1) time whether x's class occurs before or after Y in $\mathscr{L}$, and thus whether Y_a must be inserted before or after Y in $\mathscr{L}$. To restore the correct labeling on members of $\mathscr{L}$, associate either the first or last $|Y_a|$ elements of Y's interval with Y_a, depending on whether Y_a goes before Y in $\mathscr{L}$. Associate the remainder of the interval with the remaining portion of Y. All of this takes O($|Y_a|$) time, so it adds time proportional to the degree sum of the pivot sequence, which is O(m) by Lemma 4.4. □

Lemma 4.7. TransitiveOrientation *is correct.*

Proof. We must show that if the parameter v to **OrderedVertexPartition** is arbitrary, then the final partition class in the returned list gives a source or sink in a transitive orientation of G, and if v is a source or sink, the returned list gives a topological sort of a transitive orientation.

Suppose v is arbitrary. Let Y denote the rightmost class at some point during the partitioning operation, and let y be an arbitrary element of Y. Let $E(Y,y) = \{(z,y) : (z,y) \in E(G) \text{ and } z \notin Y\}$. Adopt as an inductive hypothesis that all edges in $E(Y,y)$ are in a common equivalence class induced by Γ^*. Suppose a pivot on some vertex z splits Y into two nonempty sets. Then, z is adjacent to the new rightmost class $Y' = Y \cap N(z)$. Suppose that y is also a member of Y'. For any $(w,y) \in E(Y',y) - E(Y,y)$, $w \in Y - N(z)$. Thus, $(z,y)\Gamma(w,y)$, proving that the inductive hypothesis holds for $E(Y',y)$. The truth of the inductive hypothesis when the procedure halts shows that the sole member of the rightmost class must be a source or a sink in any transitive orientation of G.

Suppose v is a source or a sink. That is, suppose that all edges (v,x) of G are in a single equivalence class of Γ^* and all edges (x,v) are in a single equivalence class of Γ^*. This time, adopt as an inductive hypothesis that all edges of G that go from an earlier to a later partition class in $\mathscr{P}$ are in a single equivalence class. This is true of the initial partition $(\{v\}, V - \{v\})$. Suppose a class X is split by a pivot vertex z. The edges of G that are subsets of $X \cap N(z) \times X - N(z)$ are each Γ related to an edge in $\{z\} \times X \cap N(z)$. The ordering of $X \cap \{z\}$ and $X - N(z)$ thus ensures that the inductive hypothesis still applies after the split. The truth of the inductive hypothesis when the procedure halts demonstrates that $\mathscr{P}$ is a linear extension of a transitive orientation of G. □

Summarizing Theorem 4.2 and Lemmas 4.6 and 4.7, we have the following:

Theorem 4.8. *Transitive orientation of a comparability graph is no harder than* **VertexPartition**.

Hereafter, we address the problem of obtaining a linear-time implementation of **VertexPartition**. Before giving the details, we consider some of the consequences of the claim that such an implementation exists.

We must first consider that when the transitive orientation algorithm is applied to a prime graph that is not a comparability graph, it produces an orientation of its edges that is not transitive. Like the $O(n^2)$ algorithm of [30], the algorithm fails to recognize whether G is, in fact, a comparability graph. In contrast, previous $O(n^{2.38})$ [31] and $O(\delta m)$ algorithms [18] perform both the orientation and the recognition, and these bounds remain the best so far for the recognition problem. The usefulness of such an algorithm comes from the fact that it makes it possible to solve a number combinatorial problems on comparability graphs where recognition is not necessary. The algorithm either provides a certificate that its answer to the problem is correct, or it demonstrates that the input graph is not a comparability graph. It fails to recognize comparability

graphs only because it may provide a solution and certificate even when the input graph is not a comparability graph.

Finding a maximum clique and a minimum vertex coloring in a comparability graph. Suppose an orientation F of the edges of G is given by our algorithm. It is an easy exercise to color each vertex according to the length of the longest directed path that begins at it in (V,F), using a postorder operation during a depth-first search of (V,F) [18]. This gives a coloring of the nodes of the input graph such that no two adjacent nodes have the same color. If G is a comparability graph, then the longest path in the graph (V,F) is a clique of G because of the transitivity of F. This gives a clique and a coloring of the graph, where the clique has the same number of nodes as the number of colors in the coloring. Since all of the nodes of any clique must have different colors, this gives a certificate that the clique is maximum clique and that the vertex coloring is a minimum one. If the input graph is not a comparability graph, the algorithm still produces an orientation F of G that is not transitive. If the longest path is a clique, then the algorithm has provided a certificate that it is a maximum clique and that the coloring is a minimum one, without ever recognizing that the input graph is not a comparability graph. If, however, the longest path does not correspond to a clique of G, then some transitive edge is missing and the algorithm has recognized that the input graph could not have been a comparability graph.

Transitive orientation of the complement of a co-comparability graph. The following lemma shows that one may compute a representation of the transitive orientation of a complement of a co-comparability graph in $O(n+m)$ time given a linear-time pivot selection strategy. This is somewhat surprising in view of the fact that direct examination of the complement of any graph requires $\Omega(n^2)$ time. The key is that it is possible to orient the complement of G by producing a compact representation of it (i.e. a linear extension of it) and by examining only the edges of G. This result is a key element in the remaining results described in this section.

Theorem 4.9. *Let G be a co-comparability graph. Any pivot selection strategy that causes* **VertexPartition** *to halt in $O(n+m)$ time on an arbitrary prime graph can be used to produce a linear extension of a transitive orientation of $\overline{G}$ in $O(n+m)$ time.*

Proof. Let T be the modular decomposition tree. T is also the decomposition tree for the complement $\overline{G}$ of G. Thus, the decomposition tree for $\overline{G}$ may be given in time proportional to the size of G. Proceeding as before, we order children of degenerate nodes of T arbitrarily, and order children of each prime node U of T according to a linear extension of a transitive orientation of $(\overline{G}|U)/children_T(U)$. The leaf ordering of the tree then gives a linear extension of a transitive orientation of $\overline{G}$.

To stay within the $O(n+m)$ bound, we observe that $(\overline{G}|U)/children_T(U)$ is the complement of $(G|U)/children_T(U)$, and we must compute a linear extension of its transitive orientation in time proportional to the size of $(G|U)/children_T(U)$.

The problem thus reduces to finding a linear extension of the transitive orientation of the complement of a prime co-comparability graph in linear time. We run

Transitive-Orientation on the prime graph using the linear-time pivot selection strategy, but change the IF-ELSE statement in the inner loop of **OrderedVertexPartition** to read:

If Y appears after X in $\mathscr{P}$ then
    Insert Y_a immediately *preceding* Y in $\mathscr{P}$
Else
    Insert Y_a immediately *following* Y in $\mathscr{P}$

In the **OrderedVertexPartition** procedure, the treatment of edges and nonedges is everywhere symmetric, except in this IF-ELSE statement. The statement given here reverses their roles, so the correctness on the complement of the graph follows from this symmetry. □

Maximum independent set and minimum clique cover in co-comparability graphs. A graph is a co-comparability graph if its complement is a comparability graph. Examples of co-comparability graphs are interval graphs and permutation graphs.

Given a linear extension of a transitive orientation of the complement of G, which we have shown is possible given a linear-time pivot selection strategy, we label each node v with the length of the longest path in this oriented complement that begins at v. Let F be a transitive orientation of $\overline{G}$. We proceed by finding a minimum vertex coloring of the complement of G. The color of a vertex v is again the length of the longest path beginning at v in F. Working inductively, this is again one plus the maximum color number of successors of v in F. However, since $|F|$ can be much larger than $O(n+m)$, there is no time to examine all the successors of v in F. Process vertices in the reverse of the order given by the linear extension of $\overline{G}$, that is, starting at the vertex that corresponds to a sink of the transitive orientation. Adopt the inductive assumption that when vertex v is reached, each successor x of v in the linear extension resides in some bucket number i, where i gives the length of the longest path that begins at x. Mark all vertices that are adjacent to v in G. Then examine vertices in descending order in the buckets until an unmarked vertex is found; this is a successor of v in F that resides in the highest bucket of any successor of v in F. The number of the bucket in which v must be inserted is one plus the number of the bucket of that vertex. Insert v in its bucket and unmark the neighbors of v. The time marking, unmarking, examining marked nodes during the descent through the buckets is charged to edges incident to v in G. Only one unmarked node is found; the cost of examining it is charged to v.

Recognition of permutation graphs and two-dimensional partial orders. We now show that given an $O(n+m)$ pivot selection strategy for **VertexPartition**, we may recognize permutation graphs in $O(n+m)$ time. We use the well-known characterization that G is a permutation graph iff G and its complement $\overline{G}$ are both transitively orientable [27]. This is the first $o(n^3)$ algorithm that makes use of this; the approach is only possible because of Theorem 4.9.

Compute a linear extension F of the transitive orientation of G, and linear extension $\overline{F}$ on the transitive orientation of $\overline{G}$. Let R be the following total order: if (x, y) is an edge of G, then xRy if $(x, y) \in F$, and if (x, y) is not an edge of G then xRy if $(x, y) \in \overline{F}$. Next, let R' be the following total order: if (x, y) is an edge of G then xRy is defined as before, and if (x, y) is not an edge of G then xRy if $(y, x) \notin \overline{F}$. R and R' are a realizer for the permutation graph [18].

We have shown that we may obtain linear extensions of F and $\overline{F}$ in linear time if we have linear-time pivot selection strategy. Given F and $\overline{F}$, R and R' may then be computed with the following $O(n + m)$ procedure. The rank of a node v in R is one $1+N1+N2$, where $N1$ is the number of neighbors of v that precede v in F, and $N2$ is the number of nonneighbors of v that precede v in F'. $N1$ may be found by counting the number members of v's adjacency list that precede v in the linear extension of F. $N2$ may be found by counting the number of members of v's adjacency list that precede v in the linear extension of F', and then subtracting this result from the total number of vertices of G that precede v in the linear extension of F'. Thus, finding the rank of v in R takes $O(|N(v)|)$ time. Finding the rank of v in R' can be accomplished in $O(|N(v)|)$ time by repeating the operation on the reverse of the linear extension of F'. Repeating this for each vertex v gives the $O(n+m)$ algorithm for finding the rank of every vertex in R and in R'; bucket sorting vertices according to their ranks gives R and R'.

R and R' always constitute a realizer for a permutation graph G'. $G' = G$ if and only if G is a permutation graph. This can be easily checked in $O(n + m)$ time by computing adjacency lists one node x at a time in the permutation graph corresponding to R and R', and comparing the adjacencies immediately to those of x of G, and halting immediately if a discrepancy is detected.

Recognition of interval and cointerval graphs: A cointerval graph is a comparability graph, since one interval coming before another is a transitive relation. Produce a linear extension of its transitive orientation, and order left endpoints according to how many neighbors of a vertex come before the vertex in the linear extension. Similarly, order the right endpoints based on how many neighbors come after. The interleaving of these two lists to create a set of intervals that realizes the graph is trivial. If the input graph is not known to be a cointerval graph, then performing this operation and then checking whether the resulting intervals realize the graph gives a linear-time recognition algorithm.

Because of Theorem 4.9, essentially the same algorithm gives a novel approach to interval graph recognition, although the time bound is not new [1]. Produce a linear extension of the complement of the graph, and order left endpoints by counting how many nonneighbors of a vertex come before the vertex in the linear extension. This is obtained by subtracting the number of neighbors that come before it from the total number of vertices that come before it. Similarly, the right endpoints can be ordered based on how many nonneighbors come after. The remaining steps are identical to those for cointerval graph recognition.

Prime graph recognition. This problem is solved by computing modular decomposition. Here, we show that **VertexPartition** alone suffices. Let us say that a call to

OrderedVertexPartition fails if some non-singleton subset A of $V - \{v\}$ is a module, since it can never separate members of the module into two different partition classes. The following shows that **VertexPartition** gives a test of whether a graph is prime.

Theorem 4.10. *An arbitrary graph is prime iff it is connected and neither call to* **OrderedVertexPartition** *fails in* **TransitiveOrientation**.

Proof. If the graph is not connected, each connected component is a module, so it is not prime. If a call to **VertexPartition** (G, v) fails, then there is a nontrivial module of G that is a subset of $V(G) - \{v\}$, and the graph is not prime.

Suppose that neither call to **VertexPartition** fails. The forcing relation on edges of G is defined even when G is not a comparability graph. If the subgraph induced by a module contains an edge e, it contains all edges that are forced by e. The first call to **VertexPartition** demonstrates that any nonsingleton module contains v. The second demonstrates that it contains x. Any nonsingleton module contains (v,x), and any module that contains (v,x) contains all edges of the graph, hence all nodes of the graph, since if G is connected then its edges span $V(G)$. $V(G)$ is the only nonsingleton module. □

4.1. $O(n + m \log n)$ *transitive orientation and a simple primality test*

Before giving the linear-time transitive orientation algorithm, we show that an $O(n + m \log n)$ bound may be obtained quite easily with **VertexPartition**. This gives a simple $O(n + m \log n)$ test of whether G is prime by Theorem 4.10. If G is prime, then the test supplies its transitive orientation directly, without resorting to a general modular decomposition algorithm. If G is not prime, then the procedure gives $O(n + m \log n)$ transitive orientation by Theorem 4.2, but it is first necessary to compute the modular decomposition with an $O(n + m \log n)$ algorithm.

The procedure uses only universal pivots. The rule for selecting a pivot vertex is that any vertex may be selected as long as the class that currently contains it is at most half as large as the class that contained it the last time it was used for partitioning. The potential danger in placing this restriction is that the rule might prevent any vertex from being selected before each partition class is a singleton. To show that this does not happen when G is prime, let X be a largest partition class at the point in the execution where no pivot vertex may be selected anywhere in G without violating the rule. Each node $x \in V(G) - X$ has been used as a pivot at least once since the last time x was in a common partition class X' with the members of X, since the class that currently contains x is at most half as large as X'. It follows that X is a module. If G is prime, X is a singleton set, and since it is a largest partition class, the procedure halts only when all classes are singletons. On the other hand, if **OrderedVertexPartition** fails, G is not prime, and $\mathscr{P}$ cannot be reduced to singleton sets, so the procedure halts while $\mathscr{P}$ contains nonsingleton members.

To implement the observation, mark each partition class with a 'previous size' label that is initially infinity, and put each partition class on a list of 'eligible classes', which contains only classes whose cardinalities are at most half their 'previous size' label. Remove a partition class X from the list of eligible classes. Change X's 'previous size' label to be $|X|$. Pivot once on each member of X. Whenever a partition class Y is split, copy Y's 'previous size' label to the two new classes it splits into, and insert either or both of them in the list of 'eligible classes' if they are at most half the size of their 'previous size' label. Since each time a universal pivot is performed on an arbitrary vertex x, the partition class that contains x is half as large as the partition class that contained it the last time there was a pivot on x. Thus, there are $O(\log n)$ universal pivots on each vertex x, which each take $O(1 + |N(x)|)$ time. Since $\sum_{x \in V(G)} N(x)$ is $O(m)$, this gives an $O((n+m)\log n)$ bound. Prime graphs are connected, so $n = O(m)$ and the bound reduces to $O(m \log n)$. Combining this with the time for modular decomposition, which allows us to perform the transitive orientation on a prime graph is $O(n+m)$, we get an $O(n+m \log n)$ for transitive orientation of arbitrary comparability graphs.

5. Linear-time transitive orientation

To get a linear time bound for the problem, we work on the time bound for **VertexPartition** and use an approach that is based on the following idea. Run the modular decomposition algorithm on the prime graph G that we want to orient. We know that G is prime, but the decomposition algorithm determines this independently. Then if we run **VertexPartition** on G, suppose that at some point a set Y is a nonsingleton member of $\mathscr{P}$. The decomposition algorithm returned a result that claims that there exists a pivot vertex $x \in V(G) - Y$ that splits Y. There must have been some point in the algorithm's execution where it acquired the information from G that allows it to draw this conclusion about Y. If we could recall the point in its execution where this occurred, we would be led to such an x, and we would have a pivot vertex that splits Y. Repeating the process eventually refines $\mathscr{P}$ until it consists of singletons. If we look up each step in the decomposition algorithm at most a constant number of times in the process, the transitive orientation algorithm should be linear, since the decomposition algorithm is linear.

Through the M trees it creates, the decomposition algorithm leaves an extensive record of when it drew inferences about subsets of $V(G)$ failing to be modules. The algorithm works primarily by inserting new nodes to an existing M1 tree or changing the label of an existing node from prime to degenerate. Each such operation excludes more sets from the family of possible modules represented by the tree without including any sets that were previously excluded. Thus, the insertion of a node into an M tree or the changing of a node's label from prime to degenerate corresponds to a precise statement by the algorithm that certain subsets of $V(G)$ are not modules. If Y is one of these subsets, then the vertices of G that justified the insertion of U during the execution of the decomposition algorithm must lead to a pivot that splits Y.

A node U of an M tree is *split* if it intersects more than one partition class. The structure of an M tree makes it easy to keep track of its split nodes as $\mathscr{P}$ evolves. Below, we develop a procedure called *Newsplits* for this. After U is split, any partition class that is neither disjoint from U nor contained in U is among the subsets of $V(G)$ that cannot be modules. Thus, the vertices of G that justified the insertion of U contain a set of pivots that refine $\mathscr{P}$ so that all members of $\mathscr{P}$ are contained in U or disjoint from it. If U is labeled prime, then all partition classes that are contained in U are contained in a child of U. We 'process' U by performing these pivots.

A partition class Y is *consistent* with an M tree if it is either a node of the M tree or a union of children of a degenerate node of the M tree. Y is *inconsistent* with the M tree otherwise. A partition $\mathscr{P}$ of nodes of G is consistent with an M tree if each partition class is consistent with the tree.

Let T be one of the M trees that appears during the decomposition algorithm, such as the P_4 tree, the M2 tree, or one of the intermediate trees that appears during the refinement of the P_4 tree to get the M2 tree. We proceed by giving what we call a *restarting* procedure. A restarting procedure processes split nodes of T until no unprocessed split nodes remain. At that point, $\mathscr{P}$ is consistent with T. We must then perform one or more pivots outside the restarting procedure. Selection of these pivots is facilitated by $\mathscr{P}$'s consistency with T. This further refines $\mathscr{P}$, resulting in new split nodes of T that have not been processed, and in new partition classes that may be inconsistent with T. We start the restarting procedure again on T, and it produces a further refinement of $\mathscr{P}$, halting only when $\mathscr{P}$ is again consistent with T. Each time around, $\mathscr{P}$ is further refined. The process is repeated until $\mathscr{P}$ consists of singleton sets. We are able to charge all operations on T in a way that demonstrates that the total time spent inside all calls to the restarting procedure is $O(n+m)$.

Definition 5.1. A procedure is a *restarting procedure* on M tree T if it halts only when $\mathscr{P}$ is consistent with T. The procedure may be started up again if other pivots make $\mathscr{P}$ inconsistent with T again. A *linear restarting procedure* is one which may be restarted $O(n+m)$ times without spending more than $O(n+m)$ total time inside the calls to it.

Here is a summary of how we proceed:

1. We assume first the existence of a hypothetical linear-time restarting procedure on the M2 tree, and use the assumption to derive a linear-time implementation of **VertexPartition**. This reduces the problem of performing transitive orientation in linear time to the problem of developing a linear restarting procedure on the M2 tree.
2. We assume the existence of a linear-time restarting algorithm on the P_4 tree and use this assumption to derive a linear-time restarting algorithm on the M2 tree. This reduces the problem of performing transitive orientation in linear time to the problem of developing a linear restarting procedure for the P_4 tree.
3. We give a linear-time restarting algorithm on the P_4 tree without making any prior assumptions, thus completing the result.

5.1. Performing partial pivots

In addition to universal pivots, we use two types of pivots called *internal pivots* and *external pivots*, which we now describe.

Definition 5.2. A set W of vertices of G is *isolated* if every member of $\mathscr{P}$ that intersects W is contained in W. Sets U and W are *separated* if no partition class intersects both of them.

Remark 5.3. If T is an M tree, then after a restarting algorithm on T halts, every split node of T is isolated.

We sort all adjacency lists in the order in which nodes appear as leaves of an M tree T for which we intend to write a restarting procedure. This may be accomplished in $O(n+m)$ time by radix sorting all edges of G with the first node as primary sort key and the second node as secondary sort key. Let W be a node of the tree. $W \cap N(x)$ is now a consecutive interval in x's adjacency list. Since the algorithm makes use of restarting procedures on more than one tree, a separate adjacency-list representation of the graph is required for each tree.

Definition 5.4. Let $ord(x,T)$ give the order of vertex x in the leaf numbering of leaves of an M tree T. Let W be a node of T. Since the algorithm uses more than one tree, we will generalize $MIN(W)$ and $MAX(W)$ from Definition 3.3, and let $MIN(W,T) = min\{ord(a,T) : a \in W\}$, and $MAX(W,T) = max\{ord(a,T) : a \in W\}$. If x is a vertex of G, let **start**(W,x,T) and **finish**(W,x,T) denote pointers to the beginning and end of the interval occupied by $W \cap N(x)$ in x's sorted adjacency list that corresponds to T. The T parameter may be dropped from any of these expressions when the tree under consideration is understood.

We will assume that in each M tree T that we work with, each node W is labeled with $MIN(W,T)$ and $MAX(W,T)$. The labeling may be accomplished in $O(n)$ time in a postorder traversal of the tree.

Let $x, X, \mathscr{P}, \mathscr{Q}$ refer to the variables by those names in **VertexPartition**. Suppose W is split. Then it is isolated when T's restarting procedure halts. We may let $\mathscr{Q}$ be the set of members of $\mathscr{P} - X$ that intersect W. If we have a pointer to the interval in x's adjacency list occupied by W, then we can clearly perform the pivot operation using x and $\mathscr{Q}$ in $O(|N(x) \cap W|)$ time, by simply ignoring everything outside of I and applying the universal pivot algorithm to this reduced segment of the adjacency list. We call such a pivot an *internal pivot* because it splits only partition classes that are contained in W. For the pivot to be 'legal', W must be isolated when an internal pivot is applied.

Let us now relax the assumption that W is split when T's restarting procedure halts. Let $\mathscr{Q}$ be the members of $\mathscr{P} - \{X\}$ that do *not* intersect W. We may perform the corresponding pivot in $O(|N(x) - W|)$ time by walking into the adjacency list for x

from the ends until we come to the first and last members of W, and then ignoring the region between these two points during the pivot. We call such a pivot an *external pivot* because it splits only those partition classes that do not intersect W. For the pivot to be 'legal,' W must be isolated or contained in a partition class.

Lemma 5.5. *Let W be an isolated node of an M tree T, and suppose that the adjacency lists representing G are sorted according to the leaf order of T. Except for the time to update new split nodes in the M trees, the following operations take time proportional to* O(1) *plus the number of edges they remove from the adjacency-list representation of G.*

- *An external pivot on W with $x \in W$.*
- *An internal pivot on W with $x \notin W$ if a pointer to an instance of a member of W in x's adjacency list is available.*

Proof. Since W is isolated, every element that is visited in x's adjacency list lies in a different partition class from x, and is thus eliminated from x's adjacency list. □

5.2. Vertex partitioning, given a restarting algorithm on the M2 tree

We now show that finding a linear-time implementation of **VertexPartition** reduces to the problem of finding a linear-time restarting procedure for the M2 tree. Assume such a restarting procedure exists; we will develop it later. Let us call it **M2resolve**.

The procedure for performing the vertex partition is just a depth-first traversal of the M2 tree, with some pivots generated at each node, followed by a call to **M2resolve**.

Procedure **LinVertexPartition**$(T, G, \mathscr{P})$

{T is an M2 tree on a prime graph G and $\mathscr{P}$ is a partition of $V(G)$
that has at least two partition classes. **M2Resolve** is a linear
restarting algorithm on T. Perform pivots, halting only when $\mathscr{P}$
consists of singleton sets.}

Run **Modules** to label each node X of T with an $x \in V(G) - X$ that splits X
Let U be the root of T
For each child W of U do
  M2DFS (W)
M2resolve ()
For each $w \in V(G)$ do
  Perform a universal pivot on w

Procedure **M2DFS** (U)
M2resolve()
Let P be the parent of U
Let $w \in V(G) - U$ that splits U
If P is labeled degenerate then
  Perform an internal pivot on P with w

Else
  Perform an internal pivot on U with w
For each non-singleton child W of U do
  M2DFS(W)

Lemma 5.6. LinVertexPartition *runs in* $O(n+m)$ *time and halts when* $\mathscr{P}$ *consists of singleton sets.*

Proof. Observe that since G is prime, **Modules** labels every internal node U in the tree with some pivot $w \in V(G) - U$ that splits U. By the definition of an M2 tree, if U's parent P is degenerate, then $w \in V(G) - P$.

Claim. *When a node* U *is visited in* **M2DFS**, *it becomes isolated, and if it is prime, its children become isolated.*

These conditions hold at the root after the initial call to **M2resolve**. Suppose U is a node at depth $k \geqslant 1$ in the tree and that the conditions hold at all nodes up through depth $k-1$. Then U's parent P exists and P is isolated. Moreover, if P is prime then U is isolated. The internal pivot performed when U is processed is legal. If P is degenerate, then the pivot node does not lie in P because it distinguishes members of U and the tree is an M2 tree. The pivot splits U, and the next call to **M2resolve** establishes the claim at U. Inductively, the claim follows at each depth.

The depth-first procedure thus splits all internal nodes in the tree. After the last call to **M2resolve**, $\mathscr{P}$ is consistent with the M2 tree by the assumption that **M2resolve** is correct. The only non-singleton members of $\mathscr{P}$ are thus sets of leaf siblings whose parents are degenerate. Since G is prime, each pair of vertices in such a set X disagrees on a third node x in G. By the definition of an M2 tree, x lies outside their parent, hence outside X. The universal pivot on x in the final loop of the main procedure splits them into separate classes. We conclude that each pair of vertices is separated by $\mathscr{P}$ when the procedure terminates, hence every member of $\mathscr{P}$ is a singleton set.

For the time bound, the call to **Modules** is $O(n+m)$, as we showed earlier. The universal pivots at the end of the main procedure are $O(n+m)$, since they happen once on each vertex. At each node of the tree, **M2DFS** performs constant-time operations, plus an internal pivot. By the foregoing claim, U or P is split, and by the assumption that **M2resolve** is correct, the preceding call to it makes P or U isolated before the internal split is performed on it. By Lemma 5.5, the cost of the internal pivot is proportional to the number of edges that it eliminates from G, so the time spent in these pivots over the course of the algorithm is $O(n+m)$. □

5.3. Keeping track of split M-tree nodes

We have now shown that linear-time transitive orientation reduces to the problem of finding a linear-time restarting procedure on the M2 tree. Before proceeding to the next step, we need an additional tool. As we indicated above, a node of an M tree T

only becomes relevant to partitioning $\mathscr{P}$ after it is split, that is, after it is no longer contained in a single partition class. In this section we show how to keep track of split nodes of T as $\mathscr{P}$ is refined by **VertexPartition**. In particular, once a split node is 'processed' by the restarting procedure on T, it is not used again, so we are interested in maintaining a list of unprocessed split nodes without exceeding $O(n+m)$ time over the course of **VertexPartition**. The methods are a variant of a technique developed in [7].

Let $\mathscr{P}' = \mathscr{P} - Y \cup \{Y_a, Y - Y_a\}$. That is, $\mathscr{P}'$ is the refinement of $\mathscr{P}$ obtained by dividing one of its partition classes into two classes during a pivot operation. A call to **Newsplits** (Y_a, T), which is defined below, labels and returns the set of nodes of T that are split by $\mathscr{P}'$ but not by $\mathscr{P}$. These nodes may then be used to update a complete list of split nodes. A pivot splits a set of classes of $\mathscr{P}$ into two classes, so the corresponding set of calls to **Newsplits** is used to find the split nodes of T that result from the pivot.

The **Newsplits** procedure assumes that each node of T is already labeled as to whether it is split by $\mathscr{P}$, and labels only those nodes that are split by $\mathscr{P}'$ but not by $\mathscr{P}$. During processing, it maintains the following information:

- **UniformList**: A set of tree nodes that are contained in Y_a, hence not split.
- **TouchedList**: A set of tree nodes known to intersect Y_a, but not yet known to be contained in Y_a.
- **UniformChildren**(U): An integer label on each internal tree node U that gives the number of children of U that are known to be contained in Y_a. **UniformChildren**(U) is assumed to be initialized to 0 for each node U in T before **Newsplits** is run, and the procedure restores this condition before returning.

The procedure follows:

```
Procedure Newsplits(Y_a, T)
  Let UniformList consist of the leaves of T corresponding to members of Y_a
  Let Newsplits and TouchedList be empty lists of nodes of T
  While UniformList is not empty
    Remove a node U of T from UniformList
    Let P be the parent of U in T
    If P is not marked split then
      UniformChildren(P) := UniformChildren(P) + 1
      If UniformChildren(P) = 1 then put P on TouchedList
      If UniformChildren(P) = |children_T(P)| then
        Move P from TouchedList to UniformList
        UniformChildren(P) := 0 {reinitialize for next time}
  While TouchedList is not empty
    Remove a tree node W from the TouchedList
    While W is not marked as split
      Mark W as split
      Insert W in Newsplits
```

UniformChildren$(P) := 0$ {reinitialize for next time}
$W := parent_T(W)$
return **Newsplits**

Lemma 5.7. *Let $\mathscr{P}' = \mathscr{P} - Y \cup \{Y_a, Y - Y_a\}$ be a refinement of a partition $\mathscr{P}$ on vertices of G. Suppose that those nodes of a union tree T on G that are split by $\mathscr{P}$ are labeled.* **Newsplits** (Y_a, T) *labels and returns the set of nodes of T that are split by $\mathscr{P}'$ but not split by $\mathscr{P}$, and takes $O(|Y_a|+k)$ time, where k is the number of nodes of T that it returns.*

Proof. By induction on the height of a node U in T, U is moved from **TouchedList** to **UniformList** at some point during execution of the first `While` loop if and only if it is contained in Y_a. A node W is inserted in **TouchedList** at some point during execution of the first `While` loop if and only if some child of W is inserted in **UniformList**. It follows that the nodes that remain in **TouchedList** at the end of execution of the first `While` loop are those that have at least one child but not all children contained in Y_a.

Every node Z that is split by $\mathscr{P}'$ but not by $\mathscr{P}$ has leaf descendants that are contained in Y_a, so each path from Z to such a descendant must contain a member of **TouchedList**. No descendant of Z is labeled as split when **Newsplits** is called, since Z is not split by $\mathscr{P}$. The paths from Z to its descendants on **TouchedList** consist of unmarked nodes, so the final loop marks Z as split. This proves the correctness of the procedure.

To obtain the time bound, note that since each node of T has at least two children and there are $|Y_a|$ leaves of T that are contained in Y_a, there are $O(|Y_a|)$ nodes of T that are contained in Y_a. Thus, $O(|Y_a|)$ nodes are inserted in **UniformList**. Each iteration of the first `While` loop removes a node from **UniformList**, so this loop can execute at most $O(|Y_a|)$ times. Constant time is spent in each iteration, so the first loop takes $O(|Y_a|)$ time. The members of **TouchedList** are parents of members of **UniformList**, so **TouchedList** may have at most $O(|Y_a|)$ elements at the end of the first loop. The second outer loop executes $O(|Y_a|)$ times. The inner loop spends constant time per iteration and iterates once for each node inserted in **Newsplits**. All iterations of this inner loop require $O(k)$ time. The total time required by the second main loop is thus $O(|Y_a| + k)$. □

Lemma 5.8. *The nodes of a union tree T that are split initially by an arbitrary starting partition may be identified in $O(n)$ time.*

Proof. Let $\mathscr{P}_s$ be the starting partition. If $\mathscr{P}_s = \{\{v\}, V - \{v\}\}$ as in **OrderedVertexPartition**, then the split nodes of T are just the ancestors of the leaf of T corresponding to v. If $\mathscr{P}_s = \{Y_1, Y_2, \ldots, Y_j\}$ is arbitrary, we may express $\mathscr{P}$ as the result of a series of splits that are suitable for processing by **Newsplits**. Let $R_1 = V$ and let $\mathscr{P}_1 = \{V\}$. Let $R_i = R_{i-1} - Y_{i-1}$ and let $\mathscr{P}_i = \mathscr{P}_{i-1} - \{R_{i-1}\} \cup \{Y_i, R_i\}$. $\mathscr{P}_s$ is just $\mathscr{P}_{j-1}$. Thus, we may obtain the nodes that are split by $\mathscr{P}_s$ by calling **Newsplits** on each of $Y_1, Y_2, \ldots, Y_{j-1}$. By Lemma 5.7, this takes $O(n)$ time. □

Lemma 5.9. *The total time required maintain a current list of all split nodes after each pivot in* **VertexPartition** *is* $O(n+m)$.

Proof. Follows from Lemmas 5.5–5.8. □

We may thus ignore **Newsplits** in the remaining analysis of the time bound of **VertexPartition**.

5.4. A restarting procedure on the M2 tree, Given one on the P_4 tree

The goal of this section is to develop a linear-time restarting procedure for the M2 tree, given a linear-time restarting procedure for the P_4 tree. This procedure is called **M2restart** in the pseudocode for **VertexPartition**. In the next section, we give the restarting procedure on the P_4 tree, completing the algorithm.

Recall that the **M1M2** procedure of the decomposition algorithm inputs the P_4 tree and outputs the M2 tree. It does this by inserting new nodes into the P_4 tree and changing the labels of some nodes from prime to degenerate. Of the procedures that are called, only **Splitters**, **SCC**, and **PQ** make refinements to the tree. Thus, the refinements to the P_4 tree come in the following stages:

1. **Splitters** produces a refinement T_1 of the P_4 tree;
2. **SCC** then produces a refinement T_2 of T_1;
3. **PQ** then produces a refinement T_3 of T_2;
4. A second call to **Splitters** produces a refinement T_4 of T_3;
5. A second call to **SCC** produces a refinement T_5 of T_4, and T_5 is an M2 tree.

We proceed as follows:

- Given an arbitrary input tree to a call to **Splitters** and the availability of a linear restarting procedure on that input tree, we derive a linear restarting procedure on the refinement of the tree produced by **Splitters**. This procedure is called **Splitresolve**.
- Given an output tree of **Splitters** and a linear restarting procedure on it, we derive a linear restarting procedure on the refinement of it produced by the main body of **SCC**. This procedure is called **SCCresolve**.
- Given an output tree of **SCC** and a linear restarting procedure on it, we give a linear restarting procedure on the refinement of the tree produced by the main body of **PQ**. This procedure is called **PQresolve**.

This is sufficient to get the result we seek. Given a linear restarting procedure on the P_4 tree, **Splitresolve** gives one on T_1. Given a linear restarting procedure on T_1, **SCCresolve** gives one on T_2. Given one on T_2, **PQresolve** gives one on T_3. Given one on T_3, **Splitresolve** gives one on T_4. Given one on T_4, **SCCresolve** gives one on the M2 tree.

Since each incarnation of each of these procedures has its own tree on which it is a restarting procedure, and its own requirements on the ordering of adjacency lists in the representation of G, assume that each has its own copy of the tree on which it is a restarting procedure, and its own copy of the adjacency-list representation of G.

Thus, there are five trees and five copies of G used in the two calls to **Splitresolve**, the two calls to **SCCresolve**, and the call to **PQresolve**. Any time one of the procedures performs a pivot, it removes edges only from its own copy of G. However, it must make the appropriate calls to *Newsplits* on all five trees to keep the split-node labeling current in all five trees. Each procedure keeps its own list of split nodes in its tree that it has not yet processed; this is updated whenever any new nodes of its tree are labeled as split. Inside a procedure, the list of split nodes is known as **Splitlist**.

5.4.1. Splitresolve

Let T denote the input tree to a call to **Splitters** and let T_s denote the tree that results from omitting the contraction steps that may occur after nodes are inserted to T. That is, we omit the final steps of the first loop, where z or X is contracted out of the tree. Omitting the step allows us to avoid inserting some special cases into the pseudocode for **Splitresolve**, and does not affect the family of sets that are consistent with the tree. T_s may therefore have an occasional node that has only one child, but each node is still defined to be synonymous with the set of vertices of G that correspond to its leaf descendants. Thus, a restarting procedure on T_s is also a restarting procedure on the output tree of **Splitters**.

T_s is obtained from T by giving each degenerate node X a new child Z that is a union of some of X's children, then labeling Z degenerate and relabeling X prime. The children of Z in T_s are the children of X in T that are not split by any vertex in $V(G)-X$. The children of X in T_s are Z and the children of X in T that are split by a vertex of $V(G)-X$. Let **StartingResolve** denote an assumed linear-time restarting procedure on T.

```
Procedure Splitresolve ()
  StartingResolve()
  While Splitlist is not empty
    Remove a tree node Z from Splitlist
    If Z is a node of T_s that was added to T by Splitters then
      Let X be Z's parent in T_s
      For each child W ≠ Z of X in T_s do
        Let w be a vertex that splits W and that does not belong to X
        Perform an internal pivot on X with w
      StartingResolve ()
```

Lemma 5.10. Splitresolve *is a linear-time restarting procedure on the output tree of a call to* **Splitters** *if* **StartingResolve** *is a linear-time restarting procedure on its input tree.*

Proof. Let $\mathcal{F}$ be the set of children of X in the input tree T. Every union of a subfamily of $\mathcal{F}$ is consistent with T. The unions of members of $\mathcal{F}$ that are consistent with T_s are $X=\bigcup\mathcal{F}$, individual members of $\mathcal{F}$, and unions of members of $\mathcal{F}$ that

are children of Z. The insertion of Z and the relabeling of X as prime causes no other change in the family of sets that are consistent with the tree. For the correctness, it suffices to show that the inner `If` statement ensures that every partition class that intersects X is contained in Z or in one of X's other children. Once this occurs, no partition class in any subsequent refinement of $\mathscr{P}$ may be among the sets made inconsistent with the tree when Z was inserted. Other iterations accomplish the same result for all other split nodes that were inserted by **Splitters**. The final call made to **StartingResolve** makes sure that $\mathscr{P}$ consistent with T. This implies then that $\mathscr{P}$ is consistent with T_s.

Splitters ensures that each child W of X other than Z is split by a vertex that lies outside of X, and provides such a vertex from the set $\{low(W), high(W)\}$ (see Definition 3.3). Since X is split, hence isolated by the most recent call to **StartingResolve**, the internal pivot on X with this node is legal and leaves W split. The procedure leaves every child of X split, with the possible exception of Z. The next call to **StartingResolve** leaves these split children of X isolated, and since X is also isolated, it leaves Z isolated. This establishes the correctness of the procedure.

We now establish the time bound. A node of T_s is inserted in **Splitlist** when it is first split, and the cost of doing this may be ignored since it is accomplished with **Newsplit**. Thus, Z is removed only once from **Splitlist**, and since only one child of X is inserted by **Splitters**, an internal pivot is employed once on X for each sibling of Z. **Splitters** found the pivot node w. A trivial modification of **Splitters** labels w with a pointer to an instance of a member of Z in w's adjacency list, to facilitate the pivot operation. Since X is isolated when the internal pivots are performed, the total cost of internal pivots in the procedure is $O(n+m)$ by Lemma 5.5. □

5.4.2. SCCresolve

Let T_r be as in the pseudocode of the **SCC** procedure, and let T_s be the output tree of the procedure. In this section we give **SCCresolve**, which is a linear restarting procedure on T_s if **Splitresolve** is a linear restarting procedure on T_r.

Definition 5.11. The existence of an edge (U,W) in a forcing graph indicates that U disagrees on some member of W. (U,W) carries a label $w(U,W)$, which is a node of W that splits U in G, and labels $a(U,W)$, and $n(U,W)$, which are nodes of U that are adjacent and nonadjacent, respectively, to $w(U,W)$.

By Remark 3.4, these labels can be supplied in $O(n+m)$ time with a trivial modification of **Splitters** and **ForcingGraphs**.

Procedure **SCCresolve** ()
  Splitresolve ()
  While **Splitlist** is not empty do
    Remove a node Y from **Splitlist**
    If Y was added to T_r by **SCC** then

{ The children of Y are a strongly connected component in a forcing graph}
Let F be the forcing graph in which Y's children are a strongly connected component
Let F' be the subgraph of F induced by children of Y.
Do a depth-first search of F' to find a forcing edge (U, W) that connects two separated nodes
Perform an external pivot on W with $w(U, W)$
Splitresolve()
Call **SCCDFS**$(F', U,)$

Procedure **SCCDFS**(F', W)
For each predecessor U of W in F' that has not yet been visited by **SCCDFS** do
Perform an external pivot on W with $w(U, W)$
Splitresolve( )
SCCDFS(F', U)

Lemma 5.12. *Let T be an M tree and let U and W be siblings in T. If $\mathscr{P}$ is consistent with T, it may be determined in* O(1) *time whether U is separated from W.*

Proof. Since no member of $\mathscr{P}$ may overlap U or W, the question reduces to a test of whether arbitrary $u \in U$ and $w \in W$ are members of the same partition class. The lemma follows from Remark 4.5. □

Lemma 5.13. *Let T_r refer to the variable by that name in* **SCCresolve**, *and let T_s be the output tree of* **SCC**. **SCCresolve** *is a linear-time restarting algorithm on T_s if* **Splitresolve** *is a linear-time restarting algorithm on T_r.*

Proof. To show that it is a restarting procedure, we claim first that when **SCCDFS** (F', W) is called on an isolated member W of a strongly connected component, it leaves every member of the strongly connected component isolated. Clearly, it performs a depth-first search on the transpose of F', and visits all members of the component, since F' is strongly connected. The external pivot with $w(U, W)$ splits U, and since W is separated from U and U is a member of T_r, the call to **Splitresolve** that follows the operation must isolate U. The claim then follows inductively for all members of the component.

In the main procedure, the first call to **Splitresolve** leaves $\mathscr{P}$ consistent with T_r. Y's children are nodes of T_r, so any two children that are not contained in the same partition class are separated. Since the children are the members of a strongly connected component of F, (U, W) exists and W is either isolated or contained in a single partition class, as required for the external pivot on W. The external pivot and the following call to **Splitresolve** isolate U, establishing correct conditions for calling **SCCDFS**. As we have shown, **SCCDFS** leaves each child Z of Y isolated. In the refinement of $\mathscr{P}$ that remains after **SCCresolve** halts, it follows that for each node Y inserted in T_r by

SCC, Y is either contained in a single partition class, or else every class that intersects Y is contained in one of Y's children. The final call to **Splitresolve** ensures that the refinement is consistent with T_r. These two statements imply that it is consistent with T_s. **SCCresolve** is a restarting procedure on T_s.

There are $O(n+m)$ calls to **Splitresolve** in any $O(n+m)$ calls to **SCCresolve**, since Y is processed at most once in all calls. The bound for the calls to **Splitresolve** is $O(n+m)$, since it is assumed to be a linear-time restarting procedure. The bound for finding (U, W) is proportional to the number of edges of F that leave members of the strongly-connected components corresponding to Y. By Lemma 5.12, searching for (U, W) and the other operations that act on forcing edges can thus be charged at constant time to each of the forcing edges they act on. There are $O(n+m)$ forcing-graph edges, since the forcing graphs are constructed explicitly by the $O(n+m)$ decomposition algorithm. Except for the first external pivot inside the main loop of **SCCresolve**, all external pivots take place on isolated nodes, so their time can be charged to edges eliminated from G, by Lemma 5.5, and their total time over all iterations is $O(n+m)$. When the first external pivot is carried out in the main procedure, W may be contained in a partition class rather than isolated. Thus, not all elements traversed in the pivot node's adjacency list are eliminated from G. However, we have shown that W is isolated after the call to **SCCDFS**. Thus, any elements that were not eliminated from the pivot node's adjacency list are eliminated the next time the external pivot is repeated, which is inside **SCCDFS**. We may charge the operations of the first pivot on W to edges that are eliminated in the call to **SCCDFS**. An $O(n+m)$ bound for $O(n+m)$ calls to **SCCresolve** follows. □

5.4.3. PQresolve

In this section, we focus on the refinements performed on the M1 tree during a call to **PQ**. Let T be the tree on which **PQ** was called, let T_r be the output of the call to **Splitters** made from **ForcingGraphs** inside **SCC**, and let T_s be the output of the call to **SCC**. Let T_t be the output of **PQ**, when we omit the step where p or q is contracted in the last line of the pseudocode of **PQ**. As before, omitting contractions has no effect on the family of sets that are consistent with the tree, so a restarting procedure on T_t is also a restarting procedure on the tree that is produced when the contraction steps are not omitted.

For each degenerate node U of T_s, order the children of U according to the topological sort on its children that was used in **PQ** when it refined T_s to produce T_t. Order children of other nodes arbitrarily. For each $x \in V(G)$, let $ord(x)$ denote the rank of x in the leaf order on this tree. Assume all adjacency lists have been organized in ascending order of $ord()$ value, as described above. Since each node of the tree constitutes an interval in the $ord()$ ordering, one node's interval precedes or follows another node's interval if neither is an ancestor of the other. Thus, $ord()$ also imposes a linear order on any set of pairwise disjoint nodes of T_s.

A cotree node U in the P_4 tree is created by a call to **cotree** on a subgraph $G|S$ of G, for some $S \subseteq V(G)$ such that $G|S$ is a cograph. Let the *representative* of a child W

of U in the P_4 tree be an arbitrary vertex $w \in (W \cap S)$. Such a representative clearly exists, since the leaf descendants of a node in the tree returned by *cotree* remain among its descendants in the P_4 tree. The representatives of the children of U induce either a complete or empty subgraph in G, since they induce a complete or empty subgraph in $G|S$.

Definition 5.14. Let D be a node of a forcing graph G_f that was computed in the call to **ForcingGraphs** inside **SCC**. Since the nodes of G_f are disjoint sets that are nodes of T_s, the $ord()$ order imposes a total order on them that is consistent with the topological sort used by **PQ** on the component graph of G_f. Let $Early(D)$ denote the earliest node W in this order such that (D, W) is an edge of G_f and D and W are not subsets of the same partition class in $\mathscr{P}$. If there is no such W, then $Early(D)$ is undefined.

By Lemma 5.12, $Early(D)$ may be found in time proportional to the number of successors of D in G_f whenever **SCCresolve** halts.

SCCresolve isolates any split node that corresponds to a strongly-connected component in the tree returned by **SCC**. The following procedure may sometimes be used to isolate such nodes even when they are not split. The parameter, U, is a member of a strongly connected component that we are interested in isolating. The procedure requires only that U have a successor in the forcing graph that does not lie in the same member of $\mathscr{P}$ as U does.

Procedure **IsolateSCC** (U)
{U is a node of a forcing graph G_f that has a successor that does
not lie in the same member of $\mathscr{P}$ as U does.
The procedure performs pivots that cause every member of U's
strongly-connected component to become isolated }

SCCresolve()
if U is not split then
$W' := Early(U)$
Let w be the representative of W'
Let W be the union of members of the strongly connected component that contains W'
Let X be the union of nodes of G_f
Perform an external pivot on X with w
Perform an external pivot on W with w
SCCresolve()
Perform an external pivot on U with $n(U, W')$ (See Definition 5.11)
Perform an external pivot on U with $a(U, W')$
SCCresolve()
Perform an external pivot on W with w

Assume that **SCCresolve** is a linear-time restarting algorithm on T_s. For the correctness, note first that if U is already split, the procedure achieves the desired result with

the first call to **SCCresolve** by the assumption that it is a restarting procedure on T_s. W' exists because of the restrictions placed on a what constitutes a valid input for U. Since X is split and a node of T_s, it is isolated after the initial call to **SCCresolve**. Thus, the external pivot on it is legal. W is also a node of T_s. It may or may not be split, so it is either isolated or contained in a single partition class after the initial call to **SCCresolve**. In either case, the first pivot on it is legal. W' is separated from U by the first call to **SCCresolve**. The external pivot on W thus splits U, hence U's strongly connected component. The second call to **SCCresolve** isolates each member of U's strongly connected component. This demonstrates that the procedure achieves the desired result. Since U is now isolated, the next two external pivots on U are legal. The third call to **SCCresolve** ensures that the final external pivot on W with w is legal.

Lemma 5.15. *After the two pivots on W with w in* **IsolateSCC**, *no subsequent external pivot on W with w when $\mathscr{P}$ is consistent with T_s can induce a proper refinement of $\mathscr{P}$.*

Proof. *Claim* 1: *The second call to* **SCCresolve** *separates w from all vertices preceding $MIN(W)$ in $ord()$.* Let C be the union of members of U's strongly connected component. If $C = W$, then the claim is immediate, since C is isolated after the second call to **SCCresolve**. Assume $C \neq W$. Since the second call to **SCCresolve** isolates C, it separates C from w. Let Y be the partition class that contains w when $W' = Early(U)$ is computed, and let Y' be the subset of $Y - C$ that precedes $MIN(W)$ in the $ord()$ numbering. It remains to show that the second call to **SCCresolve** separates Y' from w. X's isolation implies that Y' is a subset of X. No member of Y' distinguishes members of U, since $W' = Early(U)$ contains the earliest vertex in $ord()$ that splits U and is not in the same partition class as U is. However, W' contains a member that distinguishes $n(U, W)$ and $a(U, W)$. No member of U distinguishes members of W, since W is later in $ord()$ than C is. All members of W', including w, thus distinguish $n(U, W)$ and $a(U, W)$. If no member of Y' distinguishes $n(U, W)$ and $a(U, W)$ but w does distinguish them, it follows that the external pivots on U with $n(U, W)$ and $a(U, W)$ separate Y' from w.

Claim 2: *Let B be the set of vertices of G whose ord numbers are on the interval from $MAX(W) + 1$ to $MAX(X)$. No two members of B are distinguished by w.* The members of B are in strongly connected components that are later than W in the topological sort, by the definition of B. Thus w does not distinguish the members of any node of the forcing graph that is contained in B. Since w is the representative of W', it is either adjacent in G to the representatives of all of W's siblings or it is adjacent in G to none of those representatives. Thus, it cannot distinguish members of any two siblings of W' contained in B, and the claim follows.

Now that we have established the claims, let Z be the partition class that contains w after the second call to **SCCresolve**. If Z is contained in W, then W is isolated by the call to **SCCresolve**, and the subsequent external pivot on W is the last one

that can induce a proper refinement of $\mathscr{P}$. Otherwise, by the assumed correctness of **SCCresolve**, Z contains W. After the second external pivot on W with w, no future external pivot on W with w may split a future partition class that is disjoint from Z. The lemma thus follows for any future partition classes that are disjoint from Z. Clearly, it follows for any future partition classes that are contained in W, since an external pivot on W ignores such classes. In any future refinement of $\mathscr{P}$ that is consistent with T_s, any partition class that is contained in Z and not contained in W is disjoint from W. By Claim 1, $Z - W \subseteq B$. The lemma then follows from Claim 2. □

Lemma 5.16. *Any sequence of* $O(n+m)$ *calls to* **IsolateSCC** *takes* $O(n+m)$ *time if* **SCCresolve** *is a linear-time restarting procedure.*

Proof. The time spent in the calls to **SCCresolve** is $O(n+m)$ by the assumption that it is a linear restarting procedure. The time to find $Early(U)$ is charged to edges of the forcing graph that leave U at constant time per edge, by Lemma 5.12. It remains to bound the time spent in each of the five external pivots over all calls to **IsolateSCC**. Since X is split and isolated by the first call to **SCCresolve**, the first external pivot takes $O(n+m)$ time over all calls to **IsolateSCC**, by Lemma 5.5. Similarly, U is isolated by the second call to **SCCresolve**, so the external pivots on U take $O(n+m)$ time over all calls to **IsolateSCC**.

The difficult step is bounding the time spent in the two external pivots on W. Let us call these two pivots *special pivots*, since they are handled somewhat differently. If W is not marked 'off limits', perform the two special pivots normally, then mark W as off limits. If it is already marked off limits, omit the two special pivots. The omission has no effect on $\mathscr{P}$, by Lemma 5.15. The two pivots take time that is proportional to the number of elements of w's adjacency list that are members of $X - W$, since X is isolated by the first call to **SCCresolve** and the external pivot on X removed elements of $V(G) - X$ from w's adjacency list. When W is not marked off limits, pay for the two special pivots by charging each of these adjacency-list elements a constant cost. This ensures that whenever a pair of special pivots with w is performed on a node that is not marked off limits, the cost is charged to elements of w's adjacency list that have not been charged to before by any special pivot. This gives an $O(n+m)$ bound on the cost of all special pivots in all calls to **IsolateSCC**. □

We now give the pseudocode for **PQresolve**. Recall that the procedure works on T_t, and **Splitlist** is a list of split nodes of T_t. Recall the definition of *drivers*, *passengers*, 'P' and 'Q' nodes given in the pseudocode of **PQ**. Let an 'SCC' node denote any node of a strong-component graph used by **PQ**.

```
Procedure PQresolve()
  While Splitlist is not empty do
    Remove a node U from Splitlist
    If U is an unprocessed node that was added to T_s by PQ then
```

PQdescendants(U)
SCCresolve()

Procedure **PQdescendants**(U)
For each unprocessed child W of U that was added to T_s by **PQ** do
PQdescendants(W)
SCCresolve()
If U is a 'P' node then **PQprocess**(U)

Procedure **PQprocess**(P)

{P is a 'P' node. P is a union of children of some degenerate node U in T_r. Let X be P's driver. X is the union of the members of a strongly connected component $\mathscr{C}_X$ of $G(G, U, T_r)$, since X is a node of $G_c(G, U, T_r)$. The procedure causes each member of $\mathscr{C}_X$ to become isolated}

Mark P as processed
SCCresolve()
If X is not split then
Find an edge (X', Z) of $G(G, U, T_r)$ such that $X' \in \mathscr{C}_X$ and X' and Z are separated.
IsolateSCC(X')
For each $X'' \in \mathscr{C}_X$ do
For each forcing edge (X'', Z') in $G(G, U, T_r)$ such that $Z' \notin X$ do
Perform an external pivot on X with $n(X'', Z)$
Perform an external pivot on X with $a(X'', Z)$

Lemma 5.17. *Suppose* **SCCprocess** *is a restarting procedure on T_s. Let P be a P node in T_t and let X be its driver. In* **PQprocess** *either the call to* **SCCresolve**(P) *or the call to* **IsolateSCC** *isolates X.*

Proof. If X is split when **PQprocess** is called, then the result follows from the assumed correctness of **SCCprocess**. We will now assume that X is not split.

Observation ($*$): *P is a P node. From the way P was constructed in* **PQ**, *for each passenger B of P there exist an edge (X'', B'') of $G(G, U, T_r)$ such that X'' is a subset of X, and B'' is a subset of the driver of B.*

We will now define the *rank* of a P node or an SCC node. The rank preserves information about the order in which **PQprocess** was called on various P nodes. The rank of each P or SCC node is initially zero. At the moment when **PQprocess** is called on a P node, the P node assumes a new rank that is one plus the maximum of the ranks of its passengers, and which remains constant thereafter. Intuitively, the rank of a P node thus tells how high it is in the recursion tree in the call to **PQdescendants** that caused the node to be processed.

If P has rank 1, then the driver and each passenger of P is an SCC node or an unsplit P node. Any split SCC nodes are isolated by of the call to **SCCresolve** in **PQdescendants**. The driver and each passenger is either isolated or a subset of a partition class in $\mathscr{P}$. Since P is split, it follows that some passenger B is separated from X. From Observation $(*)$ above, when **PQprocess** is called on P, there exists a forcing edge that satisfies the definition of (X', Z) in the procedure. Thus, the lemma follows for P from the correctness of **IsolateSCC** and the two external pivots on X are legal.

Now assume the lemma is true for any node of rank up to $k \geqslant 1$, and suppose P is a node with rank $k+1$. When a call to **PQprocess** is initiated on P, there has already been a call to **PQprocess** on some passenger B, and B has lower rank than P. By the inductive hypothesis, B's driver is isolated. By the observation above, there exists a forcing edge satisfying the definition of (X', Z). The lemma again follows for P from the correctness of **IsolateSCC**, and the two external pivots on X are again legal. □

Lemma 5.18. *When* **PQresolve** *halts, $\mathscr{P}$ is consistent with T_t, assuming that* **SCCresolve** *is a restarting algorithm on T_s.*

Proof. The last call to **SCCresolve** before **PQresolve** halts ensures that $\mathscr{P}$ is consistent with T_s. T_t consists of T_s with some additional P and Q nodes inserted. Since each P node has two children and each Q node is degenerate, it suffices to show that no member of $\mathscr{P}$ overlaps any P or Q node in the M2 tree.

When **PQresolve** halts, it has called **PQprocess** on every split P node, since **Splitlist** is empty. No partition class that overlaps a P node may intersect its driver, by Lemma 5.17. Thus, if a partition class overlaps a Q node, it must overlap the Q-node's parent, which is a P node. It therefore suffices to show that no class may overlap a P node.

Suppose for purposes of contradiction that A is a partition class that overlaps some P node.

Each P node was created in **PQ**, when a node denoted p in **PQ** is inserted into the tree. P is given by the set of leaf descendants of p, which remain unchanged thereafter. The P nodes are created in a chronological order that is partially constrained by the topological sorts of the forcing graphs. Let P be the P node in the M2 tree that was created earliest among all P nodes that overlap A. To obtain the contradiction, we show that P must contain A.

Let X and U refer to the values of the variables by those names during the iteration of the outer loop of **PQ** that created P. X is P's driver. Let $\mathscr{C}_X$ be the set of nodes of $G(G, U, T_t)$ that are contained in X. Let $\mathscr{R}$ be the set of nodes of $G_c(G, U, T_t)$ that are marked as 'representatives' at the beginning of the iteration of the inner loop that creates P, and let $\mathscr{F}$ be the set of nodes whose drivers are members of $\mathscr{R}$. The members of $\mathscr{F}$ are just the current children of U that are candidates to become P's passengers when P is created.

Claim 1. *X was isolated either by the call to* **SCCresolve** *or by the call to* **IsolateSCC** *in* **PQprocess(P)**.

This follows from Lemma 5.17.

Claim 2. *All members of $\mathscr{F}$ are either contained in or disjoint from A.*

The last call to **SCCresolve** in the main procedure of **PQresolve** ensures that A overlaps no *SCC* nodes. By the definition of P, A overlaps no P nodes in $\mathscr{F}$.

Claim 3. *A is disjoint from X and contains one of P's passengers.*

A must be disjoint from X, since otherwise Claim 1 implies that A is contained in X and does not overlap P. Thus, one of P's passengers is contained in A, since otherwise Claim 2 implies that P does not intersect A.

Claim 4. *A is a union of some of P's passengers.*

Since U is a node of T_s, it is isolated after the first call to **SCCresolve** in **PQprocess**. Thus A is a subset of U. There is an edge of $G_c(G,U,T_r)$ from X to the drivers of each of P's passengers. By Claim 3, there is an edge of $G(G,U,T_r)$ from some $X' \in \mathscr{C}_X$ to a node Z of $G(G,U,T_r)$ that is contained in A. Thus $w(X',Z)$ is contained in A. By Claim 1, X is isolated when the final nested loops of **PQprocess**(P) perform an external pivot on X with $n(X',Z)$ and $a(X',Z)$. Thus, the partition class that contained A at that time did not contain $n(X',Z)$ or $a(X',Z)$. Since members of A were not split into different classes by these pivots, A must agree on $n(X',Z)$ and A must agree on $a(X',Z)$. Since $n(X',Z)$ and $a(X',Z)$ disagree on $w(X',Z)$, they must disagree on every node in A. It follows that there is an edge of $G(G,U,T_r)$ from X' to every node of $G(G,U,T_r)$ that is contained in A. Thus, A is a union of nodes of $G(G,U,T_r)$ that are subsets of strongly connected components that follow X in the topological sort. Every such node is contained in a member C of $\mathscr{F}$. By Claim 2, C is contained in A, so there is an edge of $G_c(G,U,T_r)$ to its driver, and C becomes a passenger of P.

Claim 4 implies that P contains A, a contradiction. □

Lemma 5.19. *Let T_s and T_t be as defined at the beginning of this section.* **PQresolve** *is a linear-time restarting procedure on T_t, provided* **SCCresolve** *is a linear-time restarting procedure on T_s.*

Proof. Each P node is inserted, examined, and removed at most once in **Splitlist**. We showed previously that the time **Newsplits** spends maintaining **Splitlist** is $O(n+m)$. Exclusive of **PQprocess**, the remainder of the operations are just a series of truncated depth-first traversals of different regions of the tree, and which collectively only visit each node of the tree once.

Thus, it remains to bound the time spent in all calls to **PQprocess**. Since each node P is processed at most once, $O(n+m)$ time is spent in the calls to **SCCresolve** and

IsolateSCC by Lemmas 5.13 and 5.16. By Lemma 5.12, the cost of finding (X', Z) and the (X'', Z') may be charged to those edges of $G(G, U, T_r)$ that are members of $\mathscr{C}_X \times V(G(G, U, T))$ at constant time per edge, since they will only be charged in this way in **Process**(P). Similarly, if each pair of pivots on X in the inner loop is assigned to (X'', Z') in the inner loop, each edge is assigned two external pivots. Since there are $O(n+m)$ edges in all forcing graphs combined, there are at most $O(n+m)$ of these pivot operations. By Lemma 5.17 and 5.5, the external pivots contribute $O(n+m)$ time to the running time of the restarting sequence. □

5.5. A restarting procedure on the P_4 tree

We have reduced the problem of performing transitive orientation in linear time to that of developing a linear restarting procedure on the P_4 tree. In this section, we give such a procedure, thus completing the transitive orientation algorithm.

Let U be a P_4 or cotree node, and let u be the node of the data structure created by **Createtree** that represents U (i.e. the node of the P_4 tree whose leaf descendants correspond to U). The *local members* of U are those that were passed to the recursive incarnation of **Createtree** or **cotree** that created u.

The existence of a node u in the P_4 tree encodes information that the algorithm discovered when it examined *local*, not global, members of u. Thus, to find what information about G the algorithm encoded by installing u in the P_4 tree, we restrict our attention to its local members. The P_4 tree is not a suitable structure for looking up the local members of a node. We use what we will call a *local tree*. For each node W of the P_4 tree, the local tree has a unique corresponding node, and the leaf descendants of that node in the local tree give the local members of W.

The local tree is created by the following procedure, which duplicates the recursion of **Createtree**.

Function **Localtree**(G)
Input: An undirected graph G
Output: A local tree corresponding to G
if G is a cograph then T = `Cotree`(G)
else
  Let (a,b), (b,c), (c,d) be a P_4 in G;
  $T_A :=$ **Localtree**$(G|A)$
  $T_B :=$ **Localtree**$(G|B)$
  $T_C :=$ **Localtree**$(G|C)$
  $T_D :=$ **Localtree**$(G|D)$
  $T_E :=$ **Localtree**$(G|E)$
  Create a tree node p and label it a "P_4" node
  Create a tree node s and label it an "S" node
  for $i = A, B, C, D, E$ do
    Make T_i be a child of p;

Select $x \in \{a,b,c,d\}$
$T_F := \textbf{Localtree}(G|(U \cup \{x\}))$
Make p be a child of s
Let T_F be the other child of s
Let T be the tree rooted at s
return T

It is instructive to verify that the local tree essentially gives the recursion tree for **Createtree**, except that each recursive incarnation creates two tree nodes, s and p, rather than just one. We assume that the same unspecified choices of $\{a,b,c,d\}$ and $x \in \{a,b,c,d\}$ are made at each step in the recursion of **Createtree** and **Localtree**. Each P_4 or cotree node in either of the two trees has a unique corresponding node in the other. However, the 'S' nodes do not have analogous nodes in the P_4 tree. Each P_4 node is paired with a parent S node that is created by the same incarnation of **Localtree**, so each S node has a unique P_4 child. Note also that since the node x mentioned in **Localtree** is passed to more than one recursive call, there will be more than one leaf of the tree that corresponds to x.

Definition 5.20. Let u be a node of a local tree T on graph G, and let X be an arbitrary set of vertices of G.

- $S(u) = \{l : l \text{ is a leaf descendant of } u \text{ in } T\}$.
- $L(u) = \{x : x \text{ is a vertex of } G \text{ that corresponds to at least one } l \in S(u)\}$.
- Given an arbitrary set X of vertices of G, $S(X)$ is the set of leaves of the local tree that correspond to a member of X. If $\mathscr{P}$ is a partition of vertices of G, $S(\mathscr{P}) = \{S(X) : x \in \mathscr{P}$.

For distinct local tree nodes u and w it is possible that $L(u) = L(w)$. However, when no ambiguity is possible, we will use u and $L(u)$ interchangeably, so that we may treat each node of the local tree as a set.

Let $\mathscr{P}$ be the current partition in **VertexPartition**. A node u of the local tree is *split* if $L(u)$ intersects more than one partition class of $\mathscr{P}$. Since more than one leaf of the local tree may correspond to the same vertex of G, the local tree is not a union tree on vertices of G. However, we may still use **Newsplits** to keep track of which nodes are split locally as $\mathscr{P}$ evolves. Let $\mathscr{P}' = \mathscr{P} - Y \cup \{Y_a, Y - Y_a\}$. That is, $\mathscr{P}'$ is the refinement of $\mathscr{P}$ obtained by dividing one of its partition classes into two classes during a pivot operation, where Y_a is the subset of Y that is adjacent to the pivot vertex. Clearly, a call to **Newsplit**$(S(Y_a), T)$, finds each node u of T such that $S(u)$ is split by $S(\mathscr{P}')$ but not by $S(\mathscr{P})$. However, $S(u)$ is split by $S(\mathscr{P})$ if and only if $L(u)$ is split by $\mathscr{P}$, and $S(u)$ is split by $S(\mathscr{P}')$ if and only if $L(u)$ is split by $\mathscr{P}'$. Thus, this call to **Newsplits** identifies each node u of T such that $L(u)$ is split by $\mathscr{P}'$ but not by $\mathscr{P}$. By applying such a call whenever a partition class Y is split into two sets Y_a and $Y - Y_a$ in **VertexPartition**, we may update a list of all nodes of the local tree that are split.

We will show with Lemma 5.21, below, that there is a way to select $x \in \{a,b,c,d\}$ in **Createtree** that guarantees that each node of G will correspond to at most two leaves of the local tree. Thus, $|S(Y_a)| = O(|Y_a|)$, so a call to **Newsplits**$(S(Y_a), T)$ takes $O(|Y_a| + k)$ time by Lemma 5.7. As before, we may ignore the cost of maintaining a list of currently split nodes of the local tree after each pivot operation.

Lemma 5.21. *A local tree where each member of V appears at most twice as a leaf may be constructed in linear time.*

Proof. We must give some additional details about the linear-time implementation of **Createtree** given in [31]. The algorithm processes each vertex of G once. Let $\{x_1,x_2,\ldots,x_n\}$ denote the order in which the vertices of G are processed. During this processing of G it maintains an evolving family of sets of vertices of G. Let $\mathscr{F}_{j-1}$ denote the state of the set family before vertex x_j is processed. $\mathscr{F}_j$ has the following property:

$(*)$ For each $X \in \mathscr{F}_j$, $G|(X \cap \{x_1,x_2,\ldots,x_j\})$ is a cograph.

The evolution of the set family tells how **Createtree** divides up the problem of computing a P_4 tree recursively. Each member of $\{x_j,x_{j+1},\ldots,x_n\}$ is in exactly one member of $\mathscr{F}_{j-1}$. As a base case, $\mathscr{F}_0 = \{V(G)\}$. For an inductive step, let Y be the member of $\mathscr{F}_{j-1}$ that contains x_j. If $G|(Y \cap \{x_1,x_2,\ldots,x_j\})$ is a cograph, then $\mathscr{F}_{j-1} = \mathscr{F}_j$. If $G|(Y \cap \{x_1,x_2,\ldots,x_j\})$ is not a cograph, then every P_4 that is contained in this subgraph must contain x_j because property $(*)$ on $\mathscr{F}_{j-1}$ implies that $G|(Y \cap \{x_1,x_2,\ldots,x_{j-1})$ is a cograph. Select such a P_4 to be $\{a,b,c,d\}$. it X_j. Compute the following partition of Y:

(1) $A := (N(b) - N(c) - N(d)) \cap Y$,
(2) $B := ((N(a) \cap N(c)) - N(d)) \cap Y$,
(3) $C := ((N(b) \cap N(d)) - N(a)) \cap Y$,
(4) $D := (N(c) - N(b) - N(a)) \cap Y$,
(5) $U := ((N(a) \cap N(b) \cap N(c) \cap N(d)) \cup (\overline{N}(a) \cap \overline{N}(b) \cap \overline{N}(c) \cap \overline{N}(d))) \cap Y$,
(6) $E := Y - A - B - C - D - U$,
$\mathscr{F}_j := (\mathscr{F}_{j-1} - Y) \cup \{A,B,C,D,E,U \cup \{x\}\}$.

Without loss of generality, suppose $x_j = a$. Since every member of A forms a P_4 with b,c, and d and $G|\{x_1,x_2,\ldots,x_{j-1}\}$ is a cograph, $\{x_1,x_2,\ldots,x_j\} \cap A = \{x_j\}$. From this observation and the fact that every P_4 in $G|(Y \cap (\{x_1,x_2,\ldots x_j\})$ contains x_j, it follows that property $(*)$ is maintained in $\mathscr{F}_j$.

We now add a small detail that ensures that each x_j corresponds to at most two leaves of the local tree. Each node corresponds to a number of leaves of the local tree that is given by one plus the number of times it was selected as x in all recursive incarnations of **Createtree** and hence of **Localtree**. If a P_4 is discovered when x_j is processed, then select x to be x_j, since x_j is a member of the P_4. Since x_j node is selected as x only when it is processed, it is selected at most once to be x during computation of the P_4 tree. □

5.6. Finding pivots with a local tree node

We will find it convenient again to label each node u with a *representative* $r \in L(u)$. The representatives will be used for pivot operations. We ensure that the degree sum of the representatives is $O(m)$ with the following procedure:

Procedure **Representatives**(u)

Input: The root u of a local tree T on undirected graph G.

Result: A labeling of each internal node of T with a 'representative' vertex of G.

For each child w of u do
  Call **Representatives**(w)
If $|U| = 1$ then let r be the sole member of $L(u)$
Else let r be a representative of children of u that has minimum degree
Assign r be the representative of u
If u is a P_4 node then
  Let a, b, c, d and A, B, C, D be as in the recursive call to **Localtree** that created u
  Reassign a, b, c, d as the representatives of the children corresponding to A,B,C,D respectively

Lemma 5.22. *Representatives runs in* $O(n + m)$ *time.*

Proof. Suppose that **Localtree** labels U with a, b, c, d and associates them with A, B, C, D when it creates U. Since only an S node may have a unique child and no S node is the parent of another, the result is immediate from Lemma 5.21, which implies that the number of nodes in the local tree is $O(n)$. □

Lemma 5.23. *Let T' be the local tree. For each node U of the local tree, let $r(U)$ denote the representative of U. $\sum_{U \in T'} deg(r(U)) = O(n + m)$, where $deg(x)$ denotes the degree of a vertex x in G.*

Proof. Consider the degree sum of representatives if the last `if` statement is omitted from **Representatives**. Let the rank of a node of the local tree be 1 if it is a leaf, and one plus the maximum of the ranks of its children otherwise. Since each internal node has at least two children, the degree sum of representatives of nodes of rank k is at most half the degree sum of representatives of nodes of rank $k - 1$. Since the degree sum of the leaves is $O(m)$ by Lemma 5.21, the bound on these representatives follows.

Createtree can be made to run in linear time [31] and in that implementation it examines the entire adjacency list of each of a, b, c, d during creation of a P_4 node. This gives an $O(n + m)$ bound on the degree sum of representatives assigned by the final *if* statement. □

Definition 5.24. A set X *coincides with the local tree* if it satisfies the following conditions:

1. For any S node S and its P_4 child P, either $X \cap P$ is empty, $X \cap P = P$, or $X \cap S$ is contained in a child of P.
2. For any cotree node W, $X \cap W$ is either empty, a union of children of W, or contained in a child of W.

A partition $\mathscr{P}$ of nodes of G coincides with the local tree if each member of $\mathscr{P}$ coincides with it.

Theorem 5.25. *$\mathscr{P}$ coincides with the local tree if and only if it is consistent with the corresponding P_4 tree.*

Proof. We proceed by induction on the height of a node in the recursion tree shared by **Createtree** and **Localtree**. If G is a cograph, then the P_4 tree and the local tree are identical, so the theorem follows in this case. Otherwise, let p and s be as in the main incarnation of **Localtree**, and let $P = L(p)$ and $S = L(s)$. Let P' be the node of the P_4 tree that corresponds to P. P' is created in the main incarnation of **Createtree**. Thus, $P = P'$ and $S = V(G)$, and the children of P in the local tree are the same sets as the children of $P' = P$ in the P_4 tree.

Let X be subset of $V(G)$ such that $X \cap P \neq P$, $X \cap P \neq \emptyset$, and X is not a subset of a child of P. X does not coincide with the local tree, and is it not consistent with the P_4 tree. The theorem is true in this case.

Suppose $X \cap P = \emptyset$ or $X \cap P = P$. The question of whether X is consistent with the P_4 tree reduces to the question of whether $X \cap (U \cup \{x\})$ is consistent with the tree returned by the recursive call to **Createtree**$(G|U \cup \{x\})$. The question of whether X is consistent with the local tree reduces to the question of whether $X \cap (U \cup \{x\})$ is consistent with the tree returned by the recursive call to **Localtree**$(G|U \cup \{x\})$. Adopting as the inductive hypothesis that the theorem holds for these two subtrees, it follows that the theorem holds for the trees returned by **Createtree**(G) and **Localtree**(G).

Suppose $X \cap S = X$ is contained in a child of P in the P_4 tree, hence in a child of $P' = P$ in the P_4 tree. Then the question of whether X is consistent with the P_4 tree reduces to the question of whether for each $Y \in \{A, B, C, D, E\}$, $X \cap Y$ is consistent with the recursive call to **Createtree**(Y). The question of whether X is consistent with the local tree reduces to the question of whether for each $Y \in \{A, B, C, D, E\}$, $X \cap Y$ is consistent with the recursive call to **Localtree**(Y). Adopting as the inductive hypothesis that the lemma is true for these these two subtrees, it follows that the theorem holds for the trees returned by **Createtree**(G) and **Localtree**(G). □

We now give the algorithm which performs a vertex partition, and does not halt until the partition coincides with the local tree. By Theorem 5.25, it follows that the algorithm is a restarting procedure on the P_4 tree. It maintains the invariant that **Splitlist** contains those nodes of the local tree that are split by $\mathscr{P}$ but that have not yet been processed, in addition to some nodes that have been processed. It assumes that the

nodes of the local tree that are split by the initial partition are correctly labeled as being split initially. The procedure maintains this invariant as it executes so that it continues to hold when it halts.

```
Procedure P4resolve()
While Splitlist is not empty
  Remove a node W from Splitlist
  If W is not marked Processed
    ProcessDescendants(W)

Procedure ProcessDescendants(W)
  Mark W as processed
  For each split child W' of W in the local tree that is not marked as
    processed do
    ProcessDescendants(W')
  Process(W)

Procedure Process(W)
  If W is a cotree node
    For each child W' of W
      Let w be the representative of W'
      Perform a universal pivot on w and update Splitlist with Newsplits
  Else if W is a P4 node
    For i := 1 to 4 do
      For W' = A,B,C,D,E do
        Let w be the representative of W'
        Perform a universal pivot on w and update Splitlist with Newsplits
```

An $O(n+m)$ bound on $O(n+m)$ calls to **P4resolve** is immediate from Lemma 5.23 and the fact that no node of the tree is processed more than once in all of the calls.

Lemma 5.26. *When* **Process** *is called on any P_4 or cotree node W in the local tree, the representatives of its children do not all lie in the same partition class.*

Proof. As a base case, suppose **Process** (W) occurred before **Process** (W') for each child W' of W. This implies that when **ProcessDescendants** is called on W, W has no split children. If W is split but has no split children, then it has children W_1 and W_2 that are subsets of different partition classes from $\mathscr{P}$. The representatives of W_1 and W_2 must thus lie in different partition classes, proving the lemma in this case.

Suppose W has a child W' such that **Process**(W') is called before **Process**(W). Adopt as an inductive hypothesis that the lemma is true for W'. For the lemma to fail at W, there must be a partition class Y that contains all representatives of children of W immediately before W' is processed. Since the lemma is true at W', Y fails to contain the representative z of some child Z of W'.

Case 1: *W is a cotree node*. W' must also a cotree node. W is a 1 node and W' is a 0 node or vice versa. Children of W' distinguish each other from all siblings of W'. Thus the representatives of children of W' distinguish each other from the representatives of siblings of W'. After the universal pivot on z that occurs when W' is processed, no representative of a child of W' is in the same partition class as any representative of a sibling of W', since the representatives of siblings of W' are members of Y and z is not a member of Y. Since one of the representatives of children of W' is the representative of W', this proves the lemma in this case.

Case 2: *W is a P_4 node*. Let $\{a,b,c,d\}$ and A,B,C,D,E be as defined in the incarnation of **Localtree** that created W. $W' \in \{A,B,C,D,E\}$, so any node of W' distinguishes a pair of members of $\{a,b,c,d\}$. Thus z distinguishes a pair of members of $\{a,b,c,d\}$. Since $\{a,b,c,d\} \subseteq Y$ and $z \notin Y$, $\{a,b,c,d\}$ do not all lie in the same partition class when **Process**(W) is called. This proves the lemma in this case. □

Theorem 5.27. P4resolve *is a restarting procedure on the P_4 tree.*

Proof. By Theorem5.25, it suffices to show that $\mathscr{P}$ coincides with the local tree. **ProcessDescendants** ensures that for each split local cotree node W, there has been a call to **Process**(W) by the time it halts.

Suppose W is a cotree node. Let the *rank* of a split cotree node in the local tree be 0 if it has no split child when P4resolve halts, or else 1 plus the maximum of the ranks of its children. If W is not split, no partition class violates condition 2 of Definition 5.24 with regard to W. If W has rank 0, none of its children is split and it follows again that no partition class violates condition 2 of Definition 5.24 with regard to W. Otherwise, let $i \geqslant 1$ be the rank of W and adopt as an inductive hypothesis that the lemma is true for split local cotree nodes of rank up to $i-1$. Assume for purposes of contradiction that X is a partition class that violates condition 2 of Definition 5.24, and thus that $X \cap W$ overlaps some child Y of W. By Lemma 5.26, when **Process**(Y) was called, no partition class contained all representatives of children of Y. After a pivot on each representative of children of Y when Y was processed, no partition class contained members of both Y and $W-Y$, contradicting the definition of X. We conclude that no class may violate condition 2 of Definition 5.24 with regard to any cotree node of any rank.

Suppose S is an S node and P is its P_4 child. If P is not split when the algorithm halts, then every partition class obeys condition 1 of Definition 5.24 with respect to S. Otherwise, let $\{a,b,c,d\}$ and A,B,C,D,E,U be as in the incarnation of **Localtree** that created S and P, and let e be the representative of E. For any partition class X that exists when **Process**(P) is called, $|X \cap \{a,b,c,d,e\}| = j \leqslant 4$ by Lemma 5.26. Suppose $j > 1$. Since $G|\{a,b,c,d,e\}$ is prime, performing a universal pivot on each of a,b,c,d,e splits X into a set $\mathscr{F}$ of partition classes such that for each $X' \in \mathscr{F}$, $|X' \cap \{a,b,c,d,e\}| \leqslant j-1$. Thus, repeating this sequence of pivots three times ensures that for each resulting partition class Z, $|Z \cap \{a,b,c,d,e\}| \leqslant 1$. By the definition of A,B,C,D,E and U, a final pivot on each of $\{a,b,c,d,e\}$ thus ensures that if $X_1, X_2 \in$

$\{A,B,C,D,E,U\}$, $X_1 \neq X_2$, $x_1 \in X_1$, and $x_2 \in X_2$, then $x1$ and x_2 are in different partition classes. It follows that for any partition class X' at this point, $X' \cap S$ is contained in one of A,B,C,D,E,U. Thus X' satisfies condition 1 of Definition 5.24 with respect to P. □

Acknowledgements

The authors would like to thank Jens Gustedt for extensive inputs to this paper.

References

[1] S. Booth, 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.
[2] B. Buer, R.H. Möhring, A fast algorithm for the decomposition of graphs and posets, Math. Oper. Res. 8 (1983) 170–184.
[3] M. Chein, M. Habib, M.C. Maurer, Partitive hypergraphs, Discrete Math. 37 (1981) 35–50.
[4] C.J. Colbourn, On testing isomorphism of permutation graphs, Networks 11 (1981) 13–21.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Algorithms, MIT Press, Cambridge, MA 1990.
[6] D.G. Corneil, H. Lerchs, L.S. Burlingham, Complement reducible graphs, Discrete Appl. Math. 3 (1981) 163–174.
[7] D.G. Corneil, Y. Perl, L.K. Stewart, A linear recognition algorithm for cographs, SIAM J. Comput. 3 (1985) 926–934.
[8] A. Cournier, M. Habib, An efficient algorithm to recognize prime undirected graphs, Technical Report R.R. LIRMM 92-023, Laboratoire D'Informatique, de Robotique et de Microelectronique de Montpellier, 1992.
[9] A. Cournier, M. Habib, A new linear algorithm for modular decomposition, in: S. Tison (Ed.), CAAP '94: 19th Internat. Coll. Edinburgh, UK, Lecture Notes in Computer Science, Springer, Berlin, 1994, pp 68–82.
[10] B. Duschnik, E.W. Miller, Partially ordered sets, Amer. J. Math. 63 (1941) 600–610.
[11] A. Ehrenfeucht, H.N. Gabow, R.M. McConnell, S.J. Sullivan, An O(n_2) divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs, J. Algorithms 16 (1994) 283–294.
[12] A. Ehrenfeucht, G. Rozenberg, Primitivity is hereditary for 2-structures, Theoret. Comput. Sci. 70 (1990) 343–358.
[13] A. Ehrenfeucht, G. Rozenberg, Theory of 2-structures, part 1: clans, basic subclasses, and morphisms, Theoret. Comput. Sci. 70 (1990) 277–303.
[14] A. Ehrenfeucht, G. Rozenberg, Theory of 2-structures, part 2: representations through labeled tree families, Theoret. Comput. Sci. 70 (1990) 305–342.
[15] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 209–221.
[16] T. Gallai, Transitiv orientierbare graphen, Acta Math. Acad. Sci. Hungar. 18 (1967) 25–66.
[17] M.C. Golumbic, The complexity of comparability graph recognition and coloring, J. Combin. Theory Ser. B 22 (1977) 68–90.
[18] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.
[19] A. Gouilà-Houri, Caracterisation des graphes non orientès dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre, C. R. Acad. Sci. Paris 254 (1962) 1370–1371.
[20] M. Habib, M.C. Maurer, On the X-join decomposition for undirected graphs, Discrete Appl. Math. 1 (1979) 201–207.
[21] W. Hsu, T. Ma, Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs, manuscript.

[22] R.M. McConnell, An O(n^2) incremental algorithm for modular decomposition of graphs and 2-structures, Algorithmica 14 (1995) 229–248.
[23] R.M. McConnell, J.P. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, 5th Ann. ACM-SIAM Symp. on Discrete Algorithms, Arlington, Virginia, 1994, pp 536–545.
[24] R.H. Möhring, Algorithmic aspects of comparability graphs and interval graphs, in: I. Rival (Ed.), Graphs and Orders, D. Reidel, Boston, 1985, 41–101.
[25] R.H. Möhring, Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and Boolean functions, Ann. Oper. Res. 4 (1985/6) 195–225.
[26] J.H. Muller, J. Spinrad, Incremental modular decomposition, J. ACM 36 (1989) 1–19.
[27] A. Pnueli, A. Lempel, S. Even, Transitive orientation of graphs and identification of permutation graphs, Canad. J. Math. 23 (1971) 160–175.
[28] D. Rotem, J. Urrutia, Circular permutation graphs, Networks 12 (1982) 429–437.
[29] J. Spinrad, J. Valdes, Recognition and Isomorphism of Two-Dimensional Partial Orders, Proc. 10th Collo. on Automata, Languages, and Programming, Lecture Notes in Computer Science, Springer, Berlin, 1983, 676–686.
[30] J.P. Spinrad, On comparability and permutation graphs, SIAM J. Comput. 14 (1985) 658–670.
[31] J.P. Spinrad, P_4 trees and substitution decomposition, Discrete Appl. Math. 39 (1992) 263–291.
[32] R. Sritharan, A linear time algorithm to recognize circular permutation graphs, Networks 27 (1996) 171–174.
[33] G. Steiner, Machine scheduling with precedence constraints, Ph.D. Thesis, University of Waterloo, Waterloo, Ont., 1982.
[34] J. Valdes, R.E. Tarjan, E.L. Lawler, The recognition of series–parallel digraphs, SIAM J. Comput. 11 (1982) 299–313.
[35] M. Yannakakis, The complexity of the partial order dimension problem, SIAM J. Algebraic Discrete Meth. 3 (1982) 303–322.