Discrete Mathematics and Theoretical Computer Science **4**, 2000, 045–060

Ordered Vertex Partitioning[†]

Ross M. McConnell[1] and Jeremy P. Spinrad[2]

[1]*Dept. of Computer Science and Engineering, University of Colorado at Denver, Denver, CO 80217-3364 USA*
[2]*Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235 USA*

received October 25, 1998, *revised August 2, 1999*, *accepted January 25, 2000*.

A transitive orientation of a graph is an orientation of the edges that produces a transitive digraph. The modular decomposition of a graph is a canonical representation of all of its modules. Finding a transitive orientation and finding the modular decomposition are in some sense dual problems. In this paper, we describe a simple $O(n+mlogn)$ algorithm that uses this duality to find both a transitive orientation and the modular decomposition. Though the running time is not optimal, this algorithm is much simpler than any previous algorithms that are not $\Omega(n^2)$. The best known time bounds for the problems are $O(n+m)$, but they involve sophisticated techniques.

Keywords: Modular Decomposition, Substitution Decomposition, Transitive Orientation

1 Motivation

Computing the *modular decomposition* of an undirected graph and a *transitive orientation*, when one exists, are problems that come up in a large number of combinatorial problems on perfect graphs and other graph classes. In this paper, we show how both problems reduce quite easily to a procedure called *vertex partitioning*. The resulting algorithm has an $O(n+m\log n)$ time bound, where n is the number of vertices, and m is the number of edges. Vertex partitioning is the only obstacle to a linear time bound. If no transitive orientation exists, the result still gives the modular decomposition. Neither of these results is optimal, since these problems can be solved in linear time [9], but the linear-time algorithms are quite involved and challenging to understand.

A reduction of modular decomposition and transitive orientation to vertex partitioning was first given in an unpublished work that was circulated in 1985 [13]. The simplified reduction we give here is a combination of ideas from that paper, and from [4]. It was circulated as an unpublished result in 1994. The linear time bound of transitive orientation of [9] was an outgrowth of it. Other subsequent papers have adopted its approach [7, 8]. In the final section, we discuss its somewhat weaker role of the approach in the parallel and sequential algorithms of [2, 3]. Because it is not the main focus of any of these papers, the simplicity of the basic approach has not been explained publicly. As we explain below, we believe that the underlying insights still hold promise for future progress in the area.

[†]This work has been made possible NSF Grant 98-20840.

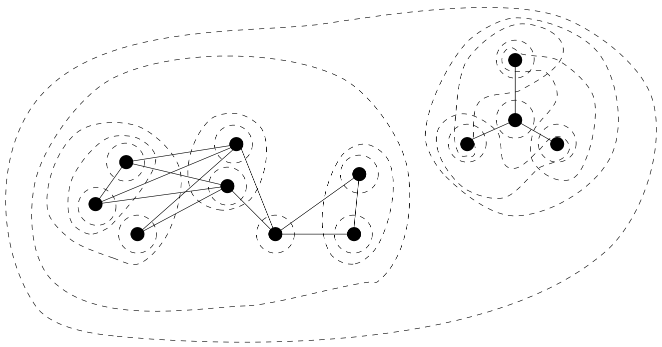

Fig. 1: A graph and its modules. A module is a set X of vertices such that for each $y \in V(G) - X$, either y is adjacent to every element of X, or y is nonadjacent to every element of X.

2 Introduction

We view an undirected graph as a special case of a directed graph, where each undirected edge (x,y) is composed of two directed arcs (x,y) and (y,x). A digraph is *transitive* if, removal of exactly one of (x,y) or (y,x) for each undirected edge (x,y). A *transitive orientation* is one where the resulting digraph is transitive. *Comparability graphs* are the class of graphs that have a transitive orientation.

A *module* of G is a set X of nodes such that for any node x not in X, either x is adjacent to every node of X, or x is nonadjacent to every node of X. $V(G)$ and its singleton subsets are the *trivial modules*. All graphs have the trivial modules; a graph is *prime* if it has no nontrivial modules. Figure 1 depicts a graph and its modules. The following is easily verified:

Theorem 2.1 *If X and Y are disjoint modules of a graph, then either every element of X is adjacent to every element of Y, or no element of X is adjacent to any element of Y.*

Thus, any pair of disjoint modules can be classified as "adjacent" or as "nonadjacent." It follows that if $\mathcal{P}$ is a partition of the nodes of G such that each member of $\mathcal{P}$ is a module, the adjacency relationships of the members of $\mathcal{P}$ to each other is itself described by a graph, as shown in Figure 2. This graph is called the *quotient* $G/\mathcal{P}$, and $\mathcal{P}$ is called a *congruence partition*. Note that if X is a set obtained by selecting one representative node from each member of $\mathcal{P}$, then $G|X$ is isomorphic to $G/\mathcal{P}$. Theorems about quotients can be applied to induced subgraphs of this type, a fact that we will occasionally use in our proofs.

The quotient $G/\mathcal{P}$ completely specifies those edges of the graph that are not in any subgraph $G|X$ induced by any $X \in \mathcal{P}$. Thus, the quotient, together with the subgraphs induced by the members of $\mathcal{P}$, gives a complete representation of the original graph.

The *modular decomposition* is a way to represent compactly all modules of a graph. Two modules X and Y *overlap* if they intersect, but neither contains the other. A *strong module* is a module that overlaps no other. The decomposition is a rooted tree. The nodes of this tree are the strong modules of the graph, and the transitive reduction of the containment relation on the strong modules gives the edges of the tree. By $MD(G)$, we denote the modular decomposition of G.

An equivalent definition of the modular decomposition is the following recursive one. Note that at least one of G and its complement is connected.

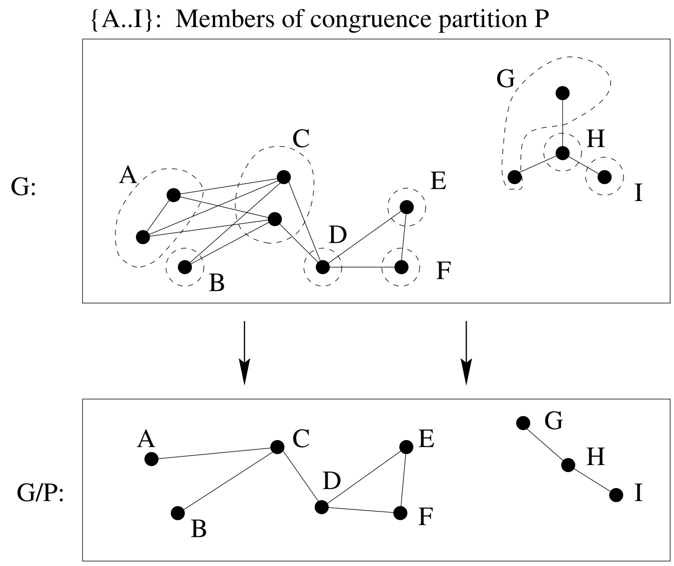

Fig. 2: A quotient on the graph of Figure 1. For any pair $\{X,Y\}$ of disjoint modules, either every element of $X \times Y$ is an edge or none is. Thus, X and Y may be viewed as *adjacent* or *nonadjacent*. If $\mathcal{P}$ is a partition of the vertices a graph into modules, the adjacencies of members of $\mathcal{P}$ can be described by a *quotient graph* $G/\mathcal{P}$. The quotient graph and the subgraphs induced by members of $\mathcal{P}$ completely specify G.

1. (parallel case) If G is disconnected, its connected components are a congruence partition, every union of components is a module, and no module overlaps a component. Thus, the modules of G can be divided into two sets: those that are a union of components and those that are a subset of a single component. Those that are a subset of a component C can be found by recursion on $G|C$.

2. (series case) If the complement of G is disconnected, apply step 1 to the complement.

3. (prime case) Otherwise, let the *highest submodules* be those modules that are not contained in any other module except $V(G)$. When both G and its complement are connected, the highest submodules are a congruence partition. The modules of G are $V(G)$ and those modules that are subsets of highest modules. Those that are a subset of a highest submodule M may be found by recursion on $G|M$.

At each step of the recursion, this algorithm finds a congruence partition. The recursion tree, together with the quotients induced by these congruence partitions, uniquely specify G. Figure 3 gives an example.

If X is a node of the tree and $\mathcal{Y}$ is its children, the *child quotient* is $(G|X)/\mathcal{Y}$. The name of the prime case of the modular decomposition is derived from the fact that if X is a prime node then its child quotient is prime.

The number of modules of a graph may be exponential, as in the case of a complete graph. However, it is easy to see from Theorem 3.1, below, that a set of vertices is a module of G if and only if it is a node of the decomposition tree or a union of children of a series or parallel node. Thus, the modular decomposition gives an implicit representation of the modules of G.

A directed graph is *transitive* if, whenever (a,b) and (b,c) are arcs, (a,c) is also an arc. An *orientation* of an undirected graph is a directed graph obtained by assigning a direction to each undirected edge. A graph is a *comparability graph* if there exists an orientation that is transitive. A transitive orientation can be found in $O(n+m)$ time [11], but this algorithm is difficult to understand. Cographs, and the complements of interval graphs are examples of comparability graphs. The fastest algorithm for recognizing permutation graphs takes advantage of the fact that a graph is a permutation graph if and only if it both the graph and its complement are comparability graphs. [11].

3 Strategy of the algorithm

It has long been recognized that there is a type of duality between the modules of a comparability graph and its transitive orientations [5]. In particular, the orientation of one edge in a transitive orientation dictates the orientation of another if and only if no module contains the endnodes of one of the two edges, but not of the other. Most approaches to transitive orientation compute the modular decomposition first. On the other hand, the modular decomposition can be constructed easily if it is known which edges force orientations of each other. The approach of our algorithm is to solve these dual problems concurrently.

The following two theorems, which are widely known, are fundamental [12]:

Theorem 3.1 *If X is a module of G, then the modules of G that are subsets of X are the modules of $G|X$.*

Theorem 3.2 *If $\mathcal{P}$ is a congruence partition on G, then $\mathcal{X}$ is a module of $G/\mathcal{P}$ if and only if $\bigcup \mathcal{X}$ is a module of G.*

Let G be an undirected graph and let v be a vertex. A *maximal module of G that does not contain v* is a module X such that X does not contain v, but for every module Y such that X is a proper subset of Y, Y contains v. Let $\mathcal{P}(G,v)$ be $\{v\}$ and the maximal modules of G that do not contain v.

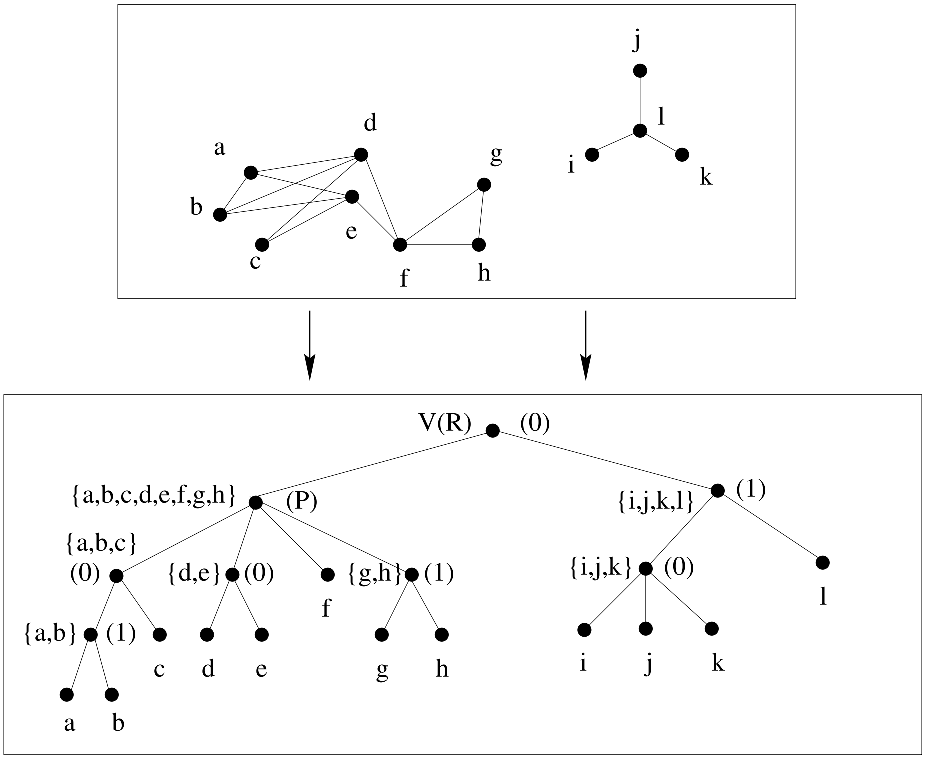

Fig. 3: The modular decomposition of the graph of Figure 1. The leaves are the vertices of the graph. The internal nodes are the strong modules, and their members are their leaf descendants. An internal node is a *parallel node* (labeled 0) if the set it denotes induces a disconnected graph, and its children are its connected components. It is a *series node* (labeled 1) if the complement of the graph it induces is a disconnected graph, and its children are the connected components of the complement. It is a *prime node* if it induces a graph that is connected and whose complement is connected; its children are the maximal modules it contains. A set of vertices is a module if and only if it is a node of the tree or a union of children of a single parallel or degenerate node.

Lemma 3.3 *A set is a member of $\mathcal{P}(G,v)$ if and only if one of the following applies:*

1. *It is $\{v\}$;*
2. *It is a child of a prime ancestor of $\{v\}$ in the decomposition tree that does not contain v;*
3. *It is the union of those children of a series or parallel ancestor of v that do not contain v.*

Proof: This is immediate from the fact that a set is a module if and only if it is a child of a prime node, or any union of one or more children of a degenerate node. □

Thus, $\mathcal{P}(G,v)$ is a partition of the vertices of G, hence a congruence partition.

The nodes of the decomposition tree are the strong modules. However, the containment relation on the nodes is still a tree if we add a weak module to the nodes. This module is a union of a set $\mathcal{X}$ of children of a series or parallel node; the containment relation dictates that it becomes a child of this node, and that the members of $\mathcal{X}$ become its children. If Y is the new node and Z is the parent, $G|Y$ is disconnected if and only if $G|Z$ is, and $\overline{G}|Y$ is disconnected if and only if $\overline{G}|Z$ is. Thus, Y is a parallel (series) node if and only if Z is. In this case, we say that Y is *superfluous*, since it is not needed to represent the modules of G. Adding superfluous nodes incrementally yields a tree with multiple superfluous nodes; this tree gives a type of modular decomposition, but it is not in *reduced form*. We will call such a tree an *unreduced modular decomposition* if it is *not necessarily* in reduced form. It has all the nodes of the modular decomposition, in addition to a (possibly empty) set of superfluous nodes. Given an unreduced decomposition, it is trivial to derive the reduced modular decomposition, by deleting series (parallel) nodes that are children of series (parallel) nodes in postorder. Thus, in the remainder of the paper, we consider only the problem of how to produce an unreduced decomposition.

Let $\mathcal{P}$ be a congruence partition. Let $\mathcal{T}'$ be a modular decomposition of $G/\mathcal{P}$ that is not necessarily in reduced form, and let $\mathcal{T}_X$ denote a modular decomposition of $G|X$ for $X \in \mathcal{P}$ that is not necessarily in reduced form. Each $X \in \mathcal{P}$ is a leaf of $\mathcal{T}'$. $\mathcal{T}'$ gives all modules of G that are unions of members of $\mathcal{P}$, while each $\mathcal{T}_X$ gives the modules of G that are contained in a member of $\mathcal{P}$. We can obtain a single tree that has all of this information by performing the *composition* of this trees. This is obtained by visiting each leaf $\{X\}$ of $\mathcal{T}'$ and attaching $\mathcal{T}_X$ as a subtree. To be precise, the composition of $\mathcal{T}'$ and $\{\mathcal{T}_X : X \in \mathcal{P}\}$ is given by $\{\bigcup Y : Y \in \mathcal{T}'\} \cup \{\mathcal{T}_X : X \in \mathcal{P}\}$.

Lemma 3.4 *Let $\mathcal{P}$ be a congruence partition on an arbitrary graph G, let $\mathcal{T}'$ be an unreduced modular decomposition of $G/\mathcal{P}$, and let $\mathcal{T}_X$ be an unreduced modular decomposition of $G|X$ for $X \in \mathcal{P}$. Then the composition of $\mathcal{T}'$ and $\{\mathcal{T}_X : X \in \mathcal{P}\}$ is an unreduced modular decomposition of G.*

Proof: If M is a strong module, then it overlaps no member of $\mathcal{P}$, by definition. Thus, M is either contained in a member of $\mathcal{P}$ or is a union of members of $\mathcal{P}$. If it is contained in a member X of $\mathcal{P}$, then it is a strong module of $G|X$, by Theorem 3.1, so it appears as a node of $\mathcal{T}_X$, hence as a node of the composition. If M is a union of a set $\mathcal{M} \subseteq \mathcal{P}$, then $\mathcal{M}$ is a strong module of $G/\mathcal{P}$, by Theorem 3.2, and appears as a node of $\mathcal{T}'$. It follows that M appears as a node of the composition. Thus, every strong module of G is a node of the composition, from which the result follows. □

Our strategy, which is summarized by Algorithm 1, is to find an unreduced decomposition of $G/\mathcal{P}(G,v)$, find an unreduced decomposition of $G|X$ for each $X \in \mathcal{P}(G,v)$, and return the composition of these trees. The vertices may be numbered arbitrarily with distinct integers.

Algorithm 1: UMD(G) Strategy for computing modular decomposition.

Let v be the lowest-numbered vertex of G;
if *G has only one vertex* **then return** $\{\{v\}\}$;
else
  Let $\mathcal{T}'$ be the modular decomposition of $G/\mathcal{P}(G,v)$;
  foreach *member Y of $\mathcal{P}(G,v)$* **do** $\mathcal{T}_Y := UMD(G|Y)$;
  return the composition of $\mathcal{T}'$ and $\{\mathcal{T}_Y : Y \in \mathcal{P}(G,v)\}$.

The correctness is immediate from Lemma 3.4. This high-level strategy was employed by the algorithm of [4]. However that algorithm had an $O(n^2)$ bound, and for some time it was not clear how to improve the bound.

4 Data structures

We represent a linear order on a set V by labeling the elements of V with unique integer keys, which need not be consecutive. Sorting V in ascending order of these labels gives the represented order. The *restriction* of π to $X \subseteq V$, denoted $\pi|X$, is just a sort of X in ascending order of these labels.

Let a *module tree* be any set of modules where the transitive reduction of the containment relation is a tree, and where the root is V and the leaves are its singleton subsets. The canonical modular decomposition is one example among many. The module tree is *ordered* if there is a left-to-right order on the children of each internal node. If $X \subseteq V(G)$, let the *restriction* of a module tree T to X, denoted $T|X$, have as its nodes the set $\{Y \cap X : Y \text{ is a node of } T\}$. The transitive reduction of the containment relation gives the parent relation. If T is an ordered module tree, then the restriction to X, is also ordered; this is done in the unique way that its leaves $\{\{x\} : x \in X\}$ have the same relative order that they do in T. This is always possible, since any node of T is a set that is consecutive on the leaf order of T, hence any node of $T|X$ is also consecutive in its leaf order.

To implement a module tree, we let an $O(1)$-size node stand for each node. This node carries a doubly-linked list of pointers to its children. If the node is a leaf, it carries a pointer to the corresponding vertex of G. The set corresponding to an internal node is then just given by its leaf descendants.

Using this representation, we obtain the following:

Lemma 4.1 *The composition step of Algorithm 1 takes $O(|\mathcal{P}(G,v)|)$ time.*

Proof: The composition is obtained by replacing each leaf $\{Y\}$ of the decomposition of $G/\mathcal{P}(G,v)$ with the tree $\mathcal{T}_Y$. This requires one pointer operation. The decomposition of $G/\mathcal{P}(G,v)$ has one leaf for each member of $\mathcal{P}(G,v)$. □

Bucket sorting a list of m items by integer keys in the range from 1 to n takes $O(n+m)$ time, even if an initialized set of buckets is provided. However, if one only wishes to group the items into groups that have identical keys, the operation may be carried out in $O(m)$ time if the initialized buckets are already available. While inserting the items to buckets, one must simply maintain a list of nonempty buckets. When all items are inserted, the list may be used to visit only the nonempty buckets, retrieving one group at each of them. The operation takes only $O(m)$ time. To distinguish this operation from bucket sorting, we will call it *bucket grouping*. Similarly, if a list of m items must be sorted by a pair of keys in the range from 1 to n, this takes $O(n+m)$ time by a radix sort that makes one call to bucket sort for each of the keys [1]. If one wishes only to partition the items into groups that share identical pairs of keys, the calls to bucket sort may be replaced by bucket groupings, and the partition takes $O(m)$ time. We will call this operation *radix grouping*.

The data structure for the graph is an adjacency-list representation, where the vertices are numbered from 1 to n, and where the adjacency lists are doubly-linked lists. The adjacency list associated with vertex x has each arc of the form (x,y). Each such arc (x,y), in turn, has a pointer to the occurrence of its "twin" (y,x) in the adjacency list of y. We also keep a partition $\mathcal{P}$ of the vertices. Each partition class of $\mathcal{P}$ is implemented with a doubly-linked list, and each vertex has a pointer to its occurrence in one of these lists, as well as a pointer to the beginning of this list.

5 The Gamma relation

An undirected graph may be viewed as a special case of a (symmetric) digraph, where each edge (x,y) is represented by two arcs, (x,y) and (y,x). An orientation of the graph is a choice of one arc from each such pair.

Let Γ be a relation on the arcs of a digraph, where $(u,w)\Gamma(x,y)$ if and only if $u=x$ and w and y are nonadjacent, or $w=y$ and u and x are nonadjacent [5, 6, 12]. Let Γ^* be the transitive reflexive closure of Γ. It is easy to see that $(u,w)\Gamma(x,y)$ in a comparability graph, then in each transitive orientation, (u,w) and (x,y) are either both present or both absent. Transitively, it follows that $(u,w)\Gamma^*(x,y)$ implies this also. Since Γ^* is an equivalence relation, this partitions the edges into groups such that for each group, either every member of the group is included in a transitive M orientation or none is. These groups are the *implication classes*.

If I is a set of arcs, let I^{-1} denote the class $\{(y,x):(x,y)\in I\}$. By symmetry, whenever I is an implication class in an arbitrary graph, I^{-1} is also an implication class.

It is easy to see that, in any undirected graph, either $I^{-1}=I$ or I^{-1} is disjoint from I. Let a *color class* be $I\cup I^{-1}$ for an implication class I. The color classes are a partition of the undirected edges of G. A graph is a comparability graph only if every implication class I is disjoint from I^{-1}; otherwise no orientation of the graph can contain the implication class. In fact, this characterizes comparability graphs: an undirected graph is a comparability graph if and only if for each implication class I, I is disjoint from I^{-1} [6]. We also make use of the following well-known result, which can be found in [12]:

Theorem 5.1 *If all edges of G are in a single color class, then all nontrivial modules of G are independent sets.*

6 Computing $G/\mathcal{P}(G,v)$ and its modular decomposition

We now describe a procedure called *vertex partitioning*, which we use to compute $\mathcal{P}(G,v)$ in Algorithm 1. The input to the procedure is a partition of the vertices, and it finds a refinement of the partition that gives the maximal modules of G that are subsets of one of the original partition classes.

The most basic operation in vertex partitioning is the *pivot*. An input is a partition $\mathcal{P}$ of the vertices of the graph, a vertex x, and the edges from x to some subset $\mathcal{Q}$ of members of $\mathcal{P}$. The operation produces a refinement $\mathcal{P}'$ such that every module that is a subset of a class in $\mathcal{P}$ is also a subset of a class of $\mathcal{P}'$, and, in addition, any class of $\mathcal{P}'$ that is a subset of a member of $\mathcal{Q}$ consists only of neighbors or only of non-neighbors of x.

To implement a pivot, let X be the class of $\mathcal{P}$ that contains x. For each neighbor y of x in a member of Q, identify the set $Y \in \mathcal{P}$ that contains y, and move y to a twin list Y' for Y. Start a new list for Y' if Y doesn't already have a twin list. In the process, keep a list of the members of $\mathcal{P}$ in which this happens. When this is finished, Y contains only non-neighbors of x, and its twin Y' contains only neighbors. Establish Y and Y' as classes in the refinement $\mathcal{P}'$. If there are k edges from x to members of Q, this operation takes $O(1+k)$ time.

We now describe what we will call the *split* operation, which we will denote $Split(X,\mathcal{P})$. Let $\mathcal{P}$ be a partition of the vertices, and let $X \in \mathcal{P}$. Given the arcs E' that have one end in X and the other in $V-X$, we may use them to subdivide X and the members of $\mathcal{P}-\{X\}$ to obtain a refinement $\mathcal{P}'$ of $\mathcal{P}$. This refinement has the property that for any $x \in X$, each class Y of $\mathcal{P}'$ contained in $V(G)-X$ consists only of neighbors or only of non-neighbors of X, and for any $z \in V(G)-X$, each class Z of $\mathcal{P}'$ contained in X consists only of neighbors or only of non-neighbors of z.

To implement the split operation, we bucket group all arcs (x,y) in E' by starting vertex x. Each group gives the neighbors of some $x \in X$ in $V(G)-X$. We perform a pivot on x to split up those members of $\mathcal{P}$ that are subsets of $V(G)-X$. This refines $\mathcal{P}$. Pivots on the other groups give further successive refinements of $\mathcal{P}$. After this has been done, each class of $\mathcal{P}$ that is a subset of $V(G)-X$ has the required properties. We then use the twin pointers of arcs in E' to find the arcs in $(V(G)-X)\times X$, and perform the foregoing steps on them to refine X. When we are done, each class of $\mathcal{P}$ that is a subset of X has the desired properties. The procedure returns the refined partition, listing those classes that are subsets of X separtely from those that are not. If the arcs $E' = E \cap (X \times (V(G)-X))$ are given, the operation takes $O(1+|E'|)$. If they are not given, it takes $O(|X|+deg(X))$, where $deg(X)$ is the sum of degrees of members of X, since the adjacency lists of members of X must be searched to find the members of E'.

The vertex partitioning algorithm is given by Algorithm 2.

Algorithm 2: $Partition(G,\mathcal{P})$, Vertex partitioning.

```
if |P| = 1 then return P;
else
    Let X be a member of P that is not larger than all others;
    Let E be the edges of G in X × (V(G) − X);
    P' = Split(X, P);
    G := G − E;
    Let Q be the classes of P' contained in X;
    Let Q' be the classes of P' contained in V(G) − X;
    return Partition(G|X, Q) ∪ Partition(G − X, Q');
```

To prove the correctness, we show that every member of the returned partition is a module. Since refinements are caused by pivots and no pivot can split up a module, it follows that the returned partition consists of the maximal modules of G that are subsets of a class in the initial partition. As a base case, if $|\mathcal{P}| = 1$, its sole member, $V(G)$, is trivially a module in G. Otherwise, assume by induction that the first recursive call returns a partition of X whose partition classes are modules in $G|X$ and subsets of classes in Q. If Y is one of the returned partition classes, every node of $X-Y$ is either a neighbor of every element of Y or a non-neighbor of every element of Y. If Z is the member of Q that contains Y, then because of the call to $Split$, each element of $V(G)-X$ is either a neighbor of every element of Z or a non-neighbor of every element of Z, hence the same can be said about Y. Thus, Y is a module in G. A symmetric argument shows that each partition class returned by the second recursive call is a module.

For the time bound, note that the call to *Split* takes $O(|X| + deg(X))$ time, since the edges from X to $V(G) - X$ are not supplied. We charge the cost of this by assessing $O(1)$ to each member of X and each edge incident to X. Since $|X| \leq |V(G)|/2$, a vertex and its edges are charged only if its partition class has at most half of the vertices passed by the call to the procedure. Since this is also true in the recursive calls, the next time $x \in X$ and its incident edges are charged charged in a recursive call, the partition class that contains it will have size at most $|X|/2$. We conclude that no element of the graph is charged more than $\log_2 n$ times, which gives an $O((n+m)\log n)$ time bound. An $O((n+m)\log n)$ bound can be reduced to an $O(n + m\log n)$ bound by spending $O(n+m)$ preprocessing time finding the connected components of G, applying the algorithm on each component, and combining the results in a trivial way, using the fact that any union of components is a module.

6.1 Ordered vertex partition

Recall that a graph is prime if it has only trivial modules. We may transitively orient prime comparability graphs and recognize arbitrary prime graphs by revising the above approach so that it keeps the partition classes in an ordered list as they are refined. Algorithm 3 gives an important subroutine, which appeared in [10].

Algorithm 3: $OVP(G, \mathcal{P})$

Run Algorithm 2, with the following changes. A linear order on the members of $\mathcal{P}$ is given initially. Maintain a linear order on $\mathcal{P}$ as it is refined, using the following rule: When a pivot on a vertex $p \in Y \in \mathcal{P}$ splits $X \in \mathcal{P}$ into a set X_a of adjacent vertices and a set X_n of nonadjacent vertices, make X_a and X_n consecutive at the former position of X in the linear order, with X_n the nearer of the two to Y in the linear order. Return the final partition and its linear order.

The significance of this ordering will soon become clear. The initial parameter $\mathcal{P}$ will always be $(\{v\}, V - \{v\})$ for a vertex v, which we call the *seed*. To facilitate the computation of the ordering, we maintain labels on the partition classes so that a class Y is labeled with (i, j), where i is one plus the sum of cardinalities of classes that precede Y and j is one less than the sum of cardinalities of classes that follow; these labels are trivial to update during a pivot, and they allow one to find out quickly whether the pivot's class precedes or follows Y. This allows us to find the correct relative order for X_a and X_n in $O(1)$ time, so the time bound for OVP is the same as that for $Partition$, $O(n + m\log n)$.

Algorithm 4: Compute $G/\mathcal{P}(G, v)$.

Algorithm 2, hence Algorithm 3 explicitly removes those edges that are not contained in a member of $\mathcal{P}(G, v)$. Let m' be the number of removed edges. The data structures of Section 4 for implementing a partition allow us to find which partition class a vertex belongs to in $O(1)$ time. Radix group the removed edges according to the two partition classes. This takes $O(m')$ time. Discard all but one *representative* edge in each group; the representative edges give the edges of $G/\mathcal{P}(G, v)$. The entire operation takes $O(m')$ time, in addition to the time required by the call to OVP.

Let the edges *exposed* by a partition of vertices be those edges that are not contained in a single partition class. Suppose that $\mathcal{P}$ is ordered so that all arcs that go from earlier to later members of $\mathcal{P}$ are in a single implication class. In this case, we say that the ordering on $\mathcal{P}$ is *consistent* with a transitive orientation of G. Using this as an inductive hypothesis, it is easy to verify that when a class is split according to the above ordering rules, the refinement of $\mathcal{P}$ and its ordering is also consistent with a transitive orientation: if a pivot p splits X into a set X_a of adjacent vertices, and X_n of nonadjacent vertices, then all members of $\{p\} \times X_a$ are in one implication class, and these force all members of $X_n \times X_a$ that are edges into this implication class. The inductive hypothesis continues to hold after the class splits. It follows that if the initial ordering of $\mathcal{P}$ is consistent with a transitive orientation, then so is the final ordering. Thus, if G is prime and v is a source vertex in a transitive orientation, then calling the algorithm with initial ordered partition $(\{v\}, V(G) - \{v\})$ gives a linear extension of the transitive orientation [10].

The problem of transitively orienting a prime comparability graph thus reduces to identifying a source vertex in a transitive orientation. Suppose $\mathcal{P}$ is ordered so that for each vertex x in the last class X, all edges of $\{x\} \times V(G) - X$ are in one implication class. Then if a an argument similar to the one above shows that if a pivot p splits X into X_a and X_n, all edges of $\{x\} \times (V(G) - X_a)$ are in one implication class. By induction, if the ordered partitioning algorithm is called with initial ordered partition $(\{v\}, V(G) - \{v\})$ for arbitrary v, the last partition class in the final ordering is a source in a transitive orientation.

Algorithm 5 summarizes the procedure for finding a transitive orientation of a prime comparability graph [10]:

Algorithm 5: TO(G) Transitive orientation of a prime comparability graph.

Let v be an arbitrary vertex ;
$\mathcal{P} := OVP(G, (\{v\}, V(G) - \{v\}))$;
Let $\{w\}$ be the rightmost class in $\mathcal{P}$;
return $OVP(G, (\{w\}, V(G) - \{w\}))$.

We now show that the same algorithm recognizes whether an arbitrary graph is prime [10]. Performing these two vertex partitions without encountering a nontrivial module proves that all edges of the graph are in a single color class. By Theorem 5.1, this implies that any nontrivial modules are independent sets. The first vertex partition finds a nontrivial module if one is a subset of $V(G) - v$. The second finds one if one is a subset of $V(G) - w$. Thus, any nontrivial modules contain both v and w. However, w is adjacent to v, since it was rightmost in the final partition, hence was among the neighbors of v, which were moved to the right when the pivot on v split $V - \{v\}$. Thus, any nontrivial module is an independent set, and any nontrivial module contains the edges (w, v). There can be no nontrivial module.

6.2 Finding the modular decomposition of $G/\mathcal{P}(G, v)$

We have shown how to compute $\mathcal{P}(G, v)$ and to compute the composition of unreduced modular decomposition trees. The only remaining problem in Algorithm 1 is computing the modular decomposition of $G/\mathcal{P}(G, v)$.

Let a graph G be *nested* if all nontrivial modules are ancestors of a particular vertex v in $MD(G)$. We call v and its siblings the *innermost* vertices.

Lemma 6.1 *For any undirected graph G and any vertex v, $G/\mathcal{P}(G, v)$ is nested, and $\{v\}$ is an innermost vertex in it.*

Proof: If $\mathcal{M}$ is a nontrivial module of $G/\mathcal{P}(G, v)$ that does not contain $\{v\}$, then $\bigcup \mathcal{M}$ is a nontrivial module of G that does not contain v, by Theorem 3.2. No member of $\mathcal{M}$ is a maximal module of G not containing v, which contradicts the definition of $\mathcal{P}$. If $\mathcal{A}$ and $\mathcal{B}$ are overlapping modules of $G/\mathcal{P}(G,v)$, then either $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{A}\Delta\mathcal{B}$ is a nontrivial module of $G/\mathcal{P}(G,v)$ that does not contain $\{v\}$, which we have just seen cannot happen. Thus, all modules of $G/\mathcal{P}$ are strong and contain $\{v\}$, from which the result follows. □

Thus, the remaining step of Algorithm 1 reduces to computing the modular decomposition of nested graphs.

Algorithm 5 simultaneously recognizes prime graphs and transitively orients them if they are comparability graphs. Recognizing a prime graph can be viewed as a special case of finding modular decomposition, which works only if the graph is prime. We generalize it so that it simultaneously computes the modular decomposition of nested graphs, and finds a transitive orientation when they are comparability graphs.

We use a variant of *OVP*, which is given by Algorithm 6. This procedure applies itself recursively inside each of the modules found by *OVP*. The vertices of G are assumed to be numbered; this determines selection of the seed vertices in calls to *OVP*.

Algorithm 6: *ROP*(G), recursive application of OVP (Algorithm 3)

if *G has an isolated vertex* **then** let w be that vertex;
else let w be the highest numbered vertex;
if *G has only one vertex* **then** return $\{w\}$;
else
  $(X_1, X_2, ..., X_k) := OVP(G, (\{w\}, V(G) - \{w\}))$;
  Let T be a module tree with one node $V(G)$;
  foreach *set X_i* **do** Let $ROP(G|X_i)$ be the i^{th} child of $V(G)$;
  return T

Lemma 6.2 *The ROP algorithm takes $O((n+m)logn)$ time.*

Proof: Using the data structure of Section 4, it takes $O(1)$ time to attach each child to $V(G)$. Since the final tree has n leaves and every internal node has at least two children, this step takes has $O(n)$, all steps except the calls to *OVP* contribute $O(n)$ to the running time of *ROP*. The running time for the calls to *OVP* is bounded by the time spend on pivots in Algorithm 2 inside these calls. In this procedure, no pivot on a vertex x occurs unless it is in a class that is at most half as large as the class that contained it the last time a pivot was performed on x in Algorithm 2 *and in a class that is half as large as the class that contained it at the beginning of the call to Algorithm 2*. From this last observation, we may conclude that whenever a pivot is performed on v, it is in a class that is half as large as the class that contained it the previous time a pivot occurred on it in any call to *OVP* generated by *ROP*. Since the cost of the pivot is $O(1+d(v))$ the total cost of all pivots in all calls to *OVP* is $O((n+m)\log n)$. □

Corollary 6.3 *The total time spent in UMD (Algorithm 1) by calls it makes to OVP is $O((n+m)\log n)$.*

Proof: These are the same as the calls to *OVP* generated by a call to *ROP*. □

Algorithm 7 finds the modular decomposition in nested graphs, if an innermost vertex v is given.

We now demonstrate the correctness. Since all modules of G contain v, the call to $OVP(G, (\{v\}, V(G) - \{v\})$ simply assigns an ordering to the vertices of G via the ordering of the returned singleton sets. This

Algorithm 7: *Chain*(G,v), Modular decomposition in a nested graph.

$\mathcal{P} := OVP(G,(\{v\},V(G)-\{v\})$;
Number the vertices of G in order of their appearance in $\mathcal{P}$;
return $ROP(G)$.

ordering is not unique, since it depends on choices of pivots during execution of the algorithm. However, not all permutations are possible. Let us say that an ordering is a *valid* result for the call if there is a sequence of pivot choices that causes it to produce that ordering.

Lemma 6.4 *If all nontrivial modules of G contain vertex v, X is a module that contains v, and* π *is the ordering of vertices produced by a call to* $OVP(G,(\{v\},V(G)-\{v\}))$*, then* $\pi|X$ *is a valid result of a call to* $OVP(G|X,(\{v\},X-\{v\}))$.

Proof: A vertex $y \in V(G)-X$ is adjacent to all members of X or nonadjacent to all of them. Thus, during a call to $OVP(G,(\{v\},V(G)-\{v\}))$, if two members of X are in a single partition class before a pivot on y, they remain in a single class after the pivot, so the pivot on y has no effect on their relative order. The final ordering of X is determined exclusively by pivots on members of X and their adjacencies in $G|X$. The production of this ordering can thus be simulated by a call to $OVP(G|X,(\{v\},X-\{v\}))$. □

In the modular decomposition of a nested graph, each internal node of the decomposition tree has at most one non-singleton child, namely, the one that contains v. Also, a series or parallel node has exactly two children; otherwise the union of two that do not contain v results in a nontrivial module that does not contain v. Let w' be the seed vertex selected to fill the role of w in the main call to *ROP* that is generated in *Chain*. If the root of the tree is a parallel node, then $V(G)$ has two children, one of which is an isolated node, and w' is selected to be this node. If the root is a series node, then $V(G)$ has two children, one of which is an isolated node in the complement of G. The ordering rule in the call to *OVP* ensures that this node will receive the highest number of any node, and w' is selected to be this node. Otherwise, the rightmost class in the numbering produced by *OVP* is a singleton class $\{w\}$. As in the proof of Algorithm 5, all edges out of $\{w\}$ are in the same implication class, and w is adjacent to v. Thus, all nontrivial modules that contain w contain v. It follows that $\{w\}$ is again a child of the root, since all nontrivial modules contain v. The call to $OVP(G,(\{w'\},V(G)-\{w'\}))$ generated in this highest-level call to *ROP* then finds the maximal modules that do not contain w', and since $\{w'\}$ is a child of the root, this call to *OVP* just returns the children of the root.

We now show that the recursive call generated inside *ROP* finds the remainder of the tree. Algorithm 7 obviously works correctly when G has only one vertex. Adopt as an inductive hypothesis that G is a nested graph with n vertices and the algorithm works on nested graphs graphs with fewer vertices. Let X be the non-singleton child of $V(G)$, the one that contains $\{v\}$. By Theorem 3.1, $G|X$ is nested with v as an innermost vertex, so a call to *Chain* on $G|X$ could be used to compute the rest of the tree. Such a call would first call *OVP* to assign an ordering to X, and then call *ROP* on $G|X$. By Lemma 6.4, the vertex numbering of X is already available in the numbering of vertices of X, so the call to *ROP* on $G|X$ returns exactly what such call to *Chain* on $G|X$ would. By the inductive hypothesis, this is the remainder of the modular decomposition of G, namely, the subtree rooted at X.

Lemma 6.5 *Chain can be run in* $O((n+m)\log n)$ *time.*

Proof: This is immediate from the fact that *OVP* and *ROP* both have this time bound. □

7 Efficient implementation of Algorithm 1

The following gives an efficient implementation of Algorithm 1.

Algorithm 8: Final modular decomposition algorithm.

Implement $UMD(G)$ as follows:
Compute $\mathcal{P}(G,v)$ with a call to Algorithm 3;
Compute $G/\mathcal{P}(G,v)$ with a call to Algorithm 4;
Compute the modular decomposition of $G/\mathcal{P}(G,v)$ with a call to Algorithm 7.

Lemma 7.1 *Algorithm 8 takes $O(n+m\log n)$.*

Proof: The set of instances of Step 1 over all recursive calls to UMD is just the set of calls to OVP generated by a single call to ROP. Since that algorithm is $O(n+m\log n)$, Step 1 contributes $O(n+m\log n)$ to the running time of UMD.

By Lemma 4, computing $G/\mathcal{P}(G,v)$ in the main call takes $O(1)$ time for each edge of G not contained in a member of $\mathcal{P}(G,v)$. We charge this cost to these edges. The edges charged in recursive calls are disjoint from these, so the total contributed by step 2 to the running time of UMD is $O(m)$.

A call to Algorithm 7 on a nested graph with n' vertices and m' edges takes $O((n'+m')\log n')$ time. $G/\mathcal{P}(G,v)$ has at most one edge for every edge of G not contained in a member of $\mathcal{P}(G,v)$. We may charge the cost of this to $O(\log n)$ *set costs* for each member of $\mathcal{P}(G,v)$, and $O(\log n)$ *edge costs* for each edge not contained in one of its members. The edges charged are disjoint from those charged in recursive calls, so Step 3 contributes $O(m\log n)$ edge charges. The containment relation on sets incurring a set cost in some recursive call of UMD is a tree, so there are $O(n)$ of them. The total contributed by Step 3 to the running time of UMD is $O((n+m)\log n)$.

Computing the composition of the trees takes $O(1)$ for each member of $\mathcal{P}(G,v)$, by Lemma 4.1. This contributes an additional $O(1)$ cost to each set incurring a set cost in Step 3. Computing compositions of trees thus contributes $O(n)$ to the running time of the algorithm.

The total running time is $O((n+m)\log n)$. However, this bound for any modular decomposition algorithm can be reduced to an $O(n+m\log n)$ bound by spending $O(n+m)$ preprocessing time to find the connected components, and if there are $k>1$ of them applying the algorithm separately to each component, and making the resulting k trees children of $V(G)$. □

8 Conclusions

Another closely-related variant of this algorithm is to perform ordered vertex partitioning, and then to perform it recursively inside each nontrivial module that is found. The final result is an ordering of the vertices. A linear extension of a transitive orientation is obtained by performing a second pass of this recursive algorithm, except that in each recursive call on a set X of vertices, the initial pivot vertex must be the rightmost member of X in the ordering produced by the first pass. Every strong module of G is identified as a module in one of the two passes. Every module that is found in one of the passes and is not strong is found to overlap a module found in the other pass. It is therefore a simple matter to obtain the modular decomposition by discarding those modules from the two passes that overlap. The proof of correctness is quite similar to the one we give above.

Another promising idea is to try to obtain the result of the vertex partitioning step without actually performing vertex partitioning. This is the approach of the parallel and sequential algorithms of [2, 3]. This elegant strategy, which is due to Dahlhaus, makes recursive calls on subgraph induced by the neighbors of v and the subgraph induced by the non-neighbors of v, and then deletes portions of these two trees to obtain the members of $\mathcal{P}(G,v)$ and their modular decompositions. However, up until now, turning this idea into an $O(n+m\alpha(m,n))$ algorithm requires conceptually difficult tricks, such as letting some parent pointers in the tree get out of date, and maintaining a union-find data structure to help simulate them. It also requires careful charging arguments to prove the time bound. The linear-time variant uses sophisticated union-find data structures.

One of the important points illustrated by the algorithm is that finding a simple linear-time vertex partitioning algorithm would give simple linear-time solutions to modular decomposition and transitive orientation. There is reason to hope that one might exist. In the first place, the basis of the difficult transitive orientation algorithm of [11] is a (difficult) linear-time implementation of Algorithm 6. Thus, there is no inherent barrier to a linear time bound. It is also worth noting that the algorithm we give here is actually linear on many classes of graphs. For instance, on graphs with fixed degree bound k, it is easy to verify that, at any point during a call to $OVP(G,\{v\},V(G-\{v\})$, only one partition class can have size greater than c. The proof of this is by induction on the number of pivots performed so far. In the proof of *ROP*, we saw that each time a pivot occurs on a vertex, it is in a class that is half as large as the class that contained it the previous time it was a pivot. This gives an $O((n+m)\log k) = O(n+m)$ bound for *ROP*. Since *ROP* is the bottleneck for the time bound of Algorithm 8, this gives a linear time bound in this case.

A similar argument shows that on graphs where the ratio of the maximum degree k to the minimum degree k' is at most c, the time bound is linear. Suppose k' is the minimum degree. Let us say a vertex is *confined* if it is in a class of size at most $k/2$, and a pivot is confined if the pivot vertex is confined at the time. Consider a confined pivot on a vertex p. The OVP algorithm removes all edges from p to other classes during a pivot. Let d be the number edges still incident to p, and let j be the size of the class that currently contains it. If $d < j$ then the cost of the pivot is $O(j)$. If $d > j$, we charge $O(d-j)$ of the $O(d)$ cost of the pivot to the $d-j$ edges thrown out of G during the pivot, at $O(1)$ per deleted edge. The total of charged costs over all confined pivots on p is $O(1)$ for each edge originally incident to p. The uncharged cost of the pivot is $O(j)$. Since the value of j drops by a factor of two each time a pivot occurs on p, the uncharged cost of pivots on p after it is confined is $O(k'/2+k'/4+k'/8+...+2+1) = O(k')$, which is $O(1)$ for each edge originally incident to p. Thus, confined pivots contribute $O(m)$ time to the running time of *ROP*. Since the size of a class containing p drops by a factor of two each time it is used as a pivot, and no pivot occurs on it when it is in a class larger than k, there are $O(\log k/k')$ unconfined pivots on p. Each pivot takes $O(1)$ time for each edge incident to p, so the total contribution of unconfined pivots to the running time of *ROP* is $O(m\log k/k') = O(m)$ for graphs where the ratio k/k' of maximum to minimum degree is bounded by a constant.

References

[1] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[2] E. Dahlhaus. Efficient parallel modular decomposition. 21^{st} *Int'l Workshop on Graph Theoretic Concepts in Comp. Sci. (WG 95)*, 21, 1995.

[3] E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical modular decomposition. *Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms*, 8:26–35, 1997.

[4] A. Ehrenfeucht, H.N. Gabow, R.M. McConnell, and S.J. Sullivan. An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *Journal of Algorithms*, 16:283–294, 1994.

[5] T. Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.

[6] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[7] M. Habib, R. McConnell, C. Paul, and L. Viennot. Transitive orientation, interval graph recognition, and consecutive ones testing. *Theoretical Computer Science (to appear)*.

[8] M. Habib, C. Paul, and L. Viennot. Partition refining techniques. *Manuscript*.

[9] R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[10] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5:536–545, 1994.

[11] R.M. McConnell and J.P. Spinrad. Linear-time transitive orientation. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 8:19–25, 1997.

[12] R.H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Order*, pages 41–101. D. Reidel, Boston, 1985.

[13] J.P. Spinrad. Graph partitioning. *Manuscript*, 1985.