# Chapter 59
# Linear-Time Modular Decomposition and Efficient Transitive Orientation of Comparability Graphs

Ross M. McConnell[†] Jeremy P. Spinrad[‡]

**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.* The modular decomposition facilitates solution of a number of combinatorial problems on certain classes of graphs, and algorithms for computing it have a lengthy history. 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, if such an assignment is possible. We give the first linear-time algorithm for modular decomposition, and a new bound of $O(n+m\log n)$ on transitive orientation and the problem of recognizing permutation graphs and two-dimensional partial orders.

## 1. Introduction

A *comparability graph* [15] is the graph obtained by ignoring the edge directions of a transitive dag (*i.e.* a partial order). A 2-dimensional partial order [8] is any partial order obtainable by intersecting two linear orders, and a *permutation graph* [29] is the corresponding comparability graph. These classes of graphs and partial orders arise in a number of combinatorial problems and have applications in scheduling theory. 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, some important combinatorial problems become solvable in linear time.

The *modular decomposition* of a graph is an $O(n + m)$-space representation of all *congruence partitions* on a graph [26,27]. This decomposition is also known as *substitution decomposition* [26,27], *prime tree decomposition* [11] and *X-join decomposition* [18]. A graph with only a trivial modular decomposition is *prime*.

We give the first general linear-time algorithm for computing the modular decomposition of an undirected graph. There have been a number of $O(n^4)$ algorithms [20,23]) $O(n^3)$ or $O(nm)$ algorithms [2, 7, 12, 13, 18, 34], and $O(n^2)$ algorithms [9, 24, 28] for graphs and special cases of graphs. 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 [4, 35]. Most recently, $O(n + m\log n)$ [6] and $O(n + m\alpha(m,n))$ [32] algorithms have been developed for undirected graphs, where $\alpha(m,n)$ is the inverse of Ackermann's function [3]. We modify the algorithm of [32] by replacing the "companion tree" data structure with simpler techniques. We then modify this algorithm to get a linear time bound.

We also give a simple node-sorting procedure that gives a unified $O(n + m\log n)$ solution to transitive orientation and recognition of 2-dimensional partial orders and permutation graphs, given the modular decomposition. The

† Dept. of Computer Science, Campus Box 430, University of Colorado, Boulder, CO 80309

‡ Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235. J. Spinrad was supported by grant R592-9632 from the National Security Agency.

same procedure recognizes arbitrary prime graphs without the decomposition. The best previous bound for recognition of 2-dimensional partial orders and permutation graphs is $O(n^2)$ [31]. The previous algorithms for transitive orientation took either $O(n^2)$ time [31], or $O(dm)$ time [14, 16, 29], where $d$ is the maximum degree of a node in the graph. As with the algorithm of [31], our algorithm fails to recognize when its input is not a comparability graph, and instead produces a non-transitive orientation of the graph. Surprisingly, this is not a problem for many applications.

The following problems can be solved on comparability graphs in $O(n + m \log n)$ time as a result of these bounds. 1) Maximum clique: Transitively orient $G$ and find the longest path in the result. If this longest path corresponds to a clique of $G$ then it is a maximum clique, even if $G$ is not a comparability graph. 2) Recognition of cointerval graphs [15]: Transitively orient $G$ and determine whether the result is an interval order. 3) Recognition of chordal comparability graphs [15]: Use the linear procedure of [22], which assumes that a transitive orientation is given. 4) Recognition of circular permutation graphs [30]: In [33] it is shown that an upper bound on recognition of circular permutation graphs is given by the bounds on transitive orientation and permutation graph recognition.

Möhring [25] gives a survey of NP-complete and other problems in graph theory and scheduling theory that may be solved more efficiently if the graph is non-prime and the decomposition is given. On graphs of *bounded decomposition diameter* [19], which are a generalization of cographs and series-parallel partial orders, many of these problems may be solved in linear time if modular decomposition is linear, although the constant of proportionality is exponential in the bound on the diameter.

## 2. Transitive Orientation and Recognition of Permutation Graphs

We assume that the adjacency-list representation is used for graphs. If $G$ is a graph or digraph, the set of nodes of $G$ is denoted *nodeset*$(G)$. A node $x$ *distinguishes* nodes $y$ and $z$ if $x$ is adjacent to exactly one of them or if exactly one of them is adjacent to $x$. A set $X$ *agrees* on $x$ if $x$ does not distinguish any two of its members. $X$ is a *module* if it agrees on each member of *nodeset*$(G) - X$. The *trivial* modules of $G$ are *nodeset*$(G)$, its singleton subsets, and the empty set. A graph is *prime* if it has only trivial modules, and *degenerate* if every subset of *nodeset*$(G)$ is a module. It is degenerate iff it is complete or edgeless.

To recognize a permutation graph in $O(n + m \log n)$ time, we use the well-known characterization that $G$ is a permutation graph iff $G$ and its complement $\overline{G}$ are both transitively orientable [15]. This approach initially appears unpromising; simply constructing $\overline{G}$ requires $\Omega(n^2)$ time. Using this approach with "forcing" algorithms for transitive orientation such as those of [16, 29, 14] would take $O(n^3)$ time even on sparse graphs, since the complement must be oriented.

Before giving the solution, we give the algorithm for orienting a comparability graph $G = (V, E)$. Given modular decomposition, transitive orientation, and recognition of 2-dimensional-partial-orders and permutation graphs reduce to the special cases of these problems on prime graphs [17, 31]. We restrict our attention to prime graphs, and use the following idea, which is based on set partitioning [1]. Beginning with the partition $\{V - \{v\}, \{v\}\}$, we repeatedly partition classes according to adjacency to a node $w$ not in the classes. To get an $O(n + m \log n)$ bound, we use $w$ only if the size of the class that currently contains it is at most half the size of its class the last time it was used. This ensures that the adjacency list for $w$ is traversed only $O(\log n)$ times. A list of eligible classes can be maintained during computation to avoid searching for them. For reasons that will be made clear shortly, the partition classes are kept in an ordered list. The procedure follows:

```
NODE_PARTITION($G, v$)

Let L be the ordered list $(V - \{v\}, \{v\})$.
Lastused $[v]$ = Lastused $[V - \{v\}] = \infty$
while there is partition class $X$ such that $|X| \leq$
      lastused $[X]/2$
  Lastused $[X] = |X|$
  For each node $w$ in $X$
    For each class $Y$ other than $X$ whose members are dis-
    tinguished by $w$
      Let $Y_1$ be the members of $Y$ that are adjacent to $w$.
```

```
        Remove $Y_1$ from $Y$
        If $X$ occurs after $Y$ in $\mathbf{L}$
          Insert $Y_1$ as a new class in $\mathbf{L}$ immediately in front
          of $Y$.
        else
          Insert $Y_1$ as a new class in $\mathbf{L}$ immediately behind
          $Y$.
        Let lastused[$Y_1$] := lastused[$Y$]
return ($\mathbf{L}$)
```

If $G$ is prime, the procedure terminates when all partition classes are singletons. For the proof, we show that the largest class $Y$ in the final partition is a module, hence a singleton set. For any $x \in V - Y$, $x$ is in a class $X$ that is no larger than $Y$. Thus, $X$ is at most half as large as the last class that contained both $X$ and $Y$, so its members have been used to partition a class containing $Y$ at some point since $X$ and $Y$ were split apart. $Y$ must agree on $x$, so $Y$ is a module.

To ensure that when the adjacency list for $w$ is traversed, at most constant time is spent on each member of the list, it suffices to keep the members of each partition class in a doubly-linked list, and to label each node with the class to which it belongs. When a node $x$ is visited in $w$'s adjacency list, its class $Y$ is retrieved, and $x$ is removed from $Y$ and installed in $Y_1$. Old partition classes that become empty are deleted. The remaining lists give the new partition classes.

It is also necessary to determine which of two partition classes occurs before the other in NODE_PARTITION. To do this, associate a subinterval of $(1, 2, ..., n)$ with each paritition class by labeling the class with the first and last elements in the subinterval. Initially $(1, 2, ..., n-1)$ is associated with $V - \{v\}$, and $(n)$ is associated with $\{v\}$. When $Y_1$ is removed from $Y$, associate either the first or last $|Y_1|$ elements of $Y$'s interval with $Y_1$, depending on whether $Y_1$ goes before $Y$ in $\mathbf{L}$. Associate the remainder of the interval with the remaining portion of $Y$. The intervals for any two partition classes then indicate which precedes the other in $\mathbf{L}$.

The following procedure solves the transitive orientation problem.

```
NODE_SORT ($G$)

Select a node $v$.
$\mathbf{L}_1$ := NODE_PARTITION($G$,$v$).
Let $\{x\}$ be the first partition class $\mathbf{L}_1$.
$\mathbf{L}_2$ := NODE_PARTITION($G$,$x$).
Return $\mathbf{L}_2$
```

If $G$ is a graph, let $P_G$ denote the transitive dag that results when the edges of $G$ are oriented. If $(x,y)$ and $(y,z)$ are undirected edges of $G$, but $(x,z)$ is not, then the orientation of $(x,y)$ forces the orientation of $(y,z)$. We say that $(x,y)$ *forces* $(y,z)$.

*Lemma 2.1: If G is a prime comparability graph, NODE_SORT returns a topological sort of* $P_G$.

Proof: Note that $v$ and $x$ are adjacent, since $x$ is a member of the first partition class in $\mathbf{L}$ after the first partition. During the first run of NODE_PARTITION, $x$ is always in the first class in $\mathbf{L}$. Suppose $(v,x)$ is initially oriented toward $x$. From the forcing relation, it follows by induction on successively smaller classes that contain $x$ that if $X$ is a class containing $x$, all edges between $V - X$ and $x$ must be oriented toward $x$. The final such class is $\{x\}$, so $x$ is a sink in any transitive orientation. Applying this fact and the forcing relation in a similar way during the second partition shows that all edges are oriented from earlier to later classes in $\mathbf{L}$. □

This solves the transitive orientation problem by giving a topological sort of $P_G$. Returning to the problem of recognizing permutation graphs, we note that all operations in NODE_SORT involve division of classes into neighbors and non-neighbors. These steps can be performed on $\overline{G}$ without constructing $\overline{G}$; the roles "neighbor" and "non-neighbor" are simply reversed from their status in $G$, and all operations can be run with the adjacency list representation of $G$. This gives a topological sort of $P_{\overline{G}}$ in $O(n + m \log n)$ time.

To complete the permutation graph recognition algorithm, we sort the nodes of $G$ with an $O(n \log n)$ comparison sort as follows. If two nodes being compared are adjacent, use their order in the topological sort for $G$, and use their order in the topological sort for $\overline{G}$ otherwise. This gives one of the two linear orders whose intersection gives $P_G$. Reversing one of the two topological sorts and performing the comparison sort again gives the other. It is then a simple exercise in list manipulation to find in $O(n + m)$ time whether the intersection of these two lists does, in fact, correspond to $P_G$. This gives $O(n + m \log n)$ permutation-graph recognition. Two-dimensional partial order recognition is carried out the same way, except that the transitive orientation of $G$ is already provided.

The $O(n \log n)$ sort in this last procedure can be replaced with the following $O(n+m)$ procedure, which shows that the only bottleneck for linear-time transitive orientation is the NODE_PARTITION procedure. Let $v$ be a node of $G$, and let $X$ be the nodes of $G$ that are adjacent to $v$ in $G$ and that precede $v$ in the topological sort of $P_G$. Let $Y$ be the nodes of $G$ that are adjacent to $v$ and that precede $v$ in the topological sort of $P_{\overline{G}}$. Let $Z$ be the nodes of $G$ that are nonadjacent to $v$ an that precede $v$ in the topological sort of $P_{\overline{G}}$. $X$ and $Y$ are easily found in time proportional to the number of neighbors of $v$ in $G$. $|Z|$ is obtained by subtracting $|Y|$ from the number of nodes that precede $v$ in the topological sort of $P_{\overline{G}}$. $X \cup Z$ are the nodes that precede $v$ in one of the two linear orders whose intersection gives $G$, so the position of $v$ in this linear order is given by $|X| + |Z| + 1$. Since $v$'s position in the linear order may thus be found in time proportional to the number of neighbors of $v$, this gives an $O(n+m)$ algorithm for constructing the linear order.

Before leaving NODE_SORT, we make a final observation. Prime graph recognition by methods that are simpler than those for full modular decomposition was investigated in [5]. NODE_SORT gives an alternative method. The procedure *fails* if one of the calls NODE_PARTITION fails to return all singletons.

*Lemma 2.2: An arbitrary graph is prime iff it is connected and NODE_SORT does not fail.*

Proof: The forcing relation is defined on arbitrary graphs. If the subgraph induced by a module contains an edge $e$, it contains all edges that are forced by $e$. The first call to NODE_PARTITION demonstrates that any non-singleton module contains $v$. The second demonstrates that it contains $x$. NODE_SORT shows that the edge $(v,x)$ indirectly forces all others, so *nodeset*$(G)$ is the only non-singleton module if $G$ is connected. □

## 3. Overview of the Modular Decomposition Algorithm

Two sets $X$ and $Y$ *overlap* iff $X - Y$, $Y - X$, and $X \cap Y$ are all nonempty. If $G$ is a graph or digraph, and $X \subseteq$ *nodeset*$(G)$, then the subgraph induced in $G$ by $X$ is denoted $G \mid X$. If **P** is a partition of the nodes of $G$, a *system of representatives* from **P** is a set consisting of one node from each member of **P**. If each member of **P** is a module of $G$, then **P** is called a *congruence partition* [26, 27], and every system of representatives induces an identical subgraph. This subgraph is denoted $G / \mathbf{P}$. The *image in G* of a set $X$ of nodes of $G / \mathbf{P}$ is the union of the members of **P** that are represented by the members of $X$. We make use of two rules [26]. The *modular subgraph rule* is that if $X$ is a module in $G$, then the modules in $G \mid X$ are those modules of $G$ that are subsets of $X$. The *quotient rule* is that a set of nodes of $G/\mathbf{P}$ is a module in $G / \mathbf{P}$ iff its image in $G$ is a module in $G$.

Let $G$ be a directed graph. The *component graph* for $G$ [3] has one node for each strongly connected component. 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 a member of $X$ to a member of $Y$.

A *partitive set family* **F** on a universe $U$ is a set family such that $U$ and its singleton subsets are members of **F**, and whenever $X$ and $Y$ are overlapping members of **F**, then $X \cap Y$, $X \cup Y$, $X - Y$ and $(X - Y) \cup (Y - X)$ are also members of **F**. A partitive family may be represented in $O(|U|)$ space as follows [26]. Let the members of the family that overlap no other be nodes in the tree; the containment relation on those sets gives the ancestor relation in the tree. The internal nodes may then each be labeled degenerate or prime so that $X \subseteq U$ is a member of **F** iff it is a node in the tree or a union of children of a degenerate node. We will call this representation the *partitive tree*. The family of modules of an undirected graph $G$ is a partitive family [26], and its modular decomposition is precisely its partitive tree. The modular decomposition of $G$ will be denoted $MD(G)$.

A *union tree* on universe $U$ is any tree whose leaves are the members of $U$. We will treat a node of the union tree as synonymous with its set of leaf descendants. If $U$ is an internal node of a union tree $T$, $children_T(U)$ denotes its children. A partitive tree may be given by a union tree whose nodes are labeled prime or degenerate.

A partitive tree $T$ on the nodes of $G$ is an *M tree* if the modules of $G$ are a subfamily of the partitive family it represents. If $T_1$ and $T_2$ are partitive trees, then $T_2$ is *stronger* than $T_1$ if the partitive family it represents is a subfamily of the one that $T_1$ represents. $MD(G)$ is clearly the strongest possible M tree. The algorithm starts with a weak M tree called a *p4 tree* and computes a sequence of increasingly stronger M trees until $MD(G)$ is obtained. We distinguish two 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 a system of representatives from $children_T(U)$ children that induces a degenerate subgraph in $G$.*

*M2: Internal nodes are labeled prime or degenerate, and for each degenerate node $U$, the members of $children_T(U)$ are modules in $G \mid U$ and $(G \mid U) / children_T(U)$ is degenerate.*

Lemma 3.1: *Let $T$ be an M2 tree for an undirected graph $G$. A non-singleton set $W$ of nodes of $G$ is a member of $MD(G)$ if and only if it is either a member of $T$ that is a module or a maximal union of children of a degenerate node that is a module.*

The proof follows from the fact that no module overlaps any member of $T$ or any member of the modular decomposition.

In [32], a linear algorithm is given that computes a p4 tree, which is an M1 tree by Theorem 4 of that paper. By Lemma 3.1, the modular decomposition may be computed from an M2 tree $T$ as follows. In [32], a simple $O(m + n)$ postorder traversal algorithm is given that may be used to compute which members of a union tree $T$ on $nodeset(G)$ are modules. These are members of the modular decomposition by Lemma 3.1. The procedure also computes, for each member $X$ of $T$, a sorted "adjacency list" consisting of the members of $nodeset(G) - X$ that are adjacent to every member of $X$. By Lemma 3.1, radix sorting the adjacency lists of modular children of a degenerate node gives the remaining members of the decomposition; these are the unions of children whose lists are not distinguished by the sort. The sum of lengths of these lists is shown to be $O(m)$, so this radix sort on all degenerate nodes requires only $O(n + m)$ time.

To complete the modular decomposition algorithm, it remains only to give an algorithm to compute an M2 tree from an M1 tree. Lemma 3.1 and the framework of M1 and M2 trees replaces the companion tree structure of [32].

## 4. Computing an M2 tree from an M1 tree in $O(n + m\alpha(m,n))$ time.

Let $U$ be a node in a union tree $T$. For a subfamily **F** of $children_T(U)$, we say we *give* **F** *a new parent in* $T$ if we create a node $Z$, and surgically make $Z$ be a child of $U$ and the parent of the members of **F**. We *refine* $U$ *with a tree* $T'$ by surgically replacing $U$ and its children with a tree $T'$ whose root is $U$ and whose leaves are the old children of $U$. A prime node in an M1 tree already satisfies the M2 property. To each degenerate node $U$ in the M1 tree, we apply an algorithm, *REFINE*$(U)$, which refines $U$ so that $T$ is a stronger M1 tree.

Let $G$ be a graph, let $T$ be an M1 tree on $G$, and let $U$ be a degenerate node in $T$. Let $\mathbf{F_f} = \{X\colon X \in children_T(U)$ and $X$ agrees on each member of $nodeset(G) - U\}$. The *forcing graph* $G_U$ denotes a graph whose nodes are the members of $\mathbf{F_f}$, and where if $X_1, X_2 \in \mathbf{F_f}$, then $(X_1, X_2)$ is an edge iff $X_1$ disagrees on some node in $X_2$. We denote with $G'_U$ the component graph of $G_U$, and view each node of $G'_U$ as the *union* of the corresponding nodes of $G_U$.

The algorithm for computing all forcing graphs on the p4 tree is given in [32], and is a variant of the postorder algorithm for computing which members of a union tree are modules. The sum of nodes in all forcing graphs is $O(n)$, and the sum of edges is $O(m)$. Their component graphs can thus be computed in $O(n + m)$ time [3].

Proposition 4.1: *Let $U$ be a degenerate node in an M1 tree $T$. A union $X$ of two or more members of $children_T(U)$ is a module of $G$ only if $X$ is a union of nodes of $G'_U$ and $X$ does not overlap the union of a weak component of $G'_U$.*

Lemma 4.2: *If $X$ is the union of a set $X'$ of nodes of $G'_U$ that are contained in a weak component of $G'_U$, then $X$ is a module of $G$ only if $X'$ is a module of $G'_U$, and $X'$ has no outgoing edges in $G'_U$.*

Because of space limitations, we omit the straightforward proofs.

We now give *REFINE*$(U)$, which is the basis of both the $O(n + m\alpha(m,n))$ algorithm and the $O(n + m)$ algorithm. We must maintain the invariant that after each modification of the tree, it continues to be an M tree. By Proposition 4.1, we may give the members of each strong component of $G_U$ a new parent that is prime. These new parents are the nodes of $G'_U$. We may then give the members of each weak component $I$ of $G'_U$ a new degenerate parent, $Z_I$, and, in turn, give all of these $Z_I$ nodes a new degenerate parent $Z_0$. We may then relabel $U$ prime. Suppose that for each $Z_I$, we know a partitive tree $T'$ for a partitive family on $I$ that has as a subfamily those modules of $G'_U \mid I$ that have no outgoing edges in $G'_U \mid I$. By Lemma 4.2, we may refine $Z_I$ with $T'$. In a postprocessing pass, those nodes of the refinement that have only one child may be collapsed to give an equivalent partitive tree.

Since $U$ is degenerate, the tree is initially an M1 tree, and the final tree is still an M tree, it follows that the final tree is a stronger M1 tree. It remains to specify how to find the tree $T'$ for refining $Z_I$. Note that $G'_U$ is a dag. If $T'$ is found with the following procedure, the result of applying *REFINE*$(U)$ to each degenerate node $U$ in an M1 tree is an M2 tree.

*DAGTREE*$(G)$

($G$ is a weakly-connected dag. Compute a partitive family on nodes of $G$ that has as a subfamily those modules of $G$ that have no outgoing edges.)

Number the nodes of $G$ in reverse topological order, where 1 is a sink and $n$ is a source. Let $G_i$ denote the subgraph induced by the nodes {1,2, ..., i}. Process the nodes in ascending order. When node $i$ is reached, we have already computed a forest that gives $DAGTREE(G')$, for each subgraph $G'$ induced by a weakly connected component of $G_{i-1}$. Create two new nodes, $x$ and $y$. Label $x$ prime and $y$ degenerate. Make $y$ be the left child of $x$ and let $i$ be the right child. For each edge $(i,j)$ of $G$, find the tree corresponding to the weak component of $G_{i-1}$ that contains $j$, and attach it as a child of $y$, if this has not already been done for that tree. If $y$ has no children, remove it from the tree. After node $n$ has been processed, collapse all nodes from the tree that have only one child in order to produce an equivalent partitive tree such that each node has at least two children.

*DAGTREE* may be implemented as a series of $O(n)$ unions and $O(m)$ finds, and thus requires $O(n + m\,\alpha(m,n))$ time [3]. An $O(n + m\,\alpha(m,n))$ time bound for *REFINE* if *DAGTREE* is used follows easily from elements of the proof of the time bound for the algorithm of [32].

In the refinement of $U$, let $X$ be a degenerate node. $X$ is the union of a set $X'$ of nodes of $G_U$. In $G_U \mid X'$, the sets of nodes that correspond to members of $children_T(X)$ are weakly-connected components. Thus, there are no edges of $G_U$ that go between members of $children_T(X)$. Every node of $G_U$ that is contained in one child of $X$ agrees in $G$ on every node in every other child of $X$. This, combined with the fact that there is a system of representatives from the nodes of $G_U$ that induces a degenerate subgraph in $G$ ensures that $(G \mid X) / children_T(X)$ is degenerate. Thus, the tree obtained by applying *REFINE* to each degenerate node of an M1 tree is an M2 tree.

## 5. Deriving an M2 tree from an M1 tree in O(n + m) time.

The obstacle to a linear-time algorithm is the sequence of *union* and *find* operations executed in computing $DAGTREE(G'_U)$. Our approach is to replace *DAGTREE* with the linear-time *WEAK_TREE*, below, to compute a refinement $T'$ of $Z_l$ given in *REFINE*. The output of *WEAK_TREE* still satisfies the requirements of $T'$ in *REFINE*, so the resulting refinement is still a stronger M1 tree. However, it is no longer guaranteed that all degenerate nodes in the refinement satisfy the M2 property. To compensate for this, we apply $REFINE(Y)$ recursively to each degenerate node $Y$ that is not known to satisfy the M2 property. $Y$ is known to satisfy the M2 property if it was $Z_0$ in a call to *REFINE*. This again guarantees that the end result is an M2 tree. However, we are able to show that the time spent in all recursive calls is $O(n + m)$.

*WEAK_TREE*$(G)$

Run $DAGTREE(G)$ with the following change. A tree in the forest has a ***representative node***. When node $i$ is processed, ignore all edges $(i,j)$ such that $j$ is not the representative node of a current tree in the forest. Make $i$ be the representative of the new tree containing $i$. When node $n$ has been processed, there may be more than one tree in the forest; in this case, create a node $X$, label it degenerate, and make the trees in the forest its children.

When a node $i$ is first processed in $WEAK\_TREE(G)$, it becomes the representative for the tree in the forest on $G_i$ that contains it, and may be labeled with a pointer to the root $x$ of the current tree for that component in contstant time. Once a representative ceases to be a representative, it may not be a representative again. Thus, it may be marked as a non-representative in constant time. It follows that *WEAK_TREE* runs in linear time.

If $Y$ is a degenerate node in the refinement of some node $U$, there is not enough time to compute $G_Y$ from scratch. Instead, we observe that $Y$ is contained in a subtree of the refinement of $U$ that is a union tree on $G_U$. We compute the forcing graphs for nodes in this subtree recursively from $G_U$. We thus modify our definition of what is meant by the forcing graph. As before, the nodes of $G_Y$ are the children of $Y$ that agree in $G_U$ on each node of $G_U$ that is disjoint from $Y$. If $(Y_1,Y_2)$ are nodes of $G_Y$, then $(Y_1,Y_2)$ is an edge of $G_Y$ if $Y_2$ contains a node of $G_U$ that distinguishes the nodes of $G_U$ that are contained in $Y_1$ *or if there is a node $U_1$ of $G_U$ that is contained in $Y_1$ and a node $U_2$ of $G_U$*

*that is contained in $Y_2$ such that $(U_1,U_2)$ is an edge if $G_U$.*

We will call $G_U$ the *parent graph* of $G_Y$. The italicized condition is the only change, so the children graphs of $G_U$ may be computed from $G_U$ with the same algorithm as the one in [32] that computes forcing graphs from $G$, with the trivial modification that incorporates this change and the fact that $G_U$ is directed. The forcing graph $G_Y$ still has the same meaning: if a module contains $Y_1$ then it must also contain $Y_2$ if $(Y_1,Y_2)$ is an edge of $G_Y$.

The following establishes that when *WEAK_TREE* is run on a weakly-connected component of $G'_U$, it returns a tree that satisfies the requirement on $T'$ that is given in *REFINE*.

Lemma 5.1: *If $G$ is a weakly-connected dag, then the family of modules of $G$ that have no outgoing edges is a subfamily of the partitive family represented by WEAK_TREE$(G)$.*

Sketch of proof: Let $X$ be a module that has no outgoing edge. Show that if $X$ contains elements of two siblings in the union tree produced by the algorithm, that $X$ must contain the two siblings. The lemma then follows, since each prime node in the tree has only two children. □

After processing halts in *REFINE*, if $Y$ is a degenerate node, then it was $Z_0$ in a call to *REFINE*$(U)$ for some formerly degenerate node $U$. The children of $Y$ are the weakly-connected components of $G_U$. By induction from parent forcing graphs to their children, it is easily seen that if $Z_1$ and $Z_2$ are weakly-connected components of a forcing graph, then the members of $Z_1 \times Z_2$ are either all edges or all non-edges in $G$. As in the previous section, this fact and the fact that the unrefined tree was an M1 tree together imply that $Y$ satisfies the M2 property.

Theorem 5.2: *Modular decomposition is computable in $O(n + m)$ time.*

Proof: Charge the operations in a call of *REFINE*$(Y)$ to the nodes and edges of $G_Y$ and to the children of $Y$ that are not nodes in $G_Y$. No more than constant cost needs to be charged to any of these items, since *WEAK_TREE* is used instead of *DAGTREE*, and computation of forcing graphs takes time proportional to their size. We thus get an $O(n + m)$ bound on the running time of all recursive calls of *REFINE* if we can show that all recursively computed forcing graphs together have $O(n + m)$ nodes and $O(m)$ edges.

An edge $(U_1,U_2)$ of a forcing graph $G_U$ "reappears" in a child graph $G_Y$ if it is one of the edges considered in computing the existence of some edge $(Y_1,Y_2)$ in $G_Y$. This happens if one of $Y_1$ and $Y_2$ contains $U_1$ and the other contains $U_2$. An edge *dies* if it does not reappear. We show that an edge of $G_Y$ is either the unique reappearance of at least two edges of $G_U$, or else it dies. This halves the bound on the total number of edges in forcing graphs with each recursive generation, giving an $O(m)$ bound on their number.

If $(Y_1,Y_2)$ and $(Y_2,Y_1)$ are both edges of $G_Y$, then they die, since $Y_1$ and $Y_2$ are members of a strongly-connected component of $G_Y$, and thus become children of a prime node with the next refinement of $Y$. When *REFINE* operated on $G_U$, then in the refinement it produced, it either made $U_2$ be a child of a prime node, or made each sibling of $U_2$ in the refinement consist of nodes of $G'_U$ that had no edge to $U_2$. If $Y_2 = U_2$, then either of these conclusions contradicts assumption that $(Y_1,Y_2)$ is an edge in a child graph of $G_U$. It follows that $Y_2$ is the union of at least two nodes of $G_U$. $(Y_1,Y_2)$ must either be the unique reappearance of at least two edges of $G_U$, or $(Y_2,Y_1)$ is also an edge of $G_Y$, and thus dies.

Using the O($m$) bound on the number of edges in all forcing graphs, it is straightforward to give an O($n + m$) bound on the number of nodes, since isolated nodes in a forcing graph become children of $Z_0$ in *REFINE*, for which no forcing graph is computed.

**Acknowledgments**

The first author would like to thank Andrzej Ehrenfeucht and Hal Gabow for enlightening discussions on this topic.

REFERENCES

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

[2] H. Buer and R.H. Möhring, A fast algorithm for the decomposition of graphs and posets, *Math. Oper. Res.*, **8** (1983), 170-184.

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

[4] D.G. Corneil, Y. Perl and L.K. Stewart, A linear recognition algorithm for cographs, *SIAM J. Comput*, **14** (1985), 926-934.

[5] A. Cournier and M. Habib, An efficient algorithm to recognize prime undirected graphs, *Rep. R.R. LIRMM No. 92-023.*, Laboratoire D'informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier.

[6] A. Cournier and M. Habib, An O($n$ + $m$ log$n$) graph substitution decomposition algorithm, *forthcoming.*

[7] W.H. Cunningham, Decomposition of directed graphs, *SIAM J. Algebraic Discrete Methods*, 3 (1982), 214-228.

[8] B. Duschnik and E.W. Miller, Partially ordered sets, *Amer. J. Math.*, **63** (1941), 600-610.

[9] 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 2-structures and modular decomposition of graphs, *Journal of Algorithms*, to appear.

[10] A. Ehrenfeucht and R.M. McConnell, A $k$-structure generalization of the theory of 2-structures, *Theoretical Computer Science*, forthcoming.

[11] A. Ehrenfeucht and G. Rozenberg, Theory of 2-structures, part 2: representations through labeled tree families, *Theoretical Computer Science*, **70** (1990), 305-342.

[12] T. Gallai, Transitiv orentbare Graphen, *Acta Math. Acad. Sci. Hungar. Tom.*, **18** (1967), 25-66.

[13] M. Golumbic, Comparability graphs and a new matroid, *J. Combin. Theory Ser. B*, **22** (1977), 68-90.

[14] M. Golumbic, The complexity of comparability graph recognition and coloring, *Computing*, **18** (1977), 199-208.

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

[16] A. Gouilà-Houri, Caracterisation des graphes non orientés dont on peut orienter les arêtes de manière à obtenir le graph d'une relation d'ordre, *C. R. Acad. Sci. Paris*, **254** (1962), 1370-1371.

[17] M. Habib, Comparability invariants, *Annals Discrete Math*, **23** (1984), 331-386.

[18] M. Habib and M.C. Maurer, On the X-join decomposition for undirected graphs, *Discrete Applied Mathematics*, 1 (1979), 201-207.

[19] M. Habib and R.H. Möhring, On some complexity properties of N-free posets and posets of bounded decomposition diameter, *Discrete Appl. Mathematics*, 12 (1985), 279-291.

[20] L.O. James, R.G. Stanton, and D.D. Cowan, Graph decomposition for undirected graphs, in *3rd South-Eastern Conf. Combinatorics, Graph Theory and Computing* (F. Hoffman and R.B. Levow, eds.), pp. 281-290, Utilitas Mathematica, Winnipeg, 1972.

[21] R. D. Lou, and M. Sarrafzadeh, Circular permutation graph family with applications, *Discrete Applied Mathematics*, **40** (1992), 433-457.

[22] T. Ma and J. Spinrad, Transitive closure for restricted classes of partial orders, *Order*, **8** (1991), 175-183.

[23] M.C. Maurer, "Joints et decompositions premières dans les graphes," These 3ème cycle, Université de Paris VI, 1977.

[24] R.M. McConnell, An O($n^2$) incremental algorithm for modular decomposition of 2-structures, submitted to *Algorithmica.*

[25] R.H. Möhring, Algorithmic aspects of comparability graphs and interval graphs, in: I. Rival, ed., *Graphs and Orders* (D. Reidel, Boston, 1985), 41-101.

[26] R.H. Möhring, Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and Boolean functions, *Annals of Operations Research*, **4** (1985/6), 195-225.

[27] R.H. Möhring and F.J. Radermacher, Substitution decomposition for discrete structures and connections with combinatorial optimization, *Annals of Discrete Mathematics*, **19** (1984), 257-356.

[28] J.H. Muller and J. Spinrad, Incremental modular decomposition, *Journal of the ACM*, **36** (1989), 1-19.

[29] A. Pnueli, S. Even and A. Lempel, Transitive orientation of graphs and identification of permutation graphs, *Canad. J. Math.*, **23** (1971), 160-175.

[30] D. Rotem, and J. Urrutia, Circular permutation graphs, *Networks*, **12** (1982), 429-437.

[31] J. Spinrad, On comparability and permutation graphs, *Siam J. Comput*, **14** (1985), 658-670.

[32] J.P. Spinrad, $P_4$ trees and substitution decomposition, *Discrete applied mathematics*, **39** (1992), 263-291.

[33] R. Sritharan, A note on circular permutatation graphs, forthcoming.

[34] G. Steiner, *Machine scheduling with precedence constraints*, Ph.D. Thesis, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ont. (1982).

[35] J. Valdes, R.E. Tarjan, and E.L. Lawler, The recognition of series-parallel digraphs, *SIAM J. Comput.*, **11** (1982), 299-313.