Efficient and Practical Algorithms for Sequential Modular Decomposition

Elias Dahlhaus, Jens Gustedt, Ross M. Mcconnell

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient and Practical Algorithms for Sequential Modular Decomposition

Elias Dahlhaus, Jens Gustedt, Ross M. McConnell

N°3804

Novembre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3804--FR+ENG

Efficient and Practical Algorithms for Sequential Modular Decomposition

Elias Dahlhaus*, Jens Gustedt†, Ross M. McConnell‡

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n ° 3804 — Novembre 1999 — 23 pages

Abstract: A *module* of an undirected graph $G = (V, E)$ is a set X of vertices that have the same set of neighbors in $V \setminus X$. The *modular decomposition* is a unique decomposition of the vertices into nested modules. We give a simpler approach to sequential linear-time modular decomposition.

Key-words: modular decomposition, linear time algorithms

(Résumé : tsvp)

* Dept. of Computer Science and Dept. of Mathematics, Univ. of Cologne, Cologne, Germany.
† LORIA and INRIA Lorraine, 54506 Vandœuvre lès Nancy, France, Jens.Gustedt@loria.fr.
‡ Dep. of Computer Science and Engineering, Univ. of Colorado at Denver, Denver, CO 80217-3364 USA.

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

Algorithmes practiques et efficaces pour la décomposition modulaire séquentielle

Résumé : Un module d'un graphe non-orienté $G = (V, E)$ est un ensemble X de sommets qui partage le même ensemble de voisins en $V \setminus X$. La décomposition modulaire est une décomposition unique des sommets en modules emboîtés. Nous donnons une approche simplifiée à la décomposition modulaire en temps linéaire.

Mots-clé : décomposition modulaire, algorithmes linéaires

1 Introduction

Modular decomposition is a tree-like decomposition of a graph that we describe below. When the resulting tree is nontrivial, it provides a way to apply a divide-and-conquer strategy to a number of NP-complete problems. Thus, a large number of NP-complete problems can be solved recursively on the tree in time proportional to the number of vertices times $2^{\Theta(k)}$, where k is the maximum number of children of a node in this tree, which gives rise to polynomial algorithms for graphs whose decomposition trees are of bounded degree.

Let n denote the number of vertices, and let m denote the number of edges. There have been a number of $O(n^4)$, $O(n^3)$, $O(nm)$, and $O(n^2)$ algorithms for finding modular decomposition, such as Buer and Möhring (1983), Ehrenfeucht et al. (1994), Golumbic (1977), Habib and Maurer (1979), McConnell (1995), Muller and Spinrad (1989), Steiner (1982), 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, see Corneil et al. (1985), Valdes et al. (1982). $O(n + m \log n)$, bounds for arbitrary undirected graphs were then given in Cournier and Habib (1993), and an $O(n+m\alpha(m,n))$ bound was given in Spinrad (1992). The first linear-time algorithm was given in McConnell and Spinrad (1994). An altogether different linear-time was given shortly thereafter in Cournier and Habib (1994). Both of these algorithms are lengthy and quite challenging to understand.

We derive a conceptually simpler linear-time sequential algorithm than was previously available. In addition, we give $O(n + m\alpha(m, n))$ variant that we believe is simple enough to be characterized as practical. We sketched this version in Dahlhaus et al. (1997).

2 Preliminaries

This section gives the basic definitions of the objects and algorithmic features that we are talking about. It is split into two parts. The first part introduces the basic notation and facts about modular decomposition. The second describes some essential data structures that are needed to handle graphs efficiently.

2.1 Modular decomposition of undirected graphs

In this paper, the term *graph* will denote an undirected graph, while the term *digraph* will denote a directed graph. We let n denote the number of vertices and m the number of edges.

Let $G = (V, E)$ be a graph with vertex set V and edge set E. If X is a subset of V, then $G|X$ is the *subgraph induced by* G. The *complement* of a graph G is the graph $\overline{G}$ with the same set of vertices, but where $\{u, v\}$ is an edge iff it is not an edge of G. A *connected component* is a maximal set of vertices that are all connected by paths. A *co-component* is a connected component of the complement. The *neighborhood* of a vertex v is the set

$$N(v) = \{w \mid \{v, w\} \text{ is an edge of } G\}.$$

A pair $\{u, v\}$ of vertices that is not an edge is a *non-edge*. The set of vertices in $V \setminus \{v\}$ that are not neighbors of v are its *non-neighbors*, and is denoted $\overline{N}(v)$.

A *strong neighbor* of a set $X \subseteq V$ of vertices is a vertex $y \in V \setminus X$ that is a neighbor of every vertex of X. A *non-neighbor* of X is a vertex $z \in V \setminus X$ that is not a neighbor of any vertex in X. A *module* of an undirected graph $G = (V, E)$ is a set X of vertices that all have the same set of neighbors in $V \setminus X$, see Figure 1. That is, X is a module iff every member of $V \setminus X$ is either a strong neighbor or a non-neighbor. Though there is a generalization of modules to digraphs, we will not refer to them in this paper.

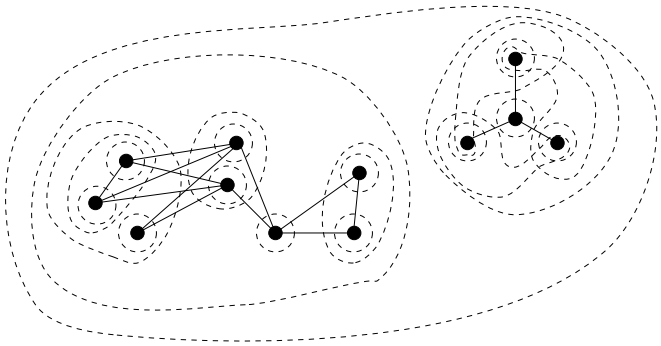

Figure 1: A graph and its modules.

Two sets X and Y *overlap* if $X \setminus Y$, $Y \setminus X$, and $Y \cap X$ are all nonempty. $X \Delta Y$ denotes $(X \setminus Y) \cup (Y \setminus X)$, the *symmetric difference* of Y and X. The following two observations are not hard to verify.

Lemma 2.1 (Möhring (1985)) *If X and Y are overlapping modules, then $X \setminus Y$, $Y \setminus X$, $Y \cap X$, $Y \cup X$, and $Y \Delta X$ are also modules.*

A pair X, Y of disjoint modules is *adjacent* if every member of Y is adjacent to every member of X, and *nonadjacent* if no member of Y is adjacent to any member of X.

Lemma 2.2 (Möhring (1985)) *Any pair X and Y of disjoint modules is either adjacent or nonadjacent.*

A *congruence partition* is a partition of V where each partition class is a module. The adjacency relation on members of a congruence partition is itself a graph, where each set of the congruence partition is treated as a single vertex. This graph is denoted $G/\mathcal{P}$. $G/\mathcal{P}$ completely specifies those edges of G that are not subsets of a single partition class (see Figure 2). A *factor* in a congruence

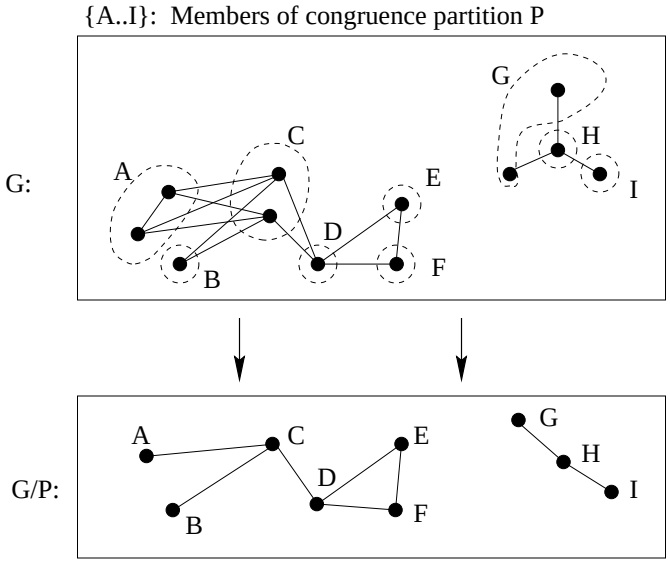

Figure 2: A graph with congruence partition and the corresponding quotient.

partition $\mathcal{P}$ is an induced subgraph $G|X$ for some $X \in \mathcal{P}$. The subgraphs induced by the parts specify

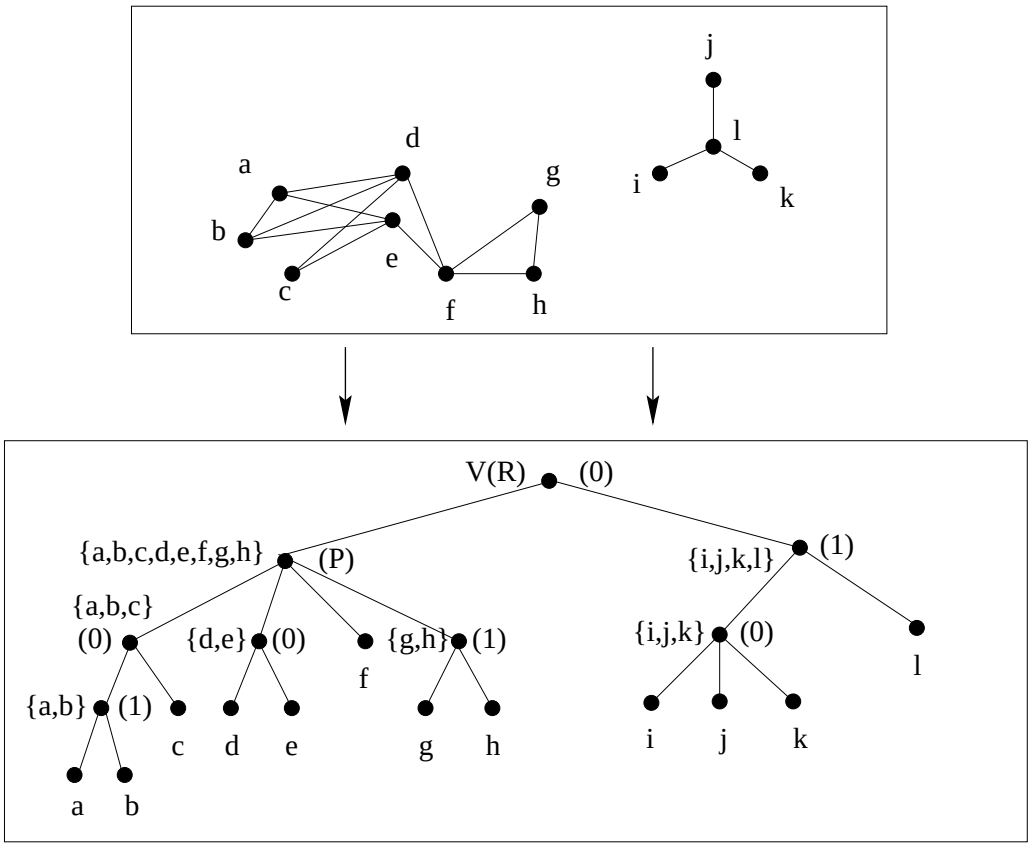

Figure 3: The modular decomposition tree of the graph of Figure 1.

precisely those edges of G that are not specified by the quotient. Thus, given a congruence partition $\mathcal{P}$, we may specify G by giving the quotient $G/\mathcal{P}$ and $G|X$ for each partition class $X \in \mathcal{P}$.

V, $\emptyset$ and the singleton subsets of V satisfy the definition of a module, and are the *trivial modules* of G. As a result, every graph has at least two congruence partitions, one with a single part V, and one with $|V|$ parts, where each part is a singleton subset of V. A graph is *prime* if it has only trivial modules, hence only the two trivial congruence partitions.

The quotient and substructures also preserve information about the arrangement of other modules of G:

Lemma 2.3 (Möhring (1985)) *If $\mathcal{P}$ is a congruence partition on G, and $\mathcal{M} \subseteq \mathcal{P}$, then $\mathcal{M}$ is a module in $G/\mathcal{P}$ iff $\bigcup \mathcal{M}$ is a module of G.*

Lemma 2.4 (Möhring (1985)) *If X is a module of $G = (V, E)$ and Y is a subset of X, then Y is a module of G iff it is a module of $G|X$.*

By these two lemmas, it follows that if no module overlaps a partition class of a congruence partition $\mathcal{P}$, the modules of the quotient and the substructures give the modules of G. This makes it desirable to specify G using congruence partitions whose parts do not overlap any modules. A *strong module* is a module that overlaps no other module of G, while a *strong quotient* is one whose parts are strong modules. The *modular decomposition* of G is the unique decomposition of G obtained by finding a coarsest strong quotient $\mathcal{P}$ that has more than one part, and then decomposing its factors recursively, see Figure 3. The decomposition of each factor specifies the factor. As observed above, the factors and the quotient then specify G.

V is the only maximal module. We will let the *highest submodules* be those modules that are not contained in any other module except V. To describe how the decomposition is derived, we use three cases.

prime case: *G and $\overline{G}$ are each connected.* One consequence of Lemmas 2.1 and 2.2 is that if a graph and its complement are both connected, the coarsest strong congruence partition of G is given by the highest submodules. To see this, note that Lemma 2.1 implies that if two highest submodules X and Y overlap, their union is a module, and since X and Y are highest submodules, their union must be V. It also implies that X and $Y \setminus X$ are modules. Thus, $X \cup Y = V$, and $\{X, Y \setminus X\}$ is a partition of V into two modules. Then Lemma 2.2 implies that either G or its complement is disconnected, a contradiction.

We conclude that if neither G nor $\overline{G}$ is disconnected, we may let $\mathcal{P}$ be the highest submodules of G. Other than V, no union of members of $\mathcal{P}$ is a module of G, since they are highest submodules. By Lemma 2.3, it follows that $G/\mathcal{P}$ has no nontrivial modules, hence we have vacuously enumerated its modules.

parallel case: *G is disconnected.* The highest submodules are not a partition of V, so we must resort to another method of finding the coarsest congruence partition. A connected component is a module that may not overlap any other module, so it must be strong. In addition, every nontrivial union of components other than V is a module, but since each such union overlaps another, no such module obtained in this way is strong. It follows that the set $\mathcal{P}$ of connected components of G is the coarsest strong congruence partition of G. Moreover, $G/\mathcal{P}$ is empty, so the modules of $G/\mathcal{P}$ are just the power set of $\mathcal{P}$.

series case: *$\overline{G}$ is disconnected.* Since the modules of G and its complement are identical, reasoning symmetric to that of the parallel case dictates that the coarsest strong congruence partition $\mathcal{P}$ of G is given by the connected components of $\overline{G}$. The quotient $G/\mathcal{P}$ is complete, since $\overline{G}/\mathcal{P}$ is empty, and the modules of $\overline{G}/\mathcal{P}$, hence the modules of $G/\mathcal{P}$, are again the power set of $\mathcal{P}$.

Summarizing, given some procedure for finding the highest submodules of a graph, we may compute the modular decomposition as follows. If G is disconnected, let $\mathcal{P}$ be the connected component, and if $\overline{G}$ is disconnected, let $\mathcal{P}$ be its components. Otherwise, let $\mathcal{P}$ be the highest submodules of G. $G/\mathcal{P}$ is a strong quotient. Recursively compute the rest of the decomposition on $G|X$ for each $X \in \mathcal{P}$. By Lemmas 2.3 and 2.4, a subset X of V is a module iff it is $\bigcup \mathcal{X}$ for some module $\mathcal{X}$ of one of the recursively computed quotients. Each such quotient either has no nontrivial modules, or else every subset of its vertices is a module.

Listing the members of all of the recursively computed congruence partitions takes $\Omega(n^2)$ space in the worst case. However, we may represent the decomposition tree in $O(n)$ space by using an $O(1)$-space representation for each node of the tree, and letting this node carry a pointer to a list of its children. In this tree, the leaf descendants of each node x give the corresponding module X. Since each node of the tree has at least two children, we may recover X from x in $O(|X|)$ time, so this representation of X is as efficient as any.

For notational convenience, we will identify x and X. For instance, we may speak of two modules X and Y as being siblings in the tree; this means that X is the leaf descendants of a tree node x, Y is the leaf descendants of a tree node y, and x and y are siblings. For any internal node X, the subgraph of G induced by a set consisting of one member of each child of Y of X gives the quotient produced by the recursive call on X.

A *degenerate node* is a parallel or series node in the decomposition tree. Every module of G that is not already a node of the tree is a union of children of a degenerate node. Every union of children of a degenerate node is a module. Thus, the decomposition also represents implicitly all modules of G.

Algorithm 1: MD(G)

if G *has only one vertex* v **then return** v;
let r be a new internal tree node;
if G *is disconnected* **then**
  label r '*parallel*' for each connected component G', make MD(G') a child of r
else if $\overline{G}$ *is disconnected* **then**
  label r '*series*' for each connected component G' of $\overline{G}$, make MD(G') a child of r
else
  label r '*prime*' for each highest submodule X, make MD($G|X$) a child of r
return r

2.2 Basic procedures

Now we present the basic data structures and algorithms that we need to efficiently represent and handle graphs and their modular decompositions.

2.2.1 Radix partitioning

Consider the following problem. Given a set of strings of integers from 1 to n and an array of empty buckets (lists of strings), we want to partition the strings into groups so that two strings are in the same group if and only if they are identical, and then leave the buckets in their initial state. This can be accomplished by radix sorting in $O(n+m)$ time, where m is the sum of lengths of the strings. However, we will find it useful to observe that we can accomplish it in just $O(m)$ time since the problem does not place any restriction on the order in which the partition classes must be presented.

Like radix sorting, the algorithm proceeds in rounds. In round i, we assume by induction that we have already found the partition classes of the strings that have length less than $i-1$, and that groups of strings that have identical $(i-1)$-integer prefixes appear together in our list.

The algorithm distributes the strings to the ordered buckets according to the integer that appears in some position i. A string is always added to the front of a bucket's list of strings when it is inserted. If a string is the first string to be inserted in a bucket, that bucket is added to a list of nonempty buckets.

Once the strings are distributed, it concatenates the lists of strings that appear in the nonempty buckets, emptying the buckets in the process. It uses the list of nonempty buckets to avoid visiting any empty buckets during this step.

After round i, a set of strings that have identical i-integer prefixes are consecutive in the list. Partition classes corresponding to i-integer strings can be removed from the list at this point. The running time can be charged at $O(1)$ to the i^{th} integers of the strings examined, and $O(1)$ to each integer of the i-integer strings that are removed from the list. Because of the similarity of this algorithm to radix sorting, we will call it *radix partitioning*.

2.2.2 Sorting Adjacency Lists

Given a numbering of vertices of a graph, we may sort all adjacency lists so that neighbors appear in ascending order, in $O(n+m)$ time, by concatenating all adjacency lists and then radix-sorting the resulting list using vertex of origin as primary sort key and ending vertex as secondary sort key. This is accomplished with two stable $O(n+m)$ sorts, a first one on the secondary key and a second one on the primary key, see Cormen et al. (1990). We obtain the adjacency lists by cutting this final list at points where the primary sort key changes.

2.2.3 Union-Find

We need a Union-Find data structure for some of our operations. This is any structure that maintains a partition $\mathcal{P}$ of a set V, and supports the following operations:

`Initialize` : Set $\mathcal{P} := \{\{v\} \mid v \in V\}$.

`Find` : Find a pointer to the member of $\mathcal{P}$ that contains a given $v \in V$.

`Union` : Given $v, w \in V$, merge the the two sets of $\mathcal{P}$ that contain v and w by replacing them with their union in $\mathcal{P}$.

An arbitrary sequence starting with one `Initialize`, on a set of size n, and a mixed sequence of $n-1$ `Union` and m `Find` operations takes $O(n\alpha(m,n))$ time, see Tarjan (1983), Cormen et al. (1990). The α is an extremely slow-growing but unbounded function that has value at most four in any practical application of the data structure.

To achieve a linear time bound for modular decomposition, a certain restriction of the union-find problem will be relevant. We say that a sequence of `Union` operations on the set V *respects* a graph $G = (V, E)$ if, for any of the subsets S that are created via the `Unions`, $G|S$ is connected. Another formulation of this property is the assertion that whenever two sets S_1 and S_2 are merged via a `Union` operation there will always be an edge in G that joins a member of S_1 to a member of S_2. For an overview of these concepts see e.g Gustedt (1998). The following theorem is crucial for the linear time bound of our algorithm.

Theorem 1 (Gabow and Tarjan (1985)) *Let $T = (V, E)$ be a tree. Any sequence of n `Union` and m `Find` operations on V that respects T can be performed in $O(n + m)$ time.*

2.2.4 Partially Complemented Representations

In a conventional adjacency-list representation of a directed graph G, each vertex carries a list of its neighbors. Such a representation clearly has a disadvantage when we also have to deal with the complement of the graph, or more generally, with non-edges: passing from a standard encoding of a graph to such an encoding of its complement rules out any possibility of a linear time algorithm. We circumvent this kind of problem with the following data structure.

Definition 2.5 *In a* partially complemented, *or* mixed *representation of $G = (V, E)$ every vertex maintains a list of either $N(v)$ or $\overline{N}(v)$ and a label that is either* standard *or* complemented *that tells which of the two cases applies for v. We define the* size *of a mixed representation to be $n + m'$, where n is the number of vertices, and m' is the sum of cardinalities of their associated lists.*

Definition 2.6 (Dahlhaus et al. (1997)) *Let the* bipartite complement *of a directed bipartite graph (V_1, V_2, A) be the graph*

$$\Big(V_1, V_2, \big((V_1 \times V_2) \cup (V_2 \times V_1)\big) \setminus A\Big). \tag{1}$$

In a mixed bipartite representation, *each vertex in V_1 (resp. V_2) has either a list of those members of V_2 (resp. V_1) that are out-neighbors, or else a list of those members of V_2 (resp. V_1) that are not out-neighbors.*

This representation has many interesting properties which are beyond the scope of this paper. The relevant result for our algorithm is the following:

Theorem 2 (Dahlhaus et al. (1997)) *Given a mixed representation of a directed graph or directed bipartite graph, finding the strongly-connected components takes time linear in the size of the representation.*

3 A Strategy for Modular Decomposition

Our approach uses elements from a previous strategy, see Ehrenfeucht et al. (1994), which we describe first. If v is a vertex of G, let $\mathcal{M}(G,v)$ denote $\{v\}$ and the set of maximal modules of G that do not contain v. That is, $X \in \mathcal{M}(G,v)$ if either $X = \{v\}$, or X is a module of G that does not contain v, and every proper superset that is a module of G contains v. $\mathcal{M}(G,v)$ is a partition of V. The following are easy consequences of Lemma 2.3.

Lemma 3.1 (Ehrenfeucht et al. (1994)) *All nontrivial modules of $G/\mathcal{M}(G,v_0)$ contain $\{v_0\}$. A subset $\mathcal{X}$ of $\mathcal{M}(G,v_0)$ is an internal node of the modular decomposition of $G/\mathcal{M}(G,v_0)$ iff $\bigcup \mathcal{X}$ is an ancestor of $\{v_0\}$ in the modular decomposition of G. All other strong modules are subsets of some member of $\mathcal{M}(G,v_0)$.*

Lemma 3.2 (Ehrenfeucht and Rozenberg (1990)) *If X is a module of G, then the strong modules of G that are proper subsets of X are the strong modules of $G|X$ that are proper subsets of X.*

The foregoing two lemmas give a strategy for finding the modular decomposition of G from the modular decompositions of subgraphs.

Algorithm 2: Ehrenfeucht et al. (1994)

Input: Let v_0 be an arbitrary vertex. $\texttt{MD}(G/\mathcal{M}(G,v_0))$ and $\{\texttt{MD}(G|X) \mid X \in \mathcal{M}(G,v_0)\}$,

Output: $\texttt{MD}(G)$

Let T be the modular decomposition of $G/\mathcal{M}(G,v_0)$;
foreach *leaf X of T* **do**
  comment: $X \in \mathcal{M}(G,v_0)$;
  Let T_X be the modular decomposition of $G|X$;
  Replace X with T_X in T;
  if *X and its parent $p(X)$ are both parallel or both series nodes in T* **then**
    Remove X from T and attach its children as children of $p(X)$

The proof of correctness follows from Lemmas 3.1 and 3.2, and an observation that the inner **if**-statement removes X if and only if X fails to be strong in G. The implementation of Ehrenfeucht et al. (1994) computes the modular decompositions of $G|X$ by recursion, and uses a separate technique to find the modular decomposition of $G/\mathcal{M}(G,v_0)$ that takes advantage of its simple structure. The best bound so far for this approach is $O(min((n^2), m \log n))$, see McConnell and Spinrad (1998). The bottleneck is a step called called *vertex partitioning*, which is used to compute $\mathcal{M}(G,v_0)$.

We now describe a significant improvement on the strategy, which is due to Dahlhaus (1995), who used it to obtain the parallel time bound for modular decomposition that we give below. We use it to obtain a linear-time sequential algorithm, and a parallel algorithm for transitive orientation. The strategy calls for computing $\texttt{MD}(G|N(v_0))$ and $\texttt{MD}\left(G|\overline{N}(v_0)\right)$ by recursion, and then finding $\{\texttt{MD}(G|X) \mid X \in \mathcal{M}(G,v_0)\}$ by a technique called *restriction*.

If $Y \subseteq V$, let the *restriction to Y* of the modular decomposition of G be the set of modular decompositions of the members of $\{(G|X) \mid X$ is a maximal module of G that is contained in $Y\}$. For $a,b \in Y$, let aRb be the relation where a and b have the same set of neighbors in $V \setminus Y$. Since this is an equivalence relation, this gives a partition of Y, which we will call the *same-neighbor partition*.

For the correctness, note that a $X \subseteq Y$ is a module of G iff all of its members have the same set of neighbors in $V \setminus X$. This happens iff it is a module in $G|Y$ and all members of X have the same neighbors in $V \setminus Y$. The procedure finds each maximal X that is a module of $G|Y$ and has only strong neighbors and non-neighbors in $G \setminus Y$, and makes it the root of a tree in a forest. That this tree is the modular decomposition of $G|X$ follows from Lemma 3.2.

Algorithm 3: The Restrict Step

Input: $Y \subseteq V$ and the modular decomposition T of $G|Y$

Output: the restriction to Y of the modular decomposition of G

Delete from T all nodes that represent sets that intersect more than one same-neighbor class;
In the remaining forest, group together any roots that are former children of a common degenerate node and are contained in a common same-neighbor class, and make them children of a new degenerate node.

Lemma 3.3 *The modular decompositions of the members of* $\{(G|X) \mid X \in \mathcal{M}(G, v_0) \setminus \{\{v_0\}\}\}$ *are given by the restrictions of the modular decomposition of* G *to* $N(v_0)$ *and* $\overline{N}(v_0)$.

Proof: The restrictions give the modular decomposition of $G|X$ for each maximal module of G contained in $N(v_0)$ and $\overline{N}(v_0)$. No such module contains v_0. If $X \in \mathcal{M}(G, v_0) \setminus \{\{v_0\}\}$, it is a maximal module of G that does not contain v_0, and since v_0 is either a strong neighbor or a non-neighbor, it must be contained in $N(v_0)$ or $\overline{N}(v_0)$. □

Algorithm 4 gives a high-level overview of the strategy. We make one final observation.

Algorithm 4: Dahlhaus (1995)

if *G has only one vertex v* **then**
  return v as a tree node

else

recurse: Select a vertex v_0, and recursively compute the modular decompositions of $G|N(v_0)$ and $G|\overline{N}(v_0)$.

restrict step: Using these trees, Algorithm 3, and Lemma 3.3, find, for each $X \in \mathcal{M}(G, v_0) \setminus \{\{v_0\}\}$, the modular decomposition of $G|X$.

v_0-modules step: Compute the modular decomposition of $G/\mathcal{M}(G, v_0)$;

assemble step: Using Algorithm 2, assemble these modular decompositions to form the modular decomposition of G.

Lemma 3.4 (Dahlhaus (1995)) *If* v_0 *is an an arbitrary vertex of* G*, then for any module* M *containing* v_0*,* $M \cap \overline{N}(v_0)$ *is a union of zero or more components of* $G|\overline{N}(v_0)$*, and* $M \cap N(v_0)$ *is a union of zero or more co-components of* $G|N(v_0)$.

Proof: If a module M contains v_0, then since v_0 is nonadjacent to every vertex in $M \setminus \overline{N}(v_0)$, every vertex in $M \cap \overline{N}(v_0)$ is nonadjacent to very vertex in $M \setminus \overline{N}(v_0)$. Thus $M \cap \overline{N}(v_0)$ is a union of zero or more connected components of $G|\overline{N}(v_0)$. That $M \cap N(v_0)$ is a union of co-components of $G|N(v_0)$ is proved in a symmetric way, by reversing the roles of edges and non-edges. □

4 Sequential Modular Decomposition in Near Linear Time

We now describe an algorithm that is based on Algorithm 4 and that runs in $O(n + m\alpha(m, n))$ time. The union-find operations are the only obstacle to a linear time bound. This algorithm is simpler than any previous algorithm with this bound.

4.1 A data structure for the decomposition tree

We use the following data structure to represent modular decomposition trees.

Data Structure 4.1 (Representation of a modular decomposition tree.)

representative: *Each node of the tree carries a pointer to a vertex of the graph which is one of its leaf descendants, called its* representative.

siblings: *Each node of the tree is a member in a list that maintains the children of its parent.*

parent: *Each node u of the tree carries a pointer to its parent. This pointer is up-to-date except under special circumstances that are described below.*

components: *Each vertex of the subgraph under consideration resides in a union-find structure that reflect the connected components of G.*

For each vertex of the graph that actually is a representative of a component, we maintain a pointer to the corresponding node of the decomposition tree.

The list of siblings maintains the operations of deleting elements and of concatenation of such lists in $O(1)$ time. This can be achieved by a circular doubly-linked list.

The parent pointer might not be up-to-date in the following cases:

1. The parent is V;
2. The parent is a connected component.

We summarize the properties of our data structure in the following lemma:

Lemma 4.2 *A data structure for modular decomposition trees can be maintained in a way that it allows the following operations in $O(f(m,n))$ amortized time, where $f(m,n)$ is the amortized bound on the union-find data structure that is used:*

`Parent` : *For any node we can access its parent in the tree (if it exists) .*

`Delete` : *Any node of a decomposition tree can be deleted at any time.*

`Fuse` : *Any two nodes u and v that are the children of the same parent p can be fused into one.*

Proof:

`Parent` : The absence of up-to-date parent pointers in some of the nodes does not prevent the data structure from finding the parent of an arbitrary node $X \neq V$ efficiently. If X does not have an up-to-date parent pointer, a `Find` operation on its representative produces the component C that contains X. If $X \neq C$, the parent is C; otherwise it is V. Timestamping parent pointers and nodes of the tree when they are created gives a way of detecting whether a parent pointer is out of date, in $O(1)$ time.

`Delete` : Deletion of a node u can be achieved by splicing it out of the doubly-linked list of its siblings, updating the child pointer of the parent p if necessary, and splicing the list of children of u into the list of children at p.

`Fuse` : We first take the lists of children of u and v and concatenate them. We give this new list to u, say, as list of children, by updating the child pointer and the union-find data structure accordingly. Then we may delete the tree node v.

□

4.2 The charging discipline

The following is our *charging discipline* for bounding the cost of operations during the inductive step of Algorithm 4.

1. There are $O(n)$ `Union` operations over an entire run of the algorithm, so the cost of these is accounted for separately, and take $O(n + m\alpha(m, n))$ time, see Cormen et al. (1990).
2. A *charge* consists of a fixed number of $O(1)$ operations and a fixed number of `Find` operations in union-find data structures.
3. We maintain the credit invariant that each node of a decomposition tree carries a credit that can pay for a charge. A node's credit can only be released to pay for operations when it is deleted.
4. Let the *active edges* of G be those edges that are not subsets of $N(v_0)$ or $\overline{N}(v_0)$. Let m' be the number of active edges. We adhere to the discipline that we incur $O(m')$ charges that are not paid for by released credits, and we deposit $O(m')$ new credits in the tree.

Each edge is active in exactly one recursive call in the recursion tree. If we adhere to the charging discipline, it receives $O(1)$ charges in this recursive call, and no charges elsewhere. This gives a bound of $O(m)$ charges over the entire recursion tree. This gives a bound of $O(m)$ `Find` operations, which is $O(n + m\alpha(m, n))$, and $O(m)$ constant-time operations.

In addition, we have an $O(n+m\alpha(m, n))$ preprocessing step. Successfully maintaining the discipline will imply the $O((n + m)\alpha(m, n))$ bound, since an edge only becomes active once, hence the number of fixed-cost operations and `Find` operations will be bounded by the $O(m)$ credits initially available on the edges. However, an $O((n+m)\alpha(m, n))$ bound for any modular decomposition algorithm implies an $O(n+m\alpha(m, n))$ bound, since, if $n > m+1$ in the original graph, we can spend $O(n+m)$ preprocessing time dividing the initial graph into its connected components, solve the problem on each component, where n is $O(m)$, and then make the root of these trees be the children of a new parallel root node, to obtain the decomposition of G.

Let the *active neighbors* of a set X of vertices be those members of $V \setminus X$ that are adjacent to a member of X on an active edge.

Lemma 4.3 *If X is a node of $\mathtt{MD}(G|N(v_0))$ or $\mathtt{MD}\left(G|\overline{N}(v_0)\right)$ that is a module of G, the charging discipline allows us to incur one charge at X for each active neighbor of X.*

Proof: We show that the sum of lengths of active neighbor lists of such nodes is $O(m')$. The sum of lengths of the active neighbor lists of of all leaves of $\mathtt{MD}(G|N(v_0))$ and $\mathtt{MD}\left(G|\overline{N}(v_0)\right)$ is exactly m'. Any internal node that remains a module of G has an active neighbor list that is identical to that of each of its children. Thus, the length of its active neighbor list is at most half of the sum of lengths of active neighbor lists of its children. Let $h(X)$ be the length of the shortest path from X to a leaf. The sum of lengths of neighbor lists of $\{X \mid h(X) = k$ and X is node of $\mathtt{MD}(G|N(v_0))$ or $\mathtt{MD}\left(G|\overline{N}(v_0)\right)$ is a module of $G\}$ is at most $m'/2^k$. Summing this over increasing values of k gives a geometric series that sums to less than $2m'$. □

Corollary 4.4 *The charging discipline allows us to incur a charge at every node of $\mathtt{MD}(G|N(v_0))$ and $\mathtt{MD}\left(G|\overline{N}(v_0)\right)$ that is not a module of G but has an incident active edge.*

Proof: If the node remains a module of G, this follows from Lemma 4.3. Otherwise, the charge is paid for by the credit that is freed when it is `Deleted` during the restrict step of Algorithm 4. (The `Delete` operation refers to the one provided by Data Structure 4.1). □

Note that we may incur a charge at every node of $\texttt{MD}(G|N(v_0))$ and $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$ that has an incident active edge; if the node is a module of G, the charge is covered by Lemma 4.3, and if it is not, the charge can be paid for by the credit that is released if it is `Deleted` during the restrict step. Thus, the only type of node at which we may not spend $O(1)$ time is one that has no incident active edge.

Let X be such a node. X must be a node of $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$, since all nodes of $\texttt{MD}(G|N(v_0))$ have v_0 as an active neighbor. X is represented by a node of Data Structure 4.1 that is returned by the recursive call to $G|\overline{N}(v_0)$. X remains a module of G, so it must be used in the assembly of $\texttt{MD}(G)$ during the inductive step. If the parent Y of X in $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$ has incident active edges, Y is not a module of G, so Y is not a node of $\texttt{MD}(G)$. In this case, X has a different parent in $\texttt{MD}(G)$ than it does in $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$, but our charging discipline does not allow us to incur a charge at X by resetting its parent pointer, or by splicing it into the child list of its new parent. Let us call any node of $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$ that has no incident active edges and a new parent in $\texttt{MD}(G)$ a *problem node.*

There are three tricks that we use for bounding the cost of splicing the problem nodes into the Data Structure 4.1 representation of $\texttt{MD}(G)$:

Trick 1: A list of problem children of a node X can be obtained without violating the charging discipline, as follows. If X remains a module of G, none of its children are problem children. Otherwise, X must have incident active edges. We may incur a charge at X and at each child of X that has an incident active edge, by Corollary 4.4. One-by-one, we splice these children out of the doubly-linked child list of X. What remains is the list of X's problem children. We may spend $O(1)$ time handling this list, since there is only one of them at X. These children become children of a single node W in $\texttt{MD}(G)$, and we can splice them as a group into the child list of W in $O(1)$ time.

Trick 2: If X is a node of $\texttt{MD}\Big(G|\overline{N}(v_0)\Big)$ that gets `Deleted`, and its children are connected components of G, or the new parent of its problem children in $\texttt{MD}(G)$ is a connected component of G, then we do not need to update explicitly the parent pointers of its problem children, by the exceptions we made for Data Structure 4.1.

Trick 3: If the union P of the problem children is their new parent in $\texttt{MD}(G)$, then the node structure x that formerly represented X may be spliced in directly as the node structure p representing P in the representation of $\texttt{MD}(G)$. The parent pointers on the problem children still correctly point to the parent in $\texttt{MD}(G)$, but we do not have to spend any time updating them.

Lemma 4.5 *The three tricks always apply to the problem children of any node that is* `Deleted` *during the inductive step.*

Proof:

X is a parallel node. The problem children are connected components of $G|X$, and, since they have no incident active edges, they are also connected components of G, hence children of the root. Tricks 1 and 2 suffice to splice them into $MD(G)$.

X is a series node with several problem children. Then the union P of problem children becomes the new parent of those same problem children in $\texttt{MD}(G)$, since P is a module of G, and no larger union of children of X is a module of G. Trick 3 suffices to splice them into $MD(G)$.

X is a series node with one problem child, or

X is a prime node. Then $G|X$ is connected, and X is contained in a single connected component of $G|\overline{N}(v_0)$. In $\texttt{MD}(G)$, every vertex in this component has the same least common ancestor W with

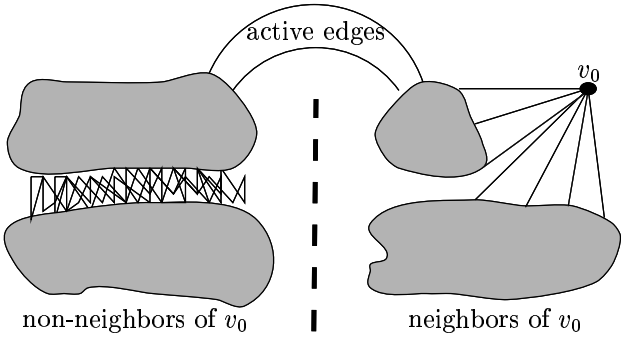

Figure 4: Splitting the graph according to a pivot

v_0, by Lemma 3.4. The problem children of X are maximal subsets of X that remain modules of G. By Algorithm 2, the problem children all become children of W, and the first condition applies. Trick 1 may be used to splice them into $MD(G)$, but not to update their parent pointers.

It remains to show that Trick 2 applies, that is, that the parent pointers of the problem children do not need to be updated. Suppose z is a neighbor of W in $V \setminus W$. It cannot be a member of $\overline{N}(v_0)$, since then it would be in the same connected component of $\overline{N}(v_0)$ as X, and would be contained in W, a contradiction. But z cannot be a member of $N(v_0)$: It is adjacent to every vertex in W, since W is a module in G, hence it is an active neighbor of the problem children of X, which, by assumption, have no incident active edges. Since W has no neighbors in G, it is either a connected component of G or a union of connected components of G. It cannot be a union of connected components of G, since there edges between some of its children, namely between any problem child of X and any child of X that is not a problem child. W is a connected component, hence Trick 2 applies.

□

4.3 Preprocessing

Let an *active group* of edges be the group of edges that becomes active when some recursive call is activated. The active groups are a partition of the edges. Clearly, we may know the active groups after Algorithm 4 has run, but we will find it useful to know them in advance. This requires us to precompute the recursion tree before we actually run Algorithm 4. For this, we run Algorithm 5. See Figure 4 for an illustration.

Algorithm 5: RecursionTree(G)

```
if G has no nodes then return nil;
else
    select v_0 using whatever criterion is used in Algorithm 4;
    r_1 = RecursionTree(G|N(v_0));
    r_2 = RecursionTree(G|N̄(v_0));
    make r_1, r_2 be the left and right children of v_0, respectively;
    return v_0
```

Call this tree the *recursion tree* R. Since the vertices of the tree are vertices of G, the least-common ancestor of an edge of G is well-defined. Two edges are in the same active group iff they have the same least-common ancestor in the recursion tree. There are simple algorithms for finding the least-common

ancestor in R of each edge (u, v) of G. One simple approach, see Cormen et al. (1990), achieves this $O(n+m\alpha(m,n))$, by maintaining a union-find structure during a traversal of the tree. Moreover, once each edge is labeled with its least-common ancestor, we may radix sort all edges with the active edge group as primary sort key, starting vertex as secondary sort key, and ending vertex as tertiary sort key. This gives, for each vertex and for each active group, a sorted list of neighbors of the vertex that are reachable on an edge in the active group. This entire preprocessing step takes $O(n + m\alpha(m, n))$ time.

We may thus assume that a sorted adjacency list of currently active incident edges is available at each vertex in a call to Algorithm 4.

4.4 The restrict step.

We assume by induction that the two recursive calls have returned Data Structure 4.1 for $G|N(v_0)$ and $G|\overline{N}(v_0)$. We must carry out the two steps of Algorithm 3. The first step requires us to `Delete` nodes of $\texttt{MD}\left(G|\overline{N}(v_0)\right)$ that are not modules in G. We do this by working inductively up the tree, starting at the leaves that have incident active edges. As a base case, leaves are vertices of G, and for each of them, the neighbors in $\overline{N}(v_0)$ are given by the sorted lists of active edges. If the children of an internal node X are modules and carry identical active neighbor lists, mark X as a module, and give X a copy of the active neighbor list of one of the children. Otherwise, mark X for deletion. This procedure marks for deletion exactly those members of $\texttt{MD}\left(G|\overline{N}(v_0)\right)$ that are not modules of G and that have incident active edges. We touch only nodes of the tree that have leaf descendants with incident active edges. All untouched nodes are modules of G. Thus, all nodes of the tree that are not modules in G are explicitly `Deleted`. In the process, each node gets a list of its children that are deleted, and a list of its children that are not deleted and have incident active edges. The conformity with the charging discipline is immediate, by Lemma 4.3.

For the second step of Algorithm 3, it suffices to identify for each deleted degenerate node those groups of children that are not deleted, but that have identical neighbor lists. If Y is a degenerate node, we remove deleted children and children with incident active edges from its doubly-linked list of children. The remaining list gives those children with no incident active edges. If there is more than one child in this list, then the union of these children is a member U of $\mathcal{M}(G, v_0)$, so we create a new node representing U, and make this list be its list of children. This takes $O(1)$ time, which we count as part of touching Y. We group those undeleted children that have nonempty neighbor lists using the radix partitioning procedure (Section 2.2), on the neighbor lists carried by those children, and create a parent for each group with more than one child. This takes time proportional to the number of elements in those neighbor lists, and thus does not affect the time bound of the algorithm.

Thus, the number of members of $\mathcal{M}(G, v_0)$ that are the union of children of a degenerate node u is at most one plus the number of active edges incident to the set of vertices that u represents. The same cannot be said of a prime node. Thus, if a prime node is deleted, we may not individually touch those members of $\mathcal{M}(G, v_0)$ that it contains and that have no incident active edges. However, we may touch all other children. For any deleted prime node p, we may obtain a doubly-linked list of the children with no incident active edges by removing all other children from its doubly-linked list of children. Call this doubly-linked list of problem children an *untouchable list*.

The restriction of the decomposition to $N(v_0)$ is found in the same way. The next remark follows from the fact that we get at most one untouchable list for each deleted node, and the observation that a node is only deleted if it has an incident active edge.

Remark 4.6 *At the end of the restrict step, the only vertices without current parent pointers are members of $\mathcal{M}(G, v_0)$. There are $O(1 + m')$ untouchable lists, and $O(1 + m')$ members of $\mathcal{M}(G, v_0)$ that are not in untouchable lists, where m' is the number of active edges.*

4.5 The v_0 modules step.

We define an equivalence relation on vertices of $\overline{N}(v_0)$, where for $x, y \in V$, xKy if and only if:

- x and y are contained in the same connected component of $G\left|\overline{N}(v_0)\right.$, or
- their connected components are both contained in a module of G that is a subset of $\overline{N}(v_0)$.

Clearly, K is an equivalence relation, so this gives a partition of $\overline{N}(v_0)$; let us call the partition classes the *basic blocks* of $\overline{N}(v_0)$.

The basic blocks of $N(v_0)$ are computed in a complementary way: xKy if and only if:

- x and y are contained in the same *co-component* or
- their co-components are both contained in a module of G that is a subset of $N(v_0)$,

Let $\mathcal{B}$ denote the basic blocks of $N(v_0)$ and let $\mathcal{B}'$ denote the basic blocks of $\overline{N}(v_0)$.

The recursive call to $\overline{N}(v_0)$ has computed the connected components of $\overline{N}(v_0)$ as a union-find structure. The restrict step in the main call has computed the maximal modules of G that are contained in $\overline{N}(v_0)$. These two structures give an implicit representation of the basic blocks of $\overline{N}(v_0)$.

The basic blocks of $N(v_0)$ are defined in a similar way, but it is not necessary to maintain a special union-find data structure for them. We have an active edge to each vertex in $N(v_0)$. If there are more than one basic block, they are the children of the root, and, since these have incident active edges from v_0, we may touch and update a block label for every member of them during the v_0-modules step.

To find strong modules containing v_0, we define a directed bipartite graph $\mathcal{F} = (\mathcal{B}, \mathcal{B}', E_{\mathcal{F}})$. For $B \in \mathcal{B}$ and $B' \in \mathcal{B}'$, (B, B') is an edge of $\mathcal{F}$ if and only if there is an edge of G between B and B', and (B', B) is an edge of $\mathcal{F}$ if and only if there is a non-edge between B and B'. We find the strongly-connected components of $\mathcal{F}$, and the *component graph*, which has one node for each strong component of $\mathcal{F}$, and edges telling which which strongly-connected components are reachable from which on a single edge of $\mathcal{F}$, see Cormen et al. (1990).

Lemma 4.7 *There is a unique topological sort of $\mathcal{F}$. A set containing v_0 is an ancestor of v_0 if and only if it is a union of $\{v_0\}$ and the members of strong components in a suffix of that sort.*

Proof: We show that a set X is a strong module containing v_0 if and only if it is a union of $\{v_0\}$ and basic blocks, and there is no edge of $\mathcal{F}$ from a basic block X to a basic block in $V \setminus X$. Since the strong modules containing v_0 are totally ordered by the inclusion relation, this establishes the claim.

Observation 4.8 *Let X be a set that contains v_0. Since v_0 is adjacent to every member of $N(v_0)$ and nonadjacent to every member of $\overline{N}(v_0)$, X is a module if and only if every member of X is adjacent to every member of $N(v_0) \setminus X$ and nonadjacent to every member of $\overline{N}(v_0) \setminus X$.*

Observation 4.9 *Every module containing v_0 is a union of v_0 and zero or more connected components of $\overline{N}(v_0)$ and co-components of $N(v_0)$.*

This follows immediately from Observation 4.8.

Observation 4.10 *If X is a union of v_0 and basic blocks, it is a module if and only if there is no edge of $\mathcal{F}$ from a basic block in X to a basic block not in X.*

To see this, note that if X is a union of v_0 and basic blocks, then it is also a union of connected components of $G\left|\overline{N}(v_0)\right.$ and co-components of $G\,|N(v_0)$. Every member of $\overline{N}(v_0) \cap X$ is nonadjacent to every member of $\overline{N}(v_0) \setminus X$, and every member of $N(v_0) \cap X$ is adjacent to every member of $N(v_0) \setminus X$. The observation then follows from Observation 1 and the definition of $\mathcal{F}$.

It remains to show that a module X that contains v_0 is a strong only if it is a union of $\{v_0\}$ and zero or more basic blocks, and a weak only if it is not such a union. Suppose X is not such a union. By Observation 2 and the definition of basic blocks, X overlaps a maximal module of G that is contained in $N(v_0)$ or $\overline{N}(v_0)$, and is therefore not strong. Suppose X is such a union. Since all modules not containing v_0 are contained in $N(v_0)$ or $\overline{N}(v_0)$, they are contained in basic blocks. Any module Y that overlaps X must contain v_0. By Lemma 2.1, $(X \setminus Y) \cup (Y \setminus X)$ is a module that overlaps X, a contradiction, so Y does not exist, and X is strong. □

For the time bound, note that we may touch each component or co-component that has an incident active edge. We may also touch those that merge with others, since there are $O(n)$ merges over a run of the algorithm. Co-components with no incident active edges merge with some other co-component, so all co-components may be touched. Components that have no incident active edges cannot be touched, but their union is a module that does not contain v_0, hence they are hidden inside a basic block, and reside under a single root of the forest produced by the restrict step. Since this block is unique in each of the $O(n)$ incarnations of the recursive algorithm, it may be touched.

$\mathcal{F}$ is too large to compute explicitly, so we work with a mixed representation where the set of neighbors of $\mathcal{B}$ is represented in a standard way and the set of neighbors of $\mathcal{B}'$ is represented with its complement. The number of edges and non-edges given explicitly in this representation is clearly bounded by the number of currently active edges. Since each component and co-component knows the number of active edges to other components, we may compute the mixed representation of the forcing graph $\mathcal{F}$ with a number of `Find` and constant-time operations proportional to the number of active edges. Creating the strong components and their topological sort then follows in time proportional to the size of this mixed representation by Theorem 2. This completes the v_0-modules step, by Lemma 4.7.

4.6 The assemble step.

Lemma 4.11 *If X is a child of an ancestor A of v_0 in $\mathtt{MD}(G)$, and X has no incident active edges, then either A or X is a connected component of G.*

Proof: We claim that if M is an ancestor of v_0 that has a member x with no incident active edges, then M has no neighbors in $V \setminus M$. This is obvious if $M = V$. Suppose $M \neq V$. All vertices of $N(v_0)$ have active edges to v_0. Thus, M must intersect $\overline{N}(v_0)$, and all vertices of M without active edges reside in $\overline{N}(v_0)$. If $N(v_0) \setminus M$ is nonempty, let y be a member of $N(v_0) \setminus M$. Since y is adjacent to v_0, it is adjacent to every member of $M \cap \overline{N}(v_0)$ via active edges, a contradiction. So $N(v_0)$ is a subset of M. There is no edge from v_0 to $\overline{N}(v_0)$, and since $v_0 \in M$ and M is a module, there is no edge from M to $\overline{N}(v_0) \setminus M$. Since M contains v_0 and $N(v_0)$, $\overline{N}(v_0) \setminus M = V \setminus M$, proving the claim.

Thus, A must have no neighbors in $V \setminus A$. If $A \neq V$, A is a connected component, and the lemma is satisfied. If $A = V$ and G is connected, then it is once again satisfied. Otherwise, A is a parallel node, and X is a connected component, and, again, the lemma is satisfied. □

The v_0-modules step gives for each basic block B the smallest ancestor A_B of v_0 that contains it. B is a union of a set $\mathcal{X}$ of members of $\mathcal{M}(G, v_0)$, each of which appears as the root of a tree in the forest produced by the restrict step. To implement Algorithm 2 without its inner **if**-statement, we must establish the members of $\mathcal{X}$ as children of A_B. If $\mathcal{X}$ has only one member, then this member is B, and we simply attach it as a child of A_B in $O(1)$ time. Otherwise, the block containing $X \in \mathcal{X}$ is just the connected component of $G|\overline{N}(v_0)$ or the co-component of $N(v_0)$ that contains it. If X has an incident active edge, then by Data Structure 4.1, we can perform a `Find` operation to find its block, and the minimum ancestor of v_0 that contains the block gives the parent in $O(1)$ time. We then establish X as a child of this ancestor, complete with an updated parent pointer, in $O(1)$ time. The remainder of members of $\mathcal{M}(G, v_0)$ reside in untouchable lists. Since each untouchable list resides in

one connected component of $G|\overline{N}(v_0)$, we perform a single *find* for each untouchable list to find its basic block. This identifies the single basic block that contains them. If the list only has one member, we add it as a child of the parent, complete with a parent pointer. Otherwise, we splice the untouchable list into its parent's child list in $O(1)$ time. We needn't update the parent pointers of the elements of the untouchable list, since, by Lemma 4.11, the parent is the connected component of G that contains them, and no parent pointer is required under Data Structure 4.1.

Next, we need to simulate the effect of the inner **if**-statement of Algorithm 2. We may spend $O(1)$ time at each ancestor of v_0, since these were created during the inductive step. If X is a parallel ancestor of v_0 with at least three children, and Y is its child that contains v_0, then the union of all siblings of Y is a single member Z of $\mathcal{M}(G, v_0)$. Thus, in the current tree, X will have two children, Y and Z. We must delete Z from the tree, and let X inherit its children. If $X = V$, we just splice Z's doubly-linked list of children into those of X's. The children of X are connected components of G, so Data Structure 4.1 does not require parent pointers. Otherwise, by Lemma 4.11, each child of X in $\texttt{MD}(G)$ has an incident active edge, so we may touch each of them, and make the parent pointer point to X. The procedure when X is a series node is similar.

To complete Data Structure 4.1 for $\texttt{MD}(G)$, we must also produce the union-find structures for G. The recursive calls return union-find structures that give the connected components and co-components of $G|N(v_0)$ and $G|\overline{N}(v_0)$. Since $\{v_0\} \cup N(v_0)$ is a connected component, and all active edges are incident to it, we must do a union on all components of $G|N(v_0)$ and $G|\overline{N}(v_0)$ that have incident active edges. Identifying these components requires a *find* on the end vertices of each active edge. No other components are affected. For the co-components, symmetric reasoning shows that all co-components of $\{v_0\} \cup \overline{N}(v_0)$ must be subsets of a single co-component. A co-component C of $G|N(v_0)$ must be merged together with this set if the number of incident active edges that go to $N(v_0)$ is less than $|N(v_0)|$.

5 Obtaining a Linear Time Bound

The only bottleneck preventing a linear time bound that remains is the $\alpha(m, n)$ term. We will show how to circumvent this factor by some preprocessing on the recursion tree.

One place where we incurred the term was in finding the least-common ancestors of edges in the recursion tree. This can be eliminated here by using the off-line least-common ancestors algorithm of Harel and Tarjan (1984). The other place was in making the connected components of the induced subgraph passed to a recursive call available in that call. These components are represented with a union-find structure in the tree data structure in Lemma 4.2. We solve this problem by restricting the possible choices for the recursion tree, so that the union-find structures can be managed without incurring the $\alpha(m, n)$ factor.

When running Algorithm 5 on vertex set X, let us number the vertices that have already been selected as the initial pivot in prior calls, in the order in which they were selected. This numbering constitutes a preorder numbering of the portion of the recursion tree already computed. No vertex in X has yet been numbered. Let the *recency* of a vertex x in X be the maximum of the preorder numbers among neighbors of x that have already been numbered. If x has no such neighbor, its recency is defined to be -1. Our modification to Algorithm 5 is to select v_0 to be the vertex with greatest recency, rather than selecting it arbitrarily.

Algorithm 6 gives the algorithm for computing a recursion tree R satisfying this constraint.

The correctness follows from the following points, which are trivial to establish by induction on the number of vertices already selected:

1. There is one doubly-linked list for each recursive call that has not yet been initiated, but whose parent has already been initiated, and this list gives the set of vertices that will be passed to the call.

Algorithm 6:

Input: A doubly-linked list L representing subset X of vertices of G, ordered by their recency. Each member of L carries a label that identifies it as a member of L

Output: a recursion tree R.

Remove the first vertex v_0 from L;
Create an empty list L_1 that will hold the neighbors of v_0;
foreach $y \in N(v_0)$ **do**
  if *y has not been selected* **then**
    if *y is a member of L* **then**
      remove y from L, put it into L_1, and label it accordingly
    else move y to the front of the list L' that currently contains it
Let R_1 be the tree obtained by recursion on L_1;
// The call to R_1 may have modified the order of vertices in L;
Let R_2 be the tree obtained by recursion on L;
Make R_1 and R_2 children of v_0;
return v_0.

2. The order of each linked list is descending in order of recency.

Together, these two invariants imply that the vertex v_0 selected as the initial pivot when a call is initiated has greatest recency among those vertices passed to the recursive call.

Lemma 5.1 *Algorithm 6 runs in linear time.*

Proof: All doubly-linked list operations take $O(1)$ time, and can be charged to an edge of v_0. Aside from the time required by the recursive calls it generates, the call requires $O(1 + N(v_0))$ time. This gives an $O(n + m)$ bound for all recursive calls, since each vertex of G has the role of v_0 in only one call. □

Let V_v denote the subset of vertices that are leaf descendants of that node, including v itself. The recursion tree R has an important property with respect to the connectivity structure of G:

Lemma 5.2 *Let $v \in V$ be a vertex and C a connected component of $G|V_v$. Then there is exists $w \in C$ that is ancestor to all other members of C in R.*

Proof: We proceed by induction from the bottom of R to the top. If $v \in C$, the claim is clearly satisfied by v. Otherwise, $C \subseteq \overline{N}(v)$, C is a connected component of $\overline{N}(v)$, and $\overline{N}(v)$ is the set of nodes in T_x, where x is one of the two children of v. By the inductive hypothesis, applied to T_x, the claim holds for C. □

Definition 5.3 *Let S be a tree defined on the vertex set V and R a recursion tree. We say that S is* coherent *with R and G if for every $v \in V$ and every connected component C of $G|V_v$ the restriction $S|C$ is connected.*

Observe that R is not necessarily coherent to itself. If we are able to compute such a coherent tree, it fulfills the necessary conditions to apply Theorem 1, and thus to perform all union-find operations in amortized linear time. Thus, the problem of eliminating the α factor reduces to computing a coherent tree for R and G.

There are many trees on G that are coherent with a given R. We will compute a unique one that is defined as follows.

Definition 5.4 *Let R be the recursion tree of graph and $v \in V$. We say that v is* subordinate *to some other vertex $w \in V$ iff*

- *w is an ancestor of v in R*
- *v and w are joined by a path in $G|V_w$.*

For each $v \in V$ that is not the root of R there is a unique minimal $w \in V$ such that v is subordinate to w. We call this unique node w the *boss* of v. The boss relation is the parent relation in a tree S on V, which we call the *subordinate tree* of R.

Lemma 5.5 *The subordinate tree S is coherent with R.*

Proof: We proceed by induction from the bottom of R to the top. Let v be some vertex of G and C a connected component of $G|V_v$ and assume that the claim is given for all V_w for $w \in V_v$, $w \neq v$. We have to show that $S|C$ is connected.

Let $y_0 \in C$ be the vertex in C with earliest preorder number. By Lemma 5.2, y_0 is an ancestor of the other members of C. If $y_0 \neq v$ the induction hypothesis shows the claim.

So suppose now that $y_0 = v$. Set G' to be the graph that is obtained by removing v and all edges that are active at v from $G|C$. G' might not be connected. Let $C_1, \ldots, C_k$ be the new components. Each C_i is completely contained in either $N(v)$ or $\overline{N}(v)$. By Lemma 5.2, C_i has a vertex y_i that is an ancestor of all other members of C_i. C_i is a connected component in $G|V_{y_i}$. Thus, by the induction hypothesis we may assume that $S|C_i$ is connected. By definition y_i is below v in R, and y_i and v are joined by a path in C and thus in $G|V_v$. So y_i is subordinate to v.

We show that v is, in fact, the boss of y_i. Suppose this is not the case. Then there must be some vertex w that is a descendant of y_i and an ancestor of v to which y_i is subordinate. But then $w \in C_i$ and y_i is not the highest node in C_i, a contradiction.

So in S, all vertices y_i are direct children of v, and v is connecting all the $S|C_i$. □

Algorithm 7: Compute a recursion tree R and its subordinate tree for a connected graph G.

Let R be a recursion tree computed with Algorithm 6;
Label each edge (x, y) of G with the least-common ancestor $lca(x, y)$ in R;
foreach *vertex v of G* **do**
  Let S be the neighbors of v with earlier preorder number than v;
  Assign $b = \max\{lca(v, s) : s \in S\}$ as boss of v;

Lemma 5.6 *Algorithm 7 is correct.*

Proof:

If a and b are two vertices, we say that $a < b$ if the preorder number of a is less than that of b.

Let u be an arbitrary vertex. Suppose the algorithm assigns no boss to u. Then u has no neighbor x such that $x < u$. It follows that u is not in the left subtree of any ancestor a of u, since then it has $a < u$ as a neighbor. Thus, every vertex of G is either a descendant of u or has earlier preorder number than u. If a descendant d of u is a neighbor of $c < u$, then d would have been selected before u, a contradiction. The vertices that are descendants of u are a union of one or more connected components of G. Since G is assumed to be connected, u must be the root, so failure of the algorithm to assign a boss to it is correct.

Let us now assume that w is the boss of u and b is the vertex the algorithm assigns as the boss of w. We must show $b = w$. It suffices to show that $b \leq w$ and $b \geq w$.

To show $b \leq w$, we let (u, x) be any edge from u to a vertex with lower preorder number than u. If $y = lca(u, x) > w$ then u or x is in the left subtree of y, and a neighbor of y. Either (y, u) or (y, x, u)

is a path, and y supersedes w status as boss of u, a contradiction. Thus, $y \leq w$, and, since b is such a y, $b \leq w$.

We now show that $w \leq b$. If u is a neighbor of w, $lca(u,w) = w$, establishing that $w \leq b$. If u is not a neighbor of w, u is in w's right subtree. Let P be a path in T_w from w to u. Since w is the boss of u, P must exist. The second vertex of P is in w's left subtree, since it is a neighbor of w. Let c be the last vertex of P that is not a member T_u, and let d be its successor on the path. Since u was selected before d, u has a neighbor z such that $c \leq z < u$. This establishes that z is in T_w, and $w \leq lca(w,z) \leq b$. □

We have already shown that the first step of Algorithm 7 is linear. We use the least-common ancestors algorithm of Harel and Tarjan (1984) to compute the least common ancestors of the edges in R. Once this is accomplished, each vertex x may be labeled with its boss in $O(|N(x)|)$ time, which gives an $O(n+m)$ bound for computing the subordinate tree.

This concludes the proof of the linear time bound of the modular decomposition algorithm on connected graphs. If G is not connected, its connected components can be found in linear time, and the modular decomposition algorithm applied to each component. Making the resulting trees children of a parallel node completes the modular decomposition of G.

As described, the algorithm to compute R and S requires two passes, first to run Algorithm 6, and then to use Harel and Tarjan to find the least-common ancestor of each edge in the resulting tree, and finally, to use the least-common ancestors of the edges to compute S. The least-common ancestors of the edges are also required by the modular decomposition algorithm, to organize the edges into groups that are active at the same time. Algorithm 6 can be modified to produce all of this information in a single pass. The modified version is given as Algorithm 8.

Algorithm 8:

Input: A doubly-linked list L representing a set X of vertices of G that are passed to a recursive call, ordered by their recency. Each member of L carries a label that identifies it as a member of L, and L is labeled with a *generator*, which is the initial pivot in the parent of the recursive call that it represents.

Output: A *parent* labeling giving a recursion tree R, a labeling of each edge with its least common ancestor in R, and a *boss* labeling giving a tree S that is coherent with R and G

```
Remove the first vertex v_0 from the front of L;
Create an empty list L_1 that will hold the neighbors of v_0;
Assign v_0 as the generator of L_1 and of L;
foreach y ∈ N(v_0) do
    if y has not been selected then
        if y is a member of L then
            remove y from L;
            put y into L_1 and relabel it as a member of L_1;
            provisionally assign v_0 as the boss of y;
            assign v_0 as the least-common ancestor of (v_0, y)
        else move y to the front of the list L' that currently contains it;
        provisionally assign the generator of L' as the boss of y;
        assign the generator of L' as the least-common ancestor of (v_0, y)
Let R_1 be the tree obtained by recursion on L_1;
// The call to R_1 may have modified the order of vertices in L;
Let R_2 be the tree obtained by recursion on what remains of L;
Make R_1 and R_2 children of v_0;
return v_0.
```

Lemma 5.7 *Algorithm 8 is correct.*

Proof: The recursion tree returned is the same as the one returned by Algorithm 6. Each vertex is chosen as the initial pivot v_0 in some recursive call. Let (x, y) be an arbitrary edge, and let u be its least-common ancestor. Assume without loss of generality that x is chosen before y. It must be that $x \in N(u)$ and $y \in \overline{N}(u)$. Let L and L_1 be the lists with those names in the recursive call where u is the initial pivot. Then x is moved to L_1, y remains in L, and u is the generator of L. L is not touched between the time when the recursive call on L_1 is generated and when it completes. During this time, x is selected as the initial pivot v_0 in some lower recursive call. At this time, the generator of L, which is u, is correctly assigned as the least-common ancestor of (x, y). When y is selected later, there is no reassignment of a least-common ancestor to (x, y); since x has already been selected, the edge is ignored. We conclude that each edge gets assigned its correct least-common ancestor.

The boss of y is the maximum preorder number of the least-common ancestors of the set $\{(x, y) : (x, y)$ is an edge of G and x has earlier preorder number than $y\}$. By the foregoing, these are the edges incident to y whose least-common ancestor has already been assigned when y is selected. Each time such an edge was assigned a least-common ancestor u, y was given u as its boss. The preorder number of the generators of the lists containing y are increasing as the algorithm proceeds, so the last boss label assigned to y is the one with largest preorder number, as desired. No boss is subsequently assigned to y, since it has already been selected. □

References

[Buer and Möhring, 1983] Buer, B. and Möhring, R. H. (1983). A fast algorithm for the decomposition of graphs and posets. *Mathematics of Operations Research*, 8:170–184.

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

[Corneil et al., 1985] Corneil, D. G., Perl, Y., and Stewart, L. K. (1985). A linear recognition algorithm for cographs. *SIAM J. Comput.*, 3:926–934.

[Cournier and Habib, 1993] Cournier, A. and Habib, M. (1993). An efficient algorithm to recognize prime undirected graphs. In Mayr, E. W., editor, *Graph-theoretic concepts in computer science, 18th international workshop, WG '92*, volume 657 of *Lecture Notes in Computer Science*, pages 114–122. Springer Verlag.

[Cournier and Habib, 1994] Cournier, A. and Habib, M. (1994). A new linear algorithm for modular decomposition. In Tison, S., editor, *Trees in Algebra and Programming, CAAP '94, 19th International Colloquium*, volume 787 of *Lecture Notes in Computer Science*, pages 68–82. Springer Verlag, Edinburgh, UK.

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

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

[Ehrenfeucht et al., 1994] Ehrenfeucht, A., Gabow, H. N., McConnell, R. M., and Sullivan, S. J. (1994). 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.

[Ehrenfeucht and Rozenberg, 1990] Ehrenfeucht, A. and Rozenberg, G. (1990). Theory of 2-structures, part 1: Clans, basic subclasses, and morphisms. *Theoretical Computer Science*, 70:277–303.

[Gabow and Tarjan, 1985] Gabow, H. N. and Tarjan, R. E. (1985). A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221.

[Golumbic, 1977] Golumbic, M. C. (1977). The complexity of comparability graph recognition and coloring. *J. Combin. Theory Ser. B*, 22:68–90.

[Gustedt, 1998] Gustedt, J. (1998). Efficient union-find for planar graphs and other sparse graph classes. *Theoret. Comput. Sci.*, 203(1):123–141.

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

[Harel and Tarjan, 1984] Harel, D. and Tarjan, R. (1984). Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355.

[McConnell, 1995] McConnell, R. M. (1995). An $O(n^2)$ incremental algorithm for modular decomposition of graphs and 2-structures. *Algorithmica*, 14:229–248.

[McConnell and Spinrad, 1994] McConnell, R. M. and Spinrad, J. P. (1994). 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.

[McConnell and Spinrad, 1998] McConnell, R. M. and Spinrad, J. P. (1998). Ordered vertex partitioning. submitted.

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

[Möhring, 1985] Möhring, R. H. (1985). Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operations Research*, 4:195–225.

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

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

[Steiner, 1982] Steiner, G. (1982). *Machine scheduling with precedence constraints.* PhD thesis, University of Waterloo, Waterloo, Ont.

[Tarjan, 1983] Tarjan, R. E. (1983). *Data structures and network algorithms.* Society for Industrial and Applied Math., Philadelphia.

[Valdes et al., 1982] Valdes, J., Tarjan, R. E., , and Lawler, E. L. (1982). The recognition of series-parallel digraphs. *Siam J. Comput.*, 11:299–313.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399