Efficient and Practical Algorithms for Sequential Modular Decomposition

Elias Dahlhaus* Jens Gustedt† Ross M. McConnell‡

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 practical algorithm with an $O(n + m\alpha(m, n))$ time bound and a variant with a linear time bound.

1 Introduction

Modular decomposition is a recursive tree-like partition of a graph into disjoint induced subgraphs. It has been used extensively on the problem of orienting a graph so that the resulting digraph is is a poset relation (Gallai (1967), Golumbic (1980), McConnell and Spinrad (1999)). It is possible to solve a number of NP-hard problems on the graph by solving them recursively on the children of the root and then combining these results to obtain a solution to the original problem. Though the time for the combination step is exponential in the number of children of the root, this strategy gives linear time bounds when the decomposition tree is of bounded degree. In some well-known classes of graphs, the tree has bounded degree (Corneil et al. (1985), Habib and Möhring (1987)). Problems that are amenable to this type of strategy include maximum independent set, chromatic number, and a number of NP-hard problems on partial orders (Möhring and Radermacher (1984), Habib and Möhring (1987)). An improvement to linear time when the degree of the decomposition is bounded can be obtained for problems with known polynomial bounds, such as recognition of poset relations, using a similar strategy. See Möhring (1985a), Möhring (1985b), and Möhring and Radermacher (1984) for surveys of applications.

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

*Dept. of Computer Science and Dept. of Mathematics, University of Cologne, Cologne, Germany , dahlhaus@suenner.informatik.uni-koeln.de

†LORIA and INRIA Lorraine, campus scientifique, BP 239, 54506 Vandœuvre lès Nancy, France, Jens.Gustedt@loria.fr. Part of this work was done while Jens Gustedt was employed by the Technische Universität Berlin.

‡Department of Computer Science and Engineering, University of Colorado at Denver, Denver, CO 80217-3364 USA, rmcconne@carbon.cudenver.edu.

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.

A parallel algorithm that runs in $O(\log^2 n)$ time on a CRCW PRAM with a linear number of processors is given in Dahlhaus (1995). This requires only $O(mlog^2n)$ work. This can be simulated on an $O(n + m)$ EREW PRAM of the same size with slowdown factor of $O(\log n)$, which, in turn, can be simulated on a single-processor machine with a slowdown factor of $O(n + m)$. Thus, this algorithm immediately gives an $O((n + m) \log^3 n)$ sequential algorithm.

In this paper, we show how to obtain $O(n+m\alpha(m, n))$ and $O(n+m)$ sequential bounds using Dahlhaus's strategy. The $O(n+m\alpha(m, n))$ variant is the more practical. Though some of the tricks we use to achieve the linear time bound are moderately challenging, this gives a conceptually simpler linear-time sequential algorithm than was previously available. For example, the data structures tricks used by the algorithm of McConnell and Spinrad (1999) are more varied and less intuitive. That algorithm moves through different "phases", each which is involved and requires a different set of tricks. The structure produced at each phase is characterized by a set of non-obvious properties that are useful in the next phase. Though the paper describing it is lengthy, it omits some challenging implementation details that were described earlier in Spinrad (1992). By contrast, the algorithm that we give here recursively develops the modular decompositions of two induced subgraphs, which are then spliced together to produce the modular decomposition of the whole graph. How the trees must be spliced is easy to work out. Most of the difficulty lies in developing low-level data structures to allow the splicing operations to be carried out efficiently.

2 Preliminaries

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 X. 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 $\overline{N}(v)$.

Let X be a set of vertices, and let y be a vertex in $V \setminus X$. Then y is a *non neighbor* of X if it is not adjacent to any vertex in X, a *strong neighbor* of X

if it is adjacent to all vertices in X, and a *weak neighbor* of X if it is adjacent to at least one vertex in X and nonadjacent to at least one vertex in X. A *module* of an undirected graph $G = (V, E)$ is a set X of vertices that has no weak neighbors. That is, X is a module iff every member of $V \setminus X$ is either adjacent to all of X or nonadjacent to all of X.

V, $\emptyset$ and the singleton subsets of V satisfy the definition of a module, and are the *trivial modules* of G. A graph is *prime* if it has only trivial 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 (see Möhring (1985b)) *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 (see Möhring (1985b)) *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 parts 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 $G/\mathcal{P}$. By Lemma 2.2, $G/\mathcal{P}$ completely specifies those edges of G that are not subsets of a single partition class (see Figures 1, 2 and 3).

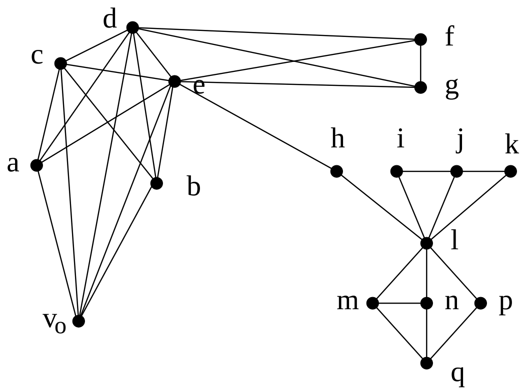

Figure 1: An example.

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

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

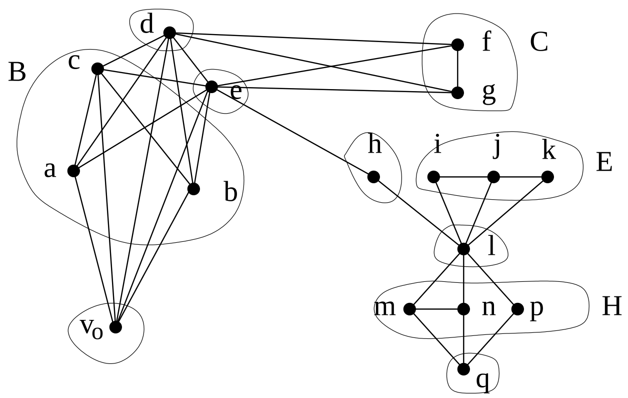

Figure 2: A partition of the vertices of the graph of Figure 1 into modules. Such a partition is called a *congruence partition.* The congruence partition depicted here is one that is denoted $\mathcal{M}(G, v_0)$, and of particular interest in this paper. It consists of $\{v_0\}$ and the maximal modules of G that do not contain v_0.

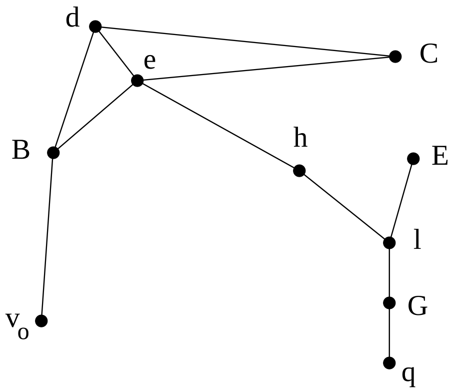

Figure 3: If $\mathcal{P}$ is a congruence partition, the quotient $G/\mathcal{P}$ is the quotient graph whose vertices are the partition classes, and whose edges indicate which pairs of classes are adjacent. This figure illustrates the quotient $G/\mathcal{M}(G, v_0)$ induced by the congruence partition of Figure 2.

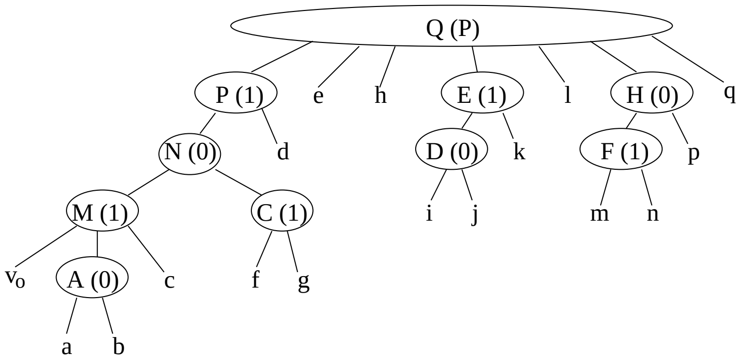

Figure 4: The modular decomposition tree of the graph of Figure 1. Parallel nodes are labeled (0), series nodes are labeled (1), and prime nodes are labeled (P).

Lemma 2.3 (see Möhring (1985b)) *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 (see Möhring (1985b)) *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 4).

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, if x is a node of the tree we will identify x and the set X of vertices that it represents. (X is given by the leaf descendants of 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 Y of X gives the quotient produced by the recursive call on X.

The subset relation on the modules is a partial order. V is the only maximal member of this relation. We will let the *highest submodules* be those modules that are proper subsets of V, but not proper subsets of any other module. It is not always the case that the highest modules are disjoint. For instance, in the subgraph induced by $\{v_0, a, b, c\}$ in the graph of Figure 1, $\{v_0, a, b\}$, $\{v_0, c\}$, and $\{a, b, c\}$ are the highest submodules. However, when G and its complement are both connected, the highest modules are a partition. Algorithm 1 gives the details.

Algorithm 1: `MD`(G) (Defines the modular decomposition of 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 with 0;
  for each connected component G', make `MD`(G') a child of r
else if *$\overline{G}$ is disconnected* **then**
  label r with 1;
  for each connected component G' of $\overline{G}$, make `MD`(G') a child of r
else
  label r with 'P';
  for each highest submodule X, make `MD`$(G|X)$ a child of r
return r

A *parallel node* is a node labeled 0 by Algorithm 1, and a *serial node* is one labeled 1. A *degenerate node* is a parallel or series node. 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.

3 A Strategy for Efficient Modular Decomposition

Our approach uses elements from a previous strategy, see Ehrenfeucht et al. (1994), which we describe first. That algorithm has an $O(n^2)$ time bound. The algorithm of McConnell and Spinrad (2000) uses a similar approach to get an $O(n + m \log n)$ bound.

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 (see Möhring and Radermacher (1984)) *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, which is shown in Algorithm 2.

Algorithm 2: Ehrenfeucht et al. (1994)

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

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

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 the fact that all nontrivial modules in this structure contain v_0. The main obstacle to a linear time bound is finding $\mathcal{M}(G, v_0)$, by using a technique called "vertex partitioning". A linear-time algorithm for vertex partitioning given in McConnell and Spinrad (1999), but computing the modular decomposition by a more involved strategy is a subproblem in deriving the solution, so this is not helpful for finding a simple algorithm.

The approach of Dahlhaus (1995) sidesteps the need for vertex partitioning. The strategy calls for computing $\mathtt{MD}(G|N(v_0))$ and $\mathtt{MD}\big(G|\overline{N}(v_0)\big)$ by recursion, and then finding $\{\mathtt{MD}(G|X) \mid X \in \mathcal{M}(G, v_0)\}$. The strategy is summarized in Algorithm 3.

Algorithm 3: 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 $G|\overline{N}(v_0)$.
- **restriction step:** Using these trees, find $\mathcal{M}(G, v_0)$, and, for each $X \in \mathcal{M}(G, v_0)$, the modular decomposition of $G|X$ (Figure 6).;
- **v_0-modules step:** Compute the modular decomposition of $G/\mathcal{M}(G, v_0)$ (Figures 7 and 8);
- **assembly step:** Assemble these modular decompositions to form the modular decomposition of G as in Algorithm 2. (Figure 5)

.

A subset of $Z \subseteq Y \subset V$ is a module of G iff it is a module of $G|Y$ and has no

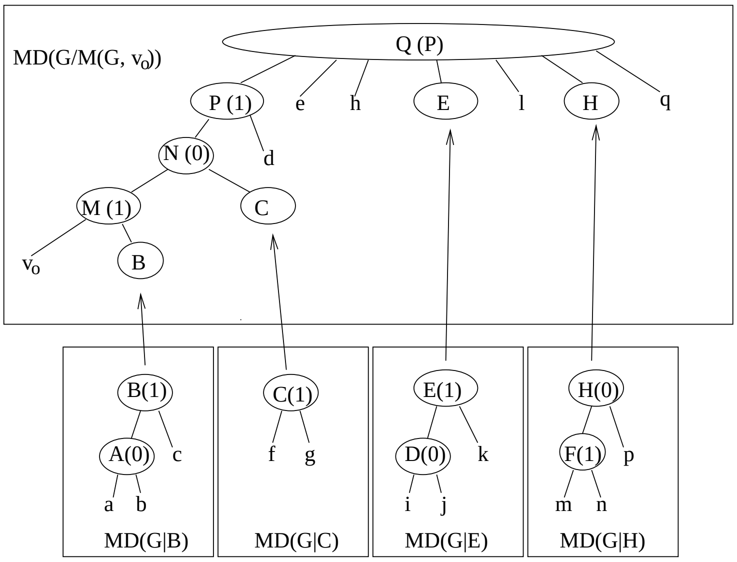

Figure 5: The strategy of the algorithm of Ehrenfeucht et al. (1994), applied to the graph of Figure 1. This is also used in the assembly step of Algorithm 3. The upper figure shows the modular decomposition of the quotient $G/\mathcal{M}(G, v_0)$ depicted in Figure 3. The lower figure shows the modular decompositions of the subgraphs induced by the nontrivial members of $\mathcal{M}(G, v_0)$. The modular decomposition is essentially obtained by replacing the nontrivial leaves of the upper figure with the corresponding decompositions from the lower figure. However, this sometimes results in a parent and child that are both labeled as 1-nodes or both labeled as 0-nodes. During a cleanup step, these cases are removed by removing the child and letting the parent adopt its children. For instance, in this figure M and B become 1-labeled nodes, B is removed, and M adopts its children, yielding the modular decomposition depicted in Figure 4.

weak neighbors in $V \backslash Y$. This simple observation gives a way of using $MD(G|Y)$ finding the part of the modular decomposition of G that describes the modules of G that are subsets of Y. The procedure is described by Algorithm 4; the correctness follows by Lemma 3.2.

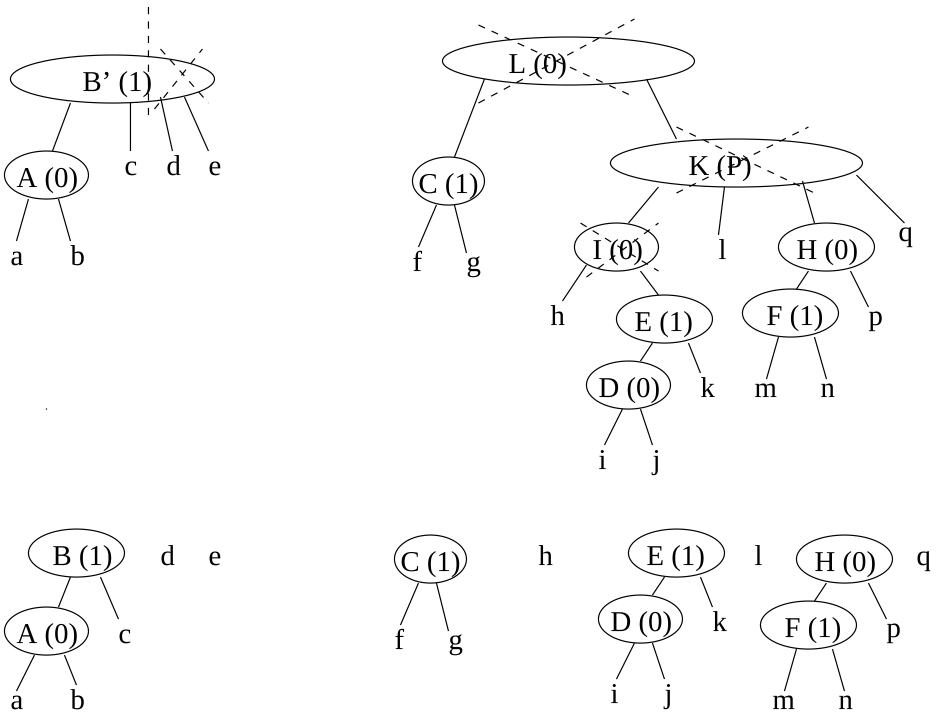

Figure 6: The restriction step of Algorithm 3, which is given as Algorithm 4. This illustration is based on the graph of Figure 1. The goal of the step is to compute $\mathcal{M}(G, v_0)$ and $MD(G|X)$ for each $X \in \mathcal{M}(G, v_0)$, using $MD(G|N(v_0))$ and $MD(G|\overline{N}(v_0))$. One must delete and split up nodes to reflect those modules of $G|N(v_0)$ and $G|\overline{N}(v_0)$ that fail to be modules of G. Any node labeled P that fails to be a module may be deleted. The same is true for nodes labeled 0 or 1, except for the following exception, which is illustrated by node B'. All unions of B' are modules of $G|N(v_0)$. Even though B' is not a module of G, a union of more than one child of B', $(A \cup \{c\})$ is a module of G. A new node for each such maximal union must be created. The algorithm of Ehrenfeucht et al. (1994) uses vertex partitioning to find the same result, which is the bottleneck for its time bound.

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,

Algorithm 4: The Restriction 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 have weak neighbors in $V \setminus Y$;

In the remaining forest, group together any roots that are former siblings with a deleted degenerate parent and that have the same neighbors in $V \setminus Y$, and make them children of a new degenerate node.

and since v_0 is either a strong neighbor or a non-neighbor, it must be contained in $N(v_0)$ or $\overline{N}(v_0)$. □

By Lemma 3.3, the restriction procedure satisfies the restriction step of Algorithm 3. The inputs to it are the two recursively-defined trees returned by the recursion step.

For the v_0-modules step, we use a combination of ideas from Dahlhaus (1995) and Ehrenfeucht et al. (1994). Let us 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 obtained in the same way, except with the roles of edges and nonedges inverted in the definition. That is, the first rule is changed by making x and y be in the same basic block if they are contained in the same co-component of $GG\left|N(v_0)\right.$.

Figure 7 shows the basic blocks of the graph of Figure 1.

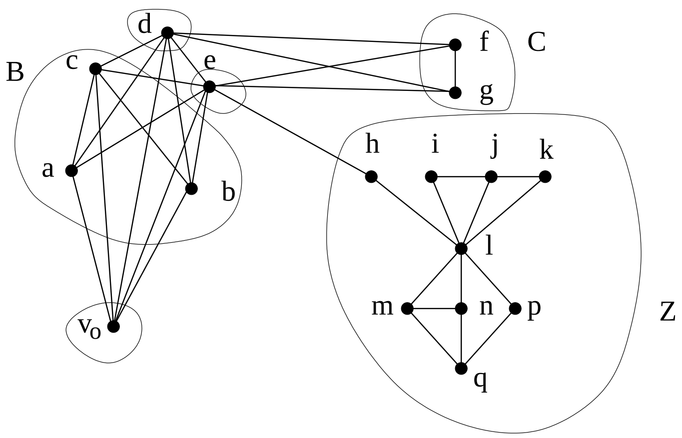

Figure 7: The basic blocks are components of $G|\overline{N}(v_0)$ and co-components of $G|N(v_0)$, except that multiple components or co-components that make up a module of G form a single block. For instance, $\{a\}$, $\{b\}$, and $\{c\}$ are each co-components, but their union is a basic block since it is a module of G.

Let $\mathcal{B}$ denote the basic blocks of $N(v_0)$ and let $\mathcal{B}'$ denote the basic blocks of $\overline{N}(v_0)$. Let us define a bipartite directed *forcing graph* $\mathcal{F} = (\mathcal{B}, \mathcal{B}', E_{\mathcal{F}})$. For $B \in \mathcal{B}$ and $B' \in \mathcal{B}'$, (B, B') is an arc of $\mathcal{F}$ if and only if there is an edge of G between B and B', and (B', B) is an arc 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 arcs telling which strongly-connected components are reachable from which on a single arc of $\mathcal{F}$ (Cormen et al. (1990)). A topological sort gives the chain of ancestors of v_0, by giving the sets of nodes that must be added to an ancestor to get its parent. The proof is given below. Figure 8 gives an example.

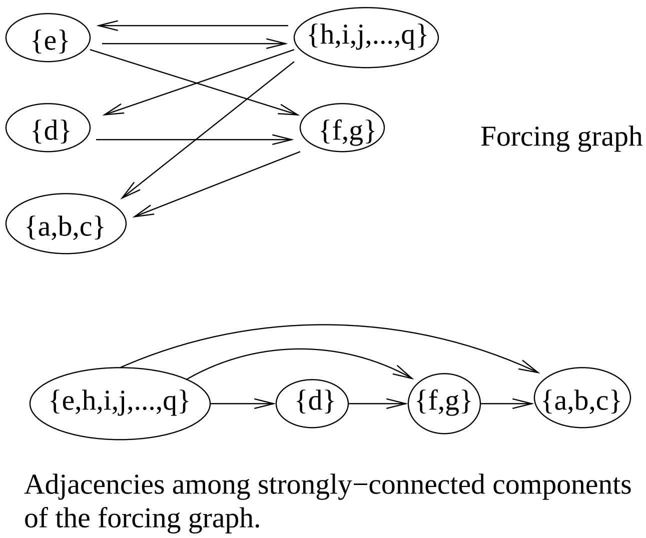

Figure 8: The forcing graph $\mathcal{F}$ is a bipartite graph on the basic blocks of $N(v_0)$ and $\overline{N}(v_0)$ from Figure 7. If X is a basic block of $N(v_0)$ and Y is a basic block of $\overline{N}(v_0)$, then (X, Y) is an arc if there is an edge of G between X and Y, and (Y, X) is an arc if there is a non-edge of G between X and Y. For instance, there is an arc from $\{e\}$ to $\{h, ..., q\}$ because $\{e, h\}$ is an edge of G, and there is an arc from $\{h, ..., q\}$ to $\{e\}$ because $\{i, e\}$ is a non-edge of G. The adjacencies of the strongly-connected components of the graph give a dag. The topological sort of it is unique, and the ancestors of v_0 in Figure 4 are given by the unions of v_0 and the suffixes of this sort, namely $\{v_0, a, b, c\}$, $\{v_0, a, b, c, f, g\}$, $\{v_0, a, b, c, f, g, d\}$, and $\{v_0, a, b, c, ..., q\}$.

Let the *active edges* be those edges of G that do not appear in $G|N(v_0)$ or $G|\overline{N}(v_0)$. These edges are incident to v_0 or go between $N(v_0)$ and $\overline{N}(v_0)$. The key observation that we make use of is that the operations of the inductive step can be carried out using only the active edges. The children of the root of the recursively obtained modular decomposition on $\overline{N}(v_0)$ gives the connected components of $\overline{N}(v_0)$ if there are more than one of them, and the co-components of $N(v_0)$ if there are more than one of them.

The main goal is to perform the inductive step in time linear in the number of active edges. If we can do this, then the total time spent over all inductive steps will then be $O(m)$, since each edge is active in exactly one of them. Together with an overhead of $O(n)$, this will give the decomposition in $O(n+m)$ time. If we allow ourselves to perform in addition to this a number of union-find operations proportional to the number of active edges, we get an $O(n+m\alpha(m,n))$ algorithm. The rest of the paper is devoted to data structures and tricks that allow us to achieve these time bounds. The main difficulty is how to perform the inductive step without touching vertices and tree nodes that have no incident active edges, as we would not be able to bound that cost in this way.

4 Basic procedures

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

4.0.1 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.*

4.0.2 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.

4.0.3 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 $v, w \in V$, perform a `Find` to see if the set of $\mathcal{P}$ containing u is different from the one containing v. If so, merge the two sets 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 m `Union` and `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 `Union`s, $G|S$ 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.*

4.0.4 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 4.1 *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 4.2 (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. However, to make efficient algorithmic use of Lemma 5.3, below, we use the following result:

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.*

5 Modular Decomposition in $O(n+m\alpha(m,n))$ Time

We now describe an algorithm that is based on Algorithm 3 and that runs in $O(n + m\alpha(m,n))$ time. The union-find operations are the only obstacle to a linear time bound. Recall that the main goal is to perform the inductive step without touching vertices and tree nodes that have no incident active edges.

5.1 The implementation

5.1.1 A data structure for the decomposition tree

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

Data Structure 5.1 (Representation of a modular decomposition tree.)

representative: *Each node of the tree carries a pointer to a vertex of the graph that is one of its leaf descendants, called its* representative*. The representative of the parent is the same as the representative of one of its children.*

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

parent: *Each node* u *of the tree carries a pointer to its parent. This pointer is allowed to be out-of-date when the parent is* V *or a connected component of* G *that is labeled 'P' in* $MD(G)$*.*

components: *A union-find structure is maintained on the connected components. When a node's parent pointer is out of date, this allows lookup of its parent anyway, through a find operation on its representative.*

We use the following basic operations for modifying the data structure during the inductive step of Algorithm 3.

Delete: *Delete a node that is no longer a module of G.* (This is used in Algorithm 4.) Mark the node for deletion. Remove deleted children and children that have incident active edges. Return them and the marked parent, together with its remaining circular list of children that have no incident active edges.

Group: *Make a set of undeleted children of a deleted degenerate node in $MD(G|\overline{N}(v_0))$ that have the same neighbors in in $N(v_0)$ into children of a new degenerate node.* (This is the last step of Algorithm 4; a symmetric operation works on $MD(G|N(v_0))$. Call `Delete` on the node. The children of the deleted node are the group of children that have no active neighbors. If there is more than one of them, remove the deletion mark from the deleted parent, and re-use it as the new parent of these children. Make its new representative be the representative of one of its children. The parent pointers of the children are up to date, since they previously pointed to the old parent. All other undeleted children have active neighbors. Group them into groups that have the same set of active neighbors using radix grouping. Create a new parent for each of these groups that has more than one child, and make the representative of one of the children be the representative of the parent. Update the parent pointer of each of these children.

Subsume: *Given a parent P and a child C that are both labeled 0 or both labeled 1, delete C and let P inherit its children.* (This is the last operation of Algorithm 2.) Splice the circular list of children of C into the circular list of children of P in $O(1)$ time. If C has active neighbors, update the parent pointers of children of C.

5.1.2 Preprocessing

For the inductive step, we would like to spend time proportional to the number of active edges, but this requires having the active edges already segregated from the rest of the edges.

To do this, we perform some preprocessing to find a partition $\mathcal{E}$ of the edges so that the edges in each partition class are those that are active in one of the recursive calls. This requires deciding in advance which vertices are going to be selected to serve the role of v_0 in each recursive call. This is quite easy, because the selection of v_0 is arbitrary. At the beginning of the preprocessing step, we select v_0, which determines the sets $N(v_0)$ and $\overline{N}(v_0)$ that will be passed to each recursive call of the algorithm. Recursively carrying out the preprocessing step on $N(v_0)$ and $\overline{N}(v_0)$ produces the recursion tree of the decomposition algorithm in advance of carrying it out. Finding the least-common ancestor of each edge of G in this tree gives the desired partition $\mathcal{E}$: for each internal node X, the set of edges with X as their least common ancestor is one of the partition classes. Figure 9 gives an example. A simple $O(n + m\alpha(m,n))$ algorithm for least-common ancestors is given in Cormen et al. (1990).

Each edge $\{u, v\}$ consists of two arcs (u, v) and (v, u). Using radix sorting, we sort all arcs by vertex of origin as primary sort key, partition class of $\mathcal{E}$ as secondary sort key, and destination vertex as tertiary sort key. During the inductive step, this gives, for each vertex, a list of active incident edges, sorted by destination vertex.

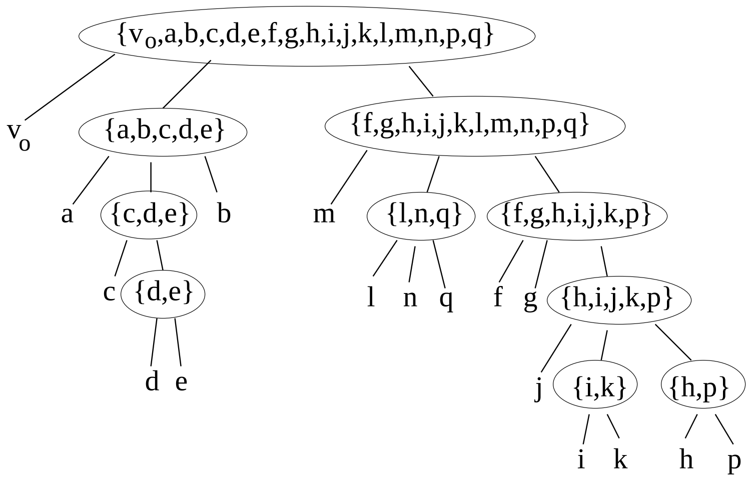

Figure 9: Preprocessing: partitioning the edges into groups that become active at the same time. The step consists of deciding in advance on the recursive calls that will be made, and then grouping edges into groups that will be active in one of the recursive calls. For example, the selection of v_0 on the graph of Figure 1 dictates that the top two recursive calls will be made on the neighbors $\{a, b, c, d, e\}$ and non-neighbors $\{f, g, h, i, j, k, l, m, n, p.q\}$. Since selection of v_0 is arbitrary, we may select a to be the vertex that will serve in the role of v_0 in the recursive call on $\{a, b, c, d, e\}$. This splits this class into neighbors $\{c, d, e\}$ and $\{b\}$. The process is carred out recursively until singleton sets are reached. Once this tree is obtained, the decomposition algorithm is then run, using this tree as its recursion tree. From the tree, it is clear before the decomposition algorithm begins that the edges $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, $\{b, c\}$, $\{b, d\}$, and $\{b, e\}$, for example, will be the active edges in the recursive call on $\{a, b, c, d, e\}$, since these are the edges that have the node for $\{a, b, c, d, e\}$ as their least-common ancestor in this tree. Finding the least-common ancestor of every edge takes $O(n + m\alpha(m, n))$ time by simple methods, see Cormen et al. (1990), and the partition may then be found in $O(m)$ time by bucket sorting with least-common ancestor as the sort key.

5.1.3 The Restriction Step

.

Using the list of active edges for the call, we identify the leaves of $N(v_0)$ and $\overline{N}(v_0)$ that have incident active edges. We then work inductively toward the roots of these trees, labeling nodes that remain modules in G with their sorted incident active edge lists, and marking the remaining nodes for deletion. At each node, we mark it for deletion if it has a child that is marked for deletion. Otherwise, we compare the sorted incident active edge lists of the children. If they are identical, we label the parent with a copy; otherwise we mark the parent for deletion. We perform the `Group` operation to create tree nodes for basic blocks that are not already nodes in one of the two recursively-obtained trees.

Below, for the assembly step, we will need a list of members of $\mathcal{M}(G, v_0)$ for each basic block that is not a module. Such a basic block is a node that gets deleted. As we travel up through nodes that get deleted, we get lists of members of $\mathcal{M}(G, v_0)$ that are formed from each deleted node's children, as described in `Delete` and `Group`. By concatenating lists as we go, we have a list of members of $\mathcal{M}(G, v_0)$ that make up a deleted node when we arrive at it; when we arrive at the basic block, this list gives the list of members of $\mathcal{M}(G, v_0)$ for the block.

5.1.4 The v_0-modules step

Let $\mathcal{B}$ be the basic blocks of $N(v_0)$ and let $\mathcal{B}'$ be the basic blocks of $\overline{N}(v_0)$. Label each $B \in \mathcal{B}$ with a list of its neighbors in $\mathcal{B}'$ in the forcing graph. Label each $B' \in \mathcal{B}'$ with a list of its *nonneighbors* in $\mathcal{B}$. There is at most one basic block with no incident active edges; if there is, it is a union of components of $\overline{N}(v_0)$. So the representation of the graph has size proportional to the number of active edges used to compute it. Use the algorithm of Theorem 2 to find the topological sort of the component graph of the forcing graph, as illustrated in Figure 8.

5.1.5 The assembly step

To get the union-find structure to reflect the connected components of G, perform a `Union` operation on the endpoints of each active edge. For any connected component X of $G|N(v_0)$ or $G|\overline{N}(v_0)$ that has incident active edges and remains a node in $MD(G)$, update the parent pointers of the children of X to point to X, since Data Structure 5.1 did not previously require them and now it does. Let B be a basic block, and let $\mathcal{M}$ be the members of $\mathcal{M}(G, v_0)$ contained in B. The restriction step produces $\mathcal{M}$ as a concatenated list of tree nodes; these become children of the ancestor A of v_0 that B goes with. We may splice $\mathcal{M}$ into the child list of A in $O(1)$ time. If A is not a connected component of G labeled 'P', update the parent pointers of its children to point to A.

5.2 Correctness and Analysis of the Time Bound

We bound the running time by charging operations during the inductive step to elements of the graph at $O(1)$ time and $O(1)$ find operations per element. This charging is not part of the algorithm; it is used in the analysis only.

Since the nodes of the decomposition tree are modules, any neighbor x of a node U is a strong neighbor. If, in addition, all edges from U to x are active edges, then x is an *active neighbor*.

If a node U is not deleted and has active neighbors in the inductive step, we can charge $O(1)$ time and find operations to each active neighbor. The total of such charges to any active neighbor by U and its descendants is $O(|U|)$. Since U is not deleted, it is a module, and each active neighbor has $\Omega(|U|)$ active edges to U, which conforms to our limit of $O(1)$ time and find operations per active edge.

Lemma 5.2 *A sibling of an ancestor of v_0 either has an active neighbor or is not required by Data Structure 5.1 to have an up-to-date parent pointer.*

Proof: Let C be a sibling of an ancestor of $\{v_0\}$, and let P be its parent. If $C \subseteq N(v_0)$, then it has v_0 as an active neighbor.

Suppose $C \subseteq \overline{N}(v_0)$. Next, suppose $P \neq V$ and P is not a connected component of G. Since P is a strong module, it is not a union of connected components of G. P has a strong neighbor q. The least-common ancestor of C and v_0 is P, and the least-common ancestor of q and v_0 is not P. Therefore, q is not in the basic block B of $\overline{N}(v_0)$ that constains C. Since B is a connected component or a union of connected components of $\overline{N}(v_0)$ and q is a neighbor of all vertices of P, hence of all vertices of C, $q \in N(v_0)$. All edges between $\overline{N}(v_0)$ and $N(v_0)$ are active, so q is an active neighbor of C.

Thus, if C has no active neighbor, P is V or a connected component of G. If P is V or a connected component of G labeled 'P', C requires no parent pointer. If P is labeled 0, it is not a connected component of G. If P is a connected component labeled 1, v_0 is a member of an sibling of C that is adjacent to C in G, and $C \subseteq N(v_0)$, contradicting the assumption that $C \subseteq \overline{N}(v_0)$. □

Each time we update a node's parent pointer, we will pre-pay the cost of an $O(1)$ operation that may occur at a later time. We may think of the prepayment as a "credit" that sits on each up-to-date parent pointer. The credit is freed up to pay for an $O(1)$ time or find operation when the node is deleted or the parent pointer becomes out of date and Data Structure 5.1 does not require that it be updated. The reason for doing this is to show bounds on the cost of operations that cannot be charged to edges that are currently active; the credit scheme charges these costs to edges that were active in a lower recursive call, while providing an accounting devide that ensures that no edge is charged more than $O(1)$ time and find operations. This accounting device is an example of *amortized analysis*, and is surveyed in Tarjan (1983) and Cormen et al. (1990).

5.2.1 Operations on Data Structure 5.1

The `Delete` operation takes $O(1+i+j)$ time, where i is the number of deleted children, and j is the number of undeleted children that have incident active edges. (The undeleted children that have no incident active edges are not touched individually, and are only returned as the remaining children of the deleted node.) The $O(1)$ and $O(i)$ costs are paid with the credit that is freed from the deleted node's parent pointer and the i credits that are freed from the parent pointers of the i deleted children. The $O(j)$ cost is charged to the active neighbors of the j undeleted children that have active neighbors.

The `Group` operation on children of a deleted degenerate node Z spends $O(1+i+j)$ time, where i is the number of deleted children, and j is the number of undeleted children that have active neighbors. The $O(i)$ is paid for out of the i credits freed from the parent pointers of the deleted children. The $O(j)$ is charged to the active neighbors of the j undeleted children that have active neighbors. The $O(1)$ cost is charged to the credit freed from Z's parent pointer.

If C has no active neighbors in `Subsume`, then its children to not get their parent pointers updated when they become children of P. By Lemma 5.2, they do not need them. `Subsume` takes $O(1)$ time if C has no active neighbors. Otherwise, it takes $O(k)$ time where k is the number of children of C. In the first case, the $O(1)$ cost can be charged to the credit freed from the deleted parent pointer of C, and in the latter case, it can be charged because each of the k children has an active neighbor.

5.2.2 The v_0-modules step

The following lemma demonstrates the correctness of the v_0`-modules` step.

Lemma 5.3 *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 5.4 *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 5.5 *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 5.4.

Observation 5.6 *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 $G|\overline{N}(v_0)$ 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 strong only if it is a union of $\{v_0\}$ and zero or more basic blocks, and a weak one 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. $\square$

5.2.3 The Assembly Step

The union-find structure starts out reflecting components of $G|N(v_0)$ and $G|\overline{N}(v_0)$. After a union on all active edges (those edges of G that are not in these subgraphs), the structure obviously reflects the connected components of G.

For the correctness of the parent pointers, note that all members of $\mathcal{M}(G, v_0)$ and all ancestors of v_0 get their parent pointers updated, except when their new parent is a connected component. All other nodes are nodes of $MD(G|N(v_0))$ or $MD(G|\overline{N}(v_0))$ that are not members of $\mathcal{M}(G, v_0)$. Let Y be such a node. Without loss of generality, suppose it is a node of $MD(G|\overline{N}(v_0))$. If Y's parent X in $MD(G|\overline{N}(v_0))$ remains its parent in $MD(G)$, then something needs to be done with Y's parent pointer only if Data Structure 5.1 now requires a parent pointer on Y but did not previously require one. This happens only if X is a connected component of $G|\overline{N}(v_0)$ labeled 'P', remains a module in G, but is not a connected component of G, hence has incident active edges. This case addressed explicitly in the algorithm. If Y's parent Z in $MD(G)$ is not its parent in $MD(G|\overline{N}(v_0))$, then, since Y is not a member of $\mathcal{M}(G, v_0)$, Z must be. Algorithm 4 makes sure that Y's parent pointer is up-to-date when it creates Z.

6 Obtaining a Linear Time Bound

One of the bottlenecks is in finding the least-common ancestors of edges in the recursion tree. This is easily eliminated by using the off-line least-common ancestors algorithm of Harel and Tarjan (1984). The only other bottleneck is in managing the union-find data structure of Data Structure 5.1. In this section, we show how to do this in linear time, by restricting the possible choices for the recursion tree.

When running the preprocessing step on vertex set X (Figure 9) 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 the preprocessing step is to select v_0 to be the vertex with greatest recency, rather than selecting it arbitrarily.

Algorithm 5 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.

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.

Algorithm 5:

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.

Lemma 6.1 *Algorithm 5 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. $\square$

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 6.2 *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. $\square$

Definition 6.3 *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 6.4 *Let R be a recursion tree of graph $G = (V, E)$, 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 6.5 *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 6.2, 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 6.2, 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 6: Compute a recursion tree R and its subordinate tree for a connected graph G.

Let R be a recursion tree computed with Algorithm 5;
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 6.6 *Algorithm 6 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 $y = lca(u, x) > w$ $\{u, x\}$ be any edge from u to a vertex with lower 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 6 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 5, 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 5 can be modified to produce all of this information in a single pass. The modified version is given as Algorithm 7.

Lemma 6.7 *Algorithm 7 is correct.*

Proof: The recursion tree returned is the same as the one returned by Algorithm 5. 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

Algorithm 7:

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 $y \in N(v_0)$ **do**
    if *y has not been selected* **then**
        if *y is a member of L* **then**
            remove y from L;
            put 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.

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.

[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.

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

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

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

[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.

[Habib and Möhring, 1987] Habib, M. and Möhring, R. H. (1987). On some complexity properties of N-free posets and posets of bounded decomposition diameter. *Discrete Mathematics*, 63:157–182.

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

[McConnell and Spinrad, 2000] McConnell, R. M. and Spinrad, J. P. (2000). Ordered vertex partitioning. *Discrete Math and Theoretical and Computer Science*, 4:45–60.

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

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

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

[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.