Efficient and Practical Modular Decomposition[*]

Elias Dahlhaus[†] Jens Gustedt[‡] Ross M. McConnell[§]

Abstract

We give a simple recursive algorithm for modular decomposition of undirected graphs that runs in $O(n+m\alpha(m,n))$ time. Previous algorithms with this bound are of theoretical use only. By adding some data structure tricks, we get a much simpler proof of an $O(n+m)$ bound than was previously available. Key components of the algorithm are variations of a procedure for finding a depth-first forest on the complement of a directed graph G in $O(n+m)$ time. This is surprising, given that it takes $\Omega(n^2)$ time to compute the complement explicitly.

1 Introduction

Computing the modular decomposition of an undirected graph is a key step in the fastest algorithms for a large number of combinatorial problems on graphs, such as finding a transitive orientation of a graph, finding maximum clique and independent set, minimum clique cover, and minimum vertex coloring if a transitive orientation exists. It is a pivotal step in the fastest algorithms for recognizing permutation graphs [9] [10]. The decomposition tree can guide a divide-and-conquer attack on a large number of NP-complete problems whenever a graph has a nontrivial decomposition [11]. The cotree decomposition of cographs [2] is a special case. When the decomposition is nontrivial, it can be used to give a compressed representation of the graph [11]. The PQ tree, which is used for recognizing interval graphs, is strongly related to the modular decomposition, and, in fact, can be computed from it [13].

Though algorithms for the problem have been studied extensively since 1974, the first linear-time algorithms did not appear until 1994 [9, 3]. These latter algorithms are unfortunately elaborate and difficult to understand, and are of strictly theoretical interest.

[*]Part of this work has been made possible by a visiting grant for Ross McConnell by the *Graduiertenkolleg Algorithmische Diskrete Mathematik* in Berlin

[†]Institut für Informatik, Universität Bonn, D-53117 Bonn, Germany; email: dahlhaus@uni-bonn.de

[‡]Technische Universität Berlin, Sekr. MA 6-1, D-10623 Berlin, Germany; email: gustedt@math.TU-Berlin.DE

[§]Dep. of Mathematics and Computer Science, Amherst College, Amherst, MA 01002 USA. Current address: Dept. of Computer Science, Willamette University, Salem, OR 97301, USA; email: rmcconne@willamette.edu

In this paper we give a simple divide-and-conquer strategy and a straightforward inductive proof of correctness. The algorithm combines elements of [4] and [5]. An implementation using elementary programming techniques gives an $O(n+m\alpha(m,n))$ time bound, where n and m are the number of vertices and edges of the graph, respectively. By making use of the Gabow-Tarjan *Union-find* strategy [6], we get a compact proof of a linear time bound for the problem.

A key element of the algorithm is a simple but non-obvious way to compute a depth-first forest for the complement of a directed graph G in time proportional to the size of G. The trick is easily extended to any *partially complemented*, or *mixed* representation of a graph, where some vertices are *complemented*, and carry a list of their non-neighbors rather than of their neighbors. This gives a depth-first forest on G in time proportional to the size of the mixed representation. The trick may be of broader use, since a depth-first search of a graph can arise as part of a solution to a problem, where the mixed representation can be computed more efficiently than a standard representation can.

2 Basic Definitions and Facts

All graphs in the following are undirected and do not have loops or multiple edges. For a graph G, the vertex set is denoted $V(G)$ and the edge set is denoted $E(G)$. The subgraph of G induced by $X \subseteq V(G)$ is denoted $G|X$.

If v is a vertex of G, the set of neighbors and non-neighbors of v are denoted by $N(v)$ and $\overline{N}(v)$, respectively. We say that v **distinguishes** two other vertices w and u if it is adjacent to one of them but not to both. We say that w and u **agree** on v if v does not distinguish w and u. Let $M \subseteq V$. M is called a **module** (or autonomous set) of G if it fulfills the following condition:

- Every $v \in V(G) - M$ agrees on every pair $m, m' \in M$.

In this paper, we will let $\mathcal{M}(G)$ denote the family of modules of G. Two sets A and B **overlap** if $A-B$, $B-A$, and $A \cap B$ are all nonempty. The following theorem can be found in [12].

THEOREM 2.1. *The family $\mathcal{M}(G)$ of modules of an undirected graph G have the following properties:*

trivial: *$\emptyset, V(G) \in \mathcal{M}(G)$ and $\{v\} \in \mathcal{M}(G)$ for each $v \in V(G)$.*

Intersection: $N \cap M \in \mathcal{M}$ *for all* $N, M \in \mathcal{M}(G)$.

Union: $N \cup M \in \mathcal{M}$ *for all* $N, M \in \mathcal{M}(G)$ *such that* $N \cap M \neq \emptyset$.

Differences: $N - M$, $M - N$, *and* $(M - N) \cup (N - M)$ *are modules for all* $N, M \in M(G)$ *such that* N *and* M *overlap.*

A **strong module** is a module that overlaps with no other module. That is, if M is a strong module, and N is any module, then $N \subseteq M$, $M \subseteq N$, or $M \cap N = \emptyset$. A **weak module** is a module that is not strong. $V(G)$ and its singleton subsets are strong modules.

Since $V(G) \in \mathcal{M}(G)$, the containment relation imposes a tree on the strong modules, where $V(G)$ is the root, the singleton sets are the leaves, and any other strong module is the least common ancestor of its singleton subsets. From the fact that no module overlaps a strong module, it follows that every weak module is a union of siblings in the tree.

So far, this fails to represent all modules of G. However, Theorem 2.1 alone is sufficient to ensure the following remarkable fact [12]:

THEOREM 2.2. *There is a labeling of each strong module in the modular decomposition tree as* ***degenerate*** *or* ***prime*** *such that* $Y \subseteq X$ *is a module if and only if it is either a strong module or a union of children of a strong module that is labeled degenerate.*

This gives rise to the following $O(n)$-space representation of the modules of G, which we will call the **decomposition tree** of G, or $T(G)$. There is one node for each strong module, labeled prime or degenerate, as appropriate, and an edge from each tree node to the node corresponding to the parent. The leaves of the tree are just the singleton sets. There are pointers in the leaves to the corresponding vertices of G.

The strong module X corresponding to an internal node x is obtained in $O(|X|)$ time by visiting the pointers in the leaf descendants. This representation is the **decomposition tree**. Though we use an $O(1)$-size node to represent X, we will find it convenient to treat that node as synonymous with the set X.

Let a **co-component** of a graph G denote a connected component of the complement of G.

THEOREM 2.3. *If* X *and* Y *are disjoint modules, then either every member of* $X \times Y$ *is an edge of* G *or none is.*

A strong degenerate module Z is of one of two types:

Parallel: the children of Z are the connected components of $G|Z$;

Series: the children of Z are the co-components of $G|Z$.

THEOREM 2.4. *If* $X, Y \subseteq V(G)$, $Y \subseteq X$, *and* Y *is a module of* G, *then it is also a module of* $G|X$.

THEOREM 2.5. *If* $X \subseteq V(G)$ *is a module of* G, *then* $Y \subseteq X$ *is a module of* G *if and only if it is a module of* $G|X$.

By Theorem 2.2, every module that is not a singleton set is a union of two or more siblings in the decomposition tree, since even a strong module is the union of all of its children. If X is a non-singleton module, let $\mathcal{F}(X)$ denote these siblings, and let $P(X)$ denote the parent of the members of $\mathcal{F}(X)$ in $T(G)$.

COROLLARY 2.1. *If* $X \subseteq V(G)$ *is a non-singleton module of* G, *the modular decomposition of* $G|X$ *is obtained by creating a node* x, *copying the prime or degenerate label of* $P(X)$ *to it, and letting the subtrees of* $T(G)$ *rooted at members of* $\mathcal{F}(X)$ *be the children of* x.

By Corollary 2.1, if X is a module, then the decomposition tree of $G|X$ is the **restriction** of $T(G)$ to X. If Y is an arbitrary subset of $V(G)$, then by Theorem 2.1, the maximal modules of G that are contained in Y are a partition of Y. The **restriction** of $T(G)$ to Y is the forest of trees obtained by restricting $T(G)$ to the maximal modules of G that are contained in Y. By Theorem 2.5 and Corollary 2.1, the restriction of $T(G)$ to Y represents the set of modules of G that are subsets of Y.

COROLLARY 2.2. *If* $X \subseteq V(G)$ *then the the modules of* G *that are subsets of* X *are given by the restrictions of* $T(G|X)$ *to the maximal modules of* G *that are subsets of* X.

This last corollary is central to part of our algorithm, as our algorithm selects a vertex v_0, computes the modular decomposition of $G|N(v_0)$ and $G|\overline{N}(v_0)$ recursively, and then restricts them to obtain a representation of those modules of G that are subsets of $N(v_0)$ and $\overline{N}(v_0)$.

3 Basic Algorithms on Partially Complemented Representations

In a conventional adjacency-list representation of a directed graph G, each vertex carries a list of its neighbors. We generalize this by allowing a vertex to carry either a list of its neighbors, or a list of its non-neighbors. Each vertex is labeled standard or complemented to indicate which case applies. We call this a *partially complemented*, or *mixed* representation of G. 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. We show that some basic graph algorithms, such as finding depth-first forests and strongly-connected components, can be carried out in $O(n + m')$ time on such a representation.

This is remarkable because the size of a mixed representation of a graph may be asymptotically smaller than the size of the graph itself. For instance, though it takes $\Omega(n^2)$ time to construct the complement of a graph, the algorithms can be performed on the complement of G in $O(n+m)$ time, by simply labeling the vertices in its adjacency-list representation as complemented and then running the mixed-representation variants on them. Moreover, when G is directed, a mixed representation may be asymptotically smaller than either the graph or its complement.

One way to implement depth-first search is by using a stack of vertices. To select a vertex to visit, pop it from S. If it has not been visited, then push copies of all unvisited neighbors on S, *deleting any lower occurrences of them in S if they already reside on the stack.* The lower occurrences may be deleted, because the existence of a higher instance of a vertex guarantees that the it will already be visited when the lower instance is popped.

It is not obvious that the approach allows us to carry out depth-first search in time proportional to the size of a mixed representation; if a node is complemented, we must delete and re-push its neighbors in time proportional to the number of its non-neighbors. We develop the following abstract data type for the purpose.

3.1 Complement stacks.

For depth-first search on a mixed representation we use a data structure, **complement stacks**, which generalizes a stack. Let X be a set and $V-X$ be its complement. One of the operations the complement stack supports is deleting lower occurrences of members of $V-X$ from the stack, and then pushing $V-X$ to the top of the stack. Surprisingly, the amortized timed bound for this operation is $O(|X|)$, not $O(|V-X|)$.

DEFINITION 3.1. *A **complement stack** is a data structure that supports the following operations:*

initialize(V) *initializes and returns an empty stack S.*

push(X,S) *removes any occurrences of members of X from S, then pushes the members of X to the top of S.*

$\overline{\textbf{push}}(X,S)$ *performs push$(V-X,S)$.*

pop(S) *returns the top element of S.*

time(S) *returns a timestamp that tells when the top element of S was pushed.*

THEOREM 3.1. *Complement stacks can be implemented in such a way that:*

- *initialize requires $O(V)$ time.*
- *push(X,S) requires $O(|X|)$ time.*
- *$\overline{push}(X,S)$ requires $O(|X|)$ time, amortized.*
- *pop and time require $O(1)$ time each.*

Proof. Let L be the members of V that are not on the stack. A, T, B lists that partition the members of the stack. A holds those elements on the stack that where last pushed by a call to *push*. T holds those elements that were pushed by the most recent call to $\overline{push}$. B holds those elements that were last pushed by a call to $\overline{push}$, but not by the most recent call to $\overline{push}$.

Each element of V keeps track of which of L, A, T, B contains it. If A or B contains it, it carries a **timestamp** that holds the time when the element was last pushed. To keep time, a global variable is incremented after each call to *push* or $\overline{push}$.

We maintain a credit invariant:

Each member of L, A, B carries a credit.

- *initialize*(V) creates a doubly-linked list L of the elements of V and assigns a credit to each of them. It creates empty doubly-linked lists A, T, B.
- *push*(X,S) traverses each member of X, splices it from the list in $\{L,A,T,B\}$ that it currently resides in, pushes it to the front of A together with a timestamp, and assigns a credit to it. This is clearly $O(|X|)$.
- $\overline{push}(X,S)$ must incur only $O(|X|)$ amortized cost. It marks each member of X and assigns a credit to it on top of any that it already has. It traverses X, removing those members of $X \cap T$ to auxiliary list T'. It then traverses L, A, and B, moving unmarked members to T, and pays for visiting their members by using up a credit at each. This still leaves credits on elements that remain in L, A, or B, since they are members of X and have just received an extra credit from X. It then assigns each member of T' a credit and labels it with a timestamp and moves it to front of B. If T is now nonempty, T is timestamped.

  This requires $O(|X|)$ amortized time, since all operations are $O(|X|)$ except for traversing L, A, B, which is paid for by using a credit sitting on each visited item.
- *pop* assigns x to be an element of T if T is nonempty, or else the top element of B if B is nonempty, or else null. If x came from T, then it gets T's timestamp. Let y be the top element of A if it is nonempty, else null. Return the member of $\{x,y\}$ that has the most recent timestamp. This clearly requires $O(1)$ time.

The time bound is observed, since each operation maintains the credit invariant, and $\overline{push}$ pays for its operations either with its budget of $|X|$ new credits or with other credits it frees up from the structure. □

COROLLARY 3.1. *Given a mixed representation of a directed graph G, constructing a depth-first forest for G takes*

$O(n+m')$ time, where m' is the number of edges and nonedges given explicitly in the representation.

Proof. When a node is first visited, call $push(N(x), S)$ if it is standard, or call $\overline{push}(\overline{N}(x), S)$ if it is complemented. This takes time proportional to its explicit list $N(x)$ if it is standard, and its explicit list $\overline{N}(x)$ if it is complemented. □

3.2 Set-complement stacks. Now let $V_1, V_2, \ldots V_p$ be a partition of a set V. A **set-complement stack** implements the complement-stack operations, but with the following changes:

initialize$(V_1, V_2, \ldots V_p)$ Initializes and returns an empty stack S.

$\overline{\textbf{\textit{push}}}(X, S, i)$ performs $push(V_i - X, S)$.

THEOREM 3.2. *Set-complement stacks can be implemented in such a way that:*

- *initialize requires $O(|V|)$ time.*
- *$push(X,S)$ requires $O(|X|)$ time.*
- *$\overline{push}(X,S,i)$ requires $O(|X|)$ time, amortized.*
- *pop and time each require $O(1)$ time, amortized.*

Proof. If $p = 1$, the result follows from Theorem 4.1. When $p > 1$, the stack may be simulated with p complement stacks, $S_1, \ldots, S_p$, one for each V_i, and an additional (ordinary) stack R that maintains the timestamps of the *push* and $\overline{push}$ operations. That is, the entries in R are pairs of the form (t, i) where t denotes a push time and i denotes the stack S_i on which an element was pushed.

initialize$(V_1, V_2, \ldots V_p)$ initializes R to the empty stack, and executes $S_i = initialize(V_i)$ for each i from 1 to p.

push(X, S) looks up for each $x \in X$ the set V_i that contains x and calls $push(\{x\}, S_i)$. In addition it then pushes the item (t, i) on the stack R.

$\overline{\textbf{\textit{push}}}(X, S, i)$ calls $\overline{push}(X, S_i)$ and pushes one instance of (t, i) on the stack R.

time(S) lets (t, i) denote the pair on top of R. While $time(S_i)$ does not equal t, it pops the top element from R and lets (t, i) denote the new top element. When done with this loop, it returns t.

pop(S) calls $time(S)$. This may delete elements from the top of R. It then looks up the pair (t, i) that is on top of R after that operation, and returns $pop(S_i)$.

Through the control of R, S clearly behaves as a single stack. Because of the properties described above for the complement stacks S_i, a push of an item causes any lower instances of that item to be deleted from S.

The time bounds clearly remain unchanged except for *time*, and for *pop*, since it calls *time*. *time* can cause a large number of items to be popped from R. We charge this cost to the calls to *push* and $\overline{push}$ that originally pushed them to R, leaving $O(1)$ operations charged to *time*. □

COROLLARY 3.2. *Given a mixed representation of a directed graph G, constructing a depth-first forest for G^T takes $O(n+m')$ time, where m' is the number of edges and nonedges given explicitly in the representation.*

Proof. Radix sort all explicit edges and nonedges so that each vertex x has a list $L_s(x)$ of standard nodes that have an edge to it in G, and a list $L_c(x)$ of complement nodes that have no edge to it in G. Let V_s and V_c denote the standard and complemented nodes of G, respectively. Call $initialize(V_s, V_c)$ to initialize a set-complement stack S. When a node is visited, push its neighbors in the transpose by making a call to $push(L_s(x), S)$ and a call to $\overline{push}(L_c(x), S, c)$. This takes $O(|L_s(x) + L_c(x)|)$ amortized time. Each explicit edge or nonedge in the mixed representation appears in exactly one list $L_s()$ or $L_c()$ in this representation of the transpose. □

Let the *bipartite complement* of a directed bipartite graph (V_1, V_2, E) be the graph $(V_1, V_2, ((V_1 \times V_2) \cup (V_2 \times V_1) - E)$. In a *mixed bipartite representation*, each vertex in V_1 (V_2) has either a list of those members of V_2 (V_1) that are neighbors, or else a list of those members of V_2 (V_1) that are not neighbors.

COROLLARY 3.3. *Given a mixed representation of a directed bipartite graph G, constructing a depth-first forest for G or G^T takes $O(n+m')$ time.*

Proof. For the depth-first forest on G, execute $S = initialize(V_1, V_2)$. When visiting a node x, suppose without loss of generality that it is in V_1. Push its neighbors in V_2 with a call to $push(N(x), S)$ on its list of neighbors, or else with a call to $\overline{push}(\overline{N}(x), S, 2)$.

For a depth-first on G^T, divide V_1 into sets $V_{1,s}$ and $V_{1,c}$, and V_2 into sets $V_{2,s}$ and $V_{2,c}$. Without loss of generality, suppose a vertex x is in V_1, and that it carries the list $L_s(x)$ of standard vertices in V_2 that have an edge to it in G, as well as the list $L_c(x)$ of complemented vertices in V_2 that do not have an edge to it in G. This is obtained for all vertices in a preprocessing step, by radix sorting the explicit edges and nonedges given in the mixed representation of G. Call $S = initialize(V_{1,s}, V_{1,c}, V_{2,s}, V_{2,c})$. When a vertex x is first visited, suppose without loss of generality that it is in V_1. Push its neighbors in V_2, with a call to $push(L_s(x), S)$ and a call to $\overline{push}(L_c(x), S, [2, c])$. □

THEOREM 3.3. *Finding the strongly-connected components of any directed or directed bipartite graph G takes $O(n+m')$ time.*

Proof. The problem reduces to depth-first searches on G and on G^T (see the textbook by Cormen et al. [1]). □

4 The Modular Decomposition Algorithm

We now describe an algorithm to compute the modular decomposition tree. The approach chosen here to compute the decomposition tree of a graph G is summarized by Algorithm 1. The correctness follows from statements made in the description. Only the last three steps require further elaboration.

Algorithm 1 The Generic Decomposition Algorithm

Input: Graph G.

Output: Decomposition tree T of G.

pivot: Choose a pivot $v_0 \in V(G)$.

recurse: Recurse on $G|N(v_0)$ and $G|\overline{N}(v_0)$. Denote the decomposition trees obtained by the recursive calls by T_1 and T_2.

restrict: Using T_1, find the maximal modules of $G|N(v_0)$ that are modules of G. By Corollary 2.2, this gives the restriction of $T(G)$ to $N(v_0)$. Using T_2, do the same to find the restriction of $T(G)$ to $\overline{N}(v_0)$.

v_0-modules: All other modules of G contain v_0, since v_0 distinguishes members of $N(v_0)$ and $\overline{N}(v_0)$. Compute a tree representation T_{v_0} of the modules that contain v_0.

assemble: Assemble $T(G)$ from $\mathcal{S}_{v_0}$, $T(G)|N(v_0)$ of G; splice them together to get the modular decomposition tree T for G.

Let the **active edges** be those edges that connect v_0, $N(v_0)$, and $\overline{N}(v_0)$. We justify the following key observation below.

REMARK 4.1. *The inductive step can be carried out without examining any inactive edges.*

Each edge becomes active at most once in the entire recursion tree. Our strategy is to pre-select which pivot nodes will be used at which points. We can then pre-partition the edges into sets that become active at the same time, so that the appropriate set will be on hand when it becomes active. We get a linear time bound by charging $O(1)$ time to each active edge during the inductive step.

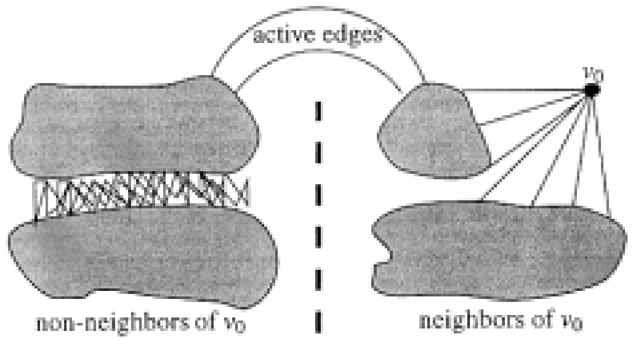

Figure 1: Splitting the graph according to a pivot

4.1 The restrict step. The **restrict** step is straightforward. By Corollary 2.2, we need only find the maximal modules of $G|N(v_0)$ that are modules of G. We first assign a list of neighbors in $\overline{N}(v_0)$ to each node of T_1 that is a module. We do this by working up the tree in postorder, starting at the leaves that have incident active edges. These leaves are modules and their neighbors in $\overline{N}(v_0)$ are given by the incident active edges. If the children of an internal node X are modules and carry identical adjacency lists, mark X as a module, and give X a copy of the adjacency list of one of the children. Otherwise, mark X for deletion. By induction, this procedure marks for deletion exactly those members of $T(G|N(v_0))$ that are not modules of G and have incident active edges. We do not touch nodes of the tree that have zero incident active edges, but all such nodes are modules of G. Thus, all non-modules are deleted.

Deleting the non-modules from the tree leaves a forest. Any roots that were children of a degenerate node and have identical lists of neighbors in $\overline{N}(v_0)$ must then be grouped together under a common vertex, as the union of such a group is a maximal module of $G|N(v_0)$ that is not distinguished by any member of $\overline{N}(v_0)$. This gives the restriction of $T(G)$ to $N(v_0)$ by Corollary 2.2. The restriction of $T(G)$ to $\overline{N}(v_0)$ is found in the same way.

The main insight for the time bound, which we discuss below, is that the size of a neighbor list of a node that is a module is at most half as large as the sum of sizes of neighbor lists of its children. The sum of sizes of all neighbor lists is thus bounded by the number of active edges. This allows us to charge the cost of traversing modules to the active edges. To bound the cost of deleting, we maintain a credit invariant, discussed below, where each tree node carries a credit that pays for its possible deletion. The credit is charged to an edge that is active when the node is created.

4.2 The v_0 modules step. We define an equivalence relation on nodes of $\overline{N}(v_0)$, where for $x,y \in V(G)$, xKy if and only if x and y are contained in the same connected component of $G|\overline{N}(v_0)$ or else if they are both contained in a module of G that is a subset of $\overline{N}(v_0)$. If $G|\overline{N}(v_0)$ has more than one connected component, then the root of $G|\overline{N}(v_0)$ is degenerate, and its children are the connected components. We have computed the maximal modules of G that are contained in $\overline{N}(v_0)$ in the restrict step. By Corollary 2.2, each such maximal module is contained in a connected component, or is a union of connected components. It is then trivial to find each equivalence class of K as a union of one or more connected components of $G|\overline{N}(v_0)$. We will call these equivalence 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 both contained in a module of G that is a subset of $N(v_0)$, *or are contained in the same **co-component***. Let $\mathcal{B}$ denote the basic blocks of $N(v_0)$ and let $\mathcal{B}'$ denote the basic blocks of $\overline{N}(v_0)$.

To find strong modules containing v_0, we define a directed bipartite graph $F = (\mathcal{B}, \mathcal{B}', E_F)$. For $B \in \mathcal{B}$ and $B' \in \mathcal{B}'$, (B, B') is an edge of F if and only if there is an edge of G between B and B', and (B', B) is an edge of F if and only if there is a non-edge between B and B'. We find the strongly-connected components of F, and the **component graph**, which has one node for each strong component of F and edges telling which which strongly-connected components are reachable from which on a single edge of F [1].

LEMMA 4.1. *The topological sort of the component graph of F is unique. A set containing v_0 is a module 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 F from a basic block X to a basic block in $V(G) - X$. Since the strong modules containing v_0 are totally ordered by the inclusion relation, this establishes the theorem.

Observation 1: 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 $N(v_0) - X$ and nonadjacent to every member of $\overline{N}(v_0) - X$.

Observation 2: 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 1.

Observation 3: If X is a union of v_0 and basic blocks, it is a module if and only if there is no edge of 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 $\overline{N}(v_0) - X$, and every member of $N(v_0) \cap X$ is adjacent to every member of $N(v_0) - X$. The observation then follows from Observation 1 and the definition of 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 Theorem 2.1, $(X - Y) \cup (Y - X)$ is a module that overlaps X, a contradiction, so Y does not exist, and X is strong. □

Thus, the v_0-*modules* step reduces to finding the strongly-connected components of F, and then producing a topological sort of them according to which component has an edge to which. This will be accomplished with Theorem 3.3 in the next section.

For the time being just observe that 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 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.

4.3 The assemble step. Let $\mathcal{M}$ be the maximal modules that do not contain v_0. The members of $\mathcal{M}$ are the roots of the trees in the forest produced by the restrict step, which gives the modules of G that do not contain v_0. Let M be a member of $\mathcal{M}$. The v_0-modules step gives the smallest ancestor A of v_0 that contains M.

To assemble the modular decomposition of G, install a parent pointer from each $M \in \mathcal{M}$ to the smallest ancestor of v_0 that contains it. Each ancestor of v_0 that becomes the parent of more than one member of $\mathcal{M}$ is labeled prime. Let B be an ancestor that becomes the parent of only one member M of $\mathcal{M}$, and let A be its child that contains v_0. Since A and B are disjoint modules of G, they are either adjacent or nonadjacent. If they are adjacent, label A as a series node, and if they are nonadjacent, label A as a parallel node. If A and M are both series or both parallel nodes, delete M from the tree and let B and M's children become the new children of A.

For the correctness, note that A is degenerate if and only if $A - B$ is a module. Thus, the labeling of A as prime or degenerate is correct. If it is prime, then the siblings of B are maximal modules that do not contain v_0, so they are the members of $\mathcal{M}$ that make up $A - B$. If it is degenerate, then the correctness of the step of removing M and letting its children become children of A follows from Theorem 2.3. That the entire trees rooted at the new children of A other than B are correct then follows from Theorem 2.5.

5 An $O(n+m\alpha(m,n))$ Bound for the Decomposition Algorithm

We use a *Union-find* data structure for keeping track of the components and co-components. We will show that we perform $O(n)$ *Union* and $O(n+m)$ *Find* operations. This gives an $O(n+m\alpha(m,n))$ time bound on these operations [16], where α is an extremely slow-growing function. All other operations take $O(n+m)$ time. In the next section, we describe a modification that gives a true linear bound.

We charge edges only when they are active, though sometimes an edge pays for a credit that is laid down in the structure to pay for a later *Find* or constant-time operation. Since each edge is active only once during the entire run of the algorithm, this ensures that there are $O(m)$ of these operations.

Let R denote the recursion tree defined by the algorithm. In this tree, we let v_0 be the root, and the recursion subtrees for $N(v_0)$ and $\overline{N}(v_0)$ be its left and right subtrees, respectively. R can be pre-computed by carrying out a first pass of the algorithm with the *restrict*, v_0*-modules*, and *assemble* steps omitted.

The partition of edges into groups that become active can be carried out in linear time with an off-line least-common ancestor algorithm [6], applied to R. A simpler way involves radix sorting the edges to get the neighbor lists sorted by a preorder numbering of the vertices in R, and taking advantage of the fact that we can spend $O(1)$ time at each node in $N(v_0)$, though we will omit the details. Radix sorting the edges again with active edge group as primary key, first vertex as secondary key, and second vertex as tertiary key gives us for each vertex with an active edge a sorted list of its neighbors that are adjacent on an active edge.

We may perform two *Find* operations at each end of each active edge to find which components and co-components of $G|N(v_0)$ and $G|\overline{N}(v_0)$ the edge is incident to. Each component or co-component may then have a count of how many active edges it has to each other component or co-component.

The connected components of G are nodes of $T(G)$. When we finish with the decomposition tree, each node of the tree has a pointer to its parent, *except possibly when its parent is a connected component of G*. In this case, it carries an outdated pointer to a defunct node, but a *Find* operation may later be used if it becomes necessary to get its parent explicitly.

5.1 The restrict step.

We maintain the invariant that each node of the tree carries a credit that pays for its deletion. Let m_a be the number of active edges. The sum of of cardinality of active neighbor lists assigned to nodes in the postorder operation is $O(m_a)$, as shown before.

Moving from a child to a parent may use a pointer, or may require a *Find* operation. Let d be the number of nodes visited but not assigned these lists. These nodes are deleted. The $O(m_a+d)$ operations are charged to edges or deletion credits.

If a degenerate parent is deleted and some of its children are not, those groups of children with identical active neighbor lists must be given a new degenerate parent. If the children have nonempty active neighbor lists, this is charged to these lists. If they have empty active neighbor lists, we avoid touching them by removing from their parent those children that have nonempty lists, and allowing the old parent to remain on as the new parent of the children with empty lists. The time spent at the parent node is paid for by the removed children's active neighbor lists, so it retains its deletion credit.

5.2 The v_0-module step.

We may touch each component or co-component that has an incident active edge or that merges with another. 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.

Since each component and co-component knows the the number of active edges to other components, we may compute the mixed representation of the forcing graph F within the required number of *Find* and constant-time operations. Creating the ancestors of F then follows within the required time by Theorem 3.3.

5.3 The assemble step.

Let X be a root or child of the root in the forest produced by the restrict operation. X must be assigned a parent A from among ancestors of v_0. If $X \in N(v_0)$, charge the operation to an edge in $X \times \{v_0\}$ Otherwise, if A does not contain all of $N(v_0)$, charge it to an edge in $X \times (N(v_0) - A)$, using Lemma 4.1. We are prevented from touching it only if A contains $N(v_0)$, in which case A is a connected component containing v_0, $N(v_0)$, and all components of $\overline{N}(v_0)$ that have incident active edges. The parent pointer is left pointing to a defunct tree node that deleted by the *restrict* step. A *Find* operation can retrieve the parent at a later point when the operation can be charged appropriately.

6 Obtaining a Linear Time Bound

The only bottleneck preventing a linear time bound is the *Union-find* problem that arises in keeping track of connected components and co-connected components of subtrees of the recursion tree. We proceed as follows:

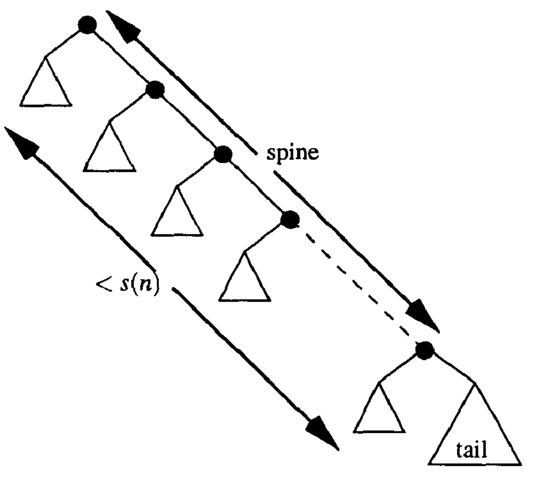

Figure 2: The Special Recursion Tree for Sparse Graphs

1. We observe that our algorithm already has a linear time bound when the graph is not sparse;

2. We solve the case of sparse graphs with a special recursion tree and data structure.

6.1 Dense Graphs. We define $s(n)$ to be $\log\log\log n$, and call G **dense** if $m \geq ns(n)$, and **sparse** otherwise. If the ratio of m/n is at least $\log^* n$, where $\log^* n$ is the extremely slow growing inverse of the tower function, then $\alpha(m,n) \leq 2$. It follows that $\alpha(n*s(n),n)$ is asymptotically less than or equal to a constant. The following is immediate:

REMARK 6.1. *On dense graphs, any Union-find problem can be solved in linear time, and the foregoing modular decomposition algorithm runs in linear time.*

6.2 Sparse Graphs. In the decomposition algorithm we have a great choice of possible pivots to use in a particular step. We will do that in a way that the recursion tree will become very biased. We do this by selecting an initial sequence S of pivots that we will call the **spine**, see Figure 2. S has the following properties:

1. S is an independent set of the graph;

2. the left subtree of each member v of S (the subtree that contains the vertices adjacent to v) is of size less than or equal to $s(n)$;

3. all vertices in the right subtree of the last member of S (the **tail** of S), have degree higher than $s(n)$.

Such a biased tree is in strong contrast to what was looked for in the parallel case [4], where a balanced tree was preferred.

By G_v we will denote the subgraph induced in G by a spine vertex v and its descendants in the recursion tree. A descendant w is **subordinate** to v if w and v are in the same connected component of G_v. From each connected component of the tail, select a representative vertex x and make the other vertices of the component subordinate to it. It is easily verified that the transitive reduction of the subordinate relationship is a tree, which we will call the **subordination tree**. We will call a vertex's parent in this tree its (immediate) **boss**.

For any spine vertex v, the connected components of G_v are also connected components of the subordination tree. Thus, if we can compute the subordination tree, we may then use Gabow-Tarjan [6] to manage the connected components as we work upward through the spine inductively, computing the modular decomposition of each G_v.

Our strategy may now be summarized as follows.

- In order to select spine vertices and compute the subordination tree, we will perform some *Union-find* operations on vertices that we know are destined to belong to the tail. We may use conventional *Union-find*, since the tail is dense.

- We use micro-encoding for the left subtrees of spine vertices. That is, using our foregoing decomposition algorithm, pre-compute the modular decompositions of all graphs on $s(n)$ vertices. Since there are at most $2^{s(n)^2} \ll n$ such graphs, this takes $O(n)$ time. We may assume that the word size is at least $\log n$. Because $s(n)$ is much smaller than $\log n$, the adjacency matrix representation of such a graph fits into a single word. We may store its decomposition in a table entry that is indexed by this word. Thus, once the table is computed, we may look up the modular decomposition for any left subtree of a spine vertex in $O(1)$ time.

- We use Gabow & Tarjan, [6], and the subordination tree as we work back up through each spine vertex v, computing connected components of G_v, and thereby the modular decomposition of G_v. Relevant co-components come only from left subtrees, so we needn't update co-components as we move up the spine.

6.3 Finding the Spine and the Subordination Tree. To compute the subordination tree, we will install a pointer from each vertex to its boss. Clearly, a spine vertex v is the boss of the vertices in its left subtree. For each connected component of the tail, we select a representative r to be the boss of the remaining vertices. It remains to show how to compute the bosses of the representatives and the spine vertices.

When we select a spine vertex v, the vertices that are destined to be its descendants in the recursion tree are those that are not in a left subtree of some spine vertex already selected. Let X denote this set. When a new spine vertex w is selected, it becomes the child of the previously selected spine vertex v. We update X by setting $X := X - (\{v\} \cup N(v))$. The spine is complete when no candidates remain in X; at this point X is the tail.

The **candidates** are those members of X that have degree less than or equal to $s(n)$; these are the candidates to become the next spine vertex selected. To select the next spine vertex, we use the following rule:

- Find the lowest spine vertex y from among those already selected that has a subordinate z among the candidates, and select z to be the new spine vertex.

If this rule is followed, then z's boss is clearly y.

Next, observe that if z is subordinate to y, then there is a path from y to z in G_y. Since all neighbors of y make up y's left subtree, this path must have an edge (s,t) passing from y's left subtree to its right subtree. If y is lowest vertex that has a subordinate, then the suffix of the path starting at t must clearly be restricted to X.

Algorithm 2 Create a Spine

Input: A sparse graph G and a list of candidates L.

```
while $L \neq \emptyset$
    first:    Take a pivot $t \in L$.
    update0:  Delete $t$ and its neighbors from $L$.
    insert:   Put $t$ as new node into the spine and its remaining neighbors to the left subtree. These are less than or equal to $s(n)$.
    update1:  Update $Q$.
    branch:   If $Q \neq \emptyset$ then let $v$ be its last element and let $t'$ an endpoint of one of the remaining crossing edges of $v$. Otherwise quit the inner block.
    update2:  Delete the corresponding edge from the list and delete $v$ from $Q$ if its list has become empty.
    choose:   Choose a pivot $t$ according to $t'$.
    iterate:  Go to insert.
```

Initially, we set up an empty stack Q of edges. Each time a spine vertex is selected, we identify any edges that go from its left subtree to X, and push them on Q. To select the next spine vertex, we pop edges from the stack until we find an edge (s,t) such that there a path from t to a candidate that does not leave X. The candidate z that the path leads to becomes the next spine vertex selected, and its boss is the boss of s. It is easily verified that this method obeys the pivot selection rule.

The entire problem thus reduces to finding efficiently whether there is a path from t to a candidate, and finding the candidate z that it leads to. To facilitate this operation, we maintain the following (pseudo-) distance function on vertices of X.

- 0 for vertices with degree at most $s(n)$ in G. These are the **candidates**, since they are candidates to be selected as spine vertices.
- 1 for vertices of X that are not candidates, but that are neighbors of candidates.
- 2 for the remaining vertices of X from which there is some path (of length at least 2) to a candidate.
- ∞ for the vertices from which there is no path to a candidate, the **losers**.

This classification tells much about the eventual relationship of vertices of X to the recursion tree:

- A candidate is destined to remain a candidate until it leaves X.
- A loser cannot leave X, and is thus destined to become part of the tail.
- A 2-vertex cannot leave X, so it is destined to become a loser, and then part of the tail.
- A 1-vertex may leave X, or else it will become a 2-vertex or a loser when all of its candidate neighbors leave X.

There is a path from t to a candidate if and only if it is not a loser. Let G_2 be the subgraph of G induced by the current 2-vertices and losers. We keep track of each connected component of G_2, using a conventional *Union-find* data structure, since G_2 is dense. For each 2-component we keep a doubly-linked list edges to 1-nodes. For each 1-node, we keep a doubly-linked list of edges to 0 nodes. Thus, given any 2-node, we may find a candidate that is reachable from it by performing a *Find* to get the list of edges from its component to 1 vertices. In $O(1)$ time, we may then look up a candidate neighbor of that 1 vertex.

When an vertex w leaves X, it is trivial to update these edge lists in time proportional to the degree of w. If w is promoted from a 1-vertex to a 2-vertex, we must perform a *Union* involving w and the components of G_2 that have neighbors of w, and concatenating their lists of 1-neighbors.

When the list of 1-neighbors of a component of G_2 becomes empty, its members become losers. When this happens, the set is destined to be a connected component of the tail. Select a representative r and make it the boss of the other losers in the set. The boss of r is the spine vertex v whose selection caused it to become a loser; r is clearly subordinate to v, and since there is no path from r to any other candidate in X, no spine vertex selected after v can be r's boss.

Summarizing, we have shown that we may find a candidate that is reachable from an arbitrary one- or two-node in $G|X$ using miscellaneous $O(1)$ operations and a *Find* operation. It suffices to do this once for each edge that is popped from Q to get the list of candidates and the boss pointers. Since these *Union* and *Find* operations are on sets of 2-vertices, their degree is greater than $s(n)$, they are "dense", and the amortized cost of the *Union* and *Find* operations is $O(n+m)$.

7 Future Work

There is a type of dual relationship between modular decomposition and the transitive orientation problem. Modular decomposition was first discovered in this context [7]. McConnell and Spinrad obtained a linear time bound for the transitive orientation problem by exploiting this relationship during execution of a linear-time decomposition algorithm [10]. The reduction requires a large number of insights and data structures, and multiplies the complexity of a decomposition algorithm that is already elaborate. It seems possible that a more tractable proof of the time bound can be obtained from the algorithm of this paper, as it decomposes the graph in a much more orderly and straightforward fashion.

Depth-first search was efficiently generalized to the complement or mixed representation of a graph. Similar tricks can be applied to breadth-first search. A key element in the most efficient time bounds for permutation-graph recognition is the ability to find a transitive orientation and compact representation of a transitive orientation of the complement of a graph in linear time [9, 10]. It is fairly easy to use them to modify the Rose and Tarjan algorithm for recognizing chordal graphs [14], so that it recognizes whether G is the complement of a chordal graph in time that is linear in the size of G. By that, one can easily recognize split graphs since they are exactly those chordal graphs where the complement is chordal, too.

It also might possible to improve the methods presented here to recognize weakly triangulated graphs in linear time. The best time bounds known so far are $O(n^2m)$, see [15].

We have given one instance where a basic algorithm on a mixed representation could be applied in a way that it could not on a standard representation or its complement. Whether mixed representations are of broader interest depends on how much the input size of graphs under investigation can be reduced. A sparser representation of certain graphs has an impact on complexity measures. It is an interesting question which other basic algorithms can be generalized efficiently to them, and whether these generalizations have uses. Algorithms that can be run on the complement in linear time are likely candidates to be generalized in this way.

References

[1] T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, Mc Graw Hill, 1990.

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

[3] A. Cournier and M. Habib, *A new linear algorithm for modular decomposition*, in 19th International Colloquium CAAP '94, S. Tison, ed., Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 68–82.

[4] E. Dahlhaus, *Efficient parallel modular decomposition, extended abstract*, in Graph-Theoretic Concepts in Computer Science, 21th International Workshop WG '95, Nagl et al., eds., vol. 1017 of Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 290–302.

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

[6] H. N. Gabow and R. E. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1984), pp. 209–221.

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

[8] J. Gustedt, *Efficient union-find for planar graphs and other sparse graph classes*, in Graph-Theoretic Concepts in Computer Science, 22nd International Workshop WG '96, Ausiello et al., eds., Lecture Notes in Computer Science, Springer-Verlag, 1996. *to appear.*

[9] R. M. McConnell and J. P. Spinrad, *Linear-time modular decomposition and efficient transitive orientation of undirected graphs*, in Proceedings of the fifth annual ACM-SIAM Symposium on Discrete Algorithms, D. D. Sleator et al., eds., Society of Industrial and Applied Mathematics (SIAM), 1994, pp. 536–545.

[10] ———, *Linear-time transitive orientation*, 1997. same volume.

[11] R. H. Möhring, *Algorithmic aspects of comparability graphs and interval graphs*, in Graphs and Orders, I. Rival, ed., D. Reidel Publishing Company, Dordrecht, 1985, pp. 41–101.

[12] ———, *Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions*, Ann. Oper. Res., 4 (1985), pp. 195–225.

[13] ———, *Computationally tractable classes of ordered sets*, in Algorithms and Order, I. Rival, ed., Kluwer Acad. Publ., Dordrecht, 1989, pp. 105–194.

[14] D. J. Rose and R. E. Tarjan, *Algorithmic aspects of vertex elimination*, SIAM J. Appl. Math., (1978), pp. 176–197.

[15] J. Spinrad and R. Sritharan, *Algorithms for weakly triangulated graphs*, Discrete Appl. Math., 59 (1995), pp. 181–191.

[16] R. E. Tarjan, *Data structures and network algorithms*, Society of Industrial and Applied Mathematics (SIAM), Philadelphia, 1983.