THESIS

MODULAR DECOMPOSITION OF UNDIRECTED GRAPHS

Submitted by

Adrian E. Esparza

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2020

Master's Committee:

Advisor: Ross McConnell

Sangmi Pallickara
Alexander Hulpke

ABSTRACT

MODULAR DECOMPOSITION OF UNDIRECTED GRAPHS

Graphs found in nature tend to have structure and interesting compositions that allow for compression of the information and thoughtful analysis of the relationships between vertices. Modular decomposition has been studied since 1967 [1]. Modular decomposition breaks a graph down into smaller components that, from the outside, are inseparable. In doing so, modules provide great potential to better study problems from genetics to compression. This paper describes the implementation of a practical algorithm to take a graph and decompose it into it modules in $O(n+m\log(n))$ time. In the implementation of this algorithm, each sub-problem was solved using object oriented design principles to allow a reader to extract the individual objects and turn them to other problems of practical interest. The purpose of this paper is to provide the reader with the tools to easily compute: modular decomposition of an undirected graph, partition an undirected graph, depth first search on a directed partially complemented graph, and stack operations with a complement stack. The provided implementation of these problems all compute within the time bound discussed above, or even faster for several of the sub problems.

ACKNOWLEDGEMENTS

I would like to thank my family for their ceaseless support. I would also like to thank my advisor Ross McConnell for his guidance and help. The Shindlebowers for treating me like family and caring for my dog. My friends Dave Bessen, Cole Frederick, and Gareth Halladay for ever more unflagging support. In addition a special recognition for Sophia Fantus for brilliant proof reading. Know that this work would not have been possible with out all of you. Thank you and congratulations on work well done.

DEDICATION

I would like to dedicate this thesis to my family, good job getting it done team Esparza.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

According to Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell [2], modules have been studied since 1967 [1]. Modules represent interesting structures within graphs. A module is some nonempty set X of vertices in a graph $G = (V, E)$, such that $\forall u \in (V - X)$, u is either a neighbor of every vertex in X, or a non neighbor of every vertex in X. A *spoiler* for X is a vertex $u \in V - X$, such that X contains both neighbors and non neighbors of u. X is a module if, and only if, it has no spoilers.

The lack of spoilers is what make modules novel structures for study. For example, optimization problems that are hard in the general case, can be solved in linear time on graphs with a non trivial modular decomposition. The beginning of this thesis will cover related research into modules, problems that have linear time solutions, and graph classes that modules can help identify. The rest of the thesis covers an $O(n+m\log(n))$ algorithm used to find the modular decomposition of a graph as well as describing our implementation of a program that runs said algorithm.

In chapter 2, we will discuss general graph terms and definitions that will be used throughout the paper. In chapter 3, we will examine the reasons modules are interesting to other types of graph theory problems. In chapter 4, we will describe how, in using modular decomposition, it is possible to find linear time solutions to *max weight independent set*, *transitive orientation*, and detecting *interval graphs* [3]. In chapter 5, we will consider an algorithm for computing the modular decomposition of a graph in $O(n + m\log(n))$ time [4]. In parallel to describing the algorithm, we will describe an implementation of said algorithm with explanations for design decisions.

Chapter 2

Preliminaries

A *directed graph* is represented by a set of vertices V, and a set E of ordered pairs of vertices, known as edges. An *undirected graph* is a special case of a directed graph, where $(x, y) \in E \iff (y, x) \in E$. In an undirected graph, the symmetric pair $\{(x, y), (y, x)\}$ can be denoted as an undirected edge xy. For simplicity, in this thesis, a graph will refer to an undirected graph, unless otherwise indicated. The variables n and m will be reserved for representing the size of the sets V and E, respectively.

A *subgraph* of G is a graph $G' = (V', E')$, where $V' \subseteq V$ and E' is some subset of E, such that every edge in E' has both its endpoints in V'. An *induced subgraph* is a graph $G' = (V', E')$, where $V' \subseteq V$ and $\forall a, b \in V', (a, b) \in E'$ iff $(a, b) \in E$. In this thesis, $G[X]$ shall represent the subgraph of G induced by X, where X represents a set of vertices in G. Unless otherwise indicated, all subgraphs referenced in this paper shall be induced subgraphs.

When (x, y) is a directed edge, y is adjacent to x. For a given vertex x, the *neighborhood of x* $N(x)$ will denote $\{y \in V : (x, y) \in E\}$. That is, $N(x)$ will represent the set of vertices in E that x is *adjacent* to. A *complement*, $\overline{G} = (V, \overline{E})$ of a graph $G = (V, E)$ retains all the vertices of the original graph, but the edges in G are non edges in $\overline{G}$ and the non edges in G are edges in $\overline{G}$. That is, $\forall a, b \in V$, $(a, b) \in \overline{E}$ iff $(a, b) \notin E$.

Two sets X and Y are *properly overlapping* if their intersection is non empty, but neither is contained within the each other. Written formally, if X, Y are properly overlapping sets then $|X \cap Y| > 0$ and $|X \cap Y| < |X|$ and $|X \cap Y| < |Y|$. A *partition* of a set X is a family $\{X_1, X_2, ..., X_k\}$ of subsets of X, such that every element in X is a member of exactly one set X_i in the family.

A *disjoint union* of two graphs takes two graphs, $G = (V, E)$ and $G' = (V', E')$, which share no vertices, and creates a new graph $H = (\{V + V'\}, \{E + E'\})$. A *path* exists between vertices x and y if there exists a set Z of edges in E, such that starting with x it is possible to traverse the

edges of Z and reach y. A *cycle* exists when there is a set of edge that can be traversed from some vertex x which leads back to x. A *DAG* is a directed graph with the condition that no cycles exist, also known as a *directed acyclic graph*. A set of vertices where a path exists between any two vertices of the set is known as a *strongly connected component*.

A *clique* C within a graph is a maximal set of vertices, such that $\forall v, w \in C : v \neq w, (v, w) \in E$. The set of cliques in a graph is denoted $\mathcal{C}_G$. The notation, $\mathcal{C}_G(x)$, denotes the set of cliques which contain some vertex x, and $\mathcal{C}_G(\bar{x})$ denotes the cliques which do not contain x. A *valid coloring* of a graph is the assignment of colors to vertices of the graph, such that no two vertices of the same color are adjacent to one another. For a graph $G = (V, E)$, $|V|$ is clearly an upper bound on the number of colors needed for a valid coloring, by assigning each vertex a unique color. A *min coloring* is the minimum number of colors necessary for a valid coloring. It is trivial to show that the size of a max clique is a lower bound on a min coloring.

Proof. By contradiction. Consider that a coloring assigns the same color to two vertices in the clique. By the definition of a clique, those two vertices are adjacent; therefore, the min coloring is not valid. □

Opposite a clique is a *kernel*, a maximal independent set of vertices in G. For some kernel, K, $\forall v, w \in K, (v, w) \notin E$. The set of kernels in a graph is denoted $\mathcal{K}_G$, while $\mathcal{K}_G(x)(\mathcal{K}_G(\bar{x}))$ represents the kernels in G that contain(do not contain) some vertex x. A graph is said to have the clique-kernel inter-section property (or *CK-property*) iff every clique of G has one vertex in common with every kernel of G.

There is a class of optimization problems called NP-hard, for which no polynomial time solution is known. A polynomial time solution for any problem in this class would solve the long standing P=NP problem.

Chapter 3

Modules

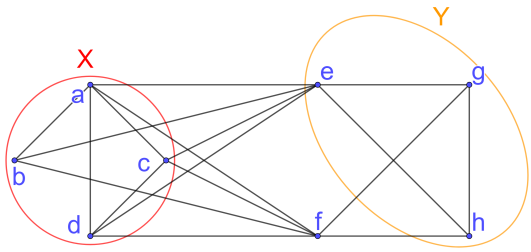

Figure 3.1: X represents a module, Y represents a potential module spoiled by f

The set V and its one-element subsets are clearly modules; these are known as the *trivial modules* of G. In figure 3.1, X is a nontrivial module because it has no spoilers. Conversely, the set Y is not a module, as it represents a set of vertices for which f is a spoiler. Vertex f has edges to g and h, but no edge to e.

A graph can be "factored" into *quotients* and *factors*, such that the original graph is uniquely obtainable from the derived quotients and factors. The left side of figure 3.2 gives an example of both a factor of G and an induced subgraph $G[X]$. The right side of figure 3.2 represents a quotient of G, $G/\{V-X, X\}$, where X is collapsed to a single point whose neighbors are the neighbors of the elements of X in $V-X$. Alternatively, as X represents a module, the quotient can be said to be an induced subgraph of G, where the set inducing the subgraph is $\{V-X$, any vertex in $X\}$. As X is a module, this process is invertible: $G[X]$ may be substituted back in for X in the quotient and the neighbors of X in $V-X$ will be neighbors of each element of X.

Lemma 3.0.1. *Let G be a graph and X be a module of G. A subset Y of X is a module of $G[X]$ iff it is a module of G*

Proof. $\boldsymbol{Y \subset X}$ **is a module of** $\boldsymbol{G[X]} \implies \boldsymbol{Y}$ **is a module in** $\boldsymbol{G}$**:** Suppose Y is a module of $G[X]$ and it therefore has no spoilers in $X-Y$. As X is a module of G, there exist no spoilers for

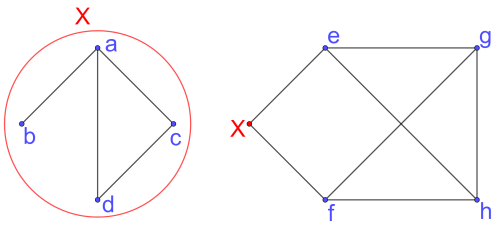

Figure 3.2: The factor is the subgraph induced by X, and the quotient is the original graph with X collapsed to a point.

Y in $V - X$, which means there are no spoilers in $V - Y$. Since no spoilers for Y exist outside of Y, Y must be a module in G.

$Y \subset X$ is a module in $G \implies Y$ is a module of $G[X]$: Suppose Y is a module of G and therefore has no spoilers in $V - Y$. As Y is a proper subset of X, the set $X - Y$ is non empty, and contained entirely within $V - Y$. No spoilers for Y exist in $V - Y$, so no spoilers for Y exist in $X - Y$, meaning Y is a module in $G[X]$. □

Since modules in a subgraph induced by another module are still modules in the larger graph, the process of finding them can be applied recursively when the factors have nontrivial modules. In the event that a graph has no nontrivial modules, it is said to be *prime*. As an example a P_4, a path on four vertices is a prime graph.

If a graph is prime, its factors are the the individual vertices and its quotient is exactly the original graph. Conversely, if a graph is not prime, it is possible for its factors and quotient to be smaller, hinting towards a recursive solution. One potential pitfall is the number of modules in a graph can be exponential. Fortunately, there is a unique way to find a recursive decomposition of the graph that implicitly represents *all* ways of decomposing the graph recursively. Equivalently, it implicitly represents all modules of a graph. This recursive decomposition is called the *modular decomposition*, and an implementation of an algorithm for finding it is the subject of this thesis.

While it has been proven that there is a linear algorithm for finding the modular decomposition of a graph [3], this work focuses on the implementation of a $O(n + m\log(n))$ [2].

Let the modular decomposition of a graph $G(V, E)$ be a rooted tree, where every node is a module. The root will be a node representing V and the trivial module of the entire graph.

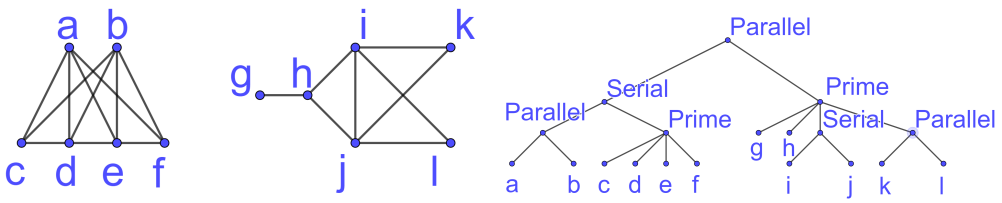

Figure 3.3: A graph with several modules within it along with its modular decomposition.

Algorithm 1: Algorithm for representing a graph by its modular decomposition.

Input: Sets V, E representing a graph.

Result: $MD(G)$ = A rooted tree representing the modular decomposition of G.

if $|V| = 1$ **then**
| It is just a one-node tree and a trivial module. Return V

else if *Case 1: If G is disconnected* **then**
| The children of V are the connected components.

else if *Case 2: The complement of G is disconnected.* **then**
| The children of V are the connected components of the complement.

else
| Case 3: None of the above. Then the maximal modules that are proper subsets of V
| are a partition of V, a fact proved below. These are the children of the root.
The sub-tree rooted at each child X of V is obtained by recursion on $G[X]$.

Looking at the cases from the algorithm above there are three potential cases for internal nodes to fall into.

- Parallel nodes (case 1)
- Series nodes (case 2)
- Prime nodes (case 3)

There are small quotients whose modular decomposition is ambiguous, namely when the quotient is a graph with two vertices. When considering the modules that represent these quotients assume them to be non prime. The fact that these modules must be either disconnected or completely connected will allow them to be classified as parallel or series nodes, respectively.

Lemma 3.0.2. *If X and Y are properly overlapping modules in G then $X \cup Y$, $X \cap Y$, $X - Y$ and $Y - X$ are also modules in G.*

Proof. **$X \cup Y$ is a module in G**: Let $z \in V - \{X \cup Y\}$. Since X is a module in G, if z has no edge to an element in X, it has no edge to any element in X. Since X properly overlaps with Y, there exists some element y of Y that is also in X. As z has no edge to anything in X, z has no edge to y. Since Y is also a module in G, and z lacks an edge to some element of Y, z has no edges to anything in Y. By symmetry, if z has an edge to X, it has edges to everything in X and therefore everything in Y, $X \cup Y$ is a module in G.

$X \cap Y$ is a module in G: For $z \in V - \{X \cup Y\}$, z could not act as a spoiler for $\{X \cap Y\}$ without spoiling both X and Y. Therefore, consider that z is an element of $V - \{X - Y\}$. X and Y are modules of G. By definition, there can exist no spoilers to a module outside of a module. There can be no spoilers in $X - Y$ for $\{X \cap Y\}$, otherwise X would contain a spoiler for Y and Y would not be a module. By a similar argument there can be no spoiler in $Y - X$ for $\{X \cap Y\}$, therefore $X \cap Y$ is a module within $G[X \cup Y]$ and within G at large.

$X - Y$ and $Y - X$ are modules in G: As discussed above, $\{X \cap Y\}$ is a module and therefore every element within it shares the same neighborhood for elements outside of the module. Since $\{X \cap Y\}$ is a contained within Y, it must also share the same neighborhood as elements in Y. The non empty set $\{Y - X\}$ contains no spoilers for X, and is also contained within the module Y. Since $\{X \cap Y\}$ and $\{Y - X\}$ are contained within the same module, they must share the same neighbors outside of the module; therefore, $\{X \cap Y\}$ contains no spoiler for $\{X - Y\}$, and $\{X - Y\}$ is a module in G. By the symmetric argument, $\{Y - X\}$ is a module in G. □

Lemma 3.0.2 allows for the exponential number of modules, as discussed earlier. Yet, the cases described above also allow for these modules to be implicitly represented in the modular decomposition. In *Case 1*, where the graph is disconnected, every subset of G is a module. In the modular decomposition of G, the root of G would be a *parallel* node and all subsets of its children would be modules. Similarly, in *Case 2*, the root of G would be a *series* node and, all subsets of its children, would be modules.

Since the leaves are already single element sets, and therefore trivial modules, the only modules left to represent in the modular decomposition must be in the *prime* nodes.

Lemma 3.0.3. *If a Node X is prime, then the maximal modules that are proper subsets of X are a partition of X*

Proof by contradiction. **Every element in V exists in at least one partition class of $\mathcal{P}$:** Assume some element y in X does not appear in any partition class of $\mathcal{P}$. Since even a single element sets can be maximal modules, we know that y must still exist in some maximal module Z. Therefore, $Z \cup \{y\}$ is a maximal module of X, and not represented by a partition class in $\mathcal{P}$. Since our definition is that maximal modules are in a single partition class, Z should not have been a partition class; rather, $Z \cup \{y\}$ is a partition class in $\mathcal{P}$ and $\{y\}$ is indeed stored in at least one partition class.

Every element in X exists in only one partition class of $\mathcal{P}$: Assume some element x is contained within two partition classes, Y and Z. These both represent maximal modules of X and, therefore, cannot be expanded any more. Accordingly, they must properly overlap. By lemma 3.0.2, the union $Y \cup Z$ is a module. It contains both Y and Z as proper subsets and is a maximal module in X. Since $Y \cup Z$ is a maximal module, neither Y nor Z is maximal, contradiction. □

Theorem 3.0.4. *The modules of G are the nodes of $MD(G)$ and the unions of children of serial and parallel nodes.*

Proof. Case 1: Parallel nodes represent disconnected quotients, where the connected components are modules represented as the children of the root. Since the quotient is disconnected, there are clearly no spoilers to a module formed by any union of children described in lemma 3.0.2. Therefore, every module of G is either a union of children or contained in a child. By symmetry of edges vs. non-edges, the same thing will apply in Case 2.

In Case 3, by Lemma 3.0.3, every module of G is either V or is contained in one of the children of the root. □

By Theorem 3.0.4, we can represent the modules of G in $O(n)$ space. Each node X is represented with an $O(1)$ data structure x that has a pointer to a list of its children. Each leaf has

an identifier that tells which node of G it is a member of. To return the set X represented by a node x, it suffices to collect the identifiers of the leaf descendants of X, using a call to depth-first search. As each node of the tree has at least two children, this takes $O(|X|)$ time. This represents explicitly that X is a module of G. In addition, each internal node is labeled according to whether it is a series, parallel, or prime node. If it is a series or parallel node, this represents implicitly the unions of children are modules of G. Since every node has at least two children, and the tree has n leaves, the number of internal nodes is at most $n - 1$.

Chapter 4

Applications

4.1 Compact representation of a graph

In order to allow the modular decomposition to represent not just the modules of a graph G, but also G itself, it suffices to label each node X with its *quotient*. The quotient is is obtained from $G[X]$ by contracting each child of X to a point. If the child is a leaf, it suffices to let its only member represent this point.

For example, in figure 3.3, the right child of the root is $\{g, h, i, j, k, l\}$. After replacing $\{i, j\}$ with a node x and $\{j, l\}$ with a node y, we obtain a quotient $H = (g, h, x, y, \{gh, hx, xy\})$.

Note, however, that the quotient at each series node is just a complete graph and, at each parallel node, is just an empty graph. There is no need to represent this explicitly; it is represented implicitly by the label of the node as a parallel or series node. Only the prime nodes need to have the quotient explicitly represented. Let us call this the *labeled modular decomposition*.

Lemma 4.1.1. *The quotient label at a prime node is a prime graph.*

Proof. Assume that within the quotient $G = (V, E)$ there exists a non trivial module X. The set of vertices contained within X share the same neighborhood in $\{V - X\}$. At this point, V or $\overline{V}$ is disconnected, the node cannot be prime, contradiction. □

G can be obtained from this labeling of the modular decomposition by working by induction from the leaves toward the root. The leaves are trivial modules. For every internal node representing some module X in G, $G[X]$ is obtained by taking its factors $G[Y]$ and plugging them into the quotient. In the construction of $G[X]$, the purpose of the labels become obvious. Each of the three internal nodes contains information for how its children are related to one another. In the case of a parallel node, $G[X]$ is disconnected and no edges need to be added among the children of X. A series node, is disconnected in the complement, so when constructing $G[X]$ add edges between all

children of X. Prime nodes contain the quotient of the $G[X]$, explicitly describing the relationship between the children of X. Add all edges as appropriate and recurse up the tree.

Because an edge in a quotient can represent many edges in G, this representation can achieve considerable savings in the space required to represent G. However, in the worst case, G is a prime graph, the labeled modular decomposition has just one internal node, V, and its quotient is G itself, resulting in no space savings. For some classes of graphs without prime modules, such as *cographs* discussed below in 4.3, this provides a better worst-case bound on space.

4.2 Max weight independent set

As discussed above, some classes of optimization problems are NP-hard; in the general case, there is little hope for a polynomial time solution. Two such problems are finding a maximum independent set, and a maximum-weight independent set. However, if a graph has nontrivial modules, the modular decomposition given by the following algorithm, which finds the weight of a maximum weight independent set, can be faster than solving the problem by brute-force on G.

The algorithm takes a graph G whose vertices are labeled with weights and returns the weight of a maximum-weight independent set.

Find the labeled modular decomposition of G.

Recursively, find the weight of the maximum-weight independent set of $G[X]$ for each child X of V.

1. If the graph is disconnected, the max weight independent set is the sum of the weights of all children; return this weight.

2. If the graphs complement is disconnected, the max weight independent set is equal to the max value of its children; return this weight.

3. If V is a prime node, let the weight of each node x of the quotient be the weight of a maximum independent set of $G[X]$, for the child X represented by x. Return the weight of the maximum-weight independent set of the quotient when it is weighted in this way.

Step three indicates that the graph G is prime and is the only case where the calculation is nontrivial. This reduces the problem of finding the max weight independent set on a graph to finding it on the prime graphs in its labeled modular decomposition.

That caveat means that just like in the example of compression, prime modules are a subgraph that prevents modular decomposition from solving this quickly, in the general case.

4.3 Cographs

Cographs also called *Complement Reducible Graphs* have been discovered independently over the years in various mathematics problems.

Lemma 4.3.1. *A definition of a cograph from Corneil [5]:*

1. *A single vertex.*
2. *The vertex disjoint union of cographs.*
3. *The complement of a cograph.*

In a cograph, a pair of vertices {x, y} are considered *twins* if they are a module. Twins are considered strong if they are adjacent and weak otherwise.

Given this definition, consider how these graphs are constructed, and how their construction ensures modules and might allow a modular decomposition tree to represent them. A single vertex is a module. The vertex disjoint union of graphs creates a parallel module where every vertex set and every union of vertex sets is a module. Those modules' membership is unaffected by complementing G, with the only change to $MD(G)$ being that parallel modules will become series and vice versa.

Thus, we only have the case of prime nodes in the modular decomposition to be mapped to structures in cographs.

Theorem 4.3.2. *The following are equivalent statements for every cograph G [5].*

1. *G is a cograph.*

2. *Any nontrivial subgraph of G has at least one pair of siblings.*

3. *Any subgraph of G has the CK-property.*

4. *G does not have a P_4 as an induced subgraph.*

5. *The complement of any nontrivial connected subgraph of G is disconnected.*

Lemma 4.3.3. *A graph is a cograph iff it contains no prime modules. By 4.3.2.2, there can be no prime modules in a cograph.*

Proof. A graph is a cograph $\rightarrow$ it contains no prime modules: Let G represent a cograph and P represent a non trivial subgraph of G, which is prime. By 4.3.2.2, P contains at least one pair of siblings. These siblings will share exactly the same neighborhood as one another, they can be factored out of the modular decomposition of P. Thus, P contains some non trivial module, and cannot be prime.

A graph contains no prime modules $\rightarrow$ it is a cograph: If a graph G lacks any prime modules in its modular decomposition then every non trivial module within it must contain at least one pair of siblings. By 4.3.2.2 above, it must be a cograph. □

Therefore, in order to be a cograph, a graph can have no prime graphs. As discussed in section 4.1, graphs with no prime modules are easily compressed using their modular decomposition. In fact, cographs can be represented in $O(n)$ space by leveraging their modular decomposition. To reconstruct the graph G, start recursively at the leaves and move up the tree. As each of the leaves is trivially a module, let us show that, given the children of an internal node, that module in the graph can be reconstructed.

Given some internal node X, the children of X will be connected according to the label of X. If X is a parallel node, then its children are disconnected, and no edges need to be added to the graph. If X is a series node, then the complement of X was disconnected in the original graph. Add edges between every pair of children of X. Recurse up the tree.

Lemma 4.3.4. *Every quotient of a prime module must have a P_4 as an induced subgraph. [6]*

Since point 4 in theorem 4.3.2 states that no cograph has a P_4 as an induced subgraph and lemma 4.3.4 indicates that every prime graph has an induced P_4. We know for certain that case three of 1, the one instance where keeping an explicit list of edges in the quotient is necessary, will never be hit in the case of a cograph.

As sections 4.1 and 4.2 described above, graphs lacking prime subgraphs do allow some problems to be solved much better than the general case. Other examples include a linear time $O(n+m)$ algorithm for finding a transitive orientation [3]. The implications of being able to find a transitive orientation in linear time will be discussed below in 4.4.

4.4 Transitive Orientation

A directed graph is *transitive* if, whenever (x, y) and (y, z) are directed edges, so is (x, z). A graph is a *comparability graph* if it is possible to orient it so that the resulting digraph is transitive. However, not all graphs are comparability graphs. Looking at the definition of a transitive graph, it should be clear that if $\{x, y\}$ are adjacent and $\{y, z\}$ are adjacent, there is a forcing relationship between the directed edges in the transitive graph. This binary relation Γ, called *gamma*, operates on the edges of an undirected graph $G = (V, E)$:

$$(a, b)\ \Gamma\ (c, d) \quad \text{iff} \quad \begin{cases} \text{either} & a = c \text{ and } (b, d) \notin E \\ \text{or} & b = d \text{ and } (a, c) \notin E \end{cases} \tag{4.1}$$

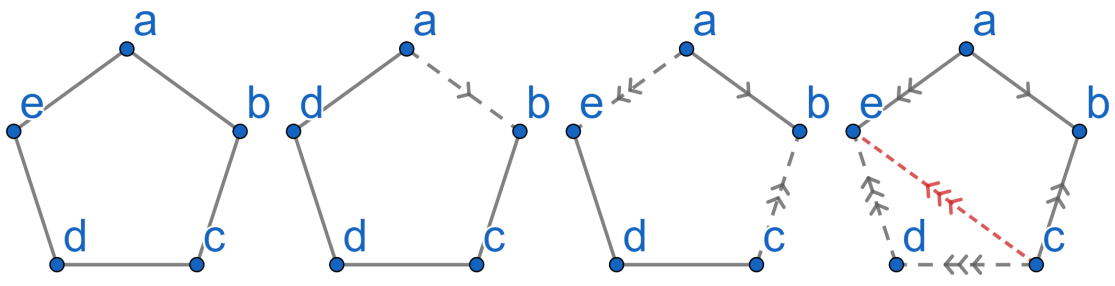

Figure 4.1: A C_5 and steps demonstrating the forcing relation, and how it shows a C_5 is not transitively orientable and thus not a comparability graph.

Figure 4.1 shows a C_5, a cycle on five vertices, and assigns an orientation to the edge between (a, b); namely, it removes the edge (b, a) that existed in the undirected graph. Using the forcing relation, we can now say that (a, e) is forced, since $\{b, e\}$ are non adjacent. Conversely, (b, c) cannot exist in the transitive orientation because $\{a, c\}$ are non adjacent, forcing (c, b). Finally, (d, e) is forced by (a, e), and (c, d) is forced by (c, b). But, since $\{c, e\}$ are non adjacent, this is not a transitive orientation.

The set of edges forced by an orientation of an edge ab is called an *implication class*. Simply, by selecting any edge in an implication class, all of the other edges in that class are forced, directly or otherwise.

This Γ relation interacts with modules in a rather elegant way. If we say that an edge is contained within a module iff both endpoints are within the module, we can refer to it as an *interior edge*.

Lemma 4.4.1. *If (a, b) is an edge interior to the module, then everything it indirectly forces is also interior to the module.*

Proof. By contradiction. Suppose X is a module, let (a, b) be an interior edge to X that forces some edge (b, c) not interior to X. The element c must clearly not be contained within X or (b, c) would be an interior edge. Since c is outside of the module, and (a, b) forces (b, c), there can exist no edge between a and c. Therefore, c is a spoiler for X, contradiction. □

By a similar proof, it is easy to show that an edge external to a module X cannot force an edge internal to X. Now that it is clear that the Γ relation stops at the boundaries of a module, let us consider how it appears within modules. In a parallel module, there are no edges between any of the children of the quotient. No edges means the Γ relation has nothing to force. It also means that in the transitive orientation, these vertices have no constraints on their relation to one another, and can be arranged in any path so long as there are no cycles in the digraph. The quotient of a series module forms a clique. Since every element in the quotient is adjacent to every other element, again there is nothing for the Γ relation to do and, again, you may arrange the elements in any path so long as there are no cycles in the digraph.

Finally, we come to the prime module. In a prime module, X, both the subgraph induced by X and its complement, are connected. Here, the Γ relation has room to act, and simplifies the work of finding a transitive orientation of the quotient.

Lemma 4.4.2. *If a module is prime, any implication class of its quotient spans every vertex in the quotient.*

Proof. By contradiction. Assume for some prime module X, whose quotient is Y and an implication class A, $\{a, b, c\} \in X$, and the span of A includes $\{a, b\}$, but not $\{c\}$. Since X is a prime module, neither Y nor its complement are disconnected. There exists at least one vertex in Y with an edge to c, and another with a non edge to c. Without loss of generality, let a have an edge to c, and b have a non edge to c. Since $a, b \in A$, there must be a forcing relation which implies the direction of edges on a path from a to b. Vertex c has an edge to one and not the other, which means that it is also forced by the same relation, meaning $\{c\} \in A$ contradiction. □

Lemma 4.4.3. *If the quotient of a prime module has a transitive orientation, there are only two possible orientations, one being the inverse of the other.*

Proof. See Theorem 5.4 from Golumbic's work, 'Algorithmic Graph Theory and Perfect Graphs' [7]. The theorem proves that the quotient of a prime module will have either one or two implication classes. In the case of a single implication class, every edge forces every other, meaning ab indirectly forces ba. Since no transitive orientation can have both those edges, this case shows that no transitive orientation can exist for this graph. In the case of two implication classes, A and A', each class spans the graph, and represents the inverse of the other, $ab \in A$ and $ba \in A'$. □

Referring back to the C_5 in figure 4.1, it should be noted that it is a prime graph. A subraph induced by removing any vertex yields a P_4 which, as discussed in 4.3.4 and above, means that there exist prime graphs which are not transitively orientable.

However, at this point, all is not lost. Recall from 4.4.3 that if a transitive orientation exists for a prime graph, then there are only two. Each directed edge exists in exactly one of the two orientations. It is possible to simply select the orientation of any edge and allow it to force the rest.

Thus far we have demonstrated that the forcing relation cannot extend past module boundaries and shown how to transitively orient each of the three types of modules. Finally, before we can say that modular decomposition reduces the problem of transitive orientation down to transitively orienting the quotients of a graph, we must show how the modules interact with one another.

Lemma 4.4.4. *Between two children in the modular decomposition all edges of vertices in those children go in one direction. [8]*

4.5 Max Clique and Min coloring in comparability graphs.

It is possible to find a transitive orientation of a comparability graph in $O(n + m)$ time [3]. Given a transitive orientation, use depth first search to find a topological sort of the graph in linear time [9]. By induction from right to left, label each vertex with the length of the longest path that begins at it.

This labeling is a proper coloring of the graph. It is simple to see that no two vertices, x and y with the same label, will be have an edge between them. If such $xy \in E$, then in the transitive orientation either (x, y) or (y, x) will be a member. Therefore, either x or y will be the start of a longer path, giving it a different label.

In addition, the number of labels represents the size of the maximum clique. Recall that in the transitive orientation, whenever (x, y) and (y, z) exist, then (x, z) must also exist. Therefore, any path in the transitive orientation represents a clique. The longest path must also be the maximum clique, since if there were a larger one, it would have had a longer path.

4.6 Related graph classes

Based on the work of Ross M. McConnell and Jeremy P. Spinrad [3], comparability graphs and their transitive orientation can be found in linear time. This work has more implications for other classes of graphs than we have examined thus far. For starters, the complement of an interval graph is a comparability graph. Interval graphs have well studied impacts on scheduling theory, resource

allocation problems, and DNA mapping [10]. In section 4.3, we discussed that cographs have no prime modules which allows them to be stored in $O(n)$ space, as discussed in section 4.1.

In addition, bipartite, permutation, and threshold graphs are all subclasses of comparability graphs, using modular decomposition these graph classes can all be detected in linear time.

Chapter 5

The Algorithm

5.1 Overview

The algorithm implemented was first described by Elias Dahlhause, Jens Gustedt, and Ross M. McConnell, in their paper, 'Partially Complemented Representations of Digraphs' [4]. In the paper, they discuss and prove an $O(n + m\log(n))$ algorithm for finding the modular decomposition of an undirected graph. Their work builds on the fact that the modular decomposition of some graph G is exactly the modular decomposition of $\bar{G}$ with the labels of series and parallel nodes swapped.

The work of this thesis has been to study said algorithm and implement it in Java. It is believed that no concrete implementation of the algorithm existed prior to this. As a result, the work of transcribing the theory described in 'Partially Complemented Representations of Digraphs' into code, was a novel project.

To begin understanding the algorithm, let us consider the restricted problem of a graph whose modular decomposition contains no degenerate nodes. This means the modular decomposition is composed of only prime nodes internally, with leaves representing individual vertices. Let x be an arbitrary vertex of G and let $\mathcal{M}$ be the set of maximal modules of $G = (V, E)$ that do not contain x. Clearly the elements of $\mathcal{M}$ are the siblings of the ancestors of x in MD(G). In addition, $\mathcal{P}$ = $\{\{x\}\} \cup \mathcal{M}$ is a partition of V, so G'=$G/\mathcal{P}$ is well defined. The first step of the algorithm is to find $\mathcal{M}$, through a process called *vertex partitioning*.

It is important to note that MD(G') is MD(G), where the only internal nodes are ancestors of x. Therefore, every module in MD(G') must contain x. Every other internal node of MD(G), siblings of ancestors of x, have been turned into leaves. Therefore, finding the ancestors of $\{x\}$ in MD(G) reduces to finding the ancestors of $\{x\}$ in MD(G').

Let V' be the vertices of G'. To find the ancestors of x in G', we define a directed graph D on V', such that, for $y, z \neq x$, there is a directed edge from y to z, if y can distinguish between

z and x. That is, z forces y into any module containing z and x because, if left out, y would act as a spoiler for such a module. No module of G' that contains x in this directed graph can have a directed edge in D coming into it from an outside vertex. Conversely, let S be a set of vertices that contains x and has no incoming directed edges in D. It is easily seen that S.

Lemma 5.1.1. *A set S of vertices of G' is a module of G' iff it has no incoming directed edges in D.*

With this, we can reduce the problem of finding the ancestors of $\{x\}$ in MD(G') and, hence, in MD(G), to finding the sets of vertices in D that have no incoming directed edges. This reduces to finding the strongly-connected components of D. As described in Cormen [11], the strongly connected components of a directed graph have a natural topological sort. Every set of vertices of G', that has no incoming directed edges in D, is clearly a union of strongly connected components. Since the modules of G' that contain x are nested, this topological sort is unique, and these sets are prefixes of this topological sort.

This gives MD(G') and the portions of MD(G) that are not contained in children of ancestors of $\{x\}$. The missing parts of MD(G) are subtrees rooted at the leaves of MD(G'). The algorithm finds the subtrees by recursing on the subgraphs of G that those modules induce. The returned trees can then be attached to the corresponding leaves of MD(G'), to give MD(G).

When we relax the assumption that all internal nodes are prime, the algorithm needs only minor changes. When Y is a degenerate node of MD(G), the union of the children of Y, other than the one that contains x, is a maximal module of G that does not contain x. In MD(G'), the entire set of them becomes a single leaf. In the event that Y has three or more children, after all the recursive calls return, they will appear as a path rooted at Y rather then children of Y. It is required that we collapse the 'children' of the recursive tree into children of Y. Therefore, after all recursive calls return we iterate over the tree and when we see a serial node that is a child of another serial node we make them siblings. Likewise for parallel nodes.

5.2 Finding the maximal modules that do not contain x

Let us define a method *Split(X, $\mathcal{P}$)*, which takes in a partition class, X, a set of partition classes, $\mathcal{P}$, and returns a refinement of $\mathcal{P}$ and X. This refinement will split each $P_i \in \mathcal{P}$ into the maximal subsets that are indistinguishable to any vertex in X. Any module in $\mathcal{P}$ will remain entirely within some P_k since it can have no spoilers in X. We can also split X using $\mathcal{P}$ to the same result, dividing the partition class X into a set of partition classes $\mathcal{X}$, each indistinguishable to every vertex in $\mathcal{P}$.

If some new partition class is created from $\mathcal{P}$ or X, we can run split using that partition class as described in algorithm 2, repeatedly calling split using newly created partition classes until it is impossible for any partition class to split any other. At this point, we know that every module in $\mathcal{P}$ is still contained entirely within some P_k. In the event that two maximal modules are contained within some P_i, they would need to be indistinguishable to every vertex outside of P_i, otherwise, they would have been split. However, if they can not be split by any exterior vertex, then neither is a maximal module, and the partition class is a single maximal module. We can see that once no partition class can be split by any other, each is exactly a maximal module in the graph.

Therefore, in order to find the maximal modules in G, which do not contain x, we can start the process by running Split($\{x\}$, $V - \{x\}$). Once it is impossible for a Split operation to yield any further refinement we know that the partition returned, $\mathcal{M}$ will be comprised of maximal modules, each contained within their own partition class.

As stated, this is not sufficient to meet our time bound. Consider an undirected graph, where vertices are sorted into a line and for vertex has an edge to every vertex that is not adjacent to it in the line. If our split algorithm begins on one end of this line, with X containing the first $n - 1$ vertices, each partition will only split a single vertex from the largest partition. Another dilemma is the number of edges that need to be considered for every iteration of split. The running time of split is proportional to the degree of edges of X; since we must find all the edges in X that have external endpoints in $\mathcal{P}$, every edge with an endpoint in X must be examined. This provides us with a worst case time bound of $O(nm)$ for using split.

Lemma 5.2.1. *Split runs in time proportional to the sum of degrees of the vertices in X.*

Proof. Split requires an ordered list of vertices across the cut between X and $\mathcal{P}$. Labeling every vertex in X can be done in time proportional to the size of X. Using a radix sort we can order the vertices in X by their assigned labels in linear time [12]. Now we can go over the edge lists of every vertex in X collecting edges with an endpoint in $\mathcal{P}$ into a sorted list. If $\mathcal{P}$ is stored as a doubly linked list we can create new partitions and add them to the list in constant time.

In order to split X using edges from $\mathcal{P}$ we reuse the edges already collected. Assign labels to the endpoints in $\mathcal{P}$ and repeat the process in the other direction. We end up labeling $2|X|$ vertices and going over the edge lists of vertices contained in X. □

5.2.1 Strategy for an efficient implementation

Using the method Split, we can derive information on the maximal modules that do not include x, our set $\mathcal{M}$. However, as described, using split without conditions on its inputs takes too long to be practical for our purposes. Enter *Vertex Partitioning*, a quicker process of using partition classes to partition the graph, with a single vertex as the seed to start the process. The key to Vertex Partitioning is in how it selects the next partition class X that will be used to split the other partition classes. In our analysis of Split, one problem that arose was the possibility of choosing a large partition class. As Split runs in time proportional to the degree of the edges in X, by lemma 5.2.1, selecting a large partition class might mean we revisit the same vertices over and over, iterating over their edges repeatedly. By selecting a partition class which contains no more than half the vertices contained in the partition, along with the recursive nature of the algorithm, we can ensure no vertex is visited more then a logarithmic number of times.

This is key to meeting our time bound, as the algorithm uses split and still has to iterate over all the edges in a partition class. Guaranteeing that no edge is visited more then O($\log(n)$) times is much better then the alternative $O(n)$. A partition class is *ripe* if it contains at most half the vertices in the partition. Any partition class marked as ripe can be used to split the rest of the partition.

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

Input: Graph G and a partition $\mathcal{P}$ of G

Result: Partition $\mathcal{P}$ containing partition classes P_1 through P_i

if $|P| = 1$ **then**

Return P

else

Let X be a member of $\mathcal{P}$ that is not larger than all others combined.

Let M be the edges of G in $X \times (V(G) - X)$

$\mathcal{P}'$ = Split(X, $\mathcal{P}$)

$G = (G - M)$

Let $\mathcal{Q}$ be the classes of $\mathcal{P}'$ contained in X

Let $\mathcal{Q}'$ be the classes of $\mathcal{P}'$ contained in $\mathcal{P} - X$

Return: Partition($G|X$, $\mathcal{Q}$) $\cup Partition$(G-X,Q')

The proof that a ripe set exists is straightforward. If $|\mathcal{P}| > 1$ then there must always be some $P_k : |P_k| \leq |\mathcal{P}|/2$. We have already covered that the partition classes in $\mathcal{P}$ do not overlap. Since the partition classes do not overlap, if $\exists P_j, P_k \in \mathcal{P} : |P_j| > |\mathcal{P}|/2$ and $|P_k| > |\mathcal{P}|/2$ then $|P_j| + |P_k| > |\mathcal{P}|$, contradiction.

After this, the Split function defines $\mathcal{Q}$, and $\mathcal{Q}'$ as distinct partitions to be operated on separately. This recursion into smaller problems is the other part of guaranteeing that this step takes, at most, O($m \log(n)$) time. Picking a partition class, at most half the size of the problem, would not be useful if we were not also reducing the size of the problem at the same time.

In addition, removing edges from X to $\mathcal{P}$ prevents future iterations from having to worry about edges that can no longer split any partition class. Every edge from $\mathcal{Q}$ to $\mathcal{Q}'$ has already been used to split partition classes. Because of that, no further refinement of the partition classes of $\mathcal{Q}'$ can ever be split by edges from $\mathcal{Q}$. Once a vertex $y \in P_i$ has been used to refine a partition class X, it has separated all of its neighbors and non neighbors in X, into neighbors X' and non neighbors X''. Now y cannot distinguish any of the vertices in X' since they are all neighbors. Regardless

of how many sets X' is subdivided into via further refinements by other vertices, the set of those partition classes is indistinguishable to y.

Once the Vertex Partition algorithm has completed, what is left is a set of partition classes that cannot be separated by each other, with the members of any particular partition class being indistinguishable to x. In short, the algorithm returns the set $\mathcal{M}$ described in 5.1. What is left is to use this information to find all the modules that do contain x.

5.2.2 Finding $\mathcal{M}$ in practice

Implementing partition proved to be one of the more challenging aspects of this work. As described in McConnell and Spinrad [13], there are several important things to track. Each vertex requires an integer id to support ordering, provided by its index in the original undirected graph, as described above. In addition, for each vertex x we will need to be able to find which partition contains x in constant time. This lookup was implemented as a list of size $|V|$, where index i contains a pointer to the partition, which contains v_i.

Each partition class P_i is modeled as a list of vertex pointers, and tracks its current size and the size of P_i the last time its members were used to partition a partition class. It also contains a pointer for another partition class labeled *prime*, which is used during the partition step. It is possible that not all of the non neighbors in P_i being moved are sequential; instead of moving them to a new partition class in the greater partition piecemeal, they are staged into this prime partition class. Once all the edges with the same endpoint in the partitioning class have been visited, every partition class with a prime list is split in order, putting the prime list ahead of it in the partition class. In addition, each partition class stores a listNode object, that contains meta information, such as whether the list is ripe for partitioning and which partition contains it.

Due to the need to add new partition classes in front of the partition class that generated them, each partition is modeled as a doubly linked list of lists, which contains the individual partition classes. Concurrent with the list of partitions classes is a list of ripePartitions, which has pointers to the partition classes which are ripe.

To start, the program simply peels some vertex x off the list of all vertices and uses it to begin partitioning. Rather then explicitly recurse into sub problems, as described in algorithm 2, the code removes edges from a mutable list of edges generated when building the graph. When removing an edge, each symmetric edge is also removed to guarantee that no time is wasted trying to partition sets that are indistinguishable to a vertex.

As partition classes become ripe, they are added to the list of ripe partitions classes. Due to not recursing into sub problems, the definition of ripe here is slightly modified. Rather than any set, which is less than half the size of the total problem, partitions classes are listed as ripe if they are, at most, half the size they were when they were last used to partition. In this way, we can guarantee that that adjacency list of each vertex is accessed, at most, a logarithmic number of times. The program pulls off the front of the ripe list until no more ripe partition classes are present. This condition concludes the first part of the algorithm. At this point, $\mathcal{M}$ is exactly equal to the returned partitionSet object, and every partition class contained within is a maximal module.

5.3 Finding Modules containing x in G'

This phase of the algorithm relies on a property of *component graphs*. The component graph of some graph G is defined as $G^{SCC} = (V^{SCC}, E^{SCC})$. Given G has strongly connected components $C_1, C_2, ..., C_k$, the vertex set V^{SCC} is $v_1, v_2, ..., v_k$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{SCC}$ iff G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$ [11].

We now have a set $\mathcal{M}$ of modules, which do not contain the original partitioning vertex x and x. Let us consider a new graph, $G' = G/\{\mathcal{M} \cup \{x\}\}$. As described above, MD(G') is exactly MD(G), where the only internal nodes are modules which contain x. We derive a directed graph D from G'. This graph, D, will have the property that every set of vertices with no incoming edges will be a module in MD(G'). Thus, in order to find the modules of MD(G'), we need only find the strongly connected components of D and a topological sort of the component graph.

The strongly connected components, SCC, can be found in linear time. As described in Cormen [11], a two pass algorithm exists, which runs DFS on D and D^T. DFS can be run in linear time, as it visits each vertex and every edge once for a time bound of O($V + E$). The SCCs described in lemma 5.1.1 will be prefixes of the topological sort of D. If D can be generated in less than the O($m \log(n)$) time it took to find $\mathcal{M}$, we are close to making the time bound of O($n + m \log(n)$).

In order to generate D, we complement every neighbor of x in G'. This does two things: the first is to assign a directed edge from every $y : (y, x) \in E'$ to each of its non neighbors. Fortunately, it is not necessary to explicitly complement each neighbor of x; simply marking them as complemented is sufficient. Otherwise, for every neighbor of x, we would need to make V operations, a cost of potentially V^2 operations total.

The second benefit of complementing neighbors of x is that now any vertex that had edges into a module containing x now has no such edges. Recall that any edge into such a module would require an edge to every element in the module, after complementing none of these edges exist. Regardless of the relationship between modules in MD(G'), no module with x in it can have an incoming edges in D. Since edges can still flow out of these sets, we have a series of nested strongly connected components which represent the ancestors of x in MD(G').

Implementing the Conversion

Implementing the D in Java was fairly straight forward. Representative vertices from the quotient are copied into a set of new directed vertices. This move allows us to remove edges internal to the module that will not have any impact in the subsequent steps. Here, we also only iterate over the edge list of vertices that were ripe at one point. If a partition class was never added to the *ripeList*, the list of partition classes that were ripe at some point in the algorithm, we never had a chance to look at its edges, and so do not have a credit to do so now.

The significant concern in terms of the time bound was how to complement the neighbors of x. As discussed, having to explicitly enumerate all the new edges and remove the old ones takes $|V|$ time for every neighbor of x. Due to this constraint, every vertex has a flag indicating whether it has

been complemented. In this way, we can complement all the neighbors of x in time proportional to the degree of x.

5.3.1 Finding the Strongly Connected Components

Given the digraph D, there are a significant number of options for finding strongly connected components. One concern is that any algorithm will need to work on a partially complemented D. As discussed, the two pass SCC algorithm, described by Cormen [11], was selected due to its simplicity.

The two pass SCC algorithm, henceforth referred to as 2-SCC, runs a depth first search on the graph, then a depth first search on the transpose D^T. The second pass is run using the finishing times of vertices from the first pass as the order it visits vertices in the transpose. 2-SCC does not take into consideration the complementation of vertices, because it is not relevant to the algorithm. Here, that concern is shifted to the DFS step. So long as there exists an algorithm that can run DFS on a partially complemented graph, 2-SCC can use the results to find the strongly connected components.

However, it remains to be proven that calculating D^T can be done efficiently. It is possible to create the transpose of an uncomplemented graph in linear time by visiting each vertex once and reversing all outgoing edges. However, D has vertices which are labeled as complemented, but never had these edges explicitly created. Creating these edges explicitly in the transpose presents the same problem as before, increasing the time bound to O(V^2). Fortunately, the solution to running DFS on a partially complemented digraph works, with minor modification, on a partially complemented transpose. So we create D^T by transposing only the explicit edges in D, leaving the complemented vertices marked as complemented. This leaves us with linear time solutions for generating D and D^T, and running DFS on them both. Finding $\mathcal{M}$ is the largest factor in determining running time at O($m \log(n)$).

DFS on Partially Complemented Graphs

The work of Dahlhaus, Gustedt, and McConnel [2] describes a *Set Complement Stack* data structure which can be used to run a variety of algorithms on partially complemented graphs. In their work on "Partially Complemented Representations of Digraphs" [4], they describe the utility of set complement stacks in DFS, BFS, topological sorts, and finding the longest path in a DAG.

The Set Complement Stack data structure, hence referred to as SCS, contains a set of *Complement Stacks* which are described in the same papers. SCS provides an amortized linear time solution for basic stack operations on partially complemented graphs. Pop and push, as well as two operations called cpush and eliminate, are used to this end. 'Cpush' stands for complement push, and is the key we used to run DFS on a partially complemented digraph. Due to SCS being a set of complement stacks, the eliminate operation is used to remove an element from whichever stack it is in. For proofs of the credit invariants and correctness of SCS please see [2] or [4].

The SCS contains four internal lists:

- **L**: which contains elements not yet pushed onto another list.
- **A**: which contains elements whose most recent operation was a push.
- **T**: which contains the elements most recently moved during a cpush.
- **B**: which contains elements whose last operation was a cpush, but not the latest to be moved during a cpush.

In addition, SCS maintains an internal timer, and stores the time an element was moved onto stacks **A** or **B**, as well as a time for when elements were moved en mass to **T**. Using these tools, we can initialize all the elements of the directed graph into **L**. When a vertex is popped off of the stack, SCS will perform either a push or a cpush depending on whether the vertex is complemented moving the appropriate neighbors or non neighbors. On a push, each element will store the time they were pushed; this will match up with the discovery time of the element that was most recently popped and is responsible for the push.

On a cpush by vertex v_i, each neighbor of v_i is staged to a new list, **A'** for the elements of **A**, **B'** for **B**, and so on. Once all the neighbors have been moved to prime lists, all of original lists are composed of only non neighbors of v_i. Now they can all be combined and added to **T** without having to interact with each non neighbor individually, and with a variable tracking the last time a cpush operation occured we can keep track of the the push time for every element in **T**. In this way we can move all non neighbors in time proportional to the degree of v_i. The pointers for the original lists will move to the lists maintained with the prime labels, with the exception of **T'**. These neighbors of v_i are moved to the top of **B** and will have the old time label of **T** applied to each of them.

With SCS, running DFS on D is now straight forward; simply start DFS by popping a vertex, and send either a push or cpush as appropriate. When popping vertex v_i from the stack, it will identify when it was last pushed, indicating which vertex was responsible for pushing it on the stack.

Implementing 2-SCC using SCS

Since *cpush* and *eliminate* are the only differences between SCS and a regular stack, we abstracted calls to the SCS behind a more normal interface for stacks that include only *pop()*, *pop(int id)*, and *push(GraphNode node)*. The *pop(int id)* is used to ensure that the next node on the discovered list is popped. This is important on the second pass of DFS which uses D^T, and the finishing times from the first pass. Since the SCS structure can be comprised of multiple stacks internally, it may not be obvious which stack currently contains a particular vertex. In order to initialize the list, the vertices are passed via the constructor, leaving the push method to push the neighbors of a vertex rather than pushing the passed vertex.

Finally, while using a single complement stack works for the first pass of DFS, in order to run on the partially complemented transpose, we must use an actual set of complement stacks. Because we did not explicitly create the partially complemented graph, the transpose poses a slight dilemma. In D, each v_i neighbor of x is partially complemented, turning all non neighbors into neighbors and vice versa. Yet, they are set through a flag rather than adding an edge. In the

transpose, the previous non neighbors of v_i now require an edge to v_i, even though as far as the vertex is concerned it has no relationship with v_i.

We move the complemented vertices into their own complement stack. Leaving us with two stacks in the SCS, CS_1 and CS_2, for uncomplemented and complemented vertices of D^T, respectively. When performing a pop of a vertex from CS_1, we push its neighbors in CS_1 normally, and cpush its neighbors in CS_2. If it is unclear why this works, consider that every vertex in CS_2 has added a conceptual edge from each of its non neighbors in G to itself. Therefore, by performing a cpush with each element from CS_1 we can use the CS structure to move the correct vertices to the top of the stack for discovery. The vertices on the CS_2 stack operate the same way. The vertices that had explicit edges to them added them when the transpose was created, so these vertices use push on the CS_1 stack. Also, just as before, they use cpush on the CS_2 stack for precisely the same reasons.

After both runs of DFS, the strongly connected components of the quotient will have been revealed and we can build the the modular decomposition tree of G'.

5.3.2 Building MD(G')

As discussed above, the strongly connected components returned by 2-SCC will represent the internal nodes in MD(G'). Each SCC will represent a maximal module that does contain x. The first component returned will represent x's parent. The subsequent SCCs in the hierarchy will be the remaining ancestors of x in ascending order.

The issues here are degenerate modules. There are only two cases for these modules when represented in the quotient. The subgraph induced by the module is disconnected, meaning every vertex will have been combined into a single partition class in $\mathcal{M}$. As discussed above, these will not be correctly marked as siblings until after the recursive step returns. The same is true for the subgraph induced by module whose complement is disconnected.

5.4 Recurse

At this point in the algorithm, every maximal module not containing x, and every module which does contain it, has been found and put into their correct place in the modular decomposition. In order to fill out the rest of the tree, we recurse into the partition class in $\mathcal{M}$, build up the trees that represent those modules, and hang them off of the leaves of MD(G'). As described in section 5.2, in the case of degenerate nodes, all the children of a degenerate node will appear as a single maximal module when building MD(G').

When hanging the results of these recursions onto the tree, a module marked as serial may have a path of serial nodes hanging off it. So when building the tree, we recurse up, and collapse these like labeled degenerate nodes to become siblings. Moving up the tree as necessary, until no degenerate node is the child of a similarly labeled node. Doing the same for parallel nodes. This final traversal need only visit each node in the tree once. Since no module can have less than two children, we can bound the number of internal nodes in the modular decomposition tree of G to, at most, $2V + 1$. This means it can be safely traversed in linear time.

Bibliography

[1] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(1):25–66, Mar 1967.

[2] Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. Efficient and practical modular decomposition. In *SODA*, 1997.

[3] Ross M. McConnell and Jeremy P. Spinrad. Linear-time transitive orientation. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 19–25, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[4] Elias Dahlhaus, Jens Gustedt, and Ross Mcconnell. Partially complemented representations of digraphs. *Discrete Mathematics Theoretical Computer Science*, 5, 12 2002.

[5] D.G. Corneil, H. Lerchs, and L.Stewart Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163 – 174, 1981.

[6] David P. Sumner. Graphs indecomposable with respect to the x-join. *Discrete Mathematics*, 6(3):281 – 298, 1973.

[7] Martin Charles Golumbic. Chapter 5 - comparability graphs. In Martin Charles Golumbic, editor, *Algorithmic Graph Theory and Perfect Graphs*, pages 105 – 148. Academic Press, 1980.

[8] Rolf Möhring. *Algorithmic Aspects of Comparability Graphs and Interval Graphs*, pages 41–101. 01 1985.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[10] Peisen Zhang, Eric A. Schon, Stuart G. Fischer, Eftihia Cayanis, Janie Weiss, Susan Kistler, and Philip E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA. *Bioinformatics*, 10(3):309–317, 06 1994.

[11] T.H. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.

[12] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1983.

[13] Ross M. Mcconnell and Jeremy P. Spinrad. Ordered vertex partitioning. In *Discrete Math and Theoretical and Computer Science*, page 2000, 2000.