Complement-equivalence classes on graphs

Ross McConnell[1]

Department of Computer Science, Willamette University, Salem, OR 97301 USA

Abstract. A *complementation operation* on a vertex of a digraph changes all outgoing edges into nonedges, and outgoing nonedges into edges. A similar operation is defined for incoming edges. This defines an equivalence relation where two graphs are equivalent if one can be obtained from the other by a sequence of such operations. We show that given an adjacency-list representation of a digraph G, many fundamental graph algorithms can be carried out on any member of G's equivalence class in $O(size(G))$ time. We use this fact to obtain a simple $O(n + m \log n)$ algorithm for modular decomposition of undirected graphs.

1 Introduction

Ehrenfeucht and Rozenberg have pointed out that graphs are not the most appropriate abstraction on which to examine some problems that had previously been thought of as graph problems, such as decomposition of a graph into modules or total orders [8, 9, 6]. In particular, the modular decomposition of a graph G is the same as the modular decomposition of $\overline{G}$. Thus, from the point of view of modular decomposition, the distinction between edges and non-edges is an artificial one, and G and $\overline{G}$ are best viewed as a single structure. This structure is simply a partition of $E_2(V) = V \times V - \{(v,v)|v \in V\}$ into two sets, where the distinction of these classes as *edges* and *nonedges* is best dropped.

The two-structure point of view played a role in the development of a linear-time algorithms for decomposing two-dimensional partial orders and permutation graphs into two total orders, and recognizing whether a graph is the complement of an interval graph [12]. The key to these bounds was the development of data structures that allow one to ignore the differences between a graph and its complement. This allows computation of many properties of the complement in linear time, instead of the $\Omega(n^2)$ *best-case* time required to construct an adjacency-list representation of the complement explicitly. The problems were solved in linear time by combining properties of G with properties of $\overline{G}$ that were obtained in this way.

In [7], the same authors examine a transformation that operations on the edges out of a vertex or else on the edges into a vertex. They define an equivalence relation on two-structures, where two two-structures are in the same equivalence class iff one of them can be obtained from another by a sequence of these operations. These structures are called *dynamical two-structures.*

A special case of this is the operation of *complementing* a vertex, which is the operation of replacing all outgoing edges with non-edges, and all outgoing non-edges with edges. The inwardly directed edges can be complemented with an analogous operation. Two graphs are *complement-equivalent* if one can be obtained from the other by a sequence of such operations.

This induces a set of equivalence classes on graphs. Let $size(G)$ denote the number of vertices and edges of G. The main result of the present paper is the following:

Claim 1 *Many fundamental algorithms can be run on any member F of G's complement-equivalence class in time that is linear in the size of G.*

What makes this remarkable is that the size of members of a class can differ greatly. An extreme case occurs when $\{A, B\}$ is a partition of V such that $|A| = |B|$, each member of A has a directed edge to all other vertices of the digraph, and no member of B has a directed edge to any other vertex. The size of this graph is $\Omega(n^2)$. However, outwardly complementing the vertices of A yields the empty graph on n vertices, and the size of this graph is $O(n)$. $O(n)$ and $\Omega(n^2)$ graphs can thus appear in the same equivalence class, so given a way of obtaining a small member G in each of a sequence of classes, one might get a sublinear-time algorithms for other members of the classes.

2 Definitions

We let n denote the number of nodes and m the number of edges, and let $size(G)$ denote $n + m$. Let $V(G)$ and $E(G)$ denote, respectively, the number of vertices and edges of G. If G is a digraph, then its *transpose*, G^T, is the digraph that is obtained by reversing the directions of all edges in G. The complement $\overline{G}$ is obtained by changing each edge into a non-edge and each non-edge into an edge. A *non-edge* of G is an edge of $\overline{G}$.

An undirected graph is *complete* if there is an edge between every pair of distinct vertices. It is *empty* if it has no edges. A digraph is a *linear order* if it has no cycles, and between each pair $\{u, v\}$ if distinct nodes, either (u, v) or (v, u) is a directed edge.

Undirected graphs will be treated as a special case of digraphs, where each undirected edge (u, v) is really two directed edges (u, v) and (v, u). Thus, algorithms that we describe for digraphs will be general also to undirected graphs.

If G is a digraph and $X \subseteq V$, then $G|X$ denotes the *subgraph of G induced by X*, namely, the graph whose vertices are X and whose edges are those edges of G that go from one member of X to another. That is, $G|X$ is the graph $(X, (X \times X) \cap E)$. If $\mathcal{F}$ is a partition of $V(G)$ and X is a union of partition classes in $\mathcal{F}$, then $\mathcal{F}|X$ is the set of partition classes that are contained in X. The *quotient* $G/\mathcal{F}$ denotes the graph whose vertices are the members of $\mathcal{F}$, and where (X, Y) is an edge of $G/\mathcal{F}$ iff there is some edge of G from a member of X to a member of Y.

If G is undirected, the set of vertices that are reachable on a single directed edge from a vertex v is denoted $N_G^+(v)$. $\overline{N}_G^+(v)$ denotes the vertices that are not reachable on a single edge from v, that is, $V - (N_G^+(v) \cup \{v\})$. The set of vertices that can reach v on a single directed edge is denoted $N_G^-(v)$, and the ones that cannot are denoted $\overline{N_G^-}(v)$. When G is undirected, we let $N_G(v) = N_G^+(v) = N_G^-(v)$.

3 Modular decomposition

A *module* in an undirected graph $G = (V, E)$ is a set M of vertices such that for each vertex $v \in V - M$, M is either contained in v's neighborhood or disjoint from v's neighborhood. Trivial examples of modules are V, the singleton sets $\{\{v\}|v \in V\}$, and the empty set. A union of connected components of G or of $\overline{G}$ is also an example of a module. However, G may have nontrivial modules even when G and $\overline{G}$ are nonempty.

It is easy to see that if M_1 and M_2 are disjoint modules, then either every member of $M_1 \times M_2$ is an edge or none is. Thus, two disjoint modules are either *adjacent* or *nonadjacent*. It follows that if there is a partition $\mathcal{P}$ of V into disjoint modules, the quotient $G/\mathcal{P}$ uniquely specifies all edges that connect members of different partition classes. We will call such a quotient *modular*. The subgraph induced in G by one of these modules is called a *factor*. Since G can be reconstructed uniquely from the factors and the quotient, this gives a decomposition of G.

If X is a module of G, then $S \subseteq X$ is a set of vertices in the factor $G|X$, and *inverse image* of a S in G is S. If $\mathcal{F}$ is a partition of G where each partition class is a module, then $\mathcal{F}' \subseteq \mathcal{F}$ is a set of vertices of $G/\mathcal{F}$, and the inverse image of $\mathcal{F}'$ in G is $\bigcup \mathcal{F}'$. The following says that modules are preserved under inverse images:

Theorem 2. *If X is a module of G, then $S \subseteq X$ is a module of $G|X$ iff its inverse image in G is a module of G. If $\mathcal{F}$ is a partition of $V(G)$ into a set of modules, then $\mathcal{F}'$ is a module of $G/\mathcal{F}$ iff its inverse image is a module of G.*

The factors and quotient may themselves contain modules, so this decomposition may be applied recursively in each of them. By Theorem 2, this discovers additional modules of G. Among all ways of performing this recursive decomposition, there is a unique canonical one that represents all others, called the *modular decomposition*. This decomposition has been rediscovered in various contexts, and is also known as the *substitution decomposition* [13], or the *prime tree family* [8, 9]. From this tree-like hierarchy of quotients, it is possible to reconstruct G and enumerate all of its modules. We give the following definition of the structure in the special case of undirected graphs:

MD(G)
    If G has only one vertex return v
    else
        create a tree node r

```
if G has more than one connected component then
    let F be the connected components
    let r be labeled a parallel degenerate node
else if Ḡ has more than one connected component then
    let F be the connected components of Ḡ
    let r be labeled a series degenerate node
else
    Let F be the maximal modules of G
    Let r be labeled a prime node
for each X ∈ F
    Let r_x = MD(G|x)
    add r_x as a child of r
return r
```

We may think of r as synonymous with $V(G)$; applying this idea recursively, we see that the nodes of the tree can be thought of as a tree-like hierarchy of subsets of $V(G)$, where each node is just the subset of $V(G)$ that make up its leaf descendants. It is not hard to see that any union of the children of a degenerate node in the returned tree is a module. The following is somewhat less immediate:

Theorem 3. *[9, 13] A subset of $V(G)$ is a module of G iff it is either a node of the decomposition tree, or a union of children of a degenerate node.*

(In the case of digraphs, a third type of *linear* tree is node is also possible, where there is a linear ordering on children of the node, and a set may be a module if it is a union of consecutive children in that ordering [9, 13].)

The *two-edges* of a set V, denoted $E_2(V)$, is the set $V \times V - \{(v,v)|v \in V\}$. A *two-structure* on V is a coloring, or partition, of the the two-edges of V. A digraph $G = (V,E)$ is a coloring of $E_2(V)$ with colors 0 and 1, but unlike a digraph, a two-structure gives no special status to either of these classes.

4 Outward, inward, and two-way equivalence classes on digraphs

If G is a digraph, we *outwardly complement* v if we replace $N_G^+(v)$ with $V - N_G^+(v) - \{v\}$, but $N_G^+(w)$ unchanged for each $w \neq v$. That is, we complement the outgoing edges without changing any of the incoming edges. We *inwardly complement* v if we change $V - N_G^-(v) - \{v\}$ and leave $N_G^-(w)$ the same for all other vertices.

Let us say that two digraphs are *outward-equivalent* if one can be obtained from the other by a sequence of outward vertex complementation operations. This is clearly an equivalence relation, so this partitions the set of undirected graphs into equivalence classes, which we may call *outward equivalence classes*. Given an undirected graph $G = (V,E)$, where $V = \{v_1, v_2, ..., v_n\}$, we may represent any member of F of G's class in $O(size(G))$ space by the pair $(G, \overline{b})$, where $\overline{b}$ is a boolean vector that has a 1 in position i if v_i must be outward complemented to get F. Similarly, two digraphs are *inward-equivalent* if one is obtained from the other by a sequence of inward complementation operations. This defines the *inward classes* on digraphs.

Two digraphs are *two-way* equivalent if one is obtained from the other by a mixed sequence of outward and inward complementations, and this defines the *two-way classes* on digraphs. We may represent any member of G's two-way class in $O(size(G))$ space with the triple $(G, \overline{b}_1, \overline{b}_2)$, where the vectors $\overline{b}_1$ and $\overline{b}_2$ have a 1 in position i if vertex v_i is inward and outward complemented, respectively. For any pair (v_i, v_j), let the *net complement* $NC(v_i, v_j)$ be obtained by assigning one point if v_i is inwardly complemented, 1 point if it is outwardly complemented, and 1 point if (v_i, v_j) is not an edge of G. Then (v_i, v_j) is an edge in the represented graph iff $NC(v_i, v_j) \bmod 2 = 0$.

We call these representations *partially complemented representations*, and call G the *base graph* of the representation.

Theorem 4. *For any digraph G, $\overline{G}$ is in G's outward class.*

Proof: Immediate, since $\overline{G}$ is obtained from G by either complementing the list of outgoing edges of each vertex, or by complementing the list of incoming edges of each vertex. Q.E.D.

Theorem 5. *If digraph F is in the equivalence class of graph G, then F^T is in the two-way equivalence class of G^T.*

Proof: Let $(G, \overline{b}_1, \overline{b}_2)$ be a partially-complemented representation of F. We claim that $(G^T, \overline{b}_2, \overline{b}_1)$ represents F^T. Let (i, j) be an ordered pair of elements of V. Then (i, j) is an edge of F iff $NC(i, j)$ is odd. Let $NC'(j, i)$ denote the net complementation of (j, i) in $(G^T, \overline{b}_2, \overline{b}_1)$. Clearly, $NC'(j, i)$ is the same as $NC(i, j)$. Q.E.D.

5 Algorithms on members of an outward equivalence class

To introduce some basic data structures tricks, I will now illustrate how to get an $O(size(G))$ algorithms for a member F of G's *outward* equivalence class. I also give an example of an application, which is obtaining an $O(n + m \log n)$ bound on the modular decomposition algorithm of [5]. The ultimate goal, which is explored in the next section, is to obtain these bounds when F is any member of G's full two-way equivalence class, since the two-way classes are examples of dynamic labeled two-structures.

5.1 Breadth-first search

As a starting exercise, let us examine the running time of breadth-first search on a member F of G's outward class. This solves the single-source shortest paths problem when the length of a path is the number of edges on it.

As in the standard algorithm, we divide V into *undiscovered* nodes, *queued* nodes, and *processed nodes*. Initially all nodes but the start node are undiscovered, and the start node is queued. To process a node, we remove one from the front of the queue, and insert any undiscovered neighbors at the back of the queue. Keep the undiscovered nodes in a linked list. When you process an uncomplemented node, proceed as in the standard case. When you process a complemented node v, mark the members of $N_G^+(v) = N_F^+(v)$. Then traverse the list of undiscovered nodes, splicing out any unmarked nodes and inserting them on the queue. Then remove the marks from $N_G^+(v)$. Marking and unmarking nodes can be charged to the edges of G that are responsible for applying the marks. Touching an unmarked node can happen at most n times, since the node is removed from the list of undiscovered nodes whenever this happens. This gives the $O(size(G))$ bound.

5.2 Depth-first search

Let us examine an algorithm discovered by Elias Dahlhaus, Jens Gustedt and me for depth-first search on F, when $F = (G, \overline{b})$.

One way to obtain a depth-first forest on G is to use a stack of vertices. Insert an initial vertex on the stack. While the stack is not empty, pop a vertex. If it is not already marked visited, mark it visited and push its neighbors on the stack. From the sequence of visits, we can create a *depth-first tree*, where the parent of each vertex is the one that had most recently pushed it at the time it was marked visited. If not all vertices were marked, the operation can be applied recursively on the remainder of the graph, producing a *depth-first forest*, which is a forest of ordered, rooted trees.

Let $d[u]$ for each vertex u be a numbering that gives the order in which the vertices were marked visited. This order is sometimes called *discovery times*, and is a preorder numbering of the forest that also records the order in which trees are generated. With a second $O(n)$ traversal of the forest, it is trivial to get a similar postorder numbering, called the *finishing times*. The finishing time of vertex u is denoted $f[u]$.

Note that when a vertex is pushed to the stack, lower instances become irrelevant, since the vertex will be marked visited by the time the lower instances are reached. Let us define a *non-duplicating push* to a stack to be like a standard push operation, except that whenever a value is pushed, any lower occurrences of the value in the stack are deleted. This guarantees that at most one occurrence of any value sits on the stack at any time, and, when used in the depth-first algorithm, does not affect the forest produced.

If we are given G, and we want a depth-first forest for F, then when we visit a vertex v, we must push $N_G^+(v)$ if v is not outwardly complmented in F, and $\overline{N}_G^+(v)$ if v is outwardly complemented. All operations except the cost of pushing $\overline{N}_G^+(v)$ may obviously be charged to vertices and edges of G as in the standard algorithm. We show next that pushing $\overline{N}_G^+(v)$ can also be accomplished in $O(N_G^+(v))$ time, amortized, giving the $O(size(G))$ bound for depth-first a forest on F.

Definition 6. A **complement stack** is a data structure that supports the following operations:

- $Initstack(V)$: 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.
- $Cpush(X, S)$ 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 7. *Complement stacks can be implemented in such a way that:*

- *$Initstack$ requires $O(V)$ time.*
- *$Push(X, S)$ requires $O(|X|)$ time.*
- *$Cpush(X, S)$ 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 are doubly-linked 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 Cpush. B holds those elements that were last pushed by a call to Cpush, but not by the most recent call to Cpush.

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 Cpush.

We maintain a credit invariant: *Each member of L, A, B carries a credit.*

- $Initstack(V)$ 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 $O(|X|)$.
- $Cpush(X, S)$ 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 Cpush pays for its operations either with its budget of $|X|$ new credits or with other credits it frees up from the structure. Q.E.D.

5.3 Topological sort

A *dag*, or *directed acyclic graph*, is a digraph that has no cycles. A *topological sort* of a digraph is an ordering of its vertices so that every edge goes from an earlier to a later vertex in that ordering. A digraph has a topological sort iff it is a dag, and, ordering the vertices of a dag in descending order of finishing time in any depth-first search gives a topological sort [3]. Thus, we may find a topological sort of any dag in G's outward class in $O(size(G))$ time.

5.4 Longest paths in a dag

We will let the *length* of a path denote the number of edges in it. The length of the longest path in a digraph is finite iff it is a dag; otherwise a cycle exists, and by repeatedly traveling the cycle, one may demonstrate paths of arbitrarily large lengths.

The following well-known approach finds a longest path in a dag F, given its adjacency-list representation in standard form. Label each vertex with the length of the longest path originating at that vertex. Given a topological sort, we may observe that as long as vertices are labeled in reverse topological order, then when it is time to label any vertex v, the members of $N_G^+(v)$ are already labeled. We may then apply the rule that the length of a maximum-length path beginning at v is one plus the maximum of the labels of $N_G^+(v)$. Given an adjacency-list representation of G, we may charge the cost of labeling v to edges out of v, and we get an $O(n+m)$-time algorithm.

We now show that if F is represented as $F = (G, \overline{b})$, we may find a longest path in F in $O(size(G))$ time. To find longest paths in F, there is not time to apply the foregoing algorithm directly, since we do not always have time to examine the labels of vertices in $N_F^+(v)$ when it is time to assign a longest-path label to v. We may nevertheless solve the problem by keeping all labeled vertices bucket-sorted according to their label. A key observation is that if a vertex w has label $k > 0$, then there is a member of $N_F^+(w)$ with label $k-1$. By induction, the buckets from k down to 0 are all nonempty. When it is time to label vertex v, then if v is not outwardly complemented, apply the standard approach of visiting the labels of $N_F^+(v)$. If v is outwardly complemented, then mark its neighbors in G; these are the vertices that are not neighbors in F. Then, starting at the highest nonempty bucket, descend through the buckets sequentially until an unmarked vertex is found. This is a neighbor in F with highest label, so assign v's label to be one plus this label, and add v to the appropriate bucket. In either case, the operation can be charged to v and edges out of v in G. Thus it takes $O(size(G))$.

5.5 Connectivity

Let us define an equivalence relation R on vertices of a digraph G where for vertices u, v, we say uRv iff there is a directed path from u to v and a directed path from v to u. In each equivalence class there is always a path from any vertex to any other, but this is never the case for two vertices in different classes. The equivalence classes are known as *strongly-connected components.* If $G = (V, E)$ is a digraph and $\mathcal{P}$ are the strongly connected components of G, then the *component graph* $G/\mathcal{P}$ is a dag. The component graph tells which components have a path to which.

An $O(size(G))$ algorithm for finding strongly-connected components in a member $F = (G, \overline{b})$ of G's outward equivalence class is obtained by modifying a standard algorithm [1]. Define the following function on vertices of G:

$$Low[v] = min(\{d[v]\} \cup \{Low[w] | w \text{ } is \text{ } a \text{ } child \text{ } of \text{ } v\} \cup \{d[u] : u \in N_G^+(v)\}. \quad (1)$$

This function differs somewhat from a similar function defined in [1], but inspection of their algorithm shows that it computes $Low[v]$ for each vertex v in postorder on the depth-first forest. This guarantees that the children's labels are in place when it is time to label v. As it goes, it occasionally removes from the forest the subtree rooted at the vertex being labeled, using the following rule: remove v's current subtree if $Low[v] \geq d[v]$. Moreover, if $\mathcal{F}$ is the set of strongly-connected components, the vertices of this subtree are a sink in the *component graph* $G/\mathcal{F}$. It then recurses on what remains of the graph, continuing to label in postorder applying the inductive rule for $Low[v]$ on the reduced vertex set. The vertices of each removed tree give the vertices of a strongly-connected component.

If $\mathcal{F}$ is the set of strongly-connected components, then $G/\mathcal{F}$ is a dag [3]. Below, we will find it useful to have a topological sort of this graph available. Because the removed component is always a sink in what remains of the component graph $G/\mathcal{F}$, it follows that the order in which the components are removed gives a topological sort of $G/\mathcal{F}$.

If G is a digraph, and we want the strongly-connected components of a digraph F in G's outward class, then there is time to find the minimum of $d[v]$ and $Low[w]$ for each child of v, since we may charge these lookups to v and to tree edges out of v. Thus, we will incur $O(n)$ time looking up these values and finding their minimum at each vertex v. We must also find $min(\{d[u] : u \in N_G^+(v)\})$. If v is not outwardly complemented, we may do so and charge the cost to edges of G that are incident to v. If v is outwardly complemented, there is not time to visit $N_G^+(v)$. Instead, we assume that we have pre-assembled a list D of vertices of G in ascending order of discovery time. We mark the members of $N_G^+(v)$; this gives all non-neighbors of v in F. We then traverse D in ascending order until we encounter an unmarked vertex. This gives the earliest discovery time of any vertex in $N_F^+(G)$, thereby allowing us to complete the computation of $Low[v]$. We may clearly charge the cost to v and edges out of v in G. Thus, this gives $O(size(G))$ time to find the strongly-connected components of F. Moreover, since the order in which components of F are removed is unaffected by the representation, this algorithm gives a topological sort of the component graph for F.

If F is undirected, then the biconnected components of F can be computed in a similar way. A standard algorithm is given in [1]. The details of adapting it are left to the reader.

5.6 An application to modular decomposition

A number of $O(n+m)$ algorithms are known for modular decomposition [2, 4, 11]. One of them makes use of a variant of the depth-first search algorithm on outward equivalence classes that I have described above [4]. Though this algorithm is arguably simpler than the others, it is still conceptually complex.

In this section, we show that if we are willing to settle for an $O(n + m \log n)$ bound, we can use one that is conceptually simple. Some of the algorithms on outward equivalence that I have described above are critical elements in the time bound.

Let G be an undirected or directed graph, and let v be a vertex. Let $\mathcal{P}(G, v)$ denote the partition of $V(G)$ given by $\{v\}$ and the maximal modules of G that do not contain v. That this is a partition of $V(G)$ is easy to verify using Theorem 3.

Since $\mathcal{P}(G, v)$ is a partition of $V(G)$, we may decompose G into the quotient $G/\mathcal{P}(G, v)$ and $\{G|X : X \in \mathcal{P}(G, v)\}$. Using Theorem 2, it is also easy to verify that inverse images of the non-singleton modules of $G/\mathcal{P}(G, v)$ are precisely the ancestors of v in the modular decomposition of G.

Here is the restriction to undirected graphs of the modular decomposition algorithm of [5]. For the moment, assume the existence of a *Partition* operation for finding $\mathcal{P}(G, v)$.

Algorithm 8 *Compute the modular decomposition of G*

```
Decomp(G)
    If G has only one node, return a one-node tree
    else
        Select a vertex v of G
        P = P(G, v) = Partition(G, v)
        For each ancestor U of {v} in G/P in descending order
```

Create a tree node t_u
Make t_u a child of the tree node corresponding to its parent in $G/\mathcal{P}$
For each member X of $\mathcal{P}$ that is a child of U
$t_x = Decomp(G|X)$
Let t_x be a child of t_u
return the root of the resulting tree

This algorithm produces a tree that is correct, except that it allows a parallel degenerate node to be a child of another, and a series degenerate node to be a child of another. To fix this, visit the nodes of the tree in postorder, deleting any such node and letting its parent inherit its children.

To get the time bound, we will precompute the recursion tree for $Decomp(G)$ (Algorithm 8). with the following:

Algorithm 9 *RecursivePartition: Precompute the recursion tree for* $Decomp(G)$ *(Algorithm 8).*

RecursivePartition (G)
Create a single root node r
If G has more than one vertex
Select a vertex v
$\mathcal{P} = Partition(G, v)$
For each class X in $\mathcal{P}$
Let $t = RecursivePartition(G|X)$;
Let t be a child of r
return r.

Once we have done this, we may precompute all quotients referred to as $G/\mathcal{P}$ in recursive calls, by applying an off-line least-common ancestors algorithm on edges of G. We have thus reduced the problem to bounding the cost of calls to $Partition$, and of finding ancestors of v in the quotient $G/\mathcal{P}(G, v)$.

Graph partitioning For finding $\mathcal{P}(G, v)$, we can use a variant of an algorithm for the problem that is due to Spinrad [16, 11]. Spinrad's original formulation was designed to give an $O(n + m \log n)$ bound for a single call to the $Partition$ function. We must modify it slightly in order to assign this same bound to the cost of all calls to this function that are generated in $RecursionTree$ (Algorithm 9). A different modification for this same problem is given in [10].

We begin with a partition $\mathcal{P}$ of $V(G)$, and start refining $\mathcal{P}$ in a way that guarantees that no module that is contained in one class is ever split up into two partition classes. The procedure halts when no further refinement is possible; at this point, the partition consists of the maximal modules of G that are contained in a single partition class of the initial partition. By setting $\mathcal{P} = \{\{v\}, V(G) - \{v\}\}$ initially, this yields $\mathcal{P}(G, v)$.

The basic driving operation is $Pivot(G, u, X)$, where X is a union of partition classes in $\mathcal{P}$, and $u \in V(G) - X$. For each partition class C contained in X, we refine $\mathcal{P}$ by splitting C into two classes C_a and C_n, which contain the vertices of C that are adjacent and nonadjacent to u, respectively.

Algorithm 10 *Given a partition $\mathcal{P}$ of $V(G)$, find the maximal modules of G that are contained in a single partition class.*

Partition$(G, \mathcal{P})$
    if $|\mathcal{P}| = 1$ return $\mathcal{P}$
    else
        Let M be a maximal member of $\mathcal{P}$
        Select $X \in \mathcal{P} - \{M\}$
        Let $Y = \bigcup(\mathcal{P} - \{X\})$
        For each $x \in X$ do
            Pivot (G, x, Y)
        For each $y \in Y$ do
            Pivot (G, y, X)
        return Partition $(G|X, \mathcal{P}|X) \cup$ Partition $(G|Y, \mathcal{P}|Y)$

Correctness: if $|\mathcal{P}| = 1$ then its only member is $V(G)$, which is a module. Otherwise, the pivots in X and Y refine the members of $\mathcal{P}$ in a way that does not split up any modules that are contained in a class. After these pivots, no subsequent pivot on a member of X can further split any member of Y, and *vice versa*, which shows that the operation may be completed with the two recursive calls.

Time bound: We show an $O(size(G) \log n)$ bound. Let $a(X)$ be the sum of cardinalities of the adjacency lists of members of X. We first show that the cost of the pivots in the main call is $O(|X| + a(X))$.

Assume that all classes and adjacency lists are implemented with doubly-linked lists, and each list is labeled with its cardinality. Since G is undirected, we may assume that each edge (u, v) in u's adjacency list has a *twin pointer* to edge (v, u) in v's adjacency list. Mark vertices in X. Remove and collect all edges that go to Y from the adjacency lists in X that go to Y. Using the twin pointers, remove the twins from adjacency lists in Y. This gives all edges connecting X and Y. Using pre-initialized buckets to group (but not sort) these edges by beginning vertex requires $O(a(X))$ time: distribute the edges to buckets, and keep a linked list of nonempty buckets in order to avoid having to touch empty buckets when they are retrieved. This gives a list of vertices in X and Y that have neighbors in Y and X, respectively, and for each of these, a list of those neighbors. The total number of elements in these lists is $O(|X| + a(X))$.

For each neighbor y of x in Y, remove it from its current partition class L and move it to a twin class L', which must be created if L doesn't already have a twin class. Update the size labels of L and L', and delete L if y was the last element of L. This takes $O(1)$ time, and, by repeating it on all edges in x's reduced adjacency list, we get the result of the pivot that uses x. Performing the pivots on all members of X is thus $O(a(X))$. The analysis for elements of Y that have neighbors in X is similar.

Thus, we incur $O(|X| + a(X))$ time, exclusive of recursive calls. If $|\mathcal{P}| = 1$, then no charges are made to X. If $|\mathcal{P}| > 1$, we charge the cost by assigning $O(1)$ to each member of X and each element of an adjacency list in X, and the members of X go to a recursive call on a graph with at most half as many vertices, since $|X| \leq |Z|$. This is true even if the next charge occurs in a separate call to *Partition* that is generated by *RecursionTree* on a subset of X. We conclude that each time a member of X is charged, it is in a recursive call with at most half as many vertices as the last time it was charged, which gives an $O(\log n)$ bound on the number of times it or its adjacency-list elements can be charged in *RecursionTree*. Summing this over all vertices and adjacency-list elements, we get an $O(size(G) \log n)$ bound. The one-time cost of initializing the buckets that are used in the pivot operations is $O(n)$.

Finding the ancestors of a vertex The foregoing shows that we may find the set $\mathcal{P}(G, v)$, To complete the time bound of *Decomp* (Algorithm 8), we must find the ancestors of $\{v\}$ in $G' = G/\mathcal{P}(G, v)$. We described above why we may assume that we have G' available in adjacency-list form when it is time to do this step.

Let $F(G, v)$ be the graph on $V(G)$ where $v \in V(G)$ is isolated, and where for each $x, y \in V - \{v\}$ there is an arc from x to y if y is adjacent to exactly one of x and v in G. It follows that if there is an edge of $F(G, v)$ from x to y, then every module that contains x and v must also contain y. On the other hand, if $X \subset V$ contains v, and there is no edge in F from $X - \{v\}$ to $V - X$, then X is a module: any vertex y in $V - X$ must have the same relationship to all members of X as it does to v.

Lemma 11. *[5] If $x \in X \subseteq V(G)$, then X is a module iff $X - \{v\}$ has no outgoing directed edges in $F(G, v)$.*

From the fact that every module of G that does not contain v is contained in a single class of $\mathcal{P}(G, v)$, it follows that all nontrivial modules of $G' = G/\mathcal{P}(G, v)$ contain $\{v\}$. If A is a degenerate ancestor of v in the modular decomposition of G, then the union X of all of its children except the one that contains v is a maximal module of G that does not contain v, by Theorem 3. So X is the inverse image of a single node of G'. It follows by Theorems 2 and 3 that the inverse images of the modules of G' in G are the ancestors of v in $MD(G)$.

Thus, the nontrivial modules of $G/\mathcal{P}(G, v)$ are nested. The following is now immediate from Lemma 11:

Theorem 12. *[5] Let G be an undirected graph, let $G' = G/\mathcal{P}(G, v)$, $F = F(G', \{v\})$ be $\{v\}$'s forcing graph on G', and let $\mathcal{S}$ be the strongly connected components of F.*

1. *The topological sort of $F/\mathcal{S}$ is unique;*

2. *$X \subseteq V$ is an ancestor of v in $MD(G)$ iff it is the union of $\{v\}$ and the members of a suffix of this sort.*

The problem for getting an $O(n + m \log n)$ bound is that we cannot guarantee that $size(F) = O(size(G))$; F may be considerably larger. For two years after development of the algorithm of [5], this appeared to be an insurmountable obstacle to bringing the time bound of that algorithm down to $O(n + m \log n)$. The solution is completely trivial in light of the foregoing results about outward equivalence classes, however:

Lemma 13. *Let G be an undirected graph, and let $F = F(G, v)$ be v's forcing graph. Then F^T is in the outward equivalence class of G*

Proof: If (y, x) is an edge of F^T, then y is adjacent to exactly one of x and v. Thus, if v is nonadjacent to y, the neighbors of y in F^T are the same as they are in G, namely $N_G^+(y)$. If v is adjacent to y, the neighbors of y in F^T are $\overline{N}_G^+(y)$, and thus obtained by outwardly complementing y. Q.E.D.

Since we have available an adjacency-list representation of $G' = G/\mathcal{P}(G, v)$, we may compute a partially complemented representation of $F^T = (G', \{v\})$ of G' in $O(size(G'))$ time. We may then find a topological sort of the strong component graph of F^T in $O(size(G'))$ time, by the results of Section 5.5. This gives the topological sort of $\mathcal{F}$ in reverse order. We may charge this cost to edges of G that connect members of $\mathcal{P}(G, v)$, and those edges are not charged in recursive calls. Over all recursive calls of *Decomp* (Algorithm 8), this step thus contributes $O(size(G))$ to the running time.

This shows that the time bound for graph partitioning is the bottleneck for further improvements in the resulting $O(n + m \log n)$ time bound for this algorithm.

6 Algorithms on Two-Way Equivalence Classes

The two-way equivalence classes have interesting mathematical properties not shared by the outward equivalence classes. Unlike the outward classes, they are a special case of dynamic-labeled two-structures. As shown in [7], there is a tree-like representation of the modules of the entire class, similar to the one for a single graph or two-structure, but unrooted.

A *comparability graph* is an undirected graph whose edges can be oriented to give a transitive digraph, hence a partial order. A *permutation graph* is a graph where adjacencies represent pairwise inversions in a permutation, and and a *circular permutation graph* is a cyclic generalization. Rotem and Urutia show the following:

Theorem 14. *[15] Let v be a vertex in G. Let G' be the result of both inwardly and outwardly complementing each member of $N_G(v)$.*

1. *If G is a comparability graph, then so G'.*

2. *G is a circular permutation graph iff it is a comparability graph and G' is a permutation graph.*

In the case of a digraph F in G's outward-equivalence class, we obtained the time bounds by allowing each vertex v to carry a list corresponding either to $N_F^+(v)$ or to $\overline{N_F^+}(v)$. In the case of a member of a two-way class, the N_F^+ depends not only on the outward complementation of v, but also on the inward complementation status of every other member of V. We can nevertheless achieve many of the same time bounds if we keep a separate doubly-linked lists for vertices that are inwardly complemented and inwardly uncomplemented.

6.1 Splitting a set of vertices

Given a vertex v in a graph G, and a representation $F = (G, \overline{b}_1, \overline{b}_2)$ in G's two-way equivalence class, we will want to perform the pivot operation on F and be able to charge it to elements of our data structure.

Let u be a vertex on which we wish to perform a pivot, let $\mathcal{P}$ be a partition of G, and let $\mathcal{P}' \subseteq \mathcal{P}$ be a set of classes we wish to split. Suppose each member $P \in \mathcal{P}'$ is represented with a pair of lists, P_c and P_u, where P_c contains members of P that are inwardly complemented and P_u contains the members that are not inwardly complemented. For each such P, we wish to create a similar representation of $A = P \cap N_F^+(u)$ and $B = P - N_F^+(u)$. If u is outwardly complemented, then $A_c = P_c \cap N_G^+(u)$, $B_c = P_c - N_G^+(u)$, $A_n = P_n - N_G^+(u)$, and $B_n = P_n \cap N_G^+(u)$. If u is not outwardly complemented, the computation is the same, but the roles of A and B are reversed. Suppose there are k directed edges of G from u to P, and we are given these edges as a list. If $k \geq 1$, it takes $O(k)$ time to split each of P_c and P_u according to adjacency in G, and then to label and pair the four resulting lists, as described, to obtain the desired representations of P_a and P_n.

6.2 Breadth-first search

When visiting a vertex u, deciding which nodes must move from the undiscovered list to the queue requires splitting the set U of undiscovered vertices into a set $U_a = U \cap N_F^+(u)$ and $U_n = U - N_F^+(u)$. This takes time proportional to $N_G^+(u)$ by the technique of section 6.1. The cost of visiting u can be charged to edges out of u in G, so breadth-first search on F takes $O(size(G))$ time.

6.3 Depth-first search, topological sort, connectivity

A data structure for depth-first search on digraph F in G's two-way class was developed by Dahlhaus, Gustedt and me in a closely related context [4]. We keep two complement stacks, one for V_u, the set of vertices that are not inwardly complemented, and and one for V_c, the set of vertices that are inwardly complemented. Together the two stacks simulate a single meta-stack with non-duplicating pushes: a pop operation looks at the timestamps of the tops of the two stacks, and pops the most recent, and a $Push$ or $Cpush$ operation on a vertex v performs the non-duplicating $Cpush$ on one of $\{V_u, V_c\}$ and the $Push$ on the other. Which one gets the $Push$ and which gets the $Cpush$ depends on whether v is outwardly complemented, since the goal is to perform a non-duplicating push on all members of $N_F^+(v)$. The depth-first search algorithm then proceeds as before, except that it uses the meta-stack to generate the next vertex to be visited.

Topological sort, strongly-connected components, and biconnected components when F is undirected now follow in $O(size(G))$ time, using the same algorithms as before. However, I have not been able to think of an $O(size(G))$ bound for longest paths in F, when F is only required to be in G's two-way class, rather than its outward class.

6.4 Graph Partitioning

A module in a directed graph is a set X of vertices such that for each $v \in (V(G) - X)$, $X \times \{v\}$ consists of either all edges or all nonedges, and $\{v\} \times X$ consists of either all edges or all nonedges. It is possible for one of these sets to consist of edges, and the other to consist of nonedges.

We wish to have an efficient algorithm to perform graph partitioning on a graph F in G's two-way equivalence class in $O(size(G))$ time. To modify the graph partitioning algorithm, we need only change the $Pivot$ step. If u is the pivot vertex, we need to consider both $N_F^+(u)$ and $N_F^-(u) = N_{F^T}^+(u)$. By Theorem 5, we may represent F^T using G^T. It takes $O(size(G))$ time to obtain G^T with a bucket-based grouping of edges by ending vertex number.

We may now use a generalization of the technique of section 6.1, where each class Y is made up of four doubly-linked lists, one for each combination of inward and outward complementation bits. Each of these lists can be split according to membership in $N_G^+(u)$, and the resulting eight lists can then be recombined into two groups of four according to membership in $N_F^+(u)$. Which set of each split pair goes into which group depends on the outward complementation status of u. This operation can then be repeated in a similar way with G^T in order to further split according to membership in $N_F^-(u) = N_{F^T}^+(u)$. When this is complete, Y has been split into four classes (some of them possibly empty), where each is represented with four doubly-linked lists corresponding to each possible combination of inward and outward complementation status.

If there are k edges of G in $\{u\} \times V$ and $V \times \{u\}$, and we are given a list of these edges, then this operation takes $O(k)$ time, using the techniques of Section 5.6. It is then easily verified that the proof of time bound for $RecursionTree$ in section 5.6 now yields an $O(size(G))$ time bound for $RecursionTree$ on F.

7 Future Work

An open question is what other graph algorithms can be generalized to these representations of two-way equivalence classes. If G is a digraph, it appears likely that one may obtain an $O(size(G))$ time bound for modular decomposition of any digraph F in G's two-way equivalence class, by modifying a linear-time modular decomposition algorithm. An $O(size(G) \log n)$ bound is strongly suggested by the results of Section 6.4. In addition, in [12], an $O(n+m)$ algorithm is given for the *Partition* procedure, but it is quite complicated. Nevertheless, it may be possible to adapt the algorithm using techniques from section 5.6, in order to obtain an $O(size(G))$ bound for graph partitioning on a member F of G's two-way equivalence class. A number of other combinatorial problems are shown in [12] to reduce in linear time to modular decomposition and graph partitioning, but it remains to establish whether the reductions generalize to members of G's equivalence class in $O(size(G))$ time.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974)
2. Cournier, A., Habib, M.: A new linear algorithm for modular decomposition, CAAP '94: 19th international colloquium, lecture notes in computer science, Sophie Tison, ed., Edinburgh, UK (1994) 68-82
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Algorithms*, MIT Press (1990)
4. Dahlhaus, E., Gustedt, J., McConnell, R.M.: Efficient and practical modular decomposition, Proc. 8th annual ACM-SIAM Symposium on Discrete Algorithms, New orleans (1997) 26-35
5. Ehrenfeucht, A., Gabow, H.N., McConnell, R.M., Sullivan, S.J.: An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs, Journal of Algorithms **16** (1994) 283-294
6. Ehrenfeucht, A., Rozenberg, G: Angular 2-structures, Theoretical Computer Science **92** (1992) 227-248
7. Ehrenfeucht, A., Rozenberg, G.: Dynamic labeled 2-structures, Math. Struct. in Comp. Science **4** (1994) 433-455
8. Ehrenfeucht, A., Rozenberg, G.: Theory of 2-structures, part 1: clans, basic subclasses, and morphisms, Theoretical Computer Science **70** (1990) 277-303
9. Ehrenfeucht, A., Rozenberg, G.: Theory of 2-structures, part 2: representations through labeled tree families, Theoretical Computer Science **70** (1990) 305-342
10. Habib, M., Paul C., Viennot, L.: Lex-BFS, a partition refining technique, manuscript.
11. McConnell, R.M., Spinrad, J.P.: Linear-time modular decomposition and efficient transitive orientation of comparability graphs, Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, Virginia (1994) 536-545
12. McConnell, R.M., Spinrad, J.P.: Linear-time transitive orientation, Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms", New Orleans (1997) 19-25
13. Moehring, R.H.: Algorithmic aspects of comparability graphs and interval graphs, in Graphs and Orders, I. Rival, ed., D. Reidel (1985) 41-101.
14. Rose, D., Tarjan, R.E., Leuker, G.S.: Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. **5** (1976) 266-283
15. Rotem, D., Urrutia, J.: Circular permutation graphs, Networks **12** (1982) 429-437
16. Spinrad, J.P.: Vertex partitioning, Manuscript.

This article was processed using the LaTeX macro package with LLNCS style