Discrete Mathematics and Theoretical Computer Science **5**, 2002, 147–168

Partially Complemented Representations of Digraphs

Elias Dahlhaus[1] and Jens Gustedt[2] and Ross M. McConnell[3]

[1] *Dept. of Computer Science and Dept. of Mathematics, University of Cologne, Cologne, Germany.*
Email: `dahlhaus@suenner.informatik.uni-koeln.de`
[2] *LORIA and INRIA Lorraine, campus scientifique, BP 239, 54506 Vandœuvre lès Nancy, France.*
Email: `Jens.Gustedt@loria.fr`.
[3] *Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873 USA.*
Email: `rmm@cs.colostate.edu`.

received Dec 12, 1999, *accepted Apr 23, 2002.*

A *complementation operation* on a vertex of a digraph changes all outgoing arcs into non-arcs, and outgoing non-arcs into arcs. This defines an equivalence relation where two digraphs 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 G' of G's equivalence class in $O(n+m)$ time, where m is the number of arcs in G, not the number of arcs in G'. This may have advantages when G' is much larger than G. We use this to generalize to digraphs a simple $O(n+m\log n)$ algorithm of McConnell and Spinrad for finding the modular decomposition of undirected graphs. A key step is finding the strongly-connected components of a digraph F in G's equivalence class, where F may have $\omega(m\log n)$ arcs.

Keywords: efficient graph algorithms, data structures, search strategies, modular decomposition

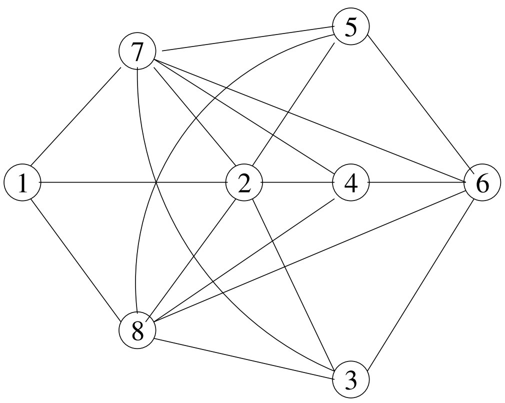
1 Introduction

It has been 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 (Ehrenfeucht and Rozenberg (1990a,b, 1992)). In particular, the family of modules of a digraph $G=(V,E)$ is the same as the family of modules of $\overline{G}$. Thus, from the point of view of modular decomposition, the distinction between arcs and non-arcs is an artificial one, and G and $\overline{G}$ are best viewed as a single structure. This structure is simply a partition of

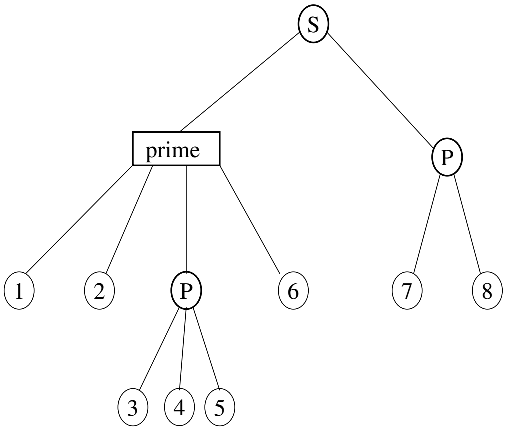
$$E_2(V) = V \times V \setminus \{(v,v) \mid v \in V\} \tag{1}$$

into two sets, where the distinction of these classes as *arcs* and *nonarcs* is best dropped.

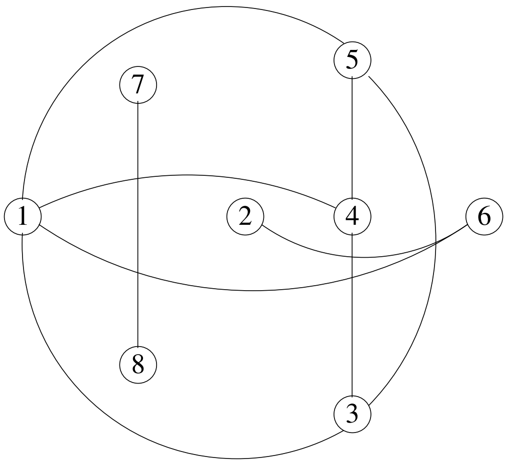
Figure 1 illustrates this fact with the so-called decomposition trees for an undirected graph and its complement: the trees only differ by the labels of internal nodes. We will see the definitions of this trees and the rules that apply below.



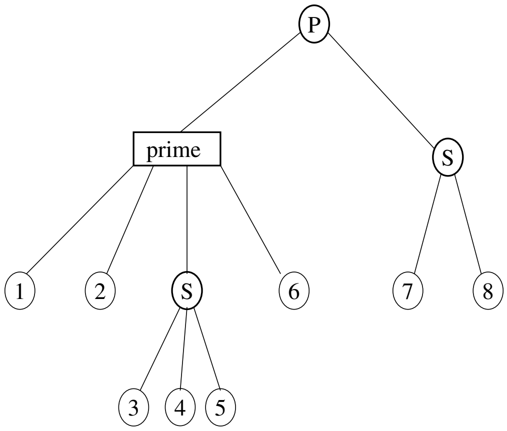
(a) An undirected graph G



(b) G's decomposition tree



(c) The complement $\overline{G}$



(d) $\overline{G}$'s decomposition tree

Fig. 1: Graph complements and decomposition trees

In this paper, we develop results that are based on this idea, and that were given in preliminary form in Dahlhaus et al. (1997) and McConnell (1997). We establish a theorem that is essential to the results of Dahlhaus et al. (2001), but whose proof we deferred to the present paper. We show how an $O(n+m\log n)$ algorithm for modular decomposition of undirected graphs given in McConnell and Spinrad (2000) can be adapted to digraphs using the strategy.

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, see McConnell and Spinrad (1999). 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.

Ehrenfeucht and Rozenberg (1994) examine a transformation that operates on the arcs out of a vertex or else on the arcs 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*.

In this paper, we explore algorithmic uses of this concept on graphs and digraphs. An *outward complementation operation* is where only the outgoing arcs of a vertex are complemented. That is, the neighbors of the vertex are turned into non-neighbors and the non-neighbors are turned into neighbors. An *inward complementation operations* is where only the inward arcs are complemented. A *symmetric complementation operation* is one where both the inward and the outward arcs are complemented.

Definition 1.1 *We define the following equivalence relations on the set of digraphs:*

- *Two digraphs are* outwardly *equivalent if one can be obtained from the other by a sequence of outward complementations;*
- *Two digraphs are* inwardly *equivalent if one can be obtained from the other by a sequence of inward complementations;*
- *Two digraphs are* two-way *equivalent if one can be obtained from the other by a mixed sequence of outward and inward complementations;*
- *Two digraphs are* symmetrically *equivalent if one can be obtained from the other by a sequence of symmetric complementations.*
- *The* outward, inward, symmetric, *and* two-way *equivalence classes are the equivalence classes induced by the corresponding equivalence relations.*

Any sequence of outward complement operations is easily seen to be represented by a Boolean variable for each vertex: re-complementing an already-complemented vertex is the same as doing nothing. Thus any representation of a digraph G that is outward equivalent to a digraph F can be used to represent F if we maintain a Boolean vector that indicates which vertices of G must be outwardly complemented to obtain F. In a similar way any two-way equivalent digraph can be used to represent F by using two such Boolean vectors.

Figure 2 shows the conventional adjacency list representation of our example graph and the savings of this representation with respect to the adjacency matrix. Observe that the outward equivalent digraph representing our undirected graph G is not symmetric anymore.

If G is a digraph, let $n(G)$ denote the number of vertices and let $m(G)$ denote the number of arcs. One of the main goals the present paper is to demonstrate the following:

Theorem 1 *Let G be a graph given in adjacency-list form. The following can computed on any member F of G's outward equivalence class in $O(n(G)+m(G))$ time.*

1. *Finding a breadth-first spanning forest;*
2. *Finding a depth-first spanning forest;*
3. *Finding a topological sort of a dag;*
4. *Finding the biconnected components of an undirected graph;*
5. *Finding the strongly-connected components of a digraph;*
6. *Determining whether a graph is chordal.*

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 an arc to all other vertices of the digraph, and no member of B has an arc to any other vertex. The size of this digraph 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 $\Omega(n^2)$ digraphs 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.

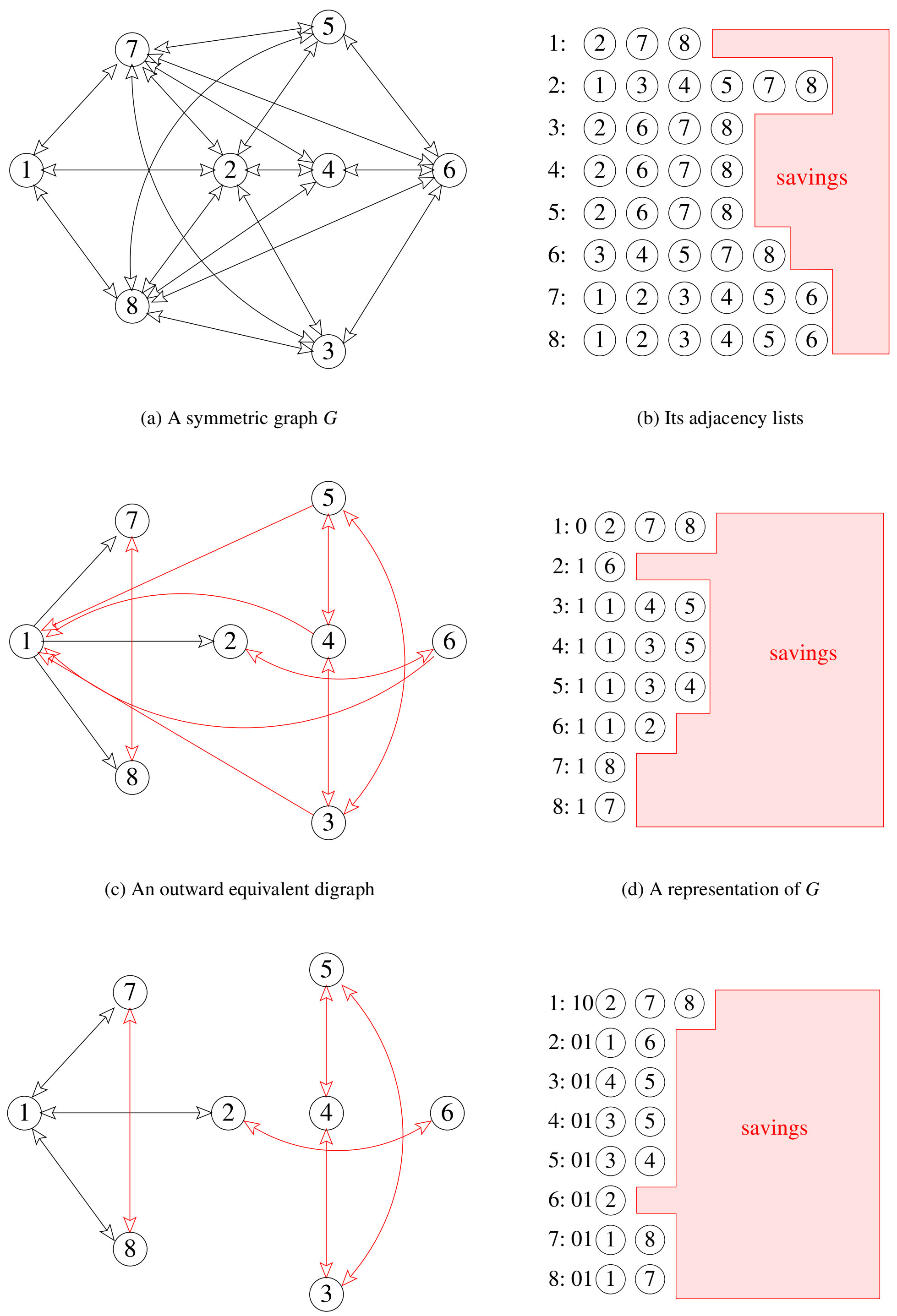
Theorem 1 is essential to the $O(n+m\log n)$ algorithm for modular decomposition of digraphs we give here and for the linear-time modular decomposition algorithm for undirected graphs we give in Dahlhaus et al. (1997, 2001). In the present paper, we also extend this to outward equivalence classes in the following way:

Theorem 2 *The modular decomposition of an undirected member F of the outward equivalence class of a given digraph G can be found in time*

$$O((n(G)+m(G))\log n(G)).$$

Modular decomposition applies to both graphs and digraphs. The history of algorithms for the problem dates back to the 1960s. The first $O(n^2)$ algorithm, for undirected graphs, appeared in Muller and Spinrad (1989), followed by an $O(n+m\alpha(m,n))$ algorithm in Spinrad (1992). The first linear-time algorithm to appear, (McConnell and Spinrad (1994, 1999)) was an adaptation of this last algorithm. It makes use of properties of undirected graphs that have no apparent generalization to digraphs. In particular, it constructs a P_4 *tree*, which is based on induced P_4s in the graph, and induced subgraphs that are *cographs*, which are a class of undirected graphs. We believe that a linear-time algorithm due to Cournier and Habib (1994) can be applied to digraphs, but it is difficult to understand, and has not appeared in final form.

A digraph is a *transitive dag* if it is acyclic, and whenever (u,v) and (v,w) are arcs, (u,w) is also an arc. The importance of transitive dags is that they model partial orders. A *transitive orientation* of an



(a) A symmetric graph *G* (b) Its adjacency lists

(c) An outward equivalent digraph (d) A representation of *G*

(e) A two-way equivalent digraph (f) A representation of *G*

Fig. 2: One-way and two-way equivalence

undirected graph is an orientation of its edges that is a transitive dag. *Comparability graphs* are the class of graphs that can be transitively oriented. We generalize another algorithm of McConnell and Spinrad (2000) for transitive orientation of comparability graphs to obtain the following:

Theorem 3 *Let F be a member of G's outward equivalence class that is a comparability graph. A linear extension of a transitive orientation of F may be found in time*

$$O((n(G)+m(G))\log n(G)).$$

There may be other graph problems that can be given bounds similar to those of Theorems 1, 2, and 3, so there are open problems in this area. It appears likely that some of the results we give here can be generalized to two-way and symmetric equivalence classes, but these are open problems. Finally, the usefulness of Theorem 1 for modular decomposition of the given graph gives hope that it may have applications to other graph-theoretic problems besides modular decomposition.

2 Basic Definitions

If G is a digraph, let $V(G)$ be the set of vertices and $E(G)$ be the set of arcs. G^T denotes the *transpose* of G, that is, the digraph that is obtained by reversing the directions of all arcs in G. The complement $\overline{G}$ is obtained by changing each arc into a non-arc and each non-arc into an arc. A *non-edge* of G is an edge of $\overline{G}$.

A digraph is a *linear order* if it has no cycles, and between each pair $\{u,v\}$ of distinct nodes either (u,v) or (v,u) is an arc. Linear orders are also known as *transitive tournaments*. Observe that for such a linear order the transpose and the complement coincide.

Undirected graphs will be treated as a special case of digraphs, where each undirected edge $\{u,v\}$ is really two arcs (u,v) and (v,u). Thus, algorithms that we describe for digraphs will be general also to undirected graphs. A *co-component* of a undirected graph G is a connected component 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.

If G is a digraph and $X \subseteq V(G)$, then $G|X$ denotes the *subgraph of G induced by X*, namely, the digraph whose vertices are X and whose arcs are those arcs of G that go from one member of X to another. That is,

$$G|X = \big(X,(X\times X)\cap E(G)\big). \tag{2}$$

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 digraph whose vertices are the members of $\mathcal{F}$, and where (X,Y) is an arc of $G/\mathcal{F}$ iff there is some arc of G from a member of X to a member of Y.

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

2.1 Modular decomposition

A *module* in a digraph G is a set M of vertices such that for each vertex $v \in V(G)\setminus M$, either every member of $\{v\}\times M$ is an arc or none is, and either every member of $M\times\{v\}$ is an arc or none is. However, $\{v\}\times M$ consists of arcs while $M\times\{v\}$ does not, and *vice versa*.

By restricting the definition of a module to an undirected graph, we see that a module M is a set such that for each vertex $v \in V(G) \setminus M$, M is either contained in v's neighborhood or disjoint from v's neighborhood. Trivial examples of modules are $V(G)$, the singleton sets $\{\{v\} \mid v \in V(G)\}$, 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 connected. The set of vertices numbered 1 to 6 in Figure 1 gives an example of such a case.

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 arc or none is. Thus, two disjoint modules are either *adjacent* or *nonadjacent*. It follows that if there is a partition $\mathcal{P}$ of $V(G)$ into disjoint modules, the quotient $G/\mathcal{P}$ uniquely specifies all arcs that connect members of different partition classes. We will call such a partition and its 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 G is disconnected and each element of $\mathcal{P}$ is a union of connected components, then $\mathcal{P}$ is a modular partition, and $G/\mathcal{P}$ is empty. Clearly, $P \in \mathcal{P}$ is a single connected component iff there is no refinement $\mathcal{P}'$ of P such that $G/((\mathcal{P} \setminus \{P\}) \cup \mathcal{P}')$ is empty. If each member of $\mathcal{P}$ is a single connected component, then $\mathcal{P}$ is a *parallel decomposition* of G.

Similarly, if $\overline{G}$ is disconnected, and each member of $\mathcal{P}$ is a union of co-components, then $G/\mathcal{P}$ is complete. $P \in \mathcal{P}$ is a single co-component iff there is no refinement $\mathcal{P}'$ of P such that $G/((\mathcal{P} \setminus \{P\}) \cup \mathcal{P}')$ is complete. If each member of $\mathcal{P}$ is a single co-component, then $\mathcal{P}$ is a *serial decomposition* of G.

In the case of digraphs, there is a third type of decomposition with analogous properties. A modular partition $\mathcal{P}$ is *linear* if $G/\mathcal{P}$ is a linear order. That is, there are arcs between each pair of partition classes, but $G/\mathcal{P}$ has no cycle. By analogy with the parallel and serial decomposition, let us say that $P \in \mathcal{P}$ is a *linear component* if there is no partition $\mathcal{P}'$ of P such that $G/((\mathcal{P} \setminus \{P\}) \cup \mathcal{P}')$ is linear. If each member of $\mathcal{P}$ is a linear component, then $\mathcal{P}$ is a *linear* decomposition of G.

Because of their restricted structures, parallel, series, and linear nodes are known as *degenerate* nodes. It is easily verified that a digraph admits at most one of these three decompositions, but it may also not admit any of the three. The smallest examples of such digraphs have as few as three vertices. The digraph

$$1 \longrightarrow 2 \longrightarrow 3$$

is such an example. The smallest example of an undirected graph that is not decomposable by any of these two operations is P_4, an undirected path on 4 vertices.

However, in such a case, the maximal modules that are proper subsets of G are a partition $\mathcal{P}$ of G, and $G/\mathcal{P}$ has only trivial modules. In this case, $\mathcal{P}$ is a *prime* decomposition of G.

Thus, G is uniquely decomposable with a parallel, series, linear, or prime decomposition. Applying this idea recursively to the factors gives a unique hierarchical decomposition of G that implicitly represents all modules of G. This is summarized by the structure produced by Algorithm 1. This decomposition, known as the *modular decomposition* of G, has been rediscovered in various contexts, and is also known as the *substitution decomposition*, see Möhring (1985), or the *prime tree family*, see Ehrenfeucht and Rozenberg (1990a,b).

Theorem 4 *(Möhring (1985); Ehrenfeucht and Rozenberg (1990a,b)) If X is a module of G, then $S \subseteq X$ is a module of $G|X$ iff it is a module of G. If $\mathcal{P}$ is a modular partition of $V(G)$, then the inverse image of $\mathcal{F}' \subseteq \mathcal{P}$ is a module of G iff $\mathcal{F}'$ is a module of $G/\mathcal{F}$.*

The following is easily established using Theorem 4:

Algorithm 1: MD(G)

```
if G has only one vertex then 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 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 node
    else if G is linearly decomposable then
        let 𝓕 be the linear components;
        let r be labeled a linear node
    else
        Let 𝓕 be the maximal modules of G;
        Let r be labeled a prime node
    for each X ∈ 𝓕 do
        Let r_x = MD(G|X);
        add r_x as a child of r
    return r
```

Theorem 5 *(Möhring (1985); Ehrenfeucht and Rozenberg (1990a,b)) A set X is a module of G if and only if it is one of the following:*

1. *A node of the decomposition tree;*
2. *A union of children of a parallel or series node;*
3. *The union of a subset of the set $\mathcal{P}$ of children of a linear node X that are consecutive in the linear order given by $(G|X)/\mathcal{P}$.*

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.

3 Properties of outward, inward, and two-way equivalence classes

Let $G=(V,E)$ be a digraph. For any $v \in V$ the *minimum outdegree* of v, $mindeg^+(v)$, is $\min\{deg^+(v), n(G)-1-deg^+(v)\}$. The minimum outdegree, $mindeg^+(G)$, of G itself is n plus the sum of these values over all the vertices. It is an important invariant of the outward equivalence class of G.

Remark 3.1 *Let $G=(V,E)$ be a digraph then $mindeg^+(G)$ is the minimum size of a digraph F that is outward equivalent to G. If $|V|$ is even, F is unique.*

Proof: Just observe that flipping the out-list of some vertex v doesn't affect the way we represent the out-neighbors of any other vertex. Thus the minimal representation is that one where we flip the interpretation if the size of the list is greater than $(n(G)-1)/2$. □

Remark 3.2 *A minimum-size member of the outward equivalence class of a digraph G can be computed in $O(n(G)+m(G))$ time, by outwardly complementing any node of G whose outdegree exceeds $(n(G)-1)/2$.*

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

Theorem 6 *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 arcs of each vertex, or by complementing the list of incoming arcs of each vertex. □

Theorem 7 *Let G be a digraph, let F be a member of G's outward equivalence class, and let $X \subseteq V(G)$. Then $F|X$ is in the outward equivalence class of $G|X$.*

Proof: Some subset of vertices of X are complemented in G to obtain F. Complementing these same vertices in $G|X$ obviously yields $F|X$. □

Theorem 8 *A digraph F is in the outward equivalence class of a digraph G if and only if F^T is in the inward equivalence class of G^T.*

Proof: Outwardly complementing a set of vertices of G and then taking the transpose of the result yields the same result as taking the transpose of G and then inwardly complementing the same set of vertices. □

4 Determining whether a member of G's outward class is undirected

In this section we illustrate the use of a classification of the vertices according to whether or not their adjacency lists are complemented.

Let G be a digraph, and let F be a member of G's outward equivalence class. Let V^1 be the set of vertices of G that must be outwardly complemented to obtain F, and let V^0 be the remaining vertices of G. Let $G^{0/1}$ be the digraph with $V(G^{0/1}) = V(G)$ and

$$E(G^{0/1}) = E(G) \cap \big((V^1 \times V^0) \cup (V^0 \times V^1)\big). \tag{3}$$

That is, let $G^{0/1}$ give the arcs of G that go back and forth between V^0 and V^1. In Figure 2(c) the sets are $V^0 = \{1\}$ and $V^1 = \{2,\ldots,8\}$.

Remark 4.1 *F is undirected iff $G|V^1$ and $G|V^0$ are symmetric, and $G^{0/1}$ is antisymmetric. This fact can be checked in $O(n(G)+m(G))$ time.*

Proof: The 'iff' claim is immediate. For the time complexity it is easy to see that the three digraphs $G|V^1$, $G|V^0$, and $G^{0/1}$ can be easily computed within the bound.

Given the adjacency-list representation for a digraph, we may determine whether it is symmetric in linear time as follows. For each arc (u,v), create a record for (u,v) and a record for its transpose (v,u). Radix sort the resulting records using vertex of origin as primary sort key and destination vertex as secondary sort key. This requires two passes, one for each sort key, and $O(n(G)+m(G))$ time per pass. The graph is undirected iff for every ordered pair (x,y) of vertices that occurs in the list occurs twice. This can be determined in $O(n(G)+m(G))$ time by traversing the sorted list of records.

To find out whether $G^{0/1}$ is antisymmetric, follow a similar procedure, but verify that each ordered pair that occurs in the sorted list of records occurs only once. □

5 Breadth-first search

Let us examine the running time of breadth-first search on a member F of G's outward class. As in the standard algorithm, divide $V(G)$ into *undiscovered* nodes, *queued* nodes, and *processed nodes*. Initially all nodes but the start node are undiscovered, and the start node is inserted to a queue. To process a node, remove it from the front of the queue, and insert any undiscovered neighbors to the back of the queue, marking them as discovered. Keep the undiscovered nodes in a doubly-linked list. This list will in fact allow us to amortize the cost of processing complemented vertices.

To process an uncomplemented node, proceed as in the standard case. When processing a complemented node v, mark the members of $N_G^+(v) = \overline{N}_F^+(v)$. Then traverse the list of undiscovered nodes, splicing out any unmarked nodes and inserting them on the queue. We then remove the marks from $N_G^+(v)$.

Marking and unmarking nodes can be charged to the arcs of G that are responsible for applying the marks. Touching an unmarked node can happen at most once per node, since the node is removed from the list of undiscovered nodes whenever this happens. This gives the $O(n(G)+m(G))$ bound.

6 Depth-first search

A *depth-first visit at v* visits all vertices reachable on a directed path from a starting vertex v, in a depth-first order. In the special case of an undirected graph, these is just the vertices in v's connected component. This generates a depth-first tree which may not contain all vertices of G, since some vertices may not be reachable from v. A *depth-first search* iteratively calls depth-first visits on unvisited vertices until all vertices have been visited. When a depth-first visit finishes, the choice of next unvisited vertex is arbitrary.

The vertex sets of the trees are a partition of the vertices of G. The postorder numbering on the nodes of a tree is a linear order on them, which can be expressed with an ordered list. The preorder numbering can be represented similarly. Ordering the postorder lists in the order in which their trees were produced and then concatenating them in this order gives a total order on vertices of G, called the *finishing times*, which can be used in the solution of other problems on the graph. Similarly, ordering the preorder lists in the order in which their trees were produced and then concatenating them gives a total order called the *discovery times*.

Rather than performing depth-first search in the standard recursive fashion, we use an explicit stack to maintain information relevant to the recursion stack. To be able to do this on any member of G's outward equivalence class, we must modify the basic operations that are supported by the stack.

One way to implement this (for a standard graph) is by using a stack of vertices, where each popped vertex is labeled with a *push time* that tells when it was last pushed. The time clock is a counter that is periodically incremented. To select a vertex x to visit, pop it from S, and record the current time as its discovery time. Look up the vertex y whose discovery time is x's push time; if y exists, it is the parent of x in the depth-first forest. Then push copies of all unvisited neighbors on S, *deleting any lower occurrences of them in S if they already reside on the stack.* The lower occurrences may be deleted, because the existence of a higher instance of a vertex guarantees that it will be visited before the lower instance is popped. When these nodes are pushed, make the current time their push time. Then increment the time clock by one unit.

To apply this technique for depth-first search on a a member of G's outward equivalence class, we have to ensure that we implement the push operation for complemented vertices efficiently. We use a data structure, *complement stack*, which generalizes a stack.

6.1 Complement stacks

The complement stack is a data structure for managing members of $V(G)$. It can hold at most one copy of any element. Let X be a set and $V(G) \setminus X$ be its complement. One of the operations the complement stack supports is purging the stack of any occurrences of members of $V(G) \setminus X$, and then pushing $V(G) \setminus X$ to the top of the stack. The amortized timed bound for this operation is $O(|X|)$, not $O(|V(G) \setminus X|)$.

Definition 6.1 *A* complement stack *is a data structure that supports the following operations:*

`Initstack`(V) *initializes and returns an empty stack S that can hold elements of V.*

`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`$(V \setminus X,S)$.

`Eliminate`(x,S) *Removes x from the universe V that applies in future calls to* `Cpush` *on S. If x is on S, it deletes it from S.*

`Pop`(S) *returns the top element of S.*

`Time`(S) *returns a timestamp that tells when the top element of S was pushed with a* `Push` *or* `Cpush` *operation.*

Theorem 9 *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.*
- `Eliminate`(x,S) *requires* $O(1)$ *time.*
- `Pop` *and* `Time` *require* $O(1)$ *time each.*

Proof: If an element x is on the stack, then it has been pushed, and possibly repushed, by a sequence of one or more calls to `Push` and `Cpush` that caused x to go to the top of the stack. If the last of these is a `Cpush`, we say that x was *last pushed* by `Cpush`; otherwise, we say it was last pushed by `Push`. The *most recent call* to `Cpush` is the most recent one of all those calls to `Cpush` that have occurred on the stack, not just the most recent of those that moved x to the top of the stack.

Let L be the members of V that are not on the stack. We implement the stack by simulating it with smaller stacks, as follows. A, T, B are doubly-linked lists that partition the members of the stack. A is a stack of elements that were last pushed by a call to `Push`. T and B may be thought of as the "top" and "bottom" of a separate stack, which holds those elements that were last pushed by a call to `Cpush`. T holds those elements that were pushed by the most recent call to `Cpush`. B holds those elements that were last pushed by an earlier call to `Cpush`, in the reverse order in which they were pushed.

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 integer variable is incremented after each call to `Pop`, `Push`, or `Cpush`.

For the elements in T we will not be able to maintain an individual timestamp when they have been pushed into T. Instead, we maintain a common timestamp for all of T since these elements have all been pushed by the same call to `Cpush`. If an element residing in T is popped, we label it with T's timestamp at that time.

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, and if it is currently a member of V, it 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 and adds one credit to each member of X. 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 T's timestamp and moves it to the front of B. If T is now nonempty, T is assigned a new timestamp, which is interpreted as the collective timestamp of all members of T, and which serves as a record of the push time of any member of T that gets popped.

  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.

- `Eliminate`(x,S) removes x from the member of $\{L,A,T,B\}$ that contains it. This takes $O(1)$ time.

- `Pop` looks at the timestamps of the top elements of T, B and A. It chooses the most recently pushed element x, pops it from the appropriate list, and returns it.

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. □

Corollary 6.2 *Computing a depth-first forest of a member of G's outward equivalence class takes $O(n(G)+m(G))$ time.*

Proof: By calling `Eliminate` on each node when it is visited, we maintain the invariant that the universe V that applies to the stack operations is the set of unvisited vertices. A vertex x is visited when it is first popped. The time of this pop is the discovery time of x, and the parent of x is the vertex whose discovery time is the the time of x's last push, minus one. Since the push times and discovery times range from 0 to $2n$, the vertices may be indexed by discovery time, allowing the push times of the vertices to serve as parent pointers in the depth-first forest. To complete the visit of x, push its unvisited neighbors with a call to `Push` if it is uncomplemented, or push its unvisited non-neighbors with a call to `Cpush` if it is complemented. This takes $O(1+|N_G(x)|)$ time, whether or not x is complemented. The corollary follows from the fact that each vertex is visited exactly once. □

6.2 Set-complement stacks

We wish to compute the strongly-connected components of a graph F that is in the same outward equivalence class as G in $O(n(G)+m(G))$ time. To do this, it will be useful to find a depth-first forest of F^T in $O(n(G)+m(G))$ time. For this, we need a generalized version these stacks.

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

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

`Cpush`(X,S,i) performs `Push`$(V_i \setminus X, S)$.

`Eliminate`(x,S) removes x from the set V_i that contains it, and deletes x from S if x is currently on S.

Theorem 10 *Set-complement stacks can be implemented in such a way that:*

- *`Initstack` requires $O(n)$ time.*
- *`Push`(X,S) requires $O(|X|)$ time.*
- *`Cpush`(X,S,i) requires $O(|X|)$ time, amortized.*
- *`Eliminate`(s,X) requires $O(1)$ time.*
- *`Pop` and `Time` each require $O(1)$ time, amortized.*

Proof: In any implementation, a `Push` or `Cpush` operation conceptually inserts a set of elements that occupy an interval I at the top of S. After subsequent pushes, I is no longer at the top of S. Eventually, a set of pop operations may pop everything above I and then pop everything in I. At this point, we may think of I has having been popped.

Exploiting this idea, we employ a stack of intervals to help model the state of S. Each of the intervals on S has a unique associated timestamp, and the timestamps are in ascending order from the bottom to the top of S. It is important to note that some of the intervals may become empty, but still remain on our stack of intervals. The reason for this is that after a `Push` or `Cpush` creates an interval I, elements of I may be removed and moved to a new interval at the top of S by a subsequent `Push` or `Cpush`. Eventually if I becomes empty, we treat it as an empty interval that continues to occupy the same place between its neighboring intervals on the stack. I is removed from the stack only when all intervals above it are explicitly popped, and then I is explicitly popped.

Using this idea, let us now prove the result. If $p = 1$, the result follows from Theorem 4.1. When $p > 1$, the stack may be simulated with p complement stacks, $S_1, \ldots, S_p$, one for each V_i, and an additional (ordinary) stack R of `Push` and `Cpush` intervals that reside on S. R represents each interval with an ordered pair (t, i), where t is the timestamp of the `Push` or `Cpush` operation that created it, and i is the index of the substack S_i that it operated on.

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

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

`Cpush`(X, S, i) calls `Cpush`(X, S_i) and pushes one instance of (t, i) on the stack R.

`Eliminate`(x, S) looks up the partition class V_i containing x and calls `Eliminate`(x, S_i).

`Time`(S) lets (t, i) denote the pair on top of R. If this corresponds to a nonempty interval, we may return t. However, it is possible that the interval corresponding to (t, i) has become empty, and it would be incorrect to return t. In this case, we must disregard the interval and try again, by popping again from R. The interval has become empty if and only if `Time`$(S_i) \neq t$. So `Time`(S) iteratively lets $(t, i) =$ `Pop`(R) until $t =$ `Time`(S_i).

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

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

The time bounds clearly remain unchanged except for `Time`, and for `Pop`, since it calls `Time`. `Time` can cause a large number of items to be popped from R. We charge this cost to the calls to `Push` and `Cpush` that originally pushed them to R, leaving $O(1)$ operations charged to `Time`. □

Corollary 6.3 *If F is a member of G's inward equivalence class, it takes $O(n(G) + m(G))$ time to compute a depth-first forest of F.*

Proof: For each vertex x, divide its neighbors into a set $L_s(x)$ of vertices that are not inwardly complemented, and a set $L_c(x)$ of nodes that are inwardly complemented. Let V_s and V_c denote the inwardly-uncomplemented and inwardly-complemented nodes of G, respectively. Call `Initstack`(V_s, V_c) to initialize a set-complement stack S. When x is visited, push its neighbors in F by making a call to

$\texttt{Push}(L_s(x), S)$ and a call to $\texttt{Cpush}(L_c(x), S, c)$. This takes $O(|L_s(x) + L_c(x)|)$ amortized time. Each arc of G appears once in $L_s()$ or $L_c()$. □

Corollary 6.4 *If F is a member of G's outward equivalence class, it takes $O(n(G) + m(G))$ time to compute a depth-first forest of F^T.*

Proof: Radix sort all arcs of G with destination as primary sort key and origin as secondary sort key. This gives an adjacency-list representation of G^T in $O(n(G) + m(G))$ time. By Theorem 8, F^T is in the inward equivalence class of G^T, so the result follows by Corollary 6.3. □

The following extension of complement representations to bipartite graphs is critical for the linear time bound of the modular decomposition algorithm of Dahlhaus et al. (2001). We deferred the proof of it to the present paper.

Let the *bipartite complement* of a directed bipartite graph (V_1, V_2, E) be the graph

$$\left(V_1, V_2, \left((V_1 \times V_2) \cup (V_2 \times V_1)\right) \setminus E\right). \tag{4}$$

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

Theorem 11 *Given a mixed representation of a directed bipartite graph G, constructing a depth-first forest for G or G^T takes time that is linear in the size of the representation.*

Proof: For the depth-first forest on G, execute $S = \texttt{Initstack}(V_1, V_2)$. When visiting a node x, suppose without loss of generality that it is in V_1. $\texttt{Push}$ its neighbors in V_2 with a call to $\texttt{Push}(N(x), S)$, or else with a call to $\texttt{Cpush}\left(\overline{N}(x), S, 2\right)$, depending on whether x is complemented.

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

7 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 arc 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, see Cormen et al. (1990). Thus, we may find a topological sort of any dag in G's outward class in $O(n(G) + m(G))$ time.

8 Longest path in a dag

We will let the *length* of a path denote the number of arcs 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 arcs out of v, and we get an $O(n+m)$-time algorithm.

We now show that if F is a member of G's outward equivalence class, we may find a longest path in F in $O(n(G)+m(G))$ 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 arcs out of v in G. Thus it takes $O(n(G)+m(G))$.

9 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.

Below, we will find it useful to compute not just the strongly-connected components of a member of G's outward equivalence class, but a topological sort of its component graph. An algorithm given in Cormen et al. (1990) for strongly-connected components also produces a topological sort of the component graph, though this is not mentioned explicitly. Most people who are familiar with the proof of that algorithm would have little trouble establishing this, but for completeness, we give a proof here.

Let us define the *finishing time* of a set S of vertices to be the latest of the finishing times of members of S.

Lemma 9.1 *In a depth-first search of G, the strongly connected components, taken in descending order of finishing time, give a topological sort of the component graph.*

Proof: Let (X,Y) be an arc of the component graph. Since the component graph is a dag, there is no directed path from Y to X. Let v be the first-discovered vertex in $X \cup Y$. If $v \in X$, then every vertex in $X \cup Y$ becomes a descendant of v, and v has latest finishing time of any vertex in $X \cup Y$. It follows that the finishing time of X is later than the finishing time of Y. If $v \in Y$, then Y finishes when v does, since every vertex in Y is reachable from v. Since there is no directed path from Y to X in the component graph, there is no directed path from v to any member of X in G. It follows that v finishes before any member of X is discovered, and once again, X has later finishing time. Thus, (X,Y) is directed from a later-finishing to an earlier-finishing component. Since (X,Y) is arbitrary, this must be true of all arcs of the component graph, yielding the result. □

This does not immediately help find the strongly-connected connected components, since, without knowing the members of the components, one cannot find the finishing times of the components. There is a fortunate exception to this: the vertex x of G with latest finishing time in all of G must give the finishing time of the component X that contains it, and X must be a source in the component graph, since it is first in topological sort of the component graph. To find X, call depth-first visit on x in G^T. Since X is a sink in G^T, no other vertices are reachable from x, and the first depth-first visit visits precisely the members of X.

Let $SCCGraph(G)$ be the component graph of G, which we seek to find. The above step shows how to find the first component X in the topological sort of $SCCGraph(G)$ given by Lemma 9.1. Next, observe that $G^T \setminus X = (G \setminus X)^T$, and $SCCGraph(G \setminus X) = SCCGraph(G) \setminus X$. Also, in any topological sort of $SCCGraph(G)$, removing X yields a topological sort of $SCCGraph(G \setminus X)$. Thus, recursing on $G^T \setminus X$ gives the remaining strongly-connected components in topological order. This recursive algorithm amounts to a second depth-first search on G^T, except that when a new depth-first visit is initiated, it must be initiated on the unvisited vertex with latest finishing time from the first depth-first search. This vertex is selected by traversing downward through a list of vertices from the first depth-search that gives the vertices in descending order of finishing time.

Theorem 12 *It takes $O(n(G)+m(G))$ time to find the strongly-connected components of a member of G's outward equivalence class, as well as a topological sort of its component graph. Given a mixed representation of a directed bipartite graph, finding the strongly-connected components and a topological sort of its component graph takes time linear in the size of the representation.*

Proof: The algorithm given above reduces the problem to a depth-first search on G, a depth-first search on G^T, and finding finishing times of the vertices. The bound follows from Corollary 6.4 and Theorem 11. □

Theorem 13 *It takes $O(n(G)+m(G))$ time to find the biconnected components of a member of G's outward equivalence class. Given a mixed representation of an undirected bipartite graph, finding the biconnected components takes time linear in the size of the representation.*

Proof: Let $d[v]$ give the preorder number for vertex v in a depth-first forest on G. Let $low[v] = min$ $\{d[v], \{low[w] : w$ is a child of v in the depth-first forest$\}\}$. Clearly, $low[]$ can be computed for all vertices in $O(n)$ time, by traversing the forest in postorder. Computation of the biconnected components reduces in linear time to computation of $low[]$, see Aho et al. (1974). □

The following extends the techniques to graphs that are not necessarily in the outward or inward equivalence classes of G or its transpose, but which may be expressed as a union of such graphs.

Theorem 14 *Let $G = (V,E)$ be a digraph, and let $H = (V, F_1 \cup F_2 \cup \ldots \cup F_k)$, where for each i from 1 to k (V,F_i) or its transpose is in the outward equivalence class of G or of G^T. It takes $O(k(n(G)+m(G)))$ time to find a depth-first forest of H, to compute its strongly-connected components, and to produce a topological sort of its component graph.*

Proof: We have shown that it takes $O(n(G)+m(G))$ time to find a depth-first forest of each F_i or its transpose. This requires the use of a stack S_i to compute the forest on F_i, and the structure of S_i depends on whether F_i or its transpose is in the outward equivalence of G or G^T. For a depth-first search of H, initialize the same set of stacks and have them collectively simulate a single depth-first search stack on H. A vertex may occur once in each of the stacks. To select the next vertex to be visited, search the tops of these k stacks for the vertex x with the earliest time stamp. `Pop` x from its stack, and then call `Eliminate` on the remaining stacks to eliminate x from them. This takes $O(k)$ time. Then visit each S_i, performing a `Push` or a `Cpush` to push the neighbors of x in F_i onto S_i. This takes $O(1+|N_G(x)|)$ time on each stack. The total time spent visiting x is $O(k(1+|N_G(x)|)$, so the whole algorithm takes $O(k(n(G)+m(G)))$ time.

To find the strongly-connected components and a topological sort of the component graph, we must also perform a depth-first search on H^T. But $H^T = {F_1}^T \cup {F_2}^T \cup \ldots \cup {F_k}^T$, so it takes $O(k(n(G)+m(G)))$ time to perform the search on H^T by the foregoing. □

10 An application to modular decomposition

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 5.

We now give the restriction to digraphs of the modular decomposition algorithm of Ehrenfeucht et al. (1994). For the moment, assume the existence of a *`Partition`* operation for finding $\mathcal{P}(G,v)$.

Algorithm 2: `Decomp`(G), compute the modular decomposition of G.

```
if G has only one node then 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 do
        Create a tree node t_U;
        Make t_U a child of the tree node corresponding to its parent in G/P;
        for each member X of P that is a child of U do
            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 the modular decomposition of G, except that it allows a parallel node to be a child of another, and a series 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.

The original time bound given there was $O(n^2)$. The bottlenecks are finding $\mathcal{P}(G,v)$ and finding the ancestors of v in $G/\mathcal{P}(G,v)$. McConnell and Spinrad (2000) gives an $O(n(G)+m(G)\log n(G))$ bound for undirected graphs, by solving the following problems:

- **Problem 1**: Find $\mathcal{P}(G,v)$ in each recursive call, without exceeding an $O((n+m)\log n)$ time bound.
- **Problem 2**: Find the ancestors of v in $G/\mathcal{P}(G,v)$ in time linear in the size of $G' = G/\mathcal{P}(G,v)$.

Their algorithm for the first problem is trivial to generalize to digraphs. Their algorithm for the second one makes use of the so-called Γ relation on edges of an undirected graph (see Golumbic (1980)), and does not generalize to digraphs. Thus, to get an $O(n(G)+m(G)\log n(G))$ bound for digraphs, it is necessary to find an alternative approach to the second problem.

Suppose $v \in V(G)$. Define the *forcing graph* to $F(G,v)$ be the following graph on $V(G)\setminus v$ as follows: there is an arc from x to y if $\{y,v\}$ is not a module in $G|\{x,y,v\}$. It follows that if there is an arc of $F(G,v)$ from x to y, then every module M that contains y and v must also contain x, i.e that y forces x into any module M common with v. On the other hand, if $X \subset V(G)$ contains v, and there is no arc in F from $V(G)\setminus X$ to $X\setminus\{v\}$, then X is a module: any vertex in $V(G)\setminus X$ must have the same relationship to all members of X as it does to v.

Using a simple argument based on these observations, Ehrenfeucht et al. (1994) show that finding the ancestors of v in G' reduces to finding a topological sort of the strongly-connected components of $F(G',v)$ in $O(n(G')+m(G'))$ time. Because $F(G',v)$ can be much larger than G', there initially seemed to be no hope of this. However, the following lemma shows that this conclusion is not true. Applying this lemma to G', then applying Theorems 12 and 14, gives the required bound of $O(n(G')+m(G'))$.

Lemma 10.1 *Let G be a digraph and v be a vertex. Then:*

1. *If G is symmetric (undirected), then $F(G,v)$ is in the outward equivalence class of $G\setminus v$;*
2. *If G is not symmetric, then $F(G,v) = (V(G-v), E(F_1)\cup E(F_2))$, where F_1 is in the outward equivalence class of $G\setminus v$, and F_2 is in the outward equivalence class of $(G\setminus v)^T$.*

Proof: If G is undirected, there is an arc from x to y if y is a neighbor of x but v is not, or else if y is not a neighbor of x but v is. Therefore, the neighbors of x in $F(G,v)$ are the same as its neighbors in $G\setminus v$ if v is not a neighbor in G and the complement of its neighbors in $G\setminus v$ if v is a neighbor. In summary, $F(G,v)$ is obtained from $G\setminus v$ by complementing the neighbors of v.

In $F(G,v)$, there is an arc from x to y under the following circumstances:

1. y is a neighbor of x and v is not;
2. v is a neighbor of x and y is not;
3. x is a neighbor of y but not a neighbor of v;
4. x is a neighbor of v but not a neighbor of y.

Let F_1 denote the digraph on $V(G)$ whose arcs are given by conditions one and two, and let F_2 denote the one whose arcs are the transposes of those given by conditions three and four. The arcs of $F(G,v)$ are the union F_1 and F_2. If v is not a neighbor of x, then x's neighbors in F_1 are the same as its neighbors in G. If v is a neighbor of x, then its neighbors in F_1 are the complement of its neighbors in G. Since x is arbitrary, F_1 is in the outward equivalence class of $G \setminus v$. By symmetry, $(F_2)^T$ is in the inward equivalence class of $G \setminus v$. By Theorem 8, F_2 is in the outward equivalence of $(G \setminus v)^T$. □

11 Proof of Theorem 2

Let H be a digraph and let G be an undirected member of the outward equivalence class of H, and suppose that we want the modular decomposition of G. For Theorem 2, we must show how to solve Problems 1 and 2 of the previous section in order to get an $O(n(H) + m(H) \log n(H))$ time bound for modular decomposition of G.

11.1 Problem 2

Let $\mathcal{P}$ be a partition of vertices of a graph. A set of *representatives* of $\mathcal{P}$ is a set that contains exactly one element from each partition class. Recall that if $\mathcal{P}$ is a partition of the vertices of G into classes that are modules, then $G/\mathcal{P}$ is isomorphic to the subgraph of G induced by any set of representatives of G. The following generalizes this idea to members of G's outward equivalence classes.

Lemma 11.1 *Let G be a digraph, let $\mathcal{P}$ be a partition of the vertices where each partition class is a module of G, and let H be a member of G's outward equivalence class. The subgraph of H induced by any set of representatives of $\mathcal{P}$ is in the outward equivalence class of $G/\mathcal{P}$.*

Proof: This is immediate from Theorem 7 and the fact that the subgraph of G induced by the representatives is isomorphic to $G/\mathcal{P}$. □

Let P be a set of representatives of $\mathcal{P}$. We must find $F(G',v)$ for $G' = G/\mathcal{P} = G|P$. By Lemma 10.1 and the fact that G is undirected, $F(G',v)$ is in the same outward equivalence class as G', and by Lemma 11.1, G' is in the same outward equivalence class as $H|P$. Transitively, $F(G',v)$ is in the same outward equivalence class as $H|P$. By Lemma 12, it takes $O(n(H|P) + m(H|P))$ to find the strongly-connected components of $H|P$, hence to find the ancestors of v in $G/\mathcal{P}$. It is now trivial to establish that the sum of costs of this step over all recursive calls is $O(n(H) + m(H))$.

11.2 Problem 1

Let $\mathcal{P}$ be a partition of vertices of G, and let X be a union of partition classes in $\mathcal{P}$, and $u \in V(G) \setminus X$. $\texttt{Pivot}(G,u,X)$ denotes the following operation. For each $C \in \mathcal{P}$ that is contained in X, we refine $\mathcal{P}$ by splitting C into two sets, $C \cap N_G(u)$ and $C \cap \overline{N}_G(u)$.

McConnell and Spinrad (2000) give an algorithm for Problem 1 that uses calls to `Pivot` on different vertices of the graph. The technique is called *vertex partitioning*. They show that in all recursive calls of Algorithm 2, each vertex is the vertex parameter to `Pivot` on $O(\log n)$ occasions. By giving an implementation of $\texttt{Pivot}(G,u,X)$ that requires $O(|N(u)|)$ time, they obtain an $O((n(G) + m(G) \log n(G))$ time bound for solving Problem 1 in all recursive calls. Thus, to get an $O((n(H) + m(H)) \log n(H))$ bound for all of these calls to `Pivot`, it suffices to implement $\texttt{Pivot}(G,u,X)$ in time proportional to the outdegree of u in H.

Because G is undirected, the procedure is identical to the original. If $C \subseteq X$, $\texttt{Pivot}(G,u,X)$ splits C into two sets, $C \cap N_G(u)$ and $C \cap \overline{N}_G(u)$. Whether or not u is outwardly complemented, $C \cap N_H(u)$ and $C \cap \overline{N}_H(u)$ induces the same partition of C. We implement each partition class C of $\mathcal{P}$ with a doubly-linked list, and maintain a pointer on each vertex to the front of its list. Extracting neighbors of u from each class C and placing them in a twin class C' can be performed in a single traversal of the adjacency list for u. When a new neighbor is examined, it is required only to check which class C contains it in order to determine which twin class C' to put it in. This takes time proportional to the outdegree of u in H, as required.

12 Proof of Theorem 3

Let the *prime quotients* in the modular decomposition tree be quotients of the form $(G|X)/\mathcal{P}$, where X is a prime node and $\mathcal{P}$ is its children.

Let H be a digraph, and let G be a comparability graph in the outward equivalence class of H. Each prime quotient $(G|X)/\mathcal{P}$ is isomorphic to $G|X'$, where X' consists of one representative of each member of $\mathcal{P}$. By Lemma 11.1, $G|X'$ is in the outward equivalence class of $H|X'$. Let us call $H|X'$ a *representation* of the prime quotient. The modular decomposition on G is a tree whose leaves are also the vertices of H. Applying the off-line least-common ancestors algorithm of Harel and Tarjan (1984), to the edges of H on the modular decomposition tree of G, it is straightforward to find a representation of each prime quotient. Details are given in McConnell and Spinrad (1999).

There is at most one vertex of a prime quotient for each node of the decomposition tree, the total number of such nodes is $O(n(G)) = O(n(H))$. No edge of H appears in more than one of the representations of prime quotients, so the total number of edges in all representations is $O(m(H))$.

Since the prime quotients are in the outward equivalence classes of their representations, we may perform a pivot on a vertex of the prime quotient in time proportional to the degree of the corresponding vertex u in the representation. McConnell and Spinrad (1994, 1999) show that if the modular decomposition of a graph is available, finding a linear extension of a transitive orientation reduces to performing $O(\log n)$ $\texttt{Pivot}$ operations on each node a prime quotient. The total time to find the linear extension of the transitive orientation of G is therefore $O((n(H)+m(H))\log n(H))$.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

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

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

E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and pratical algorithms for sequential modular decomposition. *Journal of Algorithms*, 41(2):360–387, November 2001.

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

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

A. Ehrenfeucht and G. Rozenberg. Theory of 2-structures, part 2: Representations through labeled tree families. *Theoretical Computer Science*, 70:305–342, 1990b.

A. Ehrenfeucht and G. Rozenberg. Angular 2-structures. *Theoretical Computer Science*, 92:227–248, 1992.

A. Ehrenfeucht and G. Rozenberg. Dynamic labeled 2-structures. *Math. Struct. in Comp. Science*, 4: 433–455, 1994.

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

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

R. McConnell. Complement-equivalence classes on graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*. Springer, 1997.

R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5:536–545, 1994.

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

R. M. McConnell and J. P. Spinrad. Ordered vertex partitioning. *Discrete Mathematics and Theoretical Computer Science*, 4:45–60, 2000. URL `http://dmtcs.loria.fr/volumes/abstracts/dm040104.abs.html`.

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

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

J. P. Spinrad. P_4 trees and substitution decomposition. *Discrete Applied Mathematics*, 39:263–291, 1992.