$O(m \log n)$ Split Decomposition of Strongly Connected Graphs

Benson L. Joeris, Scott Lundberg, and Ross M. McConnell

Colorado State University

Abstract

In the early 1980's, Cunningham described a unique decomposition of a strongly-connected graph. A linear time bound for finding it in the special case of an undirected graph has been given previously, but up until now, the best bound known for the general case has been $O(n^3)$. We give an $O(m \log n)$ bound.

Key words: split decomposition, graph decomposition, join decomposition
PACS:

1 Introduction

Split decomposition is a unique decomposition of arbitrary strongly-connected digraphs described by Cunningham in 1982 [1]. Because connected undirected graphs are a special case of strongly-connected digraphs, a special case of the decomposition applies to arbitrary undirected graphs. Also known as join decomposition, it is useful in many areas ranging from recognition of certain graph classes [2] to optimizations of NP-hard problems [3]. It is a proper generalization of the well known modular decomposition, also called substitution decomposition [4,5].

As a convention we denote the number of vertices of a graph as $n = |V|$ and the number of edges $m = |E|$. If X is a nonempty subset of V, by $G[X]$ we denote the subgraph of G induced by X. If $x \in V$, by $deg(x)$, we denote the degree of x.

Cunningham gave the first algorithm for computing the decomposition on arbitrary strongly-connected digraphs, which runs in $O(n^4)$ time [1]. Bouchet improved this to $O(n^3)$ [6]. This solves an interesting special case, which is determining whether a graph is *prime* with respect to the split decomposition, which means that it can be decomposed only in trivial ways (explained further

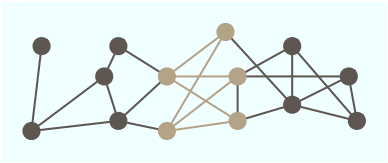

Fig. 1. The edges crossing a split induce a complete bipartite graph.

below). Spinrad gave an $O(n^2)$ algorithm for determining whether an arbitrary directed graph is prime [7], but not for finding the decomposition tree if it has a nontrivial decomposition. Since then, much work has focused on the special on the special case of undirected graphs. This work includes an $O(nm)$ algorithm by Gabor, Supowit, and Hsu [2], an $O(n^2)$ algorithm by Ma and Spinrad [8], and, finally, a linear-time ($O(n + m)$) algorithm by Dahlhaus [9].

This leaves open the possibility of improving on the previous best bound of $O(n^3)$ for finding the decomposition of strongly-connected digraphs. In this paper, we give an $O(m \log n)$ bound.

Our approach borrows generously from techniques developed by Ma and Spinrad for their $O(n^2)$ algorithm for undirected graphs [8]. In particular, we make use of a technique called *graph partitioning* or *partition refinement*.

As an historical note, it is worth noting that techniques for implementing partition refinement efficiently, and many well-known applications of it, were first described by Spinrad [10], [11], [12]. We get the $O(m \log n)$ bound by modifying a type of clever charging argument for graph partitioning, also due to Spinrad, which was circulated widely in the mid-1980's in a working manuscript about modular decomposition written by him [13]. Many of these techniques have been surveyed since in papers that mistakenly attribute their origins to subsequent papers that, like ours, borrow heavily from Spinrad's early work on the subject.

In Section 2 we give a brief overview of split decomposition. In Section 3 we note some interesting applications of split decomposition to other problems in graph theory. In Section 4 we give a special case of our algorithm for undirected graphs. In Section 5 we show how to generalize this algorithm to strongly-connected digraphs.

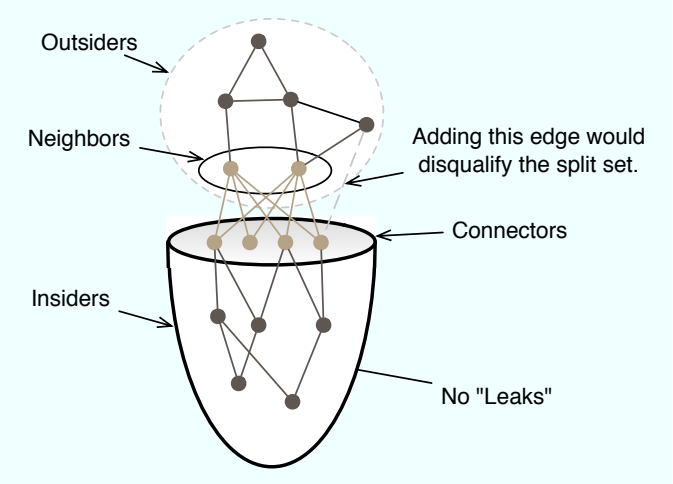

Fig. 2. The structure of a split set.

2 Split Sets and Split Decomposition Trees

2.1 Split Sets

For split decomposition, any connected component of a graph can be dealt with individually, so we assume that all graphs are connected. Let G be a undirected connected graph with vertices V and edges E. A *split* in G is a partition of V into two sets, called *split sets*, where the graph induced by the edges between the split sets forms a complete bipartite graph. The example shown in Figure 1 highlights this subgraph. Since every split forms two split sets, split sets always come in pairs that are complements of each other in V. It is easily seen that any one-element subset of V is a split set, as is its complement; these are the *trivial split sets.*

Figure 2 illustrates some elements of interest in a split set. The *connectors* are those vertices in the split set that are part of the complete bipartite graph forming the split. The *insiders* are the remaining vertices of the split set, while the *outsiders* are those vertices not contained in the split set, and the *neighbors* are those vertices adjacent to the split set. It is important to note that all the connectors are adjacent to all the neighbors, and there are no edges from insiders to outsiders; there are no "leaks."

Because all the connectors share the same set of outsiders, we can decompose the graph by replacing the outsiders with a marker vertex. Figure 3 illustrates this by showing how the vertices are mapped from the original graph to the new graph, which we will call a *quotient graph*. This mapping is a homomorphism with respect to the split sets. In other words there is a many-to-one mapping from the split sets in the original graph to those in the quotient graph. Any

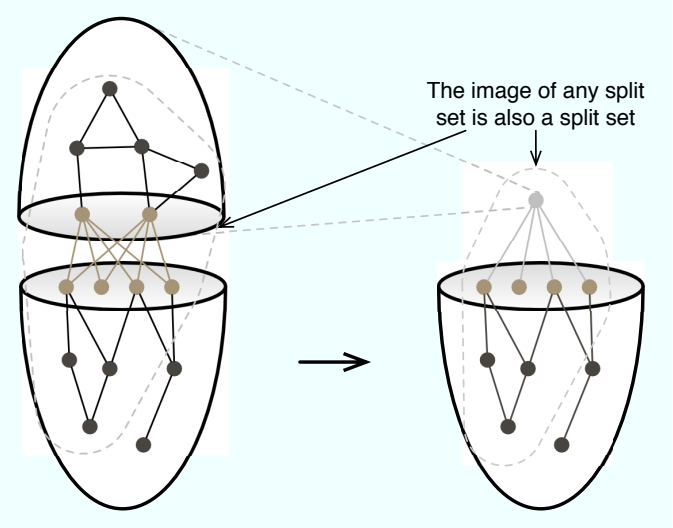

Fig. 3. Creating a quotient graph.

split set in the original graph maps to a split set in the quotient graph, and vice versa as stated by the *homomorphic rule* (see Figure 3):

Definition 1 *Let G be a graph and G' be a quotient graph derived from G. The image of a set of vertices in G is the set of vertices that it maps to in G'. The inverse image of a set of vertices in G' is the set of vertices in G that map to elements of that set.*

Lemma 1 *(***The "homomorphic rule"***) The image of any split set in a quotient G' of G is also a split set in G', and the inverse image of any split set in G' is also a split set in G.*

The quotients for two complementary split sets produce two quotient graphs, one from each split set. These new quotient graphs fully represent the original graph: the process of splitting the graph into two subgraphs and adding a marker vertex to each can easily be reversed through the *composition* of the two quotients. Because of the homomorphic rule, it is possible to recursively find splits in each of them that must all correspond to splits in the original graph.

Split decomposition uses this process of recursively building quotient graphs to find and represent all split sets in a graph G. Note that there may be an exponential number of split sets; consider the complete graph, where every non-empty proper subset of the vertices is a split set. Representing the split sets is accomplished by creating a *split decomposition tree*, which is an unrooted tree representing all the split sets of G, in space linearly proportional to G.

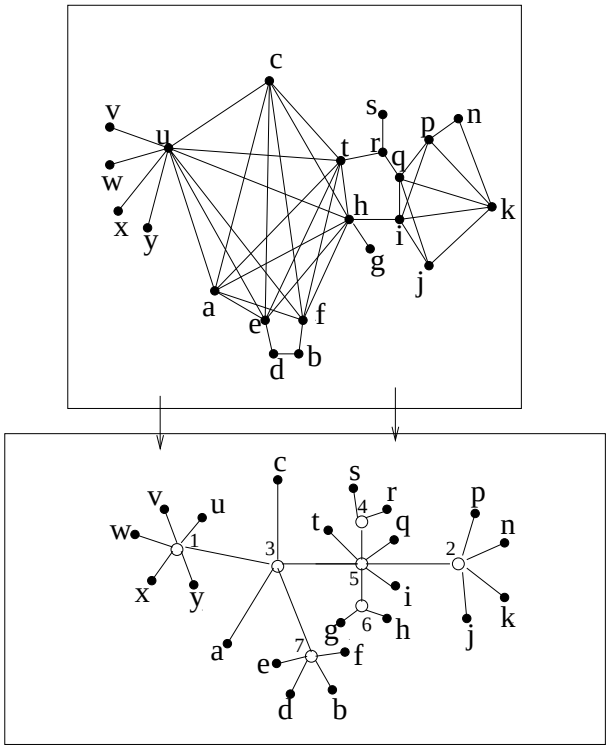

Fig. 4. The split decomposition of an undirected graph.

2.2 The Split Decomposition Tree

Cunningham [1] shows that there is a unique unrooted tree that implicitly represents all split sets of an undirected graph G. The leaves of the tree are the vertices of G. If u is an internal node, let a *neighbor set of* u be the set of leaves reachable through a neighbor v of u. That is, they are the set of vertices of G that are in v's component of the tree if the tree edge uv is removed. Each internal node can be labeled as *prime* or *degenerate*, such that a set is a split set if and only if it is one of the following:

- A neighbor set of a prime node;
- The complement of a neighbor set of a prime node;
- A union of at least one and fewer than all neighbor sets of a degenerate node.

For instance, Figure 4 gives the split decomposition of a graph. Figure 5 depicts the neighbor sets of internal node 3 of Figure 4, with their connectors circled. Each of these neighbor sets is a split set. Node 3 is a degenerate node, since all unions of at least one and fewer than all of these sets is a split set. For instance, the union of $\{u, v, w, x, y\}$, $\{c\}$, and $\{b, d, e, f\}$ is a split set with connectors $\{u, c, e, f\}$.

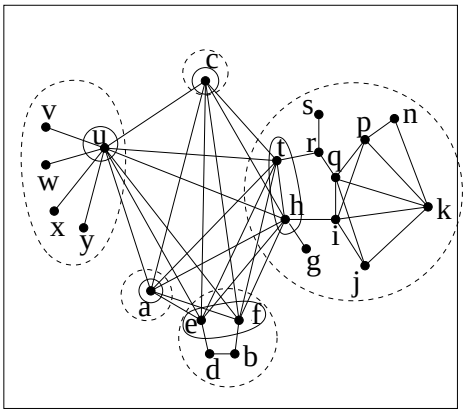

Fig. 5. The neighbor sets of node 3 of Figure 4

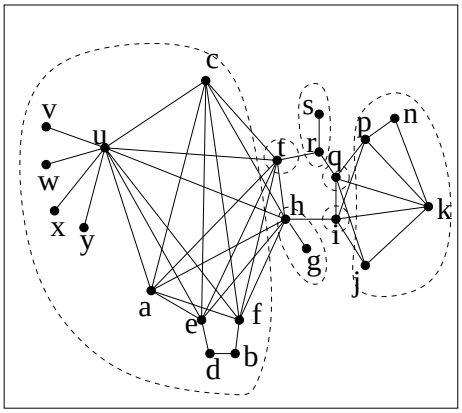

Fig. 6. The neighbor sets of internal node 5 of Figure 4

Figure 6 depicts the neighbor sets of node 5. This is a prime node, because each of its neighbor sets is a split set, but the only unions of its neighbor sets that are split sets consist either of one neighbor set or the union of all but one neighbor set.

The *associated quotient* at an internal node u is the quotient G' of G obtained by replacing each of u's neighbor sets with a marker vertex. For simplicity, if V' is the neighbor set reachable through a neighbor v, we can consider v to *be* the marker for V'.

For instance, the associated quotient at node 3 of Figure 4 is a complete graph on vertices $\{1, a, 7, 5, c\}$. The associated quotient at node 5 is a graph on vertex set $\{3, 6, i, 2, q, 4, t\}$ and edge set $\{\{6, i\}, \{i, q\}, \{q, 4\}, \{4, t\}, \{t, 6\}, \{3, t\}, \{3, 6\}, \{2, q\}, \{2, i\}\}$.

A graph is *prime* if its only split sets are the one-element subsets and their complements, and *degenerate* if all nonempty proper subsets of its vertices are split sets. (Cunningham calls the degenerate graphs *brittle*; we borrow our terminology from the literature on partitive set families, of which the split sets are

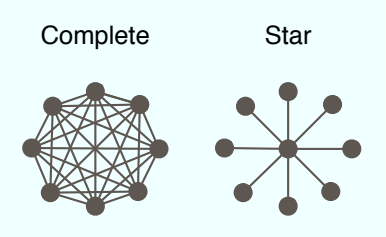

Fig. 7. The degenerate quotients in the split decomposition of a connected undirected graph are stars or complete graphs.

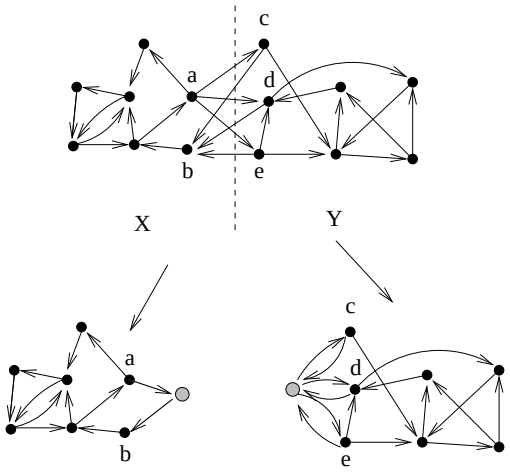

Fig. 8. Two quotients formed by a split in a directed graph.

an example.) By the homomorphic rule, the quotient associated with a prime node is a prime graph and the quotient associated with a degenerate node is a degenerate graph. The only degenerate quotients associated with nodes of the decomposition of a connected undirected graph are stars and complete graphs (see Figure 7). In Figure 4, node 1 is an example of a degenerate node with a star quotient.

It is easy to see that the process of decomposing G into the quotients at the internal nodes of the decomposition tree is invertible; G can be reconstructed by composition using the tree and its associated quotients.

2.3 Split Sets in Digraphs

Similar to the assumption that the undirected graphs were connected, in the directed case Cunningham assumed that a digraph we are decomposing, G, is strongly connected. This ensures that there is a unique decomposition tree describing the splits in G, as noted in [1].

A split in a directed graph is a partition of the graph into two parts, X and Y, where the edges directed from X to Y forms a cartesian product $X' \times Y'$,

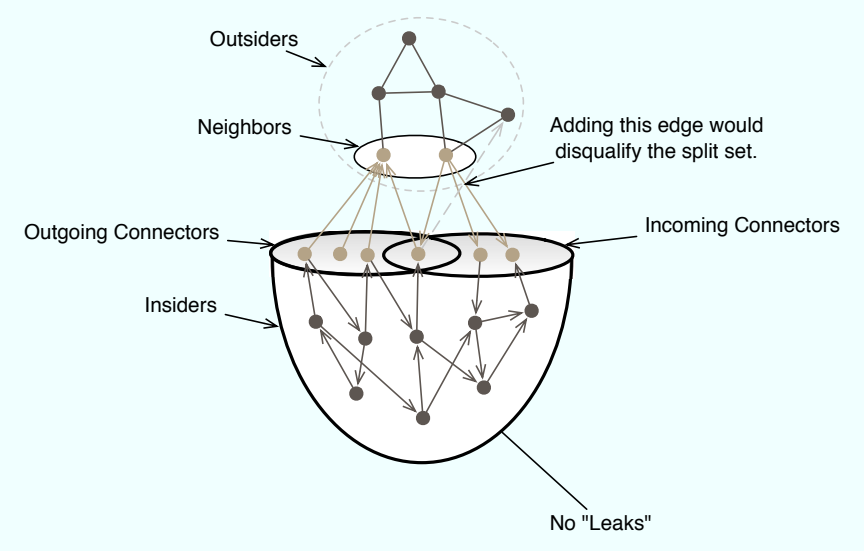

Fig. 9. Parts of interest in a directed split set.

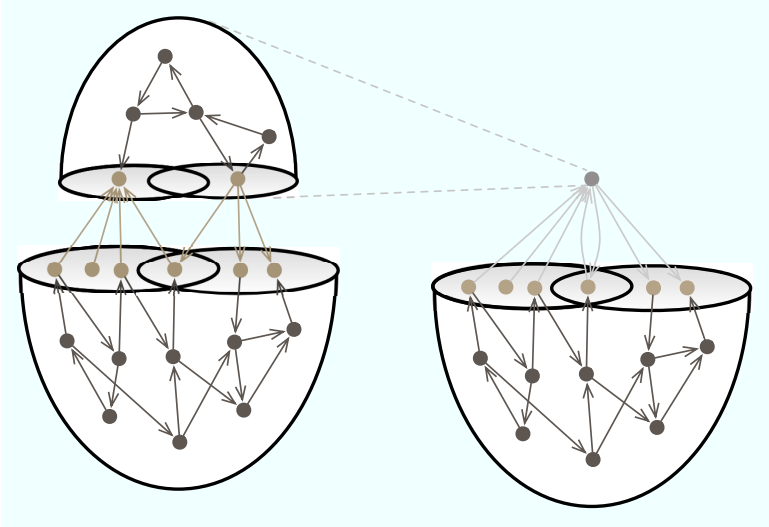

Fig. 10. Finding the quotient of a directed split.

where $X' \subseteq X$ and $Y' \subseteq Y$, and the edges directed from Y to X form a cartesian product $Y'' \times X''$, where $Y'' \subseteq Y$ and $X'' \subseteq X$. For example, in Figure 8, $X' = \{a\}$, $Y' = \{c, d, e\}$, $X'' = \{b\}$, and $Y'' = \{c, d, e\}$. It is not necessary that $X' = X''$ or $Y' = Y''$ or that they be disjoint. A quotient is formed for X by making a marker with edges from X' and edges to X'', and similarly for Y.

The primary difference is that in the undirected case, there is a single set of connectors, while in the directed case, there are two sets of connectors, the *incoming connectors*, which are neighbors of vertices outside the split set, and the *outgoing connectors*, which have neighbors outside the split set. This is illustrated in Figure 9.

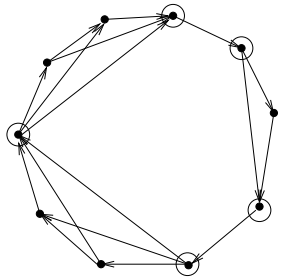

Fig. 11. The quotient associated with a circular node is a cycle of transitive tournaments. The sink of each transitive tournament is the source of the next (circled vertices), and a nonempty set of vertices is a split set if and only if the vertices are a proper subset of the vertices that is consecutive in the cyclic order.

The quotient graphs of a split set in a directed graph is once again obtained by replacing one side of the split with a marker vertex, as in Figure 10. This process preserves the split sets of the original graph in the same way as the undirected case, thus giving the following generalization of the homomorphic rule to directed graphs:

Lemma 2 *The homomorphic rule applies to quotients on directed graphs.*

2.4 The Split Decomposition of a Strongly-Connected Graph

As in the undirected case, the split decomposition tree is an unrooted tree that has the vertices of G as its leaves. The neighbor sets are defined as before, each neighbor set is a split set, and the quotient associated with an internal node is once again the graph obtained by replacing each neighbor set with a marker. Since undirected graphs are a special case, this tree may still have prime and degenerate nodes, where the associated quotient at a degenerate node can be a complete graph or a star. However, there can be an additional type of node, a *circular node*. At a circular node, the neighbors are circularly ordered, and a union of neighbor sets is a split set if and only if it is a nonempty union of at least one and fewer than all neighbor sets *that are consecutive in the circular ordering.*

By the homomorphic rule, it must be the case that the quotient associated with a circular node is a graph with a cyclic ordering on its vertices, such that a set of vertices is a split set if and only if it is a nonempty proper subset of the vertices that is consecutive in the circular ordering. Let us call such a graph a *circular graph*.

A *transitive tournament* is a directed acyclic graph with an edge between every pair of vertices. A transitive tournament has a unique source and a unique sink. If it has more than one vertex, the source and the sink are distinct. A *cycle of*

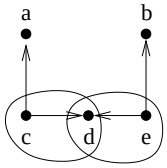

Fig. 12. A digraph that is not strongly-connected, and its maximal split sets that do not contain a or b

transitive tournaments is a cyclic ordering of transitive tournaments, where the sink of each is identified with the source of the next. Figure 11 depicts a cycle of transitive tournaments. Cunningham showed that a graph with at least four vertices is a circular graph if and only if it is a cycle of transitive tournaments. By the homomorphic rule, therefore, the quotient associated with a circular node of degree at least four must be a cycle of transitive tournaments.

So far, our distinction between prime, degenerate, and circular nodes is ambiguous for nodes of degree three. We therefore consider a node of degree three to be circular if its associated quotient is a cycle of transitive tournaments.

The definition of a split set generalizes to arbitrary directed graphs, but there is no longer a unique decomposition tree for representing them. Figure 12 illustrates some of the difficulties that set in when a graph is not strongly connected. We show below that the existence of the unique decomposition tree he describes implies that the maximal split sets that don't contain two vertices a and b are a partition of $V \setminus \{a, b\}$. This is also critical to the correctness of our algorithm. In Figure 12, the maximal split sets $\{c, d\}$ and $\{d, e\}$ that don't contain a and b fail to be a partition of $V \setminus \{a, b\}$; they properly overlap. The reason this can't happen in a strongly-connected graph is that when X and Y are two split sets that properly overlap and whose union is a proper subset of V, then $X \cup Y$, $X \cap Y$, $X \setminus Y$ and $Y \setminus X$ are also split sets. This would invalidate $\{c, d\}$ and $\{d, e\}$ as maximal split sets excluding a and b, since their union, $\{a, b, c\}$ would be a larger split set excluding a and b.

Cunningham uses these properties of properly overlapping split sets to derive the unique decomposition tree. Set families with these properties come up in other contexts, and give a tree representation identical to Cunningham's for representing the members of the family; see, for example, [14]. To prove that properly overlapping split sets have these properties, Cunningham uses the fact that in a strongly-connected digraph, for every nonempty proper subset S of vertices, there is an edge directed from S to $V \setminus S$ and an edge directed from $V \setminus S$ to S, and this is not true in arbitrary digraphs.

3 Examples of Applications

A well known application of split decomposition in graph theory is the recognition and isomorphism testing of circle graphs [15,2]. *Circle graphs* are those graphs whose vertices are each a chord of the circle, and whose edges are the pairs of chords that intersect. Equivalently, given a set of arcs on the circle, the pairs of arcs that properly overlap is a circle graph, as two arcs properly overlap if and only if the chord joining one arc's endpoints intersects the chord joining the other's endpoints. Any circle graph that is prime with respect to split decomposition has a unique chord representation, that is, the order of endpoints of chords about the circle is uniquely constrained up to reversal of their order. Conversely, any split represents two sets of chords, one of whose endpoint placements is not uniquely constrained by the other's. The split decomposition yields a gadget, analogous to the so-called PQ tree for interval graphs, that represents all possible arc endpoint placements.

Parity graphs can also be recognized using split decomposition [16,9], and it has been shown recently that split decomposition can be used when computing the coloring of a graph [3]. A graph is perfect if and only if every prime graph in the split decomposition is perfect [17].

Distance hereditary graphs are those undirected graphs that can be built by adding pendant vertices or duplicating existing vertices [18]. An alternative characterization is that all induced paths between any two vertices have the same length. A third characterization is in terms of forbidden subgraphs: they are the class of graphs that have no *gem*, *house*, *domino*, or *hole* as an induced subgraphs [19]. A fourth is that they are precisely the class of undirected graphs whose split decompositions have no prime nodes. (Their role in split decomposition is analogous to that of the so-called cographs in modular decomposition.)

Split decomposition has the potential to speed up hard computations by running the process on each component of the decomposed graph, and then combining the results to create the final answer. Finding the maximum weighted independent set is one example of an NP-hard problem on undirected graphs that can be optimized on graphs with non-trivial split decompositions. For a brief description of how to accomplish this see [1]. In the directed case, the example mentioned by Cunningham is the minimum-weight dominating set for directed graphs. The running time using this strategy is exponential in the maximum degree of a prime node, rather than exponential in the number of vertices, providing a heuristic that only fails to be useful if the graph is prime.

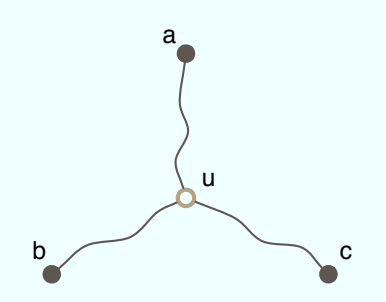

Fig. 13. The paths connecting a, b, and c cross at a node u.

4 Finding the Decomposition Tree of an Undirected Graph

To simplify the discussion, let us assume for the moment that all internal nodes of the decomposition tree are prime.

4.1 The Strategy

We have two methods at our disposal that are described below: $S(a, b, G)$ which returns the partition of V consisting of $\{a\}$, $\{b\}$, and the maximal split sets in G that don't contain a or b, and $L(a, b, c, G)$ which finds the maximal split set in G that doesn't contain a or b but does contain c. We show below that $S(a, b, G)$ is, in fact, a partition of V, and, because $L(a, b, c, G)$ is the member of $S(a, b, G)$ that contains c, it is unique. We could get $L(a, b, c, G)$ by running $S(a, b, G)$ and removing all returned sets except the one that contains c, but we use a separate procedure for efficiency reasons. Using these methods we can show how to construct the entire split decomposition tree of G.

The first step is to pick three vertices a, b, c of G. Because they are vertices of G, they must be leaf nodes of G's split decomposition tree. We don't yet know what the G's decomposition tree looks like, but we do know that the paths connecting a, b, and c in the tree must intersect at single internal node, which we will call u. Figure 13 shows u in relation to the nodes we have chosen in the final, still unknown, decomposition tree. Let A, B, and C denote the neighbor sets of u that contain a, b, and c, respectively.

Lemma 3 *(See Figure 14) If all nodes of the split decomposition are prime, $S(a, b, G)$ returns all the neighbor sets of nodes on the path from a to b that do not contain a or b.*

Proof Let v be an internal node on the path from a to b. One of its neighbor sets contains a and another contains b, and these are not members of $S(a, b, G)$ by definition. Let X be a third neighbor set. X does not contain a or b. We now show that it is a maximal split set that doesn't contain a or b. Since v

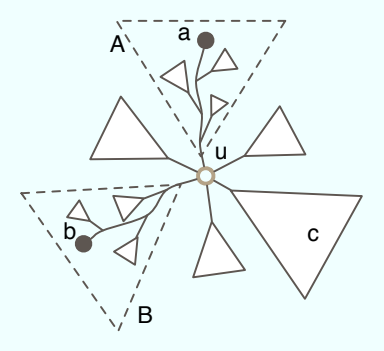

Fig. 14. What we get from calling $S(a, b, G)$.

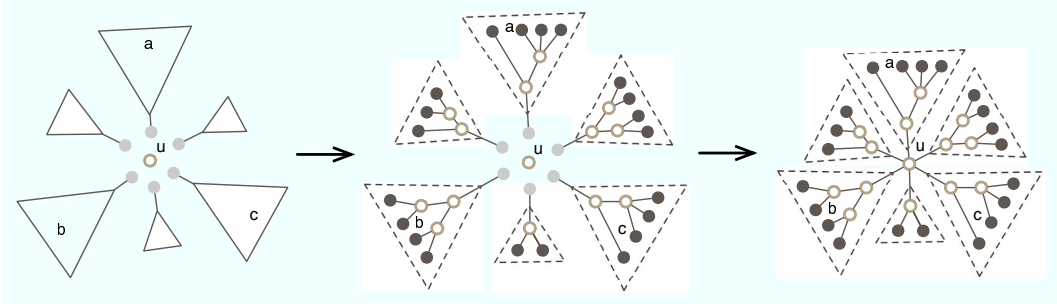

Fig. 15. Completing the decomposition tree through recursion.

is prime, the minimal split sets that properly contain X are unions of all but one neighbor set of v, hence each of them contains at least one of a and b, which reside in different neighbor sets. □

Equivalently, $S(a, b, G)$ finds all neighbor sets of u that correspond to neighbors that are not on the path from a to b, but are adjacent to nodes on the path. Therefore these sets are a partition of $V \setminus \{a, b\}$, hence $S(a, b, G)$ is a partition of V. By Lemma 3, A is a union of $\{a\}$ and neighbor sets of those nodes on the path from a to u in the decomposition tree that correspond to neighbors that do not lie on this path. Similarly, B is a union of $\{b\}$ and neighbor sets adjacent to the path from b to u.

From $S(a, b, G)$, we cannot tell which members of $S(a, b, G)$ are neighbor sets of u, which are subsets of A and which are subsets of B. To find which are subsets of A and B, it suffices to call $L(b, c, a, G)$ and $L(a, c, b, G)$, respectively. The remaining sets are neighbor sets of u, and, since we now know A and B, we now know all neighbor sets of u.

Once we have the neighbor sets of u we can add marker vertices to each set and then recurse on these quotient graphs (see Figure 15). Adding a marker y to neighbor set Y and recursing on it gives the decomposition tree of this quotient, and by the homomorphic rule, this tree is isomorphic the subtree of the decomposition of G given by paths between u and members of Y.

Repeating this operation on all neighbor sets Y and identifying the marker vertex therefore gives the full decomposition tree of G.

4.2 Introducing degenerate nodes

Let us now relax the assumption that u is prime. Because any nonempty proper subset of a degenerate node's neighbor sets is also a split set, and because $S(a, b, G)$ finds maximal split sets that don't contain a or b, if v is a degenerate node on the path from a to b in the tree, the union of all neighbor sets of v, other than the ones that contain a and b, are a single member of $S(a, b, G)$. Lemma 3 must be modified:

Lemma 4 *$S(a, b, G)$ returns $\{a\}$, $\{b\}$, and a partition of $V \setminus \{a, b\}$ where each partition class consists of the neighbor sets of a prime node on the path from a to b that do not contain a or b, or the union of all neighbor sets of a degenerate node on the path from a to b that do not contain a or b.*

Proof The characterization of neighbor sets of prime nodes is given by Lemma 3. Let v be an internal degenerate node on the path from a to b. One of its neighbor sets contains a and another contains b, and these are not members of $S(a, b, G)$ by definition. Let X be the union of all other neighbor sets of v. Because v is degenerate, X is a split set. Because there is no larger union of neighbor sets that excludes a and b, it is a set returned by $S(a, b, G)$. □

If u is degenerate, let X be the union of neighbor sets of u other than A and B returned by $S(a, b, G)$. By induction, we may assume that a recursive call on the quotient consisting of X and a marker produces the split decomposition of this graph. Let w be the marker. Making w be a neighbor of u, and doing the same for the results of recursive calls on A and B makes u a tree node of degree three. If X is the union of more than one neighbor set of u in the actual decomposition of G, then this is incorrect, because u should have one neighbor for each of these, instead of just w. We resolve this in a way that is described by Cunningham [1]. The quotient associated with u identifies one of its markers with w and the quotient at w identifies one of its markers with u. This situation is detected by checking whether the composition of the quotients associated with u and w give rise to a larger degenerate quotient (a complete graph or a star). If so, then we contract tree edge uw, and replace the quotient at the resulting node with this composition. Since A and B are each a single neighbor set of u and the recursive call on X produces the correct tree for X's quotient, no other contraction is required to produce the correct tree for G.

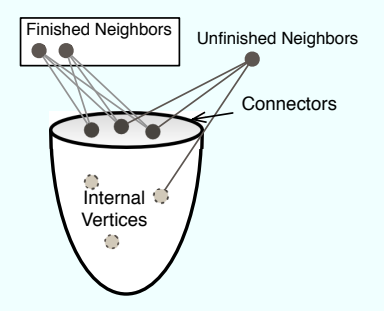

Fig. 16. Elements of interest in a partition class.

4.3 Implementation of $S(a, b, G)$ and $L(a, b, c, G)$

$S(a, b, G)$ works by starting with an initial partition $\{\{a\}, \{b\}, V \setminus \{a, b\}\}$ of V, and successively refining the partition classes until they give $\{a\}$, $\{b\}$, and $S(a, b, G)$. We maintain the following invariant:

- **Splitting invariant:** A split set that started out as a subset of a single partition class remains a subset of a single partition class after each refinement.

When no further refinement is possible without violating this invariant, the partition is $\{\{a\}, \{b\}\} \cup S(a, b, G)$. The basic operation for refining partition classes is selection of a *pivot vertex* p and using its adjacencies to try to split partition classes without violating the splitting invariant. A key point is that our procedure does not allow p to split partition class that currently contains it.

A partition class has several components of interest, listed below, and shown in Figure 16. Let S be a partition class of size greater than 1. For any vertex p in another partition class, either $p \in \{a, b\}$ or there was a first moment when a partition class containing p and the members of S was split so that the members of S were in one partition class, and p was in another. Let us call this moment their *separation point*. The *finished outsiders* of S are those vertices on which a pivot has been performed since their separation from S, and the *finished neighbors* are those finished outsiders that are neighbors of the connectors. We also maintain the following invariant during refinement of the partition:

- **Pivot invariant:** If P is the set of finished outsiders of S, S is a split set in $G[P \cup S]$.

If all elements of $V \setminus S$ are the finished outsiders of S, then the pivot invariant implies that S is a split set of G. Given the initial partition and the splitting invariant, it must be a member of $S(a, b, G)$.

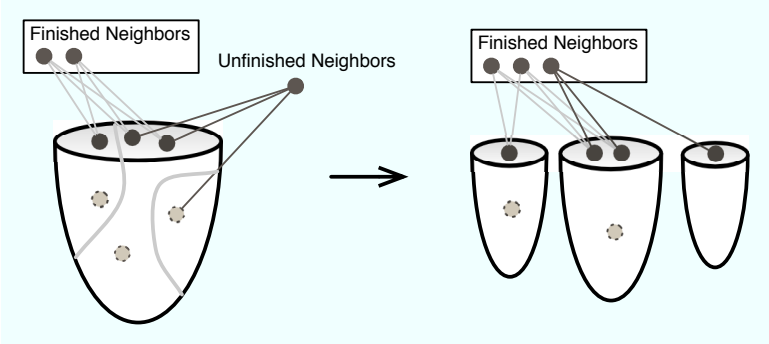

Fig. 17. Pivoting on an neighbor with respect to one partition class.

4.3.1 Implementation of a pivot

Pivoting on a vertex moves it from the unfinished to the finished outsiders for all partition classes except the one that contains the pivot. In order to place a vertex into the finished neighbors group for a partition class we must ensure that it does not violate the pivot invariant, which may require refining the partition.

As shown in Figure 17 there are up to three different subsets of the connectors formed by a pivot in a partition class S: those that are adjacent to previously finished pivots only, those that are adjacent both to previously finished pivots and to the pivot, and those that are adjacent only to the pivot. After the pivot has been completed, these three sets will have different sets of finished neighbors outside the partition class. Therefore, they cannot remain in a common split set that is a subset of S, since that would violate the pivot invariant. Let us call each of these sets the *founding set* of a new partition class that is a subset of S. We must split S into one subclass for each of these founding sets.

We must now decide which other vertices from S go into which subclass. For this, we set up a process we will call a "land grab," where vertices that are known to be in one of the subclasses of S plants a flag on its unclaimed neighbors from S, thereby claiming them for their own subclass. Initially, the vertices in each founding set have an implicit flag identifying the subclass they are founding. To find which insiders go with which split set, we select a vertex that has a flag and hasn't attempted to plant any flag, and having it plant a flag on all of its neighbors in S that don't already have a flag planted on them. The process continues until all vertices of S have been claimed for one of the subclasses.

Lemma 5 *The splitting invariant is maintained during a pivot.*

Proof Identical flags will be planted in all connectors of a split set S' that starts out as a subset of S; they are either planted by the pivot if they are connectors of S, or else by the first neighbor of the connectors that attempts to

plant flags. This is because all connectors of S' have identical neighborhoods outside of S'. Once this happens, the flags on the connectors block access to all competing land grabs, as S' has no leaks. At this point, all of S' is destined to end up with identical flags, hence in the same partition class. □

Lemma 6 *The pivot invariant is maintained during a pivot.*

Proof Let P be the finished outsiders before the pivot that splits class S. The connectors of the class all have P as their neighbor set outside S before the pivot. The connectors of the new subclasses are the connectors of S that have only P as their finished neighbor set outside of S, $P \cup \{p\}$ as their finished neighbor set outside of S, or $\{p\}$ as their neighbor finished neighbor set outside of S. Each of the three subclasses satisfies the pivot invariant. □

$L(a, b, c, G)$ performs a subset of the operations of $S(a, b, G)$. Since we are only interested in the class S that contains c, we keep only this class instead of a partition of $V \setminus \{a, b\}$. Whenever this class splits, we discard the subclasses that do not contain c.

4.4 Bounding the number and density of marker elements

Definition 2 *A* **marker vertex** *is one that was inserted as a marker at any point during execution of the algorithm. A* **non-marker vertex** *is therefore a vertex that existed in the adjacency-list representation of the original graph. The* **vertex size measure** *of a partition class is the number of non-marker vertices in the class. The* **edge size measure** *is the number of non-marker vertices plus the sum of sizes of their adjacency lists.*

Lemma 7 *The degree of a non-marker vertex in a recursive call is never greater than its degree in the main call.*

Proof If the vertex is a connector of the split set passed to a recursive call, it gains a marker neighbor, but this marker replaces one or more neighbors in a separate split set that got passed to a separate recursive call. It gained at most one (marker) neighbor and lost at least one neighbor. If it is an internal node of the split set, then all of its neighbors were also neighbors in the parent call. In each case, its degree in a recursive call is bounded by its degree in the parent call. □

Note that for the correctness, there is no restriction on the choice of a and b. Therefore, we adopt the following rule:

- Suppose in a recursive call on $G' = (V', E')$ (which might be G), we make a call to $S(a, b, G')$. Inside the recursive call G_A on the split set A containing

a, we constrain the choice of parameters to $S()$ by calling $S(a, m, G_A)$, where m is the new marker representing $V' \setminus A$. Similarly, inside the recursive call G_B on the split set B containing b, we constrain the choice of parameters to $S()$ by calling $S(b, m', G_B)$, where m' is the new marker representing $V' \setminus B$. The choice of parameters to $S()$ in all other recursive calls is unconstrained.

Lemma 8 *There are at most two markers in any recursive call G', and if there are any, they are used as a or b in the call to $S(a, b, G')$.*

Proof The proof is by induction on the depth of the call. Suppose that it is true for the call on G'. Then only a and b can be markers. a and b are passed to different calls, one on G_A, with added marker m, and one on G_B, with added marker m', so in G_A, only a and m can be markers and they are used as parameters for the call on $S()$ in G_A, and in G_B, only b and m' can be markers, and they are used as parameters for the call on $S()$ in G_B. □

Corollary 1 *In a recursive call or partition class with at least three vertices that occurs during a call to $S(a, b, G)$ or $L(a, b, c, G)$, let k be the sum of degrees of the vertices and let k' be the edge size measure for the call. Then $k = O(k')$.*

Proof By Lemma 8, at most one edge fails to be incident to a non-marker, and there is at least one non-marker. □

4.5 Recurring implementation tricks

4.5.1 The ripe rule

Execution of $S()$ involves incremental refinement a partition of the vertices of G. A recurring trick in what follows is to charge vertices and their adjacency-list elements for time spent by the algorithm. A key element of the charging scheme is to charge at intervals that ensure that they are not charged too often. In particular, we charge an element for $O(1)$ time only if one of the size measures of the class that currently contains it has half the same size measure of the class that contained it the last time the was charged. Whether we use the vertex size measure or the edge size measure depends on the circumstances. A key requirement of the size measure of the class containing an element never increases, and can decrease if the class containing the element is split into two classes, in which case the element finds itself in a smaller class.

Since the size measure of a class containing an element can halve $O(\log m) = O(\log n^2) = O(\log n)$ times, and there are $O(m)$ vertices and adjacency-list elements, this ensures that the total charges are $O(m \log n)$. We obtain the $O(m \log n)$ bound by charging in this way all costs that are difficult to bound by a simpler method.

The reason for using the non-marker vertices as a measure of the size of a class is that it satisfies the requirement that the size measure of the class containing a vertex never grows. A measure based on all vertices in a class fails this criterion, since marker vertices are added to classes when they become recursive calls, giving rise to moments when the class grows.

To make the accounting easier, whenever we charge an element in a class, we charge all elements to the class. This maintains the invariant that all elements of a class become eligible to be charged at the same time.

Definition 3 *A partition class is* **ripe** *if the last time its members were charged, they were in a partition class that had at least twice the size measure of their current class. A vertex and its adjacency-list elements are* **ripe** *if it is a member of a ripe partition class.*

The rule can be summarized as charging only elements that are in ripe classes. The measure of the size of a partition class depends on the circumstances. Let us refer to this as the **ripe rule**.

Proposition 1 *Whenever a partition class is split into two partition classes, at least one of the subclasses becomes ripe.*

When a class S splits into two smaller classes, S_1 and S_2, at least one of them has at most half the size measure of S. This is the motivation for including only non-marker vertices in the size measures. If marker vertices were included, the proposition could fail if S_1 and S_2 then have marker vertices added to them and both could have more than half the size measure of S.

Even though the non-marker vertices determine whether a class becomes ripe, it allows us to charge marker vertices and their adjacency lists in a ripe class; this presents no problem for the argument, as we show below that the number of marker elements and their adjacency-list elements that are added in all recursive calls is $O(m)$, and the ripe rule ensures that each of them is charged $O(\log n)$ times.

4.5.2 The complement trick

When we wish to perform some operation on a class S that is not ripe, for instance, removing adjacency-list elements from it that point to vertices in other partition classes, we often find that S is the only class that is not ripe, because of Proposition 1. Since all elements in the complement of S are ripe, we can operate on S by traversing all adjacency lists of all vertices not in S, charging them according to the ripe rule. In the process we can identify all edges from elements outside of S to elements inside of S, which identifies edges of S to elements outside of S without traversing adjacency-list elements in S

or charging to them. These can then be removed from S if the adjacency lists are implemented with doubly-linked lists.

The complement trick comes up in various guises in what follows. On occasion, we touch elements in a class that is not ripe; this is allowed as long as we can charge the cost of touching them to elements in a different class that is ripe.

4.6 Data Structures

For the graph, we use a dynamic adjacency-list representation of the graph G where each adjacency-list element is in a doubly-linked list, and where each vertex is labeled with its current degree, and each adjacency-list element has a pointer to the vertex whose adjacency list it resides in. In addition, if v is an element in u's adjacency list, then this element has a pointer to its *twin* edge, that is, the occurrence of u in v's adjacency list. We do this when we split G into induced subgraphs for the recursive calls. Clearly, we can perform the inverse of this operation in $O(1)$ time, which we do when we insert a marker.

Proposition 2 *When we remove a vertex v from the adjacency list of u, it takes $O(1)$ time to remove u from the adjacency list of v. It takes $O(1)$ time to look up the current degree of a vertex, given a pointer to it.*

Definition 4 Partition Class: *We use the following data structure to implement a partition class:*

- *A doubly-linked list of known connectors of the class, where each element has a pointer to the header of this list and the header has a pointer to the last element of the list.*
- *A doubly-linked list of internal vertices in the class, where each element has a pointer to the header of this list and the header has a pointer to the last element.*
- *A pointer from the header of each of these two lists to a class header, which keeps a count of its degree sum, number of vertices, number of non-marker vertices, number of known connectors, and other counters as needed.*

Proposition 3 *Using the above data structure to implement a partition class, we get the following operations:*

- *Remove a connector or an internal vertex from a class, given a pointer to it in $O(1)$ time;*
- *Add a known connector or an internal vertex to a class by inserting it at the end of the appropriate doubly-linked list in $O(1)$ time;*
- *Identify the class that a vertex belongs to and to determine whether it is a known connector, given a pointer to the vertex in $O(1)$ time;*

- *Retrieve the current vertex size measure of the class in $O(1)$ time;*
- *Retrieve the current edge size measure of the class in $O(1)$ time;*
- *Merge two instances of* `Partition Class` *in time proportional to the number of vertices in the smaller of the two, where the new set of connectors is the union of the two old sets of connectors, and the new set of internal vertices is the union of the two old sets of internal vertices.*
- *Support a* `next-vertex` *iterator, which returns a vertex that* `next-vertex` *hasn't already returned since the last time that the iterator was initialized or the vertex was inserted to the class. It returns* null *if such a vertex doesn't exist. Each iteration takes $O(1)$ time.*

The `next-vertex` iterator is implemented with a pointer to the next unreturned element in the list of known connectors and the next unreturned element in the list of internal vertices. Since newly inserted elements are appended to the lists, the iterator can return elements that were inserted after the iterator was initialized.

4.7 Preparing recursive calls

When we generate recursive calls on a set of split sets, we remove each adjacency-list element corresponding to an edge that goes from a vertex in one recursive call to a vertex in another. We then add a marker to create the subgraph that is passed to the recursive call. This avoids passing any extraneous edges to a recursive call that are not part of the recursive subproblem. Let us call this the *cleanup operation*.

The operation serves as an illustration of an application of both the ripe rule and the complement trick. For this operation, we use the *vertex size measure* (Definition 3) in applying the ripe rule.

The sets $\{S_1, S_2, \ldots, S_k\}$ passed to recursive calls are split sets. By Proposition 3, we can identify the split set S_i with the largest number of non-marker vertices in $O(k)$ time. We traverse all adjacency lists in each class $S_j \neq S_i$, removing all adjacency-list elements that go to a vertex in a class other than S_j. S_j is ripe. By Proposition 3, it takes $O(1)$ time to identify whether an adjacency-list element goes to a vertex in another class. When we remove v from u's adjacency list, we remove u from v's adjacency list in $O(1)$ time, by Proposition 2. When we have done this for all classes other than S_i, S_i has no vertex v with an adjacency-list element pointing to a vertex u in any $S_j \neq S_i$, as this was removed when v was removed from u's adjacency list. S_i was cleaned up even though it may not have been ripe; this illustrates an example of the complement trick.

We use the edge size measure in determining ripeness of a class when performing pivots in a call to $S()$. We perform pivots on ripe vertices until no vertex is ripe.

Whenever we perform a pivot on a vertex p, we must traverse its adjacency list. Whenever we find an adjacency-list element that points to a vertex q in another class from the one that contains p, we move its twin element to the front of q's adjacency list. This takes $O(1)$ time given the doubly-linked representation of the adjacency lists, the twin pointers, and pointer of each adjacency-list element to the vertex whose list it is in. Therefore, the cost of this reshuffling of adjacency lists is subsumed by the other costs of the pivot. Let us call this the **reshuffling operation**.

Lemma 9 *If all vertices outside a class S have performed a pivot since the last time they were in the same class as members of S, then the neighbors outside the class of occupy prefixes of the adjacency lists of connectors of S.*

Proof Let c be a connector of S. When each neighbor of S performed a pivot, it was moved to the front of c's adjacency list. No other vertices are moved forward in c's adjacency list. Since a pivot has been performed on all neighbors of S, they occupy a prefix of c's adjacency list. □

Lemma 10 *Let S be a partition class with the largest edge size measure when no vertex is ripe. A pivot has been performed on all vertices outside of S since the last time they were in the same class as members of S.*

Proof Each vertex v outside of S is in a class with at most the size measure as S has, hence at most half the size measure of the class that most recently contained both v and S. Therefore, v must have become ripe since splitting from S, and since it is no longer ripe, a pivot has been performed on v. □

Corollary 2 *When no vertex is ripe, a partition class S with the largest size measure is a member of $S(a, b, G)$.*

Proof By Lemma 10, a pivot has been performed on all vertices outside of S since the time they last shared a partition class with members of S. All neighbors of S are therefore known neighbors. That S is a valid split set follows from the pivot invariant, and that it is a member of $S(a, b, G)$ follows from the split invariant and the fact that it is a subset of the partition class $V - \{a, b\}$ that contained it at the beginning of the call to $S(a, b, G)$. □

Since each partition class has at most two markers, we use as a base case a class with two vertices, which can be handled trivially.

To select pivots, we select a ripe class P and perform a pivot on all members of the class. We then label the class with a *last-touched* label that gives its current size measure, which takes $O(1)$ time to retrieve, by Proposition 3. Recall that none of these pivots are allowed to split P, but they can split other partition classes. When a class is split by a pivot, the resulting subclasses inherit its last-touched label. By Proposition 3, for each subclass, we can retrieve its current size measure sum in $O(1)$ time, and by comparing this with its *last-touched* label, we can determine whether the class is ripe in $O(1)$ time and insert it in a list of ripe classes if it is. Since the ripe classes are kept in a list, it takes $O(1)$ time to find a ripe class when one is needed or else determine that no class is ripe.

By Proposition 3, given a vertex p, it takes $O(deg(p))$ time to find the founding sets in each partition class S that p has neighbors in. This is a matter of removing neighbors of p from the connectors and starting a founding set and removing the neighbors of p from the internal vertices and starting a founding set. The third founding set is what remains of the connectors of S. We create a fourth instance of `Partition Class` by moving the list I of internal vertices of S to it, in $O(1)$ time. Since p is ripe when this happens, the total number of times p can be used as a pivot is $O(\log n)$, and since the sum of degrees of vertices at all times is $O(m)$, the cost of finding founding sets in all calls to $S()$ is $O(m \log n)$, by Lemma 1.

Below, we show how to bound the cost of all land grabs in all calls to $S()$. First, let us examine the consequences of only pivoting on ripe vertices. A risk is that this constraint might cause the partition refinement to halt before the partition classes are split sets. However, by Corollary 2, a class with the largest edge size measure is a member of $S(a, b, G)$.

We then pivot once on a connector c from this X (it doesn't matter which one, since they all have the same neighbors outside of X), and then remove X from consideration. This pivot may split some more classes, generating more ripe sets, and restarting the partitioning.

To charge the cost of a restarting pivot on c, observe that only edges of c to neighbors lying outside of S are relevant to the pivot on c, since a pivot is not allowed to split the class that contains the pivot vertex. By Lemma 9, these edges form a prefix of the adjacency list for c, so we may perform the pivot by traversing only this prefix. Because the removed class containing c now constitutes a recursive call, these elements are removed from the adjacency

list of c by the cleanup operation before a recursive call is started. We can charge the cost of the restarting pivot to elements that are removed from c's list, ensuring that no edge of c is charged twice during execution of the split decomposition algorithm.

4.10 Bounding the cost of land grabs in calls to $S()$

To finish bounding the cost of all calls to $S()$, it it remains to bound the cost of all the land grabs in all calls to $S()$. When a class S is split, the pivot identifies up to three founding sets, S_1, S_2, and S_3, which we implement as instances of `Partition Class`. In addition, the pivot produces an instance I of `Partition Class` that contains the internal vertices of S.

The land grab now serves to remove elements from I and assign them to S_1, S_2, and S_3 as vertices from these sets claim them as internal vertices for their own class.

Lemma 11 *If v is a member of S_1, S_2, or S_3, it takes $O(deg(v))$ time for it to "plant flags" by claiming any neighbors in I as internal vertices for its own class.*

Proof For each element in v's adjacency list, we must determine whether it is in I, which takes $O(1)$ time by Proposition 3, and move it to the list of internal vertices of S_1, S_2, or S_3, also in $O(1)$ time by Proposition 3. □

If a partition class S contains no neighbors of a pivot, it is not split by the pivot, and not touched.

Otherwise, if two of S_1, S_2, or S_3 are empty, S is not split by the pivot, and the founding set is the connector set of S. The list I may be restored as its list of internal vertices in $O(1)$ time by Proposition 3, and S can subsequently be ignored in the call to $S()$.

Otherwise, at least two of S_1, S_2, and S_3 are non-empty, and S is properly split by the pivot on p. The main obstacle we must overcome is that one of the subclasses, $S_i \in \{S_1, S_2, S_3\}$, that S is split into by the pivot, may fail to be ripe at the end of the land grab. We do not know that S_i will fail to be ripe until we perform the land grab, as this determines the edge size measure of S_i after the split. We must therefore perform flag-planting operations in S_i, only to find out after the fact that it didn't become ripe in time to charge its elements for the cost of the land grab inside it. The way we get around this is to arrange to charge the cost of the land grab in S_i to elements in another class from $\{S_1, S_2, S_3\}$ that is ripe, at $O(1)$ per element.

S_1, S_2, or S_3 becomes **closed** when all of its current vertices have attempted to plant flags. This means that the class cannot grow anymore, and the remaining members of I must be distributed among the remaining members of $\{S_1, S_2, S_3\}$. If one of them is empty, it is closed initially. When only one member S_h of $\{S_1, S_2, S_3\}$ remains open, all remaining members of I are known to belong in S_h. Rather than inserting them individually, we merge what remains of I with S_h. By Proposition 3, this takes time proportional to the number of internal vertices already in S_h, hence proportional to the time already expended by this point in inserting internal vertices to S_h. Note that this is independent of how large I is.

We now show how to ensure that if S_i is not ripe, the cost of the land grab can be charged to vertices and adjacency-list elements in some $S_j \neq S_i$ that is ripe. We do this by running the land grabs on S_1, S_2, and S_3 concurrently, so that they compete for members of I. We run them in a way that keeps the total amount of work spent in each of these classes roughly equal at all times.

To accomplish this, we use a running counter in each of S_1, S_2, and S_3 of the number of vertices in each class that have attempted to plant flags, plus the degree sum of these vertices. Call these counters n_1, n_2, and n_3, respectively. We keep these counters as equal as possible. To do this, we use the `next-vertex` iterator of the data structure of Proposition 3 to produce the next candidate flag-planter in each of the three classes. Call them v_1, v_2, and v_3. For each $v_k \in \{v_1, v_2, v_3\}$, $n_k + 1 + deg(v_k)$ will be number plus degree sum of flag-planters in S_k if v_k is chosen next. We choose v_k to minimize $n_k + 1 + deg(v_k)$, and then set $n_k := n_k + 1 + deg(v_k)$.

Lemma 12 *Let d_1, d_2, and d_3 be the number of vertices plus degree sums of S_1, S_2, and S_3, respectively, and let t_1, t_2, and t_3 be the times spent on their land grabs, respectively. If S_k becomes closed while another class S_j remains open, $t_k = O(d_j)$. If S_k is the last class closed and S_j is the class that remains open, $t_j = O(d_k)$.*

Proof At each point, the time spent so far on a land grab in each class S_k is $O(n_k)$, by Lemma 11. At the point where S_k becomes closed, it was the last set to have a vertex v_k plant flags, and n_k is now d_k. S_j still has a vertex v_j that hasn't planted flags, and the degree sum of its vertices that have attempted to plant flags is n_j. Because v_k was chosen over v_j, $n_k \leq n_j + 1 + deg(v_j)$. Since $d_k = n_k$ and $n_j + deg(v_j) \leq d_j$, $d_k \leq d_j$. By Lemma 11, $t_k = O(d_k)$, hence $t_k = O(d_j)$.

If S_j is the last class open when S_k is closed, $t_j = O(n_j)$, and we have shown that $n_j \leq n_k = d_k$, so $t_j = O(d_k)$. □

By Lemma 11, if S_h is ripe after the split, then $t_h = O(d_h)$, so the cost of its land grab can be charged to elements of S_h at $O(1)$ apiece. By Lemma 12, if S_h

is a member of $\{S_1, S_2, S_3\}$ that is unripe after the split, the cost t_h of the land grab in S_h can be charged to the d_ℓ vertices and adjacency-list elements in some other member of $\{S_1, S_2, S_k\}$. Since only one of the three is unripe, this charges all costs to elements in a ripe set at $O(1)$ apiece. Identifying which of the other two is large enough to charge the costs to takes $O(1)$ time by Proposition 3.

One final subtlety in the charging argument remains. Consider the depiction of a call to $S(a, b, G)$ in Figure 14. The neighbor sets A and B of u that contain a and b, respectively are recursive calls. However, $S(a, b, G)$ has computed smaller partition classes inside these recursive calls, and we have charged elements in those classes when their numbers of non-markers have halved. If we threw these classes away and started a new recursive call using A or B as the partition class containing these elements, we would have increased the sizes of the partition classes containing vertices in A and B. Our analysis that an element can be charged $O(\log n)$ times assumes that the size of a partition class of an element doesn't ever increase.

Lemma 13 *Let A be the partition classes containing a after a call to $S(a, b, G)$, and let m be the marker created for the recursive call G_A on A. The members of $S(a, b, G)$ that are subsets of A are the same as the members of $S(a, m, G_A)$.*

Proof This is immediate from the homomorphic rule, as m is the image of $V \setminus A$ in the quotient passed to the recursive call. □

By Lemma 13, we don't need to make this call, since the the vertices are already in the partition classes such a call would put them in by Lemma 13. By symmetry, we may handle partition classes inside of B in the same way. The partition class containing a vertex is thereby prevented from ever increasing before a new call to $S()$.

4.11 Bounding the cost of calls to $L()$

We describe how to charge the cost of a call to $L(a, c, b, G)$; the call to $L(b, c, a, G)$ is handled symmetrically.

A call to $S()$ maintains a partition of the vertices. There is no reason that a vertex can't belong to three instances of the `Partition Class` data structure without affecting the cost of the operations of Proposition 3 on each.

The simplest way to implement $L(a, c, b, G)$ is by starting with all vertices of $V - \{a, c\}$ in its own instance S of `Partition Class`, and perform pivots on vertices outside of S, splitting S into subclasses and letting S be the subclass that contains b. The process continues until a pivot has been performed on all

vertices outside of S. By the pivot and splitting invariants, S is now the maximal split set B containing b and excluding a and c, as required. $L(b, c, a, G)$ is implemented similarly.

Lemma 14 *$L(a, c, b, G)$ takes time proportional to the degree sum of vertices that lie outside of the set B that it returns.*

Proof A pivot on p takes $O(1 + deg(p))$ and so does using a vertex v to plant flags. Each vertex outside of B is used once as a pivot and once to plant flags. □

Definition 5 *A* **boulder** *is a recursive call whose size measure is more than half the size measure of the parent.*

Note that there can be at most one boulder. The boulder could be an element of $S(a, b, G)$, or it could be A or B. As in the discussion above on preparing recursive calls, we can touch all elements except those that reside in the boulder, as the boulder is the only partition class that might not be ripe.

If the boulder is a member of $S(a, b, G)$, this can be detected in $O(1)$ time when it is produced in the call to $S(a, b, G)$, by Proposition 3. To avoid having the call to $L(a, c, b, G)$ touch elements internal to the boulder, we replace the boulder with a marker, yielding a quotient G'. The edges incident to the marker are generated in $O(1)$ time apiece by Lemma 9. We remove the elements of the boulder from the initial partition class $V \setminus \{a, c\}$ employed by $L(a, c, b, G)$ by traversing elements of $S(a, b, G)$ other than the boulder, removing them from this partition class, and forming a new partition class from them that excludes the boulder. This takes $O(1)$ time for each element not in the boulder, thereby avoiding charging elements of the boulder. The remaining elements of the partition class are kept so that they can be passed into the recursive call on the boulder for use as the initial instance of `Partition Class` by one of the two calls to $L()$ in that call. This is necessary, because assembling the class at the beginning of the call would incur charges to elements of the boulder before they were ripe.

Let us call this operation a **boulder extraction**.

We then run $L(a, c, b, G')$. By the homomorphic rule and the fact that the boulder is disjoint from B this call generates B, as required. By Corollary 1, the cost of generating and touching edges incident to the marker during the call to $L(a, c, b, G')$ can be charged to elements outside of the boulder. $L(b, c, a, G)$ is handled similarly, and it generates a second partition class consisting of the elements of the boulder, for use by the second call to $L()$ inside the recursive call on the boulder.

The remaining case is when no member of $S(a, b, G)$ is a boulder. This leaves

open the possibility that A or B is a boulder. Suppose B, is the boulder. If we call $L(a, c, b, G)$ before $L(b, c, a, G)$, then we can charge all costs to elements external to B. We can then perform a boulder extraction on B for the call to $L(b', c, a, G')$, which yields A by the homomorphic rule, avoiding charges to elements internal to B.

Unfortunately, we could call $L(b, c, a, G)$ before $L(a, c, b, G)$, since we have no way of knowing initially which of A or B might turn out to be the boulder. Suppose B turns out to be the boulder. The call to $L(b, c, a, G)$ would charge elements internal to B, and we would realize this too late to undo the damage to the time bound. To prevent this possibility, we run $L(b, c, a, G)$ and $L(a, c, b, G)$ in parallel, keeping the degree sum of vertices touched so far in each call equal to each other, using the technique we used for running land grabs in parallel during a call to $S()$. That is, if the next vertex to be operated on in $L(b, c, a, G)$ has degree d_1 and the next one to be operated on in $L(a, c, b, G)$ has degree d_2, add $1 + d_1$ to the sum of degrees of vertices operated on so far in $L(b, c, a, G)$ and add $1 + d_2$ to the sum of degrees of vertices operated on so far in $L(a, c, b, G)$, and choose the vertex that minimizes this sum to be the next operation. When $L(a, c, b, G)$ returns the boulder, we can detect this by Proposition 3. $L(a, c, b, G)$ has charged no elements internal to the boulder.

If the parallel call to $L(b, c, a, G)$ is still running, we charge the costs that it has incurred so far to elements charged by the call to $L(a, c, b, G)$, which has charged at least as many elements. We undo the refinement it has performed so far so that once again $\mathcal{P}_A$ is in its initial state, $\{V' \setminus b, c\}$. As this takes no longer than refining the classes, we may also charge its cost to elements charged by $L(a, c, b, G)$. We then perform a boulder extraction on B, replacing it with a marker. Let G' be the resulting quotient. $L(b, c, a, G')$, which charges to no element that is internal to the boulder, returns A, as required, by the homomorphic rule, without charging to elements internal to the boulder.

5 Generalizing to Strongly-Connected Digraphs

Given the undirected version of the split decomposition algorithm described above, the directed version can be described as a set of modifications to the undirected case. Recall that the homomorphic rule still applies. The only modifications that are nontrivial are the modification of the high-level strategy of Section 4.1 to incorporate the possibility of circular nodes, and the modification to the low-level pivot operation. Once these are addressed, all other elements of the algorithm are straightforward.

Suppose for the moment that all nodes of the decomposition tree are prime or degenerate. Since the reduction of the problem to procedures for finding $S(a, b, G)$ and $L(a, b, c, G)$ is based on the homomorphic rule, the reduction in the directed case is identical to the undirected case as long as all nodes of the tree are prime or degenerate.

5.1.1 Introducing circular nodes

Let us now relax the assumption that all nodes are prime or degenerate. Let u be a circular node on the path from a to b in the decomposition tree. Because any nonempty proper subset of a circular node's neighbor sets that is *consecutive* in the circular ordering of neighbor sets, the union X_1 of neighbor sets that lie clockwise from A and counterclockwise from B is a maximal split set that excludes a and b, hence one member of $S(a, b, G)$. The same is true of the union X_2 of neighbor sets that lie counterclockwise from A and clockwise from B. One of X_1 or X_2 may be empty, in which case we ignore it.

A similar argument applies to any circular node on the path from a to b. As in the undirected case, all members of $S(a, b, G)$ are unions of neighbor sets of nodes on the path from a to b in the decomposition tree, and partition $V \setminus \{a, b\}$. By induction, we may assume that recursing on the quotient consisting of X_1 and a marker produces the split decomposition of this quotient. Let w_1 be the neighbor of the marker. Replacing the marker with u makes w_1 be a neighbor of u, and doing the same for the results of recursive calls on A, B, and X_2 makes u a tree node of degree at most four (depending on whether one of X_1 and X_2 is empty).

Since the quotient at u has at most four vertices, it takes $O(1)$ time to determine that it is a cycle of transitive tournaments (that is, that u is a circular node). By induction, we may assume that the recursive calls on X_1 and X_2 determine whether w_1 and w_2 are circular nodes.

As described by Cunningham, the composition of two cycles of transitive tournaments is a cycle of transitive tournaments, so we contract the edge uw_1 and perform the composition on their quotients if they are both circular. We do the same for uw_2. This has the effect of making each neighbor set that comprised X_1 be a neighbor set of u in the final tree, and similarly for X_2.

We select a and b as before. The proof of Lemma 8 goes through without change; each recursive call has at most two markers. There are still at most two directed edges that are not incident to non-marker vertices. Since G is strongly-connected, Corollary 1 therefore still applies; at any point, the total

number of directed edges in recursive calls, counting those incident to markers, is $O(m)$, where m is the number of directed edges of G.

5.1.2 Implementation of a pivot

We must now use two types of pivots, *incoming pivots*, and *outgoing pivots*. Below is a breakdown of how this ends up affecting the structure of a partition class. Since there are two types of pivots, a vertex outside a partition class S is said to have been *finished* if both types of pivots have been performed on it since the time that it was separated from S.

Of interest in a partition class S are the following:

- *Known incoming connectors* - These are vertices in S that have incoming edges from finished outsiders.
- *Known outgoing connectors* - These are vertices in S that have outgoing edges to finished outsiders.
- *Finished incoming neighbors* - These are outside vertices that are known to have edges to known incoming connectors.
- *Finished outgoing neighbors* - These are outside vertices that are known to have edges to known incoming connectors.
- *Insiders internal to the split set* - Vertices inside the S that don't have any edges incident finished neighbors.

Note that the outgoing and incoming connector sets may overlap.

Our **incoming pivot invariant** is that the known incoming connectors all have incoming edges from the same set of finished outsiders. Our **outgoing pivot invariant** is that the known outgoing connectors all have outgoing edges to the same set of finished outsiders.

Assume that both the incoming and outgoing pivot invariants apply before an outgoing pivot on a vertex p. The goal of an outgoing pivot on a vertex p is to turn p into a known incoming neighbor of all partition classes in which p has neighbors, partitioning the classes in order to satisfy both the incoming and outgoing pivot invariant while observing the split invariant. The goal of an incoming pivot on p is defined symmetrically.

The way we accomplish this for an outgoing pivot is to perform the pivot exactly as we did in the directed case, performing the same operations on the adjacency lists as we did before. For an outgoing pivot, we do this on the transpose of the graph, where the adjacency list of each vertex v is its list of vertices that have directed edges to v. We show that performing pivots in this way preserves both the incoming and outgoing pivot invariants.

Recall that in the undirected case, when no vertex is ripe in a call to $S()$, all vertices outside of a largest class S have been used as pivots since the last time they were in a common class with S. This will still be true for our version of the algorithm for strongly-connected digraphs. As before, this ensures that all connectors and neighbors are known, except that now we say that all incoming and outgoing connectors and all incoming and outgoing neighbors are known. Because the incoming and outgoing pivot invariants apply, S must be a split set. Because the split invariant applies, S must be a largest split set that started in a single partition class, hence a member of $S(a, b, G)$ or $L(a, b, c, G)$.

5.2 Data Structures

For the graph, we use a dynamic adjacency-list representation of the graph G where each vertex v carries two doubly-linked adjacency lists: one that gives the neighbors of v (the out-neighbors) and one that gives the vertices that have v as a neighbor (the in-neighbors). Given an adjacency-list representation of the graph where only the lists of out-neighbors are given, it takes $O(m)$ time to find the in-neighbors of each vertex, by radix sorting a copy of the list of edges using destination vertex as the primary key and vertex of origin as the secondary key, and then cutting the list into segments that share the same destination vertex. If (u, v) is an edge, then the occurrence of v in u's out-neighbor list carries a pointer to the occurrence of u in v's in-neighbor list, and the occurrence of u in v's in-neighbor list carries a pointer to the occurrence of v in u's out-neighbor list. When v is removed from u's out-neighbor list, this allows u to be removed from v's in-neighbor list, or vice versa, in $O(1)$ time.

The *degree* of a vertex v, denoted $deg(v)$, is the number of in-neighbors and out-neighbors, and each vertex is labeled with its current degree. Let $deg^+(v)$ denote the *out-degree*, that is, the number of out-neighbors, and $deg^-(v)$ denote the *in-degree*, that is, the number of in-neighbors.

Definition 6 *The edge size measure of a partition class in a directed graph is the number of non-marker vertices plus the sum of their degrees.*

The structure of partition classes is similar to those in the undirected case. However, in the directed case, there are two types of connectors, the *outgoing connectors* and the *incoming connectors*. There are also internal vertices that are not connectors. We modify the `Partition Class` structure of Definition 4 by giving it three lists, one for the outgoing connectors, one for the incoming ones, and one for the internal vertices. As before, these lists are doubly-linked, the elements have pointers to the list headers, the list headers have pointers

to a class header, which keeps track which keeps a count of its vertex size measure, edge size measure, and other counters as needed.

Proposition 4 *Using the above data structure to implement a partition class, it takes $O(1)$ time to perform any of the following operations:*

- *Remove an outgoing connector, an incoming connector, or an internal vertex from a class, given a pointer to it;*
- *Add an outgoing connector, an incoming connector, or an internal vertex to a class by inserting it at the end of the appropriate doubly-linked list;*
- *Identify the class that a vertex belongs to and to determine whether it is a known connector, given a pointer to the vertex;*
- *Retrieve any of the counters stored in the class header;*
- *Merge two instances of* `Partition Class` *in time proportional to the number of vertices in the smaller of the two, where the new set of incoming connectors is the union of the two old sets of incoming connectors, the new set of outgoing connectors is the union of the old ones, and the new set of internal vertices is the union of the old ones.*
- *Support a* `next-vertex` *iterator, which returns a vertex that* `next-vertex` *hasn't already returned since the last time that the iterator was initialized or the vertex was inserted to the class. It returns* null *if such a vertex doesn't exist.*

5.3 Preparing recursive calls

As before, when we generate recursive calls on a set of split sets, we remove edges that go from elements in one recursive call to another, and we use the vertex size measure in determining when a class is ripe.

Lemma 7 goes through with trivial changes:

Lemma 15 *The in-degree and out-degree of a non-marker vertex in a recursive call is never greater than these degrees in the main call.*

Proof If the vertex is an incoming connector of the split set passed to a recursive call, it gains a marker neighbor, but this marker replaces one or more in-neighbors in a separate split set that got passed to a separate recursive call. It gained at most one (marker) neighbor and lost at least one neighbor. By a symmetric argument, if it is an outgoing connector, its out-degree does not increase. If it is an internal node of the split set, then all of its in-neighbors were also in-neighbors in the parent call and all of its out-neighbors were also out-neighbors in the parent call. In each case, its degree in a recursive call is bounded by its degree in the parent call. □

The sets $\{S_1, S_2, \ldots, S_k\}$ passed to recursive calls are still split sets. By Proposition 4, we can identify the split set S_i with the largest number of non-marker vertices in $O(k)$ time. We traverse all out-neighbor and in-neighbor lists in each class $S_j \neq S_i$, removing all elements residing in a class S_k other than S_j, and, while we're at it, we remove the twin adjacency elements in S_k. This takes $O(1)$ time for each adjacency-list element in all sets other than S_i, but removes all adjacency-list elements in S_i that reside in classes other than S_i. Since each class other than S_i has half the non-marker vertices that the parent call has, each adjacency-list is traversed $O(\log n)$ times in all calls to cleanup operations.

5.4 Implementation of a pivot

We describe how an outgoing pivot on vertex v and partition class S works; an incoming pivot is defined symmetrically. It works just as in an undirected graph, where the outgoing adjacency lists assume the role of the undirected adjacency lists and the known incoming connectors assume the role of the known connectors. With this change, the founding sets are defined as before: those known incoming connectors that have the pivot as an in-neighbor those known incoming connectors that don't have the pivot as an in-neighbor, and those that are in neither of these categories, but have the pivot as an in-neighbor.

Once again, a pivot is not allowed to split the partition class that contains it. By Proposition 4, it takes $O(deg^+(p))$ time to obtain instances S_1, S_2, S_3 of the `Partition Class` data structure for representing these the founding sets for each partition class S, as well as an instance of the `Partition Class` data structure representing the set $I = S \setminus (S_1 \cup S_2 \cup S_3)$.

As before, in traversing the adjacency list of p, whenever we find an adjacency-list element that points to a vertex in another class from the one that contains p, we move it to the front of the adjacency list for p and we move its twin element to the front of its adjacency list. This takes $O(1)$ time given the doubly-linked representation of the adjacency lists, the twin pointers, and pointer of each adjacency-list element to the vertex whose list it is in. Therefore, the cost of this reshuffling of adjacency lists is subsumed by the other costs of the pivot. Let us again call this the **reshuffling operation**.

Lemma 16 *If pivots have been performed on each vertex outside of a class S since it was last in the same class as members of S, all in-neighbors occupy a prefix of each incoming connector's in-neighbor list, and all out-neighbors occupy a prefix of each outgoing connector's out-neighbor list.*

The proof is a trivial variant of the proof of Lemma 9.

An *outgoing flag planting* is the operation of allowing a vertex v of S_1, S_2, and S_3 to move its out-neighbors in I to the partition class in $\{S_1, S_2, S_3\}$ that contains v.

An outgoing pivot on class S once again consists of finding the incoming founding sets $\{S_1, S_2, S_3\}$, then letting these sets grow by planting outgoing flags until all vertices in the three sets have planted outgoing flags.

Lemma 17 *If the incoming and outgoing pivot invariants are observed by S before an outgoing pivot, then the partition classes vertices of S_1, S_2, and S_3 have planted outgoing flags, these three partition classes observe the incoming and outgoing pivot invariants.*

Proof Let P be the finished incoming outsiders before the outgoing pivot that splits class S. The connectors of the class all have P as their in-neighbors outside S before the pivot. The incoming connectors of the new subclasses are the connectors of S that have only P, $P \cup \{p\}$, or $\{p\}$ as their finished incoming neighbors outside of S. They thus satisfy the incoming pivot invariant. The outgoing connectors of the new classes are subsets of the outgoing connectors of S. If Q is the set of known out-neighbors of S, it remains the set of known out-neighbors of each of $\{S_1, S_2, S_3\}$. All out-connectors of each of the classes have Q as their known out-neighbors, so they satisfy the outgoing pivot invariant. □

Lemma 18 *After all vertices of S_1, S_2, and S_3 have planted outgoing flags, the partition of S observes the splitting invariants.*

Proof Identical flags will be planted in all incoming connectors of a split set S' that starts out as a subset of S; they are either planted by the pivot if they are connectors of S, or else by the first neighbor of the connectors that attempts to plant flags. This is because all connectors of S' have identical neighborhoods outside of S'. Once this happens, the flags on the connectors block access to S' to all competing land grabs, as S' has no incoming edges other than those into its incoming connectors. Because G is strongly-connected, there are paths from every incoming connector of S' to all other members of S'. The land grab will expand out along these paths, giving all of S' identical flags, hence S' will reside in a single partition class. □

An incoming pivot is defined symmetrically, and symmetric versions of Lemmas 17 and 18 apply. It follows from Lemma 17 and its symmetric version that when all outside in-neighbors and out-neighbors of a class are known, it satisfies the requirements of a split set. It therefore follows from Lemma 18 and its symmetric version that when all outside in-neighbors and out-neighbors of a class are known, it is a member of $S(a, b, G)$.

As before, we perform a pivot on a vertex p only if the edge-size measure of its class has halved since the last time it was used as a pivot.

When we perform a incoming or outgoing pivot on p, we perform both incoming and outgoing pivots on all vertices in its class. By Proposition 4, it takes $O(deg^+(p))$ to find the founding sets in all partition classes that contain neighbors of p and do not contain p. We use *last-touched* labels on the classes, just as before, to identify classes of vertices that are ripe. Since at all times the number of edges in all recursive calls is $O(m)$ and the eligibility requirement ensures that each time $O(1)$ time is spent on the adjacency-list elements of a vertex, the number of non-marker vertices in its class halves, it takes $O(m \log n)$ time to find founding sets in all calls to $S()$ made during the course of the algorithm.

The risk in observing this eligibility requirement is once again that there may be no eligible vertices at some point before $S(a, b, G)$ has been computed. However, once again, at this point, a class S with a largest number of non-marker vertices must be a valid split set.

Performing one pivot on an outgoing c_1 connector of S and one on an incoming connector c_2 of S restarts the partitioning, and S can be removed, as it is a known member of $S(a, b, G)$. As before, Lemma 16 ensures that the adjacency-list elements of each class S that point to vertices outside of S are prefixes of the outgoing adjacency list of c_1 and the incoming adjacency list of c_2. These are the only edges that are relevant to performing the partition, and since S is a split set, it constitutes a recursive call, and these edges are discarded from these adjacency lists before the recursive calls start, by the cleanup operation. Therefore, no edge of c_1 or c_2 is charged twice for a restarting pivot during execution of the algorithm.

5.6 Bounding the cost of land grabs in all calls to $S()$

By Proposition 4, Lemma 11 generalizes to the directed case, except that planting the flags takes time proportional to the out-degree of the flag planter if the pivot is an outgoing pivot and time proportional to the in-degree if the pivot is an ingoing pivot.

We discuss the cost of bounding the cost of outgoing pivots; the bound is obtained in a symmetric way for incoming pivots. As before, class S is split into instances S_1, S_2, S_3 and I of the `Partition Class` data type by an incoming neighbor p, where S_1, S_2, and S_3 are the founding sets, and I is the

set of vertices that will be claimed by the land grab. S_1, S_2, and S_3 are the incoming connectors of the new subclasses that will form. I is formed in $O(1)$ time by moving the lists of outgoing connectors and internal vertices of S to a new instance of the `Partition Class` data type.

The flag-planting operation is similar to the one in the undirected case, except that the vertices in I are classified as outgoing connectors or internal vertices, and they retain this status when moved to S_1, S_2, and S_3. In addition, if the flag planter is an incoming connector that is also an outgoing connector, it moves its occurrence in I as an outgoing connector to the list of outgoing connectors of its own class. This adds $O(1)$ to the cost of flag-planting, which is subsumed by other costs.

A class becomes **closed** when all vertices have been used as outgoing flag planters. When all but one class, S_i, is closed, the remaining vertices in I are moved to the open class by transferring the two lists of outgoing connectors and internal vertices to S_i, and then appending the current outgoing connectors and internal vertices to these lists, just as before. The cost of moving the lists is $O(1)$ and the cost of appending is ones that are already there is subsumed by the cost incurred before all but S_i was closed.

As before, we let a class become ripe each time the number of vertices plus the degree sum halves. We let n_1, n_2, and n_3 be the number of flag-planters used so far, plus the degrees of these flag planters in S_1, S_2, and S_3, respectively. We use an iterator from each class to produce the three candidates, v_1, v_2, and v_3 to be the next flag-planters, and choose v_k that minimizes $n_k + 1 + deg^+(v_k)$.

For the generalization of Lemma 12, we let d_1, d_2, and d_3 be the number of vertices plus degree sums of the vertices. The times t_1, t_2, t_3 spent at any point are $O(d_1)$, $O(d_2)$, and $O(d_3)$ as before, as the number of vertices plus out-degrees in a class is bounded by the number of vertices plus degrees. With these observations, the proof of Lemma 12 goes through unchanged. Therefore, if one of S_1, S_2, and S_3 is not ripe after the land grab, we can charge the time spent on the land grab to elements of one of the other classes, which is ripe, by the fact that Corollary 1 applies also in the directed case.

We again consider the depiction of a call to $S(a, b, G)$ in Figure 14. The neighbor sets A and B of u that contain a and b, respectively, are recursive calls. However, $S(a, b, G)$ has computed smaller partition classes inside these recursive calls, and we have charged elements in those classes when their numbers of non-markers have halved. Because it is based on the homomorphic rule, which is general to strongly-connected digraphs, Lemma 13 continues to apply.

As before, if G' is G_A in the parent call, then we use the call $S(a, m, G_A)$. By Lemma 13, we don't need to make this call, since the vertices are already in the partition classes such a call would put them in by Lemma 13. By

Lemma 8, there are at most two markers to choose from in selecting a and b. By symmetry, we may handle partition classes inside of B in the same way. The partition class containing a vertex is thereby prevented from ever increasing before a new call to $S()$.

5.7 Bounding the cost of calls to $L()$

As before, we implement $L(a, c, b, G)$ as the subset of operations of $S(a, b, G)$ that lay claim to vertices in the set that contains b. The argument is the same as the one before, except that we perform both an outgoing and incoming pivot on a vertex after it is known not to be in the class containing b that the operation is going to be returned. Instead of being used once as a pivot and once as a flag-planter, each vertex is used twice for each of these operation: once as an outgoing pivot and once as an outgoing flag planter, and once as an incoming pivot and once as an incoming flag planter. We have established that each outgoing pivot takes $O(1)$ time for each element in its outgoing adjacency list and each flag-planter takes $O(1)$ time for each element in its outgoing adjacency list. The same is true for incoming adjacency-list elements on incoming pivots and incoming flag planters. Lemma 14 therefore applies without change.

Because the splitting and pivot invariants are maintained on the class containing b, the algorithm halts when we have the maximal split set containing b and excluding a and c, as required.

To avoid charging elements internal to a boulder, we run $L(a, c, b, G)$ and $L(b, c, a, G)$ in parallel, as before, keeping the work equal in both calls by choosing from the next vertex in each call to be operated on and choosing the one that will minimize the amount of work for its call after the operation. We calculate this by 1 plus the out-degree to the work so far for the call if the next operation is an outward pivot or outward flag planting, and 1 plus the in-degree if the next operation is an inward pivot or inward flag planting. If $L(a, c, b, G)$ returns a boulder, it will again halt in time to extract the boulder in the call to $L(b, c, a, G)$, and to charge the cost of incurred by $L(b, c, a, G)$ touching elements internal to the boulder to operations performed in $L(a, c, b, G)$, as before.

6 Acknowledgments

We would like to acknowledge Haim Kaplan for helpful discussions and feedback on this work.

References

[1] W. H. Cunningham. Decomposition of directed graphs. *SIAM J. Algebraic Discrete Methods*, 3:214–228, 1982.

[2] C.P. Gabor, K.J. Supowit, and W.-L. Hsu. Recognizing circle graphs in polynomial time. *Journal of the ACM*, 36:435–473, 1989.

[3] M. Rao. Solving some NP-complete problems using split decomposition. *Discrete Applied Mathematics*, 2007.

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

[5] 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.

[6] A. Bouchet. Digraph decompositions and eulerian systems. *SIAM Journal on Algebraic and Discrete Methods*, 8, 1987.

[7] J. Spinrad. Prime testing for the split decomposition of a graph. *SIAM Journal on Discrete Mathematics*, 2:590–599, 1989.

[8] T.H. Ma and J. Spinrad. An $O(n^2)$ algorithm for undirected split decomposition. *Journal of Algorithms*, 16:145–160, 1994.

[9] E. Dahlhaus. Parallel algorithms for hierarchical clustering, and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36:205–240, 2000.

[10] J. P. Spinrad. *Two Dimensional Partial Orders*. PhD thesis, Princeton University, 1982.

[11] J. P. Spinrad and J. Valdes. Recognition and isomorphism of two-dimensional partial orders. In *Proceedings of the 10th Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science*, pages 676–686. Springer-Verlag, Berlin, 1983.

[12] J. P. Spinrad. On comparability and permutation graphs. *Siam J. Comput.*, 14:658–670, 1985.

[13] J. P. Spinrad. Graph partitioning. unpublished manuscript, 1985.

[14] W.L. Hsu and R.M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296:59–74, 2003.

[15] A. Bouchet. Reducing prime graphs and recognizing circle graphs. *Combinatorica*, 7:243–254, 1987.

[16] S. Cicerone and di Stefano G. On the extension of bipartite to parity graphs. *Discrete Applied Mathematics*, 95:181–195, 1999.

[17] G. Cornuejols and W.H. Cunningham. Compositions for perfect graphs. *Discrete Math.*, 55:245–254, 1985.

[18] H.J. Bandelt and H.M. Mulder. Distance-hereditary graphs. *Journal of Combinatorial Theory Series B*, 41:182–208, 1986.

[19] P. L. Hammer and F. Maffray. Completely separable graphs. *Discrete Applied Mathematics*, 27:85–99, 1990.