

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

Benson L. Joeris¹, Scott Lundberg², and Ross M. McConnell²

¹ Trinity College, Cambridge, CB2 1TQ, UK
blj24@cam.ac.uk

² Computer Science Department, Colorado State University, Fort Collins, CO
80523-1873
{lundberg,rmm}@cs.colostate.edu

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.

Keywords: split decomposition, join decomposition, strongly connected graphs.

1 Introduction

Split decomposition is a unique decomposition of arbitrary strongly-connected digraphs described by Cunningham in 1982 [3]. Because 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 [5] to optimizations of NP-hard problems [9]. It is a proper generalization of the well known modular decomposition, also called substitution decomposition [8,7].

As a convention we denote the number of vertices of a graph as $n = |V|$ and the number of edges $m = |E|$.

Cunningham gave the first algorithm for computing the decomposition on arbitrary strongly-connected digraphs, which runs in $O(n^4)$ time [3]. Bouchet improved this to $O(n^3)$ [2]. 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 below). Spinrad gave an $O(n^2)$ algorithm for determining whether an arbitrary strongly-connected directed graph is prime [10], but not for finding the decomposition tree if it has a nontrivial decomposition. Since then, much work has focused on the special case of undirected graphs. This work includes an $O(nm)$ algorithm by Gabor, Supowit, and Hsu [5], an $O(n^2)$ algorithm by Ma and Spinrad [6], and, finally, a linear-time ($O(n + m)$) algorithm by Dahlhaus [4].

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 [6]. In particular, we make similar 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 [11], [14], [13]. We get the improvement to Ma and Spinrad's time bound, a generalization to strongly-directed graphs, and a substantial simplification of their approach, by modifying a clever charging argument for partition refinement, called *halving before re-use*. The use of this trick for modular decomposition, transitive orientation, and many related problems was circulated widely in the mid-1980's in a working manuscript, also written by Spinrad [12]. He obtained the trick by showing how to modify a related one, developed by Hopcroft, for state minimization in deterministic finite automata [1]. These techniques and applications have been mistakenly attributed in the literature to subsequent papers that, like the present paper, borrow heavily from Spinrad's early work on the subject.

2 Split Sets and Split Decomposition Trees

We view an undirected graph as the special case of a directed graph where every undirected edge is a directed two-cycle. 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 . If G is a directed graph, then we let $deg(x)$ denote the number of in-neighbors plus the number of out-neighbors.

A *split* in a directed graph is a partition of the vertices into two sets, X and Y , where the edges directed from X to Y forms a cartesian product $X' \times Y'$, for some $X' \subseteq X$ and some $Y' \subseteq Y$, and the edges directed from Y to X form a cartesian product $Y'' \times X''$, for some $Y'' \subseteq Y$ and some $X'' \subseteq X$. For example, in Figure 1, $X' = \{a\}$, $Y' = \{c, d, e\}$, $X'' = \{b\}$, and $Y'' = \{c, d, e\}$. X and Y are *split sets* if they satisfy these requirements. X' is the set of *outgoing connectors* of X , and X'' is the set of *incoming connectors* of X . Y' is the set of *outgoing neighbors* of X , and Y'' is the set of *incoming neighbors* of X . Applying these definitions to Y reverses the roles of outgoing vs. incoming and connectors vs. neighbors. In a directed graph, it is not necessary that $X' = X''$ or $Y' = Y''$ or that they be disjoint, but in an undirected graph, symmetry dictates that $X' = X''$ and $Y' = Y''$, in which case we can refer to X' and Y' simply as the *connectors* and *neighbors* of X .

Figure 1 shows how, if we find a split, we can represent the graph with two *quotients*, where, in one quotient, Y is replaced with a single *marker vertex* whose out-neighbors and in-neighbors are the in-connectors and out-connectors of X , and X is similarly replaced with a marker vertex in the other. This process is invertible; we can reconstruct the original graph by a *composition* of the two quotients.

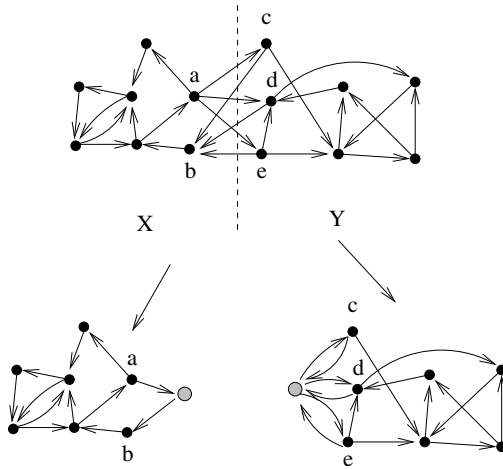


Fig. 1. Two quotients formed by a split in a directed graph

This differs from modular decomposition, where a graph is broken into a quotient, which receives a marker, and a factor, which does not. This asymmetry is due to the fact that, unlike split decomposition, where the complement of a split set is a split set, the complement of a module in modular decomposition is not a module.

The quotient consisting of X and its marker y can be considered a many-to-one mapping of V to $X \cup \{y\}$, where each element of X maps to itself and each element of Y maps to y . This mapping has the property that any split set of G maps to a split set of the quotient, and the inverse image of any split set of the quotient is a split set of G . Let us call this the *homomorphic rule*.

There can be an exponential number of split sets (consider the complete graph, where all nonempty proper subsets of V are split sets). However, they can be represented implicitly in $O(n)$ space with a *split decomposition tree*, which is an unrooted tree whose leaves are the vertices of G . We first describe how this is accomplished for an undirected graph. 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 of the tree is labeled *prime* or *degenerate*. A set is a split set if and only if it is a neighbor set of an internal node, a union of all but one neighbor set of an internal node, or any union of at least one and fewer than all neighbor sets of a degenerate node. Cunningham showed that there is a unique tree with these properties.

For instance, Figure 2 gives the split decomposition of an undirected graph. The neighbor sets of node 3 are $\{c\}$, $\{y, x, w, v\}$, $\{a\}$, $\{e, d, b, f\}$, and $\{g, h, i, j, k, n, p, q, r, s, t\}$. This is a degenerate node, since every union of these neighbor sets is a split set. For instance, the union of $\{u, v, w, x, y\}$, $\{c\}$, and $\{b, d, e, f\}$ is a

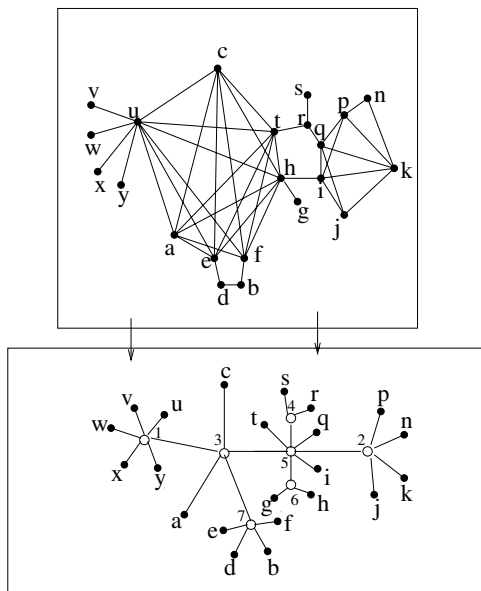


Fig. 2. The split decomposition of an undirected graph

split set with connectors $\{u, c, e, f\}$. Node 5, on the other hand is prime: the only unions of its neighbor sets that are split sets are one neighbor set or 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 2 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. 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. 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.

If the graph is a strongly-connected digraph, the decomposition tree is similar, except that, in addition to prime and degenerate nodes, it may have *circular nodes*. At a circular node, there is a cyclic ordering of its neighbor sets, and a union of neighbor sets is a split set if and only if it is a union of at least one and fewer than all of the neighbor sets *that are consecutive in the cyclic 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 *cycle of transitive tournaments* is a cyclic ordering of transitive tournaments, where the sink of each is identified with the source of the next. Figure 3 gives an example. 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.

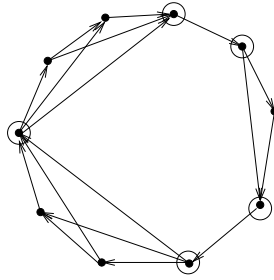


Fig. 3. The quotient associated with a circular node is a cycle of transitive tournaments. The sink of each transitive tournament is the source of the next (circled vertices), and a nonempty set of vertices is a split set if and only if the vertices are a proper subset of the vertices that is consecutive in the cyclic order.

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.

2.1 The Strategy

Let a, b, c be vertices of G . We have two methods at our disposal whose implementation is described below: $S(a, b, G)$, which finds 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 a partition of $V \setminus \{a, b\}$, 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 discarding all returned sets except the one that contains c , but we use a separate procedure that omits the unnecessary steps for efficiency reasons. Before showing how to compute $S(a, b, G)$ and $L(a, b, c, G)$, we show how to reduce finding the split decomposition to calls to these procedures.

Because a, b, c 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

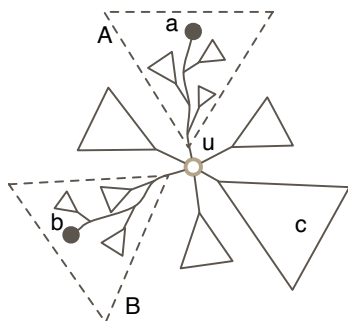


Fig. 4. What we get from calling $S(a, b, G)$

at single internal node, which we will call u . (See Figure 4.) Let A , B , and C denote the neighbor sets of u that contain a , b , and c , respectively.

It is easy to see from the relationship between the split sets and neighbor sets of prime nodes that if all nodes of the split decomposition are prime, $S(a, b, G)$ returns precisely the neighbor sets of nodes on the path from a to b that do not contain a or b . It follows that $S(a, b, G)$ is a partition of $V \setminus \{a, b\}$, that each neighbor set of u , other than A and B , is a member of $S(a, b, G)$, and that $A \setminus \{a\}$ and $B \setminus \{b\}$ each are unions of the remaining members of $S(a, b, G)$, since they are the neighbor sets of the nodes other than u on the path from a to b . A call to $L(b, c, a, G)$ finds A and a call to $L(a, c, b, G)$ finds B . We now know all neighbor sets of u .

To find the remainder of the decomposition tree, we replace each neighbor set of u with a quotient graph, recursively find the decomposition trees of these quotients, and then join them by identifying their markers with u (see Figure 5). The correctness of this algorithm when each node of the decomposition tree is prime is immediate from the homomorphic rule.

Introducing degenerate and circular nodes. From the relationship between the split sets and the decomposition tree, it is easy to see that once the possibility of degenerate and circular nodes is introduced, the following are the members of $S(a, b, G)$ for some node v on the path from a to b : a neighbor set of v that does not contain a or b if v is prime, the union of all neighbor sets that do not contain a or b if v is degenerate, the union of neighbor sets clockwise from A and counterclockwise from B or the union of neighbor sets clockwise from A and counterclockwise from B if v is a circular node. (In the case of a circular node, one of the two sets is empty if A and B are adjacent in the cyclic order.)

As before, we find A and B by calls to $L(b, c, a, G)$ and $L(a, c, b, G)$, and recursively find the decomposition trees of the quotients for A , B , and those members of $S(a, b, G)$ that are not subsets of A or B , that is, that are unions of one or more neighbor sets of u . If u is prime, this requires one recursive call for each neighbor set of u . If u is degenerate, this requires three recursive calls, one for A , one for B , and one for the union of all other neighbor sets. If u is circular,

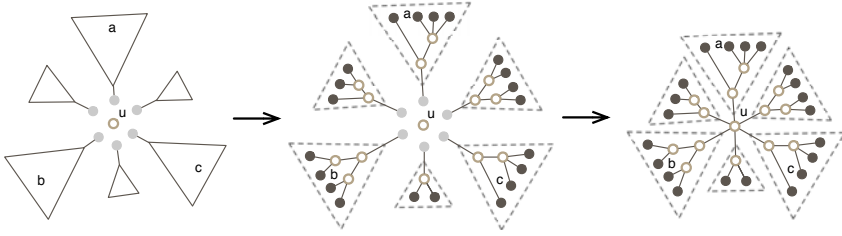


Fig. 5. Completing the decomposition tree through recursion

it requires up to four recursive calls, as explained above. We then identify the markers with u to perform a composition of these trees as before.

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. Identifying u and the marker for X , 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. Note that w carries the missing neighbor sets that should be u 's. Since u has degree at most four, it takes $O(1)$ time to find its associated quotient and determine whether it is a complete graph, a star, or a circular graph (cycle of transitive tournaments). By induction, we assume that this has been determined for w by the recursive call. If u and w both have complete or circular quotients, the composition of their quotients obtained by identifying the markers corresponding to u and w is a larger complete or circular quotient. We accomplish this by contracting the edge uw in the tree, letting u stand for the resulting node. This allows u to inherit its individual neighbor sets from w . Similarly, if both of u and w have associated quotients that are stars, we can determine in $O(1)$ time whether the composition of their two stars is a larger star, and perform the contraction if it is.

2.2 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 $S(a, b, G)$.

For simplicity, let us first assume that G is undirected. We find $S(a, b, G)$ by selecting a vertex as a *pivot vertex*, p , and perform a *pivot operation* on it, which may refine the partition by splitting partition classes that do not contain p . The *outsiders* of a partition class S are those vertices $V \setminus S$. For a vertex v that starts out in $V \setminus \{a, b\}$ the only initial outsiders of its class are a and b . Every time the class containing v is split, it finds itself in a smaller class that has more outsiders. If S is a partition class, the *known outsiders* are those on

which we have performed a pivot since the point when they became outsiders of S , and the *known connectors* of S are those vertices of S that have edges to known outsiders.

We show that if the following invariants apply before a pivot, then they also apply after the pivot:

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

The invariants apply to the initial partition because no partition class has been split and no partition class has any finished outsiders. When all outsiders of every partition class are known, it follows by the pivot invariant that each partition class is a split set. Since the initial partition is $\{\{a\}, \{b\}, V \setminus \{a, b\}\}$, the splitting invariant ensures that the final partition is $\{a\}, \{b\}$, and the maximal split sets that do not contain a or b , in other words, the final partition is $\{\{a\}, \{b\}\} \cup S(a, b, G)$.

Implementation of a pivot. We first describe the implementation of a pivot operation in the undirected case. Pivoting on a pivot vertex p moves it from the unfinished to the finished outsiders of each partition class except the one, P , that contains p and is not allowed to split during the pivot. In $S(a, b, G)$, for each partition class S other than P , we identify the following *founding sets* for subclasses that S will be subdivided into during the pivot: the known connectors of S that are non-neighbors of p , the known connectors that are neighbors of p , and the members of S that were not known connectors, but that are neighbors of p .

Some of these founding sets may be empty, and if S has at most one non-empty founding set, it will still satisfy the pivot invariant once p is included among its known outsiders. Otherwise, members of different founding sets in S have different sets of known outside neighbors, so they cannot remain in the same partition class without violating the pivot invariant. They become the known connectors of new subclasses that S will be partitioned into. It remains to determine which other members of S can be placed with which founding sets. We select a vertex z that has already been claimed for a subclass S' of S , and have it claim all of its neighbors in S for S' , except for those neighbors that have already been claimed. Let us call z a *claim staker*. A second claim-staking by z during a split of S cannot claim any new vertices for its subclass, so each vertex is selected at most once to stake claims during a pivot. Once all members of S have been claimed for a subclass, these subclasses become the refinement of S .

Except for the requirement that a vertex already be claimed before it is selected as a claim-staker, the order in which we select vertices as claim-stakers does not matter in maintaining the splitting invariant, which can be seen as follows. Let S'' be a split set that is a subset of S . If it intersects the known connectors of S , then the connectors of S'' must be its intersection with the known

connectors of S , and since its connectors have the same neighbors outside of S , its known connectors are a subset of a single founding set. All paths into S'' contain one of its connectors, the claiming of vertices in S'' is initiated at its connectors, and claims to other members of S'' are sealed off from claims to other members of S'' . All of S'' ends up in the subclass founded by this founding set. Similarly, if S'' doesn't intersect a founding set, its connectors are all claimed at once by the first outsider of S'' that claims vertices for its class, since a neighbor of one of its outsiders is a neighbor of all of them. This again seals off the rest of S'' from claims by competing classes, and S'' ends up in a single partition class, as required.

After the split of S , each subclass's founding set is its set of known connectors, and they all have the same set of known outsiders that are neighbors, including p . The pivot invariant is preserved.

$L(a, b, c, G)$ is implemented using pivots also, with the only difference being that we keep only the partition class that contains c . Every time this class is split by a pivot, we discard the resulting subclasses that do not contain p .

For a strongly-connected digraph, we run the same procedure for a pivot on p , with the only difference that we reinterpret the neighbors of a vertex to be its out-neighbors. Let us call this the *outward pivot*. This refines the partition, but does not refine the class containing p . We then re-run the procedure on the refined partition, this time reinterpreting the neighbors of a vertex to be its in-neighbors. This is the *inward pivot*. A *pivot* on p consists of an outward pivot and then an inward pivot. Using trivial variants of the above arguments, and the fact that the graph is strongly-connected, which means that every nonempty proper subset of V has both incoming and outgoing directed edges, it is easy to see that the splitting and pivot invariants are maintained.

2.3 The $O(m \log n)$ Time Bound

When the algorithm generates a recursive call, it adds a marker to it. That recursive call may, in turn, generate others, the marker is passed to one of them, and a new marker is added to that call. This shows that multiple markers can occur in a single call deep in the recursion.

To avoid proliferation of markers in any one call, when we select a , b , and c , we give priority to markers in selecting a and b . We claim that this ensures that there are at most two markers in any recursive call. The proof is by induction on the depth of the call. Suppose that it is true for a recursive call on a quotient, G . Since priority is given to markers in selecting a and b , no other vertices are markers. Since a and b are passed to different recursive calls, each recursive call, which receives a new marker, still has at most two markers.

Let the *non-marker degree sum* in a recursive call be the sum of in-degrees and out-degrees of the non-markers. Because each recursive call has at most two non-markers, there are at most two directed edges that fail to be incident to a marker. The following is immediate:

Lemma 1. *If k is the non-marker degree sum in a recursive call, then the total degree sum of all vertices in the call is $O(k)$.*

We keep a list of in-neighbors and a list of out-neighbors on each vertex. We implement each partition class with a doubly-linked list, where each element has a pointer to the front of the list that supports identifying its partition class in $O(1)$ time. In addition, we keep a doubly-linked list of the known in-connectors and a doubly-linked list of the known out-connectors, and mark the vertices in these lists according to which they are members of, which may be both of them.

During the outward pivot, this representation allows creating lists of the founding sets of all partition classes that don't contain p , in time proportional to $\text{deg}(p)$, by traversing p 's list of out-neighbors, identifying the neighbor's class S , removing it from S , and from the list of incoming connectors if it is an incoming connector, and putting it in one of two doubly-linked lists of founding sets for the subclasses of the class. Which founding set it goes into depends on whether the neighbor is already a known inward connector. This gives at most two founding sets for each partition class S : neighbors of p that were not already know inward connectors and neighbors of p that were not. The third founding set for S , the non-neighbors of p that were already known inward connectors of S , is what's left of the doubly-linked list of inward connectors after neighbors of p have been removed, and we have not touched them. We therefore spend $O(\text{deg}(p))$ time getting a doubly-linked list of the members of each founding set for all partition classes that don't contain p . If S has only one founding set, then it is unnecessary to split it to get it to continue to satisfy the pivot invariant, so this founding set is just restored as the known inward connectors of S , and no further work is done on it.

If S does not contain p and has more than one founding set, then it is split. When $z \in S$ is selected as a claim-staker, it traverses its list of out-neighbors, and ignores the neighbor if it is either marked or a member of a different class from S . Otherwise it marks it and moves it to its own class. We conclude that, during an outward pivot, we spend $O(\text{deg}(z))$ time for each member z of a class that is properly split by the pivot on p . A similar analysis applies to the inward pivot.

The insight that leads to the strategy for getting the $O(m \log n)$ bound is to suppose for the moment that every time a class S is split into a set of subclasses by a pivot on p , each of these side classes has at most half the non-marker degree sum of S , and that we select a vertex p as a pivot only if it is *ripe*, that is, only if the non-marker degree sum of the class that contains it is at most half that of the class that contained it the last time it was used as a pivot. Then, by the foregoing, we spend $O(\text{deg}(x))$ time on each vertex each time the degree sum of its class halves, for a total of $O(\text{deg}(x) \log m) = O(\text{deg}(x) \log n)$ over all uses of the vertex as a pivot or a claim staker. The total time over all vertices is $O(m \log n)$.

Let us now repair this strategy once we consider the possibility that if a class is split, one of the resulting subclasses it is split into may have greater than half

the non-marker degree sum of the original. We begin by bounding the cost of all calls to $S(a, b, G)$ over all recursive calls.

The cost of calls to $S(a, b, G)$. When vertices in up to three subclasses of a class S that is being split lay claim to vertices for their own class, we work on the subclasses in parallel, keeping the degree sum of the vertices that have made claims so far equal among the three calls. A class becomes *closed* if all of its members have attempted to lay claim to new members, and has no hope of capturing further members. When two classes become closed, we can put the remaining list of unclaimed members at the front of the list of the third class without touching them individually, and update the data structures for the three classes in time proportional to the time we've spent so far on the class. One of the other two subclasses has non-marker degree sum at most half of that of S , so we can charge all costs to the non-marker degree sum of this subclass, plus the degree of p . *We can still charge all costs to the degree sum of vertices that find themselves in classes whose non-marker degree sum has halved since the last time they were charged.*

We must also bound the cost of identifying ripe pivots. When we pivot on a vertex, we pivot on all vertices in the class. We label each class with the non-marker degree sum of the most recent class that contained its members. When we split a class, each of the subclasses inherits this label, and the ones whose current degree sum is half this label are put in a list of ripe classes.

A risk in adhering to the discipline of only pivoting on ripe vertices might cause the partition refinement to halt, due to the absence of ripe vertices, before the partition classes are split sets. The key insight is that if we ever run out of ripe vertices to pivot on, the class X with the largest non-marker degree sum is a split set. This is because every other partition class Y has at most the non-marker degree sum that X has, so Y has at most half the non-marker degree sum as the most recent partition class that contained both X and Y . Y has become ripe at some point since it was separated from X , and since it is not ripe, all of its vertices have been used as pivots since that time. All members of Y are finished outsiders of X . Applying this to all partition classes $Y \neq X$, we see that all members of $V \setminus X$ are finished outsiders of X . By the pivot invariant, X is a split set. We then pivot once on a connector 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 vertices, or else we may repeat the argument and remove another class X from consideration. Despite the new constraint, the procedure for finding $S(a, b, G)$ halts when we have found and removed every member of $\{\{a\}, \{b\}\} \cup S(a, b, G)$, and we get $S(a, b, G)$ in $O(m \log n)$ time.

It remains to show that the cost of calls to $S(a, b, G)$ over all recursive calls of the split decomposition algorithm is also $O(m \log n)$ time. Other than A and B , each recursive call is a member D of $S(a, b, G)$. In the recursive call on D , when we choose a new a and b , this creates two ripe vertices that can be used to re-start the partitioning on D without violating the constraint that we only pivot on ripe vertices. Unfortunately, A and B are each a union of members of

$S(a, b, G)$, each of which may have smaller non-marker degree sum than A or B . However, as depicted in Figure 5, the members of $S(a, b, G)$ that are adjacent to vertices on the subpath from a to u are just $S(a, u, G_A)$, where G_A is quotient passed to the recursive call on A . We avoid overcharging vertices in A for calls to $S()$ by passing in the members of $S(a, u, G_A)$, which are computed as a side-effect of finding $S(a, b, G)$, instead of making a call to find $S(a, u, G_A)$. This avoids the need to re-charge vertices in A before they become ripe again.

The cost of calls to $L(a, c, b, G)$ and $L(b, c, a, G)$. In a call $L(a, c, b, G)$, we keep only one of the partition classes that we would during a call to $S(a, c, G)$, namely, the one that contains b . We make use of a pivot p exactly once, when it is no longer in the partition class S that contains b . We ignore neighbors of p that are not in S , obtaining the founding sets of S , as before, in $O(deg(p))$ time. If S has only one founding set, we do no work on it, as before. Otherwise, since S is the only class of interest, we make use of a vertex to stake claims only if it is in S . If the subclass S' that contains b has more than half of the non-marker degree sum of S , then we charge the cost of the inward pivot to vertices in a different subclass. If it is less than half of the non-marker degree sum of S , we could charge the cost to the degree sum of S' , but since the degree sum of the other subclasses exceeds this, we can still charge it to vertices in other subclasses. The invariant this maintains is that at no point have we charged any costs to the degrees of vertices in the class that contains b . This gives the following:

Lemma 2. *If B is the set returned by $L(a, c, b, G)$, the cost of the call to $L(a, c, b, G)$ is the degree sum of $V \setminus B$.*

When it is time to call $L(a, c, b, G)$ and $L(b, c, a, G)$, we already know $S(a, b, G)$. If some element D of $S(a, b, G)$ has non-marker degree sum larger than that half that of G , we replace D with a marker before the calls to avoid charging elements internal to D . By the homomorphic rule and the fact that the sets returned by $L(a, c, b, G)$ and $L(b, c, a, G)$ are unions of sets in $\{\{a\}, \{b\}\} \cup S(a, b, G)$, this does not affect the result of the calls. Touching the marker for D can be charged to the recursive call on D , and since there are $O(n)$ recursive calls overall, this contributes $O(n) = O(m)$ to the total running time. The edges incident to the marker can be found by looking elements of D in all adjacency lists of vertices outside of G , and we are allowed $O(1)$ charges for each such adjacency-list element, since it is not in D .

If there is no such D , it may be that A or B , say, B has more than half the non-marker degree sum of G . If we knew this in advance, we could avoid charging to elements internal to B by running $L(a, c, b, G)$ to find B first, charging this to elements in $V \setminus B$. By Lemma 2, we could then replace B with a marker b' to create a quotient graph G' , and since A is disjoint from B , a call to $L(b', c, a, G')$ returns the same result as $L(b, c, a, G)$.

Unfortunately, we don't know in advance which of A and B might turn out to have more than half the degree sum of G . We therefore we run the calls $L(a, c, b, G)$ and $L(b, c, a, G)$ in parallel, keeping the degree sum of vertices used as pivots and claim-stakers equal. If $L(a, c, b, G)$ halts first, we have touched

all edges incident to vertices in $V \setminus B$ during the call, so we can subtract their non-marker degree sum from that of G to obtain that of B . If this is more than half the degree sum of G , then since both calls have spent the same amount of time, we can charge the cost of both calls so far to elements of $V \setminus B$, even if the call to $L(b, c, a, G)$ has touched elements internal to B . We then replace B with a marker, just as we did with D , above, before finishing out the call to $L(b, c, a, G)$. This prevents elements internal to B from getting charged.

Summarizing, every time a vertex or an element of its adjacency list is charged during a call to $L()$, the non-marker degree sum of the partition class that contains the vertex is at most half what it the previous time these elements were charged, and the charges pay for all costs of running the algorithm. The sum of these charges is $O(m \log m) = O(m \log n)$.

3 What Goes Wrong If G Is Not Strongly-Connected

One question that Cunningham did not dwell on is what goes wrong when a digraph is not strongly-connected. Using the existence of Cunningham's decomposition tree, we have shown that the maximal split sets that do not contain a or b is a partition of $V \setminus \{a, b\}$. Figure 6 shows that this is not the case when G is not strongly-connected, and therefore, the representation with the unique decomposition tree as we have described it does not exist. Cunningham's proof that it does exist makes use of the fact that, in a strongly-connected graph, for every nonempty proper subset S of V , there is an edge from an element of S to an element of $V \setminus S$, and this is not true for arbitrary directed graphs.

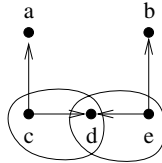


Fig. 6. A digraph that is not strongly-connected, and its maximal split sets that do not contain a or b

Nevertheless, a split is well-defined on an arbitrary directed graph, and the question of whether an arbitrary directed graph is prime is also well-defined. Spinrad gives an $O(n^2)$ algorithm for testing whether a strongly-connected graph is prime. As far as we know, there has been no work on the problem of determining whether an arbitrary directed graph is prime.

Acknowledgments

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

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Bouchet, A.: Digraph decompositions and eulerian systems. *SIAM Journal on Algebraic and Discrete Methods* 8 (1987)
3. Cunningham, W.H.: Decomposition of directed graphs. *SIAM J. Algebraic Discrete Methods* 3, 214–228 (1982)
4. Dahlhaus, E.: Parallel algorithms for hierarchical clustering, and applications to split decomposition and parity graph recognition. *Journal of Algorithms* 36, 205–240 (2000)
5. Gabor, C.P., Supowit, K.J., Hsu, W.-L.: Recognizing circle graphs in polynomial time. *Journal of the ACM* 36, 435–473 (1989)
6. Ma, T.H., Spinrad, J.: An $O(n^2)$ algorithm for undirected split decomposition. *Journal of Algorithms* 16, 145–160 (1994)
7. Möhring, R.H.: Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operations Research* 4, 195–225 (1985)
8. Möhring, R.H., Radermacher, F.J.: Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics* 19, 257–356 (1984)
9. Rao, M.: Solving some NP-complete problems using split decomposition. *Discrete Applied Mathematics* (2007)
10. Spinrad, J.: Prime testing for the split decomposition of a graph. *SIAM Journal on Discrete Mathematics* 2, 590–599 (1989)
11. Spinrad, J.P.: Two Dimensional Partial Orders. Ph.D thesis, Princeton University (1982)
12. Spinrad, J.P.: Graph partitioning (1985) (unpublished manuscript)
13. Spinrad, J.P.: On comparability and permutation graphs. *Siam J. Comput.* 14, 658–670 (1985)
14. Spinrad, J.P., Valdes, J.: Recognition and isomorphism of two-dimensional partial orders. In: *Proceedings of the 10th Colloquium on Automata, Languages and Programming*. LNCS, pp. 676–686. Springer, Berlin (1983)