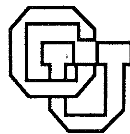
Applications of a Poset Representation to Edge Connectivity and Graph Rigidity

Harold N. Gabow

CU-CS-545-91 September 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

Applications of a Poset Representation to Edge Connectivity and Graph Rigidity

Harold N. Gabow

CU-CS-545-91 September 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax

Applications of a Poset Representation to Edge Connectivity and Graph Rigidity

Harold N. Gabow*

Department of Computer Science

University of Colorado at Boulder

Boulder, CO 80309

hal@cs.colorado.edu

Abstract. This paper investigates a poset representation for a family of sets defined by a "labelling algorithm." It gives poset representations for the family of minimum cuts of a graph, and shows how to compute them quickly. For undirected graphs a representation was previously known but we find it faster. For directed graphs we know of no previous compact representation. The representations are the starting point for algorithms that increase the edge connectivity of a graph, from λ to a given target $\tau = \lambda + \delta$, adding the fewest edges possible. For undirected graphs the time bound is essentially the best-known bound to test if a graph is τ-edge-connected; for directed graphs the time bound is roughly a factor δ more. We also construct poset representations for the family of rigid subgraphs of a graph, when graphs model structures constructed from rigid bars. The link between these problems is that they all deal with graphic matroids.

---

* Research supported in part by NSF Grant No. CCR-8815636.

1. Introduction.

Algorithms for optimization problems like network flow, matroid intersection and many others are often based on a labelling procedure. The labelling procedure can often be adapted to carry out sensitivity analysis, or even give a representation of all solutions, often as a poset. This paper investigates the poset representation for problems related to graphic matroids. We obtain representations for the family of minimum cuts of a graph. We use these representations as a starting point to design efficient algorithms that increase the edge connectivity of a graph. We also construct representations for the family of rigid subgraphs of a graph, when graphs model structures constructed from rigid bars.

To elaborate, consider a family $\mathcal{F}$ of sets such that each element is in a unique minimal set, i.e., each element e is in a set $M(e)$ of $\mathcal{F}$ such that any set of $\mathcal{F}$ containing e contains $M(e)$. An obvious example of $\mathcal{F}$ is a family closed under intersection. In general, any set of $\mathcal{F}$ is a union of sets $M(e)$, and the sets $M(e)$ can be represented as follows. For any element e let $[e]$ be the set of all elements f with $M(e) = M(f)$. Define a partial order P whose elements are the sets $[e]$, and $[f]$ is a successor of $[e]$ if $M(f) \subseteq M(e)$. Thus $M(e)$ consists of all elements in successors of $[e]$. P is called the *poset representation of* $\mathcal{F}$. For many applications it is unnecessary for P to have all transitive edges. To emphasize that all transitive edges are explicitly included we write P^c; P^r denotes the transitively reducted poset; P denotes any relation whose transitive closure is P^c. If the family of sets is not clear we write it as an argument, e.g., $P^c(\mathcal{F})$.

The poset representation is precisely the minimal difference poset in [GI], used to represent all stable marriages. In network flow, it is the representation of Picard and Queyranne for all minimum cuts of a flow network [PQ]; it is the Dulmage-Mendelsohn representation for maximum bipartite matchings [DM]. Nakamura [N] proposes a variant as a representation of all rigid subgraphs of a bar-and-joint structure. It is closely related to various notions of the principal partition of a matroid or polymatroid [NakI]. Section 3 suggests that the source of this representation may be

broader – graph and network labelling algorithms often induce such posets.

Consider first edge connectivity. Fix a vertex a. An *a-set* is a nonempty set of vertices not containing a. An *a-cut* is the set of edges directed from $V - S$ to S, for an a-set S. The minimum number of edges in an a-cut is denoted λ_a. An a-cut of λ_a edges is an *a-mincut*; in this case the above set S is an *a-minsink*.

The *edge connectivity* of a directed graph G, denoted $\lambda(G)$ or λ, is the smallest value of λ_a when a ranges over all vertices. Such a set of λ edges is a *mincut*. For any a, an a-mincut in G or in G with all edges reversed is a mincut. The edge connectivity of an undirected graph is the same as that of the corresponding directed graph having each edge oriented in both directions. We often refer to a *spanning tree* of a directed graph – for this, ignore edge directions.

Edmonds established the relation between edge connectivity and graphic matroids: Fix a vertex a and integer k. A set of $k(n-1)$ edges T is a *complete k-intersection (for a, on G)* if it contains precisely k edges directed to each vertex except a, and it can be partitioned into k spanning trees. (T is a maximum cardinality matroid intersection of two matroids, one of which is derived from the graphic matroid of G.) *Edmonds' Theorem* states that for any a, the largest k for which G has a complete k-intersection for a is λ_a [E69, E72]. This relation is the basis of an algorithm that finds an a-mincut in time $O(\lambda m \log(n^2/m))$, the best-known bound [G91].

We investigate the poset representation for a-minsinks. We show that in an undirected graph the poset explicitly gives *all* the mincuts. Figure 1 shows a 2-edge connected graph and the corresponding poset for the a-minsinks (the figure is discussed fully in Section 3). Another representation of all mincuts is known, the cactus representation of Dinits, Karzanov and Lomosonov [DKL]. Our poset is either a tree or more generally a graph we call an m-tree, which allows limited sharing of children. The m-tree is easily converted to the cactus representation, although our applications work equally well with either. Our algorithm finds the m-tree representation in time $O(m + \lambda^2 n \log(n/\lambda))$. This improves the algorithm of [KT] that constructs the cactus representation

in time $O(\lambda n^2)$ (this bound is never smaller than ours since $\lambda \leq n$).

The poset representation also gives a representation of the a-minsinks of a directed graph. We know of no previous compact representation, nor any striking properties of the poset. We construct this poset in time $O(\lambda m \log(n^2/m) + n^2)$. However various properties of the poset can be determined in time $O(\lambda m \log(n^2/m))$ and this suffices for the applications below.

The representation of all a-minsinks provides the starting point for our algorithms to increase edge connectivity. In the *edge connectivity augmentation problem*, we are given a graph G with edge connectivity λ. The task is to increase the edge connectivity by some positive quantity δ to a target connectivity $\tau = \lambda + \delta$, adding as few edges as possible. The solution graph is allowed to have parallel edges.

We present efficient algorithms for directed and undirected graphs. The high-level approach of both algorithms is that of Naor, Gusfield and Martel [NGM] for undirected graphs: repeatedly increase the connectivity by one, using a good representation of all mincuts. To do this we solve the $\delta = 1$ augmentation problem for directed graphs in time $O(\lambda m \log(n^2/m))$. This problem is easier for undirected graphs – [NGM] shows it can be done by a depth-first traversal of the representation of all mincuts. Thus our poset representation solves the undirected problem in time $O(m + \lambda^2 n \log(n/\lambda))$. These two bounds for the $\delta = 1$ augmentation problem are precisely the best-known bounds to determine the edge-connectivity λ [G91].

Naor et. al. solve the general augmentation problem for undirected graphs in time $O(\delta^2 nm + n^{5/3}m)$ [NGM]. We improve this to $O(m + \tau^2 n \log n)$, close to the best-known bound to test τ-edge-connectivity. Frank solves the augmentation problem for directed multigraphs in time $O(n^5)$ [F] (this allows edges with arbitrarily large multiplicities). Our algorithm for directed graphs uses time $O(\delta\tau(m + \delta n)\log^2 n)$ and space $O(m + \delta\tau n)$ space. For example when $\delta = O(1)$ this is within a logarithmic factor of the best-known bound to test τ-edge-connectivity. A more space-efficient implementation uses time $O(\delta^2\tau(m + \delta n)\log^2 n)$ and space $O(m + \delta n)$. Both [F] and [NGM] use

network flow as the main technical tool; we use the edge connectivity algorithm of [G91].

Our poset representation algorithm is stated as a combination of depth-first search plus the labelling algorithm for the problem at hand. Our specific implementation is for labelling algorithms related to graphic matroids, like the minsink poset above. We obtain efficient poset representation algorithms for other problems related to graphic matroids.

Specifically the poset representation is important in the study of graph rigidity. A graph can represent a bar-and-body framework or a bar-and-joint framework – each is a physical structure connected together by rigid bars that end in universal joints. We wish to decide the rigidity properties of the structure. (For a textbook treatment see [R].) For bar-and-body frameworks, White and Whitely [WW] show several rigidity properties are revealed by our poset representation for all minimally rigid subgraphs. Nakamura suggests the poset representation for the minimally rigid subgraphs of a bar-and-joint framework. For both types of frameworks, we construct the poset representation of an n vertex graph in time $O(n^2)$.

The rest of this paper is organized as follows. Section 2 gives an algorithm to increment the connectivity of a directed graph by one. It assumes certain characteristics of the poset representation of all minsinks can be found efficiently. Sections 3–5 discuss the poset representation, with emphasis on families of sets related to graphic matroids. Section 3 gives an algorithm that finds the nodes of the poset representation, in general and for the minsink poset. This completes the algorithm of Section 2. Section 4 discusses combinatoric properties of posets related to graphic matroids, in particular the minsink poset for undirected graphs. Section 5 presents algorithms to construct the complete poset, including the minsink posets and poset representations for graph rigidity. Section 6 solves the edge connectivity augmentation problem for undirected graphs. Section 7 solves this problem for directed graphs, drawing on ideas from Sections 2 and 6. The rest of this section gives notation, definitions and some related background.

Consider sets S and T in a universe U. If e is an element then $S+e$ denotes $S \cup \{e\}$ and $S-e$

denotes $S - \{e\}$. For integers i and j, $[i..j] = \{k \mid k \text{ is an integer}, i \leq k \leq j\}$; similarly for $(i..j]$, etc. We use both set containment $S \subseteq T$ and proper set containment $S \subset T$. Sets S and T *nest* if $S \subseteq T$ or $T \subseteq S$. The sets *meet* if $S \cap T$ is nonempty. The sets *cross* if each set $S - T$, $T - S$, $S \cap T$ and $U - S - T$ is nonempty. A *subpartition* of a set S is a collection of disjoint subsets of S.

For any graph G, $V(G)$ and $E(G)$ denote the vertex set and edge set of G, respectively. V and E denote these sets for the input graph of a problem, and $n = |V|$, $m = |E|$. For vertices v and w, the notation vw denotes an undirected edge joining v and w, or a directed edge from v to w; it will be clear from context which is meant. An undirected graph G has an associated directed graph $D(G)$, which contains each edge of G in both directions.

Consider a directed graph. For $S \subseteq V$, $\rho(S)$ denotes the set of all edges directed from $V - S$ to S. If the graph H is not clear from context we write $\rho_H(S)$. The function δ is analogous to ρ for edges directed from a set; thus $\delta(S) = \rho(V - S)$. For any sets of vertices X and Y, $|\rho(X)| + |\rho(Y)| \geq |\rho(X \cap Y)| + |\rho(X \cup Y)|$; a similar inequality holds if each ρ is changed to δ. These two inequalities are referred to as *submodularity.*

Similarly consider an undirected graph. For $S \subseteq V$, $d(S)$ denotes the set of all edges joining $V - S$ and S; similarly $d_H(S)$ refers to graph H. In addition to the submodular identity d satisfies the following inequality: For any sets of vertices X and Y, $|d(X)|+|d(Y)| \geq |d(X-Y)|+|d(Y-X)|$.

We summarize the *round robin connectivity algorithm* [G91]. Given a directed graph and a vertex a, it computes λ_a by finding a complete k-intersection for a, where k takes on successive values $1, \ldots, \lambda_a$. (The algorithm determines the value λ_a by finding that no complete $(\lambda_a + 1)$-intersection exists.) Each complete k-intersection is found by executing the *round robin algorithm.* Its input is a complete $(k-1)$-intersection, partitioned into $k-1$ spanning trees; its output is a similar partitioned complete k-intersection, if such exists. The time for round robin is $O(m \log n)$ on a multigraph and $O(m \log (n^2/m))$ on a graph. The time to compute the edge connectivity λ is $O(\lambda m \log n)$ on a multidigraph, $O(\lambda m \log (n^2/m))$ on a directed graph. (Actually λ is found

by using the above connectivity algorithm to compute two values λ_a.) For an undirected graph a preprocessing step reduces the time to compute the edge connectivity to $O(m + \lambda^2 n \log(n/\lambda))$.

We do not use results from matroid theory although there is a close relationship. For instance a set of edges that can be partitioned into k forests is precisely an independent set in $\mathcal{G}^k$, the matroid sum of k copies of the graphic matroid of G. This notion appears in Edmonds' Theorem and is used throughout this paper.

2. Incrementing directed edge connectivity.

In this section we are given a directed graph G with edge connectivity λ. We wish to increase the connectivity by one, adding as few edges as possible. We present an algorithm that combines ideas of [F] and [NGM].

Frank's solution to the general edge connectivity augmentation problem specializes to the following result for $\delta = 1$: The minimum number of edges needed to increase the edge connectivity by one is precisely the maximum number of pairwise-disjoint sets that are all minsources or all minsinks. We begin with several lemmas that provide an alternate proof of this result (essentially Lemma 2.4). In addition the lemmas form the basis of our algorithm.

Define a *cover* to consist of two sets $\mathcal{X}$ and $\mathcal{Y}$ where $\mathcal{X}$ meets every minsource and $\mathcal{Y}$ meets every minsink. The first lemma is the basis of an algorithm that uses a cover to increase the edge connectivity by one. It needs one more concept. Given a cover, for each vertex $y \in \mathcal{Y}$ define $Y(y)$ as the maximal minsink satisfying $Y(y) \cap \mathcal{Y} = \{y\}$; if a unique such set does not exist then $Y(y) = \emptyset$. If $|\mathcal{Y}| = 1$ we may have $Y(y) = \emptyset$ because there are many such sets; we will only use $Y(y)$ when $|\mathcal{Y}| > 1$. In that case, if some minsink S satisfies $S \cap \mathcal{Y} = \{y\}$ then set $Y(y)$ is well-defined. This follows because the family of minsinks S satisfying $S \cap \mathcal{Y} = \{y\}$ is closed under union. (This is proved using submodularity and the assumption $|\mathcal{Y}| > 1$.) Similarly define $X(x)$ for $x \in \mathcal{X}$.

Lemma 2.1. Let $\mathcal{X}$, $\mathcal{Y}$ be a cover.

(i) Adding all edges xy for $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ gives a graph with connectivity $\lambda + 1$.

(ii) Let $|\mathcal{X}|$, $|\mathcal{Y}| > 1$. Adding an edge xy with $x \in \mathcal{X} - Y(y)$ and $y \in \mathcal{Y} - X(x)$ gives a graph with cover $\mathcal{X} - x$, $\mathcal{Y} - y$.

Proof. (i) It suffices to show that for each minsink S of G, $\rho(S)$ contains an added edge. Clearly S contains some $y \in \mathcal{Y}$. Since $V - S$ is a minsource, it contains some $x \in \mathcal{X}$. Hence edge $xy \in \rho(S)$.

(ii) Let S be a minsink of G with $S \cap \mathcal{Y} = \{y\}$. As noted above $S \subseteq Y(y)$. Thus $x \notin S$ and S is not a minsink after xy is added. (This argument is vacuous if $Y(y) = \emptyset$.) ∎

The lemma motivates the following *basic procedure*. Given a cover $\mathcal{X}$, $\mathcal{Y}$, it increases the connectivity by one by adding $\max\{|\mathcal{X}|, |\mathcal{Y}|\}$ edges: While both $|\mathcal{X}|$ and $|\mathcal{Y}|$ are larger than 1, apply Lemma 2.1(ii). When $|\mathcal{X}|$ or $|\mathcal{Y}|$ equals 1, apply Lemma 2.1(i).

We show the basic procedure is correct by proving that vertices x, y of Lemma 2.1(ii) always exist. The main reason is the following property.

Lemma 2.2. Let $\mathcal{X}$, $\mathcal{Y}$ be a cover with $|\mathcal{X}|, |\mathcal{Y}| > 1$. For any vertex $y \in \mathcal{Y}$, $y \in X(x)$ implies $X(x)$ and $Y(y)$ nest. Similarly for any $x \in \mathcal{X}$.

Proof. Suppose $y \in X(x)$. We show that if $X(x)$ and $Y(y)$ do not nest then

$$|\rho(Y(y) - X(x))| = \lambda. \tag{1}$$

But this means $Y(y) - X(x)$ is a minsink disjoint from $\mathcal{Y}$, a contradiction.

First suppose that $X(x)$ and $Y(y)$ cross. Thus $X(x)$ and $V - Y(y)$ are two minsources that cross. Submodularity implies their union is a minsource. This is equivalent to (1).

Next suppose that $X(x) \cup Y(y) = V$. Then (1) follows since $X(x)$ is a minsource. ∎

Now we show the basic procedure is correct. It suffices to show that the vertices x and y of Lemma 2.1(ii) always exist. Choose any $x \in \mathcal{X}$. Since $X(x)$ is a minsource, its complement is a

minsink, whence there is some $y \in \mathcal{Y} - X(x)$. (The conclusion is true even if $X(x) = \emptyset$.) If $x \notin Y(y)$ then x, y is the desired pair. Otherwise $x \in Y(y)$. There is some $x' \in \mathcal{X} - Y(y)$. Clearly $y \notin X(x')$ since otherwise Lemma 2.2 shows $x \in Y(y) \subseteq X(x')$, contradicting the definition of $X(x')$. Thus the basic procedure is correct.

We will find the smallest number of edges to add by finding a smallest cover. Using terminology similar to [NGM], an *in-extreme set* is a set S that is a minimal minsink, i.e., $|\rho(S)| = \lambda$ but no proper subset satisfies this condition. Similarly for *out-extreme set*; *extreme set* refers to either. (In Sections 6–7 these terms have a more general meaning.) An *a-minsource* is an a-set S that has $|\delta(S)|$ minimum. A *minsource* is a set of vertices S that has $|\delta(S)| = \lambda$. Clearly in the definition of cover we can replace minsinks by in-extreme sets and minsources by out-extreme sets.

Lemma 2.3. Either (i) all in-extreme sets are pairwise disjoint, or (ii) any two in-extreme sets meet and cover V. Similarly for out-extreme sets.

Proof. First observe that if S and T are in-extreme sets that meet then $S \cup T = V$. For S and T do not nest, by definition of in-extreme. They do not cross, since submodularity would imply that $S \cap T$ is in-extreme, a contradiction. The only remaining possibility is $S \cup T = V$.

Next observe that if an in-extreme set S meets another in-extreme set T then it meets every in-extreme set. For if U is in-extreme and disjoint from S then $U \subset T$, a contradiction. ∎

Let a_y be a vertex in some in-extreme set. Form the set $\mathcal{Y}$ by including a_y plus an arbitrarily chosen vertex from each in-extreme a_y-set. (Note that Lemma 2.3 shows for any vertex a, the a-sets that are in-extreme are pairwise disjoint. Possibly there are none.) Similarly form $\mathcal{X}$, with vertex a_x, for out-extreme sets. Clearly $\mathcal{X}, \mathcal{Y}$ is a cover; call it an *extreme-set cover.*

Lemma 2.4. For any extreme set cover $\mathcal{X}, \mathcal{Y}$, $\max\{|\mathcal{X}|, |\mathcal{Y}|\}$ is precisely the minimum number of edges needed to increase the connectivity by one.

Proof. By the basic procedure and symmetry it suffices to show that $|\mathcal{Y}|$ edges are needed. This is obvious if Lemma 2.3(i) holds. Suppose Lemma 2.3(ii) holds. This implies $|\mathcal{Y}| \leq 2$. Clearly we can assume $|\mathcal{Y}| = 2$. Let S and T be two in-extreme sets corresponding to $\mathcal{Y}$. Sets $S-T$ and $T-S$ are disjoint minsources. Thus at least two edges are needed to increase the connectivity. ∎

We have shown that the basic procedure, executed on an extreme-set cover, solves our problem. We turn to making this algorithm efficient. We wish to avoid calculating the sets $X(x)$, $Y(y)$. We do this by maintaining a partition of each set $\mathcal{X} - a_x$ and $\mathcal{Y} - a_y$. We now define the partition of $\mathcal{X} - a_x$; that of $\mathcal{Y} - a_y$ is analogous. Say that vertex x *elicits* vertex y if any a_y-minsink containing x contains y. Each vertex $y \in \mathcal{Y} - a_y$ has an associated set $E(y)$. The sets $E(y)$ form a partition of $\mathcal{X} - a_x$ such that any $x \in E(y)$ elicits y.

The key property of the partition is that $Y(y) \subseteq E(y)$. This follows since $Y(y)$ is an a_y-minsink. Hence if $x \in Y(y) \cap E(y')$ then $y' \in Y(y)$. Since $Y(y) \cap \mathcal{Y} = \{y\}$ this implies $y' = y$ as desired. Note that in this argument all sets refer to the current graph.

This gives a simple rule for choosing vertices x and y as required by Lemma 2.1(ii): Choose x and y so that $x \notin E(y)$ and $y \notin E(x)$. Clearly this implies $x \notin Y(y)$ and $y \notin X(x)$ as desired.

The main tool to implement the algorithm is the poset representation P of all a-minsinks. It is easy to see that the in-extreme sets not containing a are precisely the sinks of P. Now we present the entire algorithm.

Initialize Step. Find an in-extreme set and choose a vertex a_y in it. Let P_y be the poset for the a_y-minsinks. Form $\mathcal{Y}$ as a_y plus a vertex from each sink of P_y. Similarly define a_x, P_x and $\mathcal{X}$.

Partition $\mathcal{X} - a_x$: Place each $x \in \mathcal{X} - a_x$ in a set $E(y)$ where $y \in \mathcal{Y}$ and $[y]$ is a successor of $[x]$. (If x is not in any a_y-minsink then place x in an arbitrary set.) Similarly partition $\mathcal{Y} - a_y$.

Add_edges Step. Repeatedly execute the case below that applies until the Low Case halts.

High Case $\min\{|\mathcal{X}|,|\mathcal{Y}|\} \geq 4$. Choose vertices x and y so that $x \in E(y')$ for some $y' \neq y$ and $y \in E(x')$ for some $x' \neq x$. Add edge xy to the graph. Remove x from $\mathcal{X}$ and $E(y')$; remove y from $\mathcal{Y}$ and $E(x')$. Merge $E(y)$ into $E(y')$ and $E(x)$ into $E(x')$.

Middle Case $\min\{|\mathcal{X}|,|\mathcal{Y}|\} \in [2..3]$. Find all sets $X(x), x \in \mathcal{X}$ and $Y(y), y \in \mathcal{Y}$. Choose vertices x and y so that $x \in \mathcal{X} - Y(y)$ and $y \in \mathcal{Y} - X(x)$. Add edge xy to the graph. Remove x from $\mathcal{X}$ and y from $\mathcal{Y}$.

Low Case $\min\{|\mathcal{X}|,|\mathcal{Y}|\} = 1$. Add all possible edges from a vertex of $\mathcal{X}$ to a vertex of $\mathcal{Y}$. Halt. ∎

Correctness follows from the preceding discussion plus the fact that the High Case updates the partition correctly. To prove this fact consider any $z \in E(x)$. Let S be an a_x-minsource containing z. Since z elicits x, $x \in S$. If S is a minsource after xy is added then $y \in S$. Since y elicits x', $x' \in S$ as desired.

The efficiency depends on further details of the implementation. We first discuss the High Case. The partition sets $E(x)$ and $E(y)$ are maintained using an algorithm for disjoint set merging. Observe that the desired vertices x and y can be found by considering 3 arbitrary vertices of $\mathcal{X} - a_x$ and 3 arbitrary vertices of $\mathcal{Y} - a_y$. These vertices form 9 pairs x, y, of which 6 can be disqualified by relations $x \in E(y)$. Hence the desired pair can be found among the chosen vertices. Thus it is easy to see that the total time for the High Case is the time for $O(n)$ *unions* and $O(n)$ *finds*, which is $O(n\alpha(n,n))$. (Alternatively there is a simple algorithm that represents each set by a linear list and uses total time $O(n)$. However this improvement is asymptotically irrelevant.)

The remaining details of the algorithm are implemented using the posets P_x and P_y. We do not construct these orders explicitly. Instead we rely on three operations on each poset representation P:

(i) Find the subpartition of $V(G)$ into the nodes of P.

(ii) For each node of P, find a successor that is a sink.

(iii) Find all nodes of P having only one successor that is a sink.

Note that operation (ii) identifies all the sinks. It is clear that operations (i) and (ii) suffice to implement the Initialize Step. We need only show that operation (iii) suffices to implement the Middle Case.

Consider an execution of the Middle Case. We show that the algorithm preserves the following relation between $\mathcal{Y}$ and the minsinks of the current graph. (A similar relation holds for sources, we prove only the sink case.) Let P_y be the poset for a_y-minsinks, in the current graph.

(a) Each sink of P_y contains a vertex of $\mathcal{Y} - a_y$. Each vertex of $\mathcal{Y} - a_y$ is in a sink of P_y.

The first statement follows since $\mathcal{Y}$ is a cover. For the second statement it suffices to assume that the algorithm has added at least one edge. This implies that the initial graph has $|\mathcal{Y}| \geq 3$. Lemma 2.3 shows that all in-extreme sets are pairwise disjoint. Thus an in-extreme set containing a vertex $y \in \mathcal{Y}$ remains in-extreme until an edge directed to y is added.

Property (a) motivates how the Middle Case finds all sets $Y(y)$, $y \in \mathcal{Y} - a_y$: For each $y \in \mathcal{Y} - a_y$ let $Y(y)$ consist of all vertices v such that $[v]$ has $[y]$ as its unique sink successor. Do this using operation (iii) on the poset P_y of a_y-minsinks for the current graph.

To prove this is correct, property (a) shows that the set $Y(y)$ constructed is the union of all minsinks $M(v)$ having $M(v) \cap \mathcal{Y} = \{y\}$. This is the desired set $Y(y)$ (if a minsink S has $S \cap \mathcal{Y} = \{y\}$, any $v \in S$ has $M(v)$ included in the above union).

We now give the last details of the Middle Case. For simplicity we implement the Middle Case so the algorithm works with only two posets – for the a_x-minsinks and a_y-minsinks. The Middle Case finds all sets $X(x)$, $x \in \mathcal{X} - a_x$ and $Y(y)$, $y \in \mathcal{Y} - a_y$. Choose an $x \in \mathcal{X} - a_x$ and $y \in \mathcal{Y} - a_y$. If edge xy cannot be added then by symmetry we can assume $x \in Y(y)$ and $X(x) \subseteq Y(y)$. If $|\mathcal{Y}| = 2$

then choose any $x' \in \mathcal{X} - Y(y)$, as in the discussion after Lemma 2.2. If $|\mathcal{Y}| \geq 3$ we use a different procedure (since the previous procedure may choose $x' = a_x$). Clearly the sets $Y(y)$ are disjoint. Thus we can choose x plus any $y' \in \mathcal{Y} - \{a_y, y\}$.

Section 3 shows that each of the operations (i) – (iii) can be done in $O(m)$ time, given a complete λ-intersection. Such an intersection is found in time $O(\lambda m \log(n^2/m))$ using the round robin connectivity algorithm of [G91]. (Note that in the Middle Case, although the graph changes after the Initialize Step there is no need to recompute the complete λ-intersection – it is still valid. This also depends on the fact that a_x and a_y do not change.) Since the Middle Case is executed at most twice it uses total time $O(m)$.

Theorem 2.1. The directed edge connectivity augmentation problem for $\delta = 1$ can be solved in time $O(\lambda m \log(n^2/m))$ and space $O(m)$. ∎

3. The node-finding algorithm.

This section gives an algorithm that finds the nodes $[e]$ of a poset representation. The node-finding algorithm is stated in general, and a specific implementation is given for the minsink poset. Extensions of the algorithm determine other properties of the poset, e.g., those needed for the algorithm of Section 2. We begin by characterizing the minsinks.

Consider a directed graph G. Fix a vertex a. Let T be a complete λ_a-intersection for a on G (recall Edmonds' Theorem, Section 1). A set of vertices S is *cut by* T if $\rho(S) \subseteq T$ and *cospanned by* T if each tree of T contains a spanning tree of S. (Note that a single vertex is cospanned by T.) The following result is implicit in [G91].

Lemma 3.1. The a-minsinks are precisely the a-sets that are cut and cospanned by T.

Proof. Let S be an a-set. Let $|S| = s$ and let T have t edges with both ends in S. Since T consists

of λ_a trees, $t \leq \lambda_a(s-1)$. Thus $|\rho_T(S)| \geq \lambda_a s - \lambda_a(s-1) = \lambda_a$. This implies that S is an a-minsink if and only if $\rho_T(S) = \rho(S)$ and $t = \lambda_a(s-1)$, i.e., S is cut and cospanned by T. ∎

Let $\mathcal{F}$ denote the family of sets that are cut and cospanned by T; let $\mathcal{F}_a$ denote the restriction of $\mathcal{F}$ to a-sets. It is easy to see that both families are closed under intersection. Thus as discussed in Section 1, both families have poset representations. (In contrast, the family of all minsinks does not have such a poset representation. For example in Figure 1.a there are two minsinks containing vertex d that are minimal, $\{a, b, \ldots, h\}$ and $\{d, i, j, \ldots, n\}$.) The a-minsinks are represented by $P(\mathcal{F}_a)$ (by Lemma 3.1). It is convenient to construct $P(\mathcal{F})$ first and use it to derive $P(\mathcal{F}_a)$. Thus we begin by letting P denote $P(\mathcal{F})$. Also let k denote λ_a, so T consists of trees T_i, $i = 1, \ldots, k$.

Let us review and extend the notation of Section 1. For a vertex v, $M(v)$ denotes the smallest set of $\mathcal{F}$ that contains v. (This always exists since $V(G) \in \mathcal{F}$.) Extend this notation so that for an edge e, $M(e)$ denotes the smallest set of $\mathcal{F}$ that contains both ends of e.

The sets of $\mathcal{F}$ containing at least two vertices are precisely the sets that can be written as $\bigcup\{M(e) \mid e \in R\}$ where the set of edges R forms a subtree of T_1.

Several of our applications use a representation of sets $M(v)$ for vertices v. These sets are easily defined in terms of sets $M(e)$: if $|\rho(v)| = k$ then $M(v) = \{v\}$; otherwise $M(v) = M(e)$ for any edge $e \in \rho(v) - T$. However rather than treat sets $M(v)$ as a special case, we will use the following transformation. Given graph G, form multigraph $\overline{G}$ by adding for each vertex v of in-degree k a new vertex $\overline{v}$, along with $k+1$ edges $v\overline{v}$ and $k+1$ edges $\overline{v}v$. There is a natural one-to-many correspondence between complete k-intersections T on G and complete k-intersections $\overline{T}$ on $\overline{G}$. For corresponding intersections T and $\overline{T}$, there is a one-to-one correpondence between the sets of $\mathcal{F}$ for T and $\overline{T}$, in which a set $M(v)$ for $v \in V(G)$ corresponds to $M(v\overline{v})$ in $\overline{T}$. Furthermore $\overline{G}$ is undirected if G is. Thus in the remainder of this section, unless noted otherwise, the poset P refers to the representation for sets $M(e)$, e an edge.

We turn to constructing the poset representation. We give a general algorithm, illustrating it for the minsink poset. As in Section 1 consider a family of sets over a universe E, such that each $e \in E$ is in a unique minimal set $M(e)$. The sets $M(e)$ can be specified by a function $\mathcal{L} : E \rightarrow 2^E$ that gives the relations

$$M(e) = \{e\} \cup \bigcup \{M(f) \mid f \in \mathcal{L}(e)\}. \tag{1}$$

A trivial choice of $\mathcal{L}$ is given by $\mathcal{L}(e) = M(e)$. We interpret (1) as defining the family $M(e)$, i.e., sets $M(e)$ are the smallest sets satisfying (1), and they are unique.

For many combinatorial problems the function $\mathcal{L}$ is given by an efficient labelling algorithm. For instance to represent all minimum cuts of a flow network, we use the standard Ford-Fulkerson labelling algorithm for maximum value network flow [PQ].

Given $\mathcal{L}$, define a directed graph L having vertex set E, and edges ef for $f \in \mathcal{L}(e)$. Thus $M(e)$ consists of all the successors of e in L. Let P denote the graph L with each strongly connected component contracted to a node. It is easy to see that the nodes of P are precisely the sets $[e]$ and P is the poset representation of the given family of sets.

For the set families of this paper P can be constructed without generating all of L. We adapt Tarjan's strong connected components algorithm [Tar72] (unlike other algorithms [see CLR] Tarjan's uses adjacency lists in only one direction, which corresponds to the information given by $\mathcal{L}$). We briefly summarize the relevant part of the algorithm. For $e \in V(L)$ recall that $[e]$ denotes the strong component containing e. The algorithm does a depth-first search, building up the sets $[e]$ and outputting each one as it is completed. At any time during the algorithm let $\lfloor e \rfloor$ denote the subset of $[e]$ containing e that the algorithm has built up. We describe the algorithm in terms of a stack that is a variant of the one used in [Tar72]. At any time denote the stack as the sequence of vertices $e_1, \ldots, e_s$ (where s is the top of the stack). $\bigcup_1^s \lfloor e_i \rfloor$ contains precisely the vertices e that have been reached in the depth-first search but $[e]$ has not been output; furthermore L has a path

containing vertices in $\lfloor e_i \rfloor$, $i = 1, \ldots, s$ in that order (more precisely the path contains a positive number of vertices in $\lfloor e_1 \rfloor$ followed by a positive number of vertices in $\lfloor e_2 \rfloor$, etc.)

The algorithm scans an edge ef where $e \in \lfloor e_s \rfloor$. If f has not yet been reached in the search it is added to the depth-first spanning forest, also $s \leftarrow s+1$ and $e_s \leftarrow f$. If f has been reached then nothing is done if $\lfloor f \rfloor$ equals $[f]$ or $\lfloor e \rfloor$; otherwise for r such that $f \in \lfloor e_r \rfloor$ the algorithm merges $\lfloor e_i \rfloor$, $i = r, \ldots, s$ into one and sets $s \leftarrow r$. (This makes $\lfloor e \rfloor = \lfloor f \rfloor$.) If every edge from a vertex in $\lfloor e_s \rfloor$ has been scanned, the algorithm outputs $\lfloor e_s \rfloor = [e_s]$, sets $s \leftarrow s-1$ and continues.

Recall also that this search amounts to a depth-first search of P. A set $[e]$ is output when it is postvisited [Tar83]. The total time is linear in the size of L.

For the problems involving graphic matroids treated in this paper, the labelling functions are adapted from the labelling algorithm for the graphic matroid partitioning algorithm [R]. An efficient labelling algorithm is the *cyclic scanning rule* [RT, GW]: Let the complete k-intersection T consist of trees T_i, $i = 0, \ldots, k-1$. For $e \in T_i$, let $\mathcal{L}(e)$ be the fundamental cycle of e in T_{i+1}, where addition is modulo k. Using this function $\mathcal{L}$ in (1) defines sets $M(e)$ as the smallest set containing both ends of e that is cospanned by T.

We extend this function $\mathcal{L}$ to give the smallest set that is both cut and cospanned by T, i.e., the desired set $M(e)$. For a directed edge vw define $\mathcal{L}'(vw) = \rho(v) \cup \rho(w) - T$. Then the labelling function $\mathcal{L}_c(e)$ is given as follows: if $e \in T$ then $\mathcal{L}_c(e) = \mathcal{L}(e) \cup \mathcal{L}'(e)$; if $e \notin T$ then $\mathcal{L}_c(e)$ is the fundamental cycle of e in T_j, where j can be chosen arbitrarily. It is not hard to show that the sets $M(e)$ are correctly defined by using $\mathcal{L}_c$ in (1). More precisely (1) specifies $V(M(e))$ as the smallest set of $\mathcal{F}$ that contains both ends of e. This is easy to verify using [G91].

Note that the labelling function $\mathcal{L}_c(e)$ is a variant of the function used in the round robin connectivity algorithm [G91]. The two labelling fuunctions differ in their definition of $\mathcal{L}'$; the round robin algorithm uses $\mathcal{L}'(vw) = \rho(w) - T$. Neither definition can be substituted in the other algorithm. So the poset representation goes beyond network and matroid structures.

Our basic algorithm, *find_nodes*, computes the nodes $[e]$ of P. It is given the complete k-intersection T and trees T_i. To specify *find_nodes* we need only give an algorithm that generates a subset of the edges of L that allows the strong components algorithm to compute the nodes of P. Specifically to generate the next edge ef directed from e we can restrict attention to edges $f \in \mathcal{L}_c(e)$ such that $\lfloor f \rfloor$ does not equal $[f]$ or $\lfloor e \rfloor$.

The algorithm uses several data structures. First, the strong components algorithm maintains sets $\lfloor e \rfloor$ using the set merging algorithm. (The implementation in [Tar72] does not do this, but the algorithm is easily modified.)

To process the edges not in T, for each vertex v, $f(v)$ is an arbitrarily chosen edge of $\rho(v) - T$; the remaining edges of $\rho(v) - T$ are stored in a list $L(v)$. ($f(v)$ is Λ if $\rho(v) - T = \emptyset$. The value Λ is used to mark the end of list $L(v)$.)

For the data structures consider each tree T_i to be rooted at vertex a. Each tree T_i is labelled so that in $O(1)$ time we can test given vertices v and w to see if v is a descendant of w. (For instance a preorder numbering of T_i supports this operation.)

In addition we maintain several partitions of the edges of each tree T_i into subtrees. To do this we use the static tree set merging algorithm of [GT]. It performs n *union* and m *find* operations in time $O(m + n)$. (The algorithm of [GT] maintains a partition of the vertices of a given tree into subtrees. It is a simple matter to adapt the algorithm to maintain an edge partition.) Before defining the partitions note that in each tree T_i a set $M(e)$ induces a subtree but $[e]$ may induce a forest (i.e., it need not be connected).

There are two universes for set merging, called universes U_1, U_2. Each universe maintains a partition of the edges of T; more precisely each set of the partition is a subtree of some tree T_i. Let $C = \{e \in T \mid \lfloor e \rfloor = [e]\}$. In U_1 each set is either a singleton edge or a subtree of C in some tree T_i. In U_2 each set is a subtree with precisely one edge f incident to its root and every edge contained in $C \cup \lfloor f \rfloor$. If S is a set in either universe, *root*(S) denotes its root vertex; if S is in universe U_2,

$root_edge(S)$ denotes the edge of S incident to its root. In set merging operations a subscript of i indicates the relevant universe U_i, e.g., $union_2(x,y)$ (we use other set merging universes U_i below).

In both universes the subtrees forming sets are maintained to be maximal in a lazy sense. To elaborate, instead of the usual *find* operation we use a "lazy find" *l_find*. To do *l_find*(e) let e' be the edge joining $root(find(e))$ to its parent (if it exists). If the specification of the data structure implies that e and e' can be in the same set, perform $union(find(e), find(e'))$. (Note that we check the data structure specifications using the data structures of the strong components algorithm. For universes U_1 and U_2 a test $\lfloor e \rfloor = [e]$ is done in $O(1)$ time. For U_2 a test $\lfloor e \rfloor = \lfloor f \rfloor$ is done using two *find* operations.) Repeat this step until e and e' cannot be in the same set (or e' does not exist).

Now we state the *find_nodes* algorithm. It is a subroutine of the strong components algorithm. Its input is a vertex e of L, plus the index i such that $\mathcal{L}_c(e)$ contains edges of the fundamental cycle $C(e, T_i)$. It returns an edge $f \in E(G)$ corresponding to an edge ef of L to be processed by the strong components algorithm; it returns Λ if no such edge remains to be processed. It is organized using two steps. The T_edge Step sets f to an appropriate edge of $C(e, T_i)$, if it exists; the Non-T_edge Step sets f to an appropriate edge not in T, if it exists.

The *find_nodes* algorithm works as follows. Write $e = v_0v_1$. For a vertex x let x' denote its parent in tree T_i. Set $f \leftarrow \Lambda$. Execute the T_edge Step and the Non-T_edge Step as follows, stopping as soon as some execution sets f to an edge: If $e \in T$ execute the T_edge Step for $j = 0, 1$, then execute the Non-T_edge Step for $j = 0, 1$. If $e \notin T$ execute the T_edge Step for $j = 0, 1$. In both cases, stop as soon as an execution sets f to an edge and return this edge; if no execution sets f to an edge then return the value Λ.

T_edge Step. Set $x \leftarrow x_j$. Set $u \leftarrow root(l_find_1(xx'))$. If $\lfloor uu' \rfloor = \lfloor e \rfloor$ then set $x \leftarrow root(l_find_2(xx'))$ and again set $u \leftarrow root(l_find_1(xx'))$. (At this point $\lfloor uu' \rfloor$ is the first edge in the tree path from x_j to the root that is not in $C \cup \lfloor e \rfloor$.) If u is not an ancestor of x_{1-j} then set $f \leftarrow uu'$. Continue as

specified above. (In this entire step, if any parent x' or u' that gets computed does not exist then continue as specified above.)

Non-T_edge Step. Set $x \leftarrow x_j$. Set $g \leftarrow f(x_j)$. If $g \neq \Lambda$ and $\lfloor g \rfloor \neq [g]$ then proceed as follows: if $\lfloor e \rfloor \neq \lfloor g \rfloor$ then set $f \leftarrow g$ else ($\lfloor e \rfloor = \lfloor g \rfloor$) remove the first edge f from list $L(x_j)$. Continue as specified above. ∎

To see this routine is correct first consider the T_edge Step. If u is an ancestor of x_{1-j} then any edge $g \in C(e, T_{i+1})$ in the tree path from x_j to the root is in $C \cup [e]$, i.e., edge eg need not be processed by the strong components algorithm. If u is not an ancestor of x_{1-j} then edge uu' is in $C(e, T_{i+1})$.

Next consider the Non-T_edge Step. For a given edge e this step does not return all edges of $\mathcal{L}'(e)$. To show this is correct it suffices to prove that for any vertex v the algorithm adds every edge $g \in \rho(v) - T$ to $\lfloor f(v) \rfloor$.

The fundamental cycle $\mathcal{L}_c(g)$ contains an edge $e \in T$ incident to v. Suppose e is added to the stack of the strong components algorithm after g. The Non-T_edge Step for e ensures that $e, g \in \lfloor f(v) \rfloor$. Suppose e is added before g. The Non-T_edge Step that returned g ensures that $\lfloor f(v) \rfloor$ is the entry immediately preceding g on the stack. Thus the T_edge Step for g ensures that $g, f(v) \in \lfloor e \rfloor$. We conclude the algorithm is correct.

Now consider the time for the algorithm. In the T_edge Step, each *find* operation results in a *union*. There are two set merging data structures, each defined on a forest of kn nodes. Thus the total time for this step is $O(kn) = O(m)$. The Non-T_edge Step returns edge $f(v)$ at most once for each edge in $\rho(v) \cap T$, and returns any edge of $L(v)$ once in the entire algorithm. Thus the total time is $O(m)$.

Lemma 3.2. Consider a directed graph, given with a complete λ_a-intersection T for a. Let $\mathcal{F}$ be the family of sets that are cut and cospanned by T. The nodes of P can be found in $O(m)$ time.

▮

We turn to $\mathcal{F}_a$, the family of a-minsinks, equivalently the a-sets that are cut and cospanned by T. We wish show that each of the following operations can be done in time $O(m)$ on $P(\mathcal{F}_a)$:

(i) Find the subpartition of $V(G)$ into the nodes of P.

(ii) For each node of P, find a successor that is a sink.

(iii) Find all nodes of P having only one successor that is a sink.

(iv) For a given set $S \subseteq V(P)$, find all nodes of P having a proper successor in S.

(v) Find the maximal a-minsinks.

Operations (i) – (iii) are used in Section 2. Section 5 uses (iv) and Section 7 uses (i) – (iii) and (v).

The operations are implemented using the following primitive. It computes a label $\ell([e])$ for each node $[e]$, in time $O(m)$. It is convenient to introduce another value $\ell'([e])$ that can be computed in $O(1)$ time from $\ell([e])$. (Usually ℓ' equals ℓ.) The labels $\ell([e])$ have two properties. Let I denote the set of immediate successors of $[e]$ in P.

(a) $\ell([e])$ can be computed in time $O(|I|)$ from $\{\ell'([i]) \mid i \in I\}$.

(b) $\ell([e])$ can be computed in time $O(1)$ given two nodes in I with different ℓ' values.

The algorithm computes labels using two set merging universes U_3, U_4, each defined on the trees T_i, $0 \leq i < k$. Say that an edge e is "labelled" or not depending on whether $\ell([e])$ has been computed. In U_3 each set is either a singleton edge that is unlabelled or a subtree of edges f having the same value $\ell'([f])$. U_4 ensures that an edge is examined only once in U_3; "examined" is defined precisely in the algorithm below. In U_4 each set is either a singleton edge, or a subtree of labelled edges, or a subtree of edges rooted at an examined edge with each edge either examined or labelled. Each universe is maintained in a lazy fashion (like U_1 and U_2).

It is helpful to recall the following description of the set of immediate successors I of a node $[e]$: I equals $X - [e]$ where X is a set of nodes defined as follows. Consider any edge $f \in [e]$. X contains all edges of the fundamental cycle of f in T_j, where $j = i+1$ if $f \in T_i$ and $j = 0$ if $f \notin T$. Additionally if $f = vw \in T$ then X contains $[v]$ and $[w]$.

The algorithm to compute labels executes *find_nodes*. When the strong components algorithm postvisits a node $[e]$ of P, $\ell([e])$ is computed. To do this use U_3 to scan fundamental cycles in I, as long as the edges encountered are in $[e]$ or have the same ℓ' value. In U_4, an edge e is declared "examined" the first time it is found in a fundamental cycle in U_3. If, in U_3, the algorithm encounters an edge that has been examined, it uses U_4 to find the next fundamental cycle edge. If the algorithm encounters a second ℓ' value in U_3, it determines $\ell([e])$ (using (b)) and stops. Otherwise it computes all fundamental cycles and determines $\ell([e])$ using (a).

The time to process $[e]$ is $O(|[e]|)$. This follows from two properties. First in each fundamental cycle, an edge $f \in [e]$ is encountered after each U_3 set of edges G not in $[e]$. (If an edge $g \notin [e]$ is encountered after G, then $\ell'([g])$ is different from the ℓ' value for G, and $\ell([e])$ is+ determined using (b).) Second an edge $f \in [e]$ is encountered only once in a fundamental cycle, because of U_4. Thus the total time is $O(m)$.

Now we briefly discuss operations (i) –(v). Operation (i) goes beyond algorithm *find_nodes* since it determines which nodes of P have a successor that contains vertex a. To do this define $\ell([e])$ as a bit indicating whether or not such a successor exists, and $\ell' = \ell$. Clearly properties (a) – (b) hold.

For operation (ii), define $\ell([e])$ as a successor of $[e]$ that is a sink, and $\ell' = \ell$. For operation (iii) define $\ell([e])$ as the unique sink-successor of $[e]$ if it exists, else Λ; also $\ell' = \ell$. For operation (iv) define $\ell([e])$ as a bit indicating whether or not $[e]$ has a proper successor in S. Take $\ell'([e])$ to be true if $\ell([e])$ is true or $[e]$ is in S. In all cases (a) –(b) hold, and $\ell([e])$ is the information returned by the operation.

Consider operation (v). The maximal a-minsinks are precisely the trees of the forest $T_1 - A$, where A is the set of edges not in an a-minsink. This set is known from operation (i). (As a footnote, it is not hard to show that the maximal a-minsinks are precisely the weak components of $P(\mathcal{F}_a)$.)

Corollary 3.1. For a directed graph, let $\mathcal{F}$ be the family of a-minsinks. Given a complete λ_a-intersection T for a, each of the operations (i) – (v) can be done in $O(m)$ time. ∎

4. Poset properties.

This section presents combinatorial properties of poset representations for set families related to graphic matroids, especially the minsink poset for directed and undirected graphs.

Directed and undirected graphs have some differing characteristics: For directed graphs, even if $m = O(n)$ P^r can require $\Omega(n^2)$ edges and there can be $\Omega(2^n)$ minsinks. In contrast for undirected graphs P^r has $O(n)$ edges and there are $O(n^2)$ minsinks. The facts for undirected graphs follow from Theorems 4.1(i) and 4.2 below, although the second fact is well-known. The facts for directed graphs are shown in Figures 2 – 4: Figure 2.a shows a directed graph with $2k+1$ vertices and $4k$ edges; Figure 2.b shows that P^r has $\Theta(k^2)$ edges. Note that each node $[e]$ is labelled with the vertices and edges in $[e]$ (for instance for any index $i \in (k..2k]$ there is a node corresponding to vertex i and edges $1i$, ki). Figure 3 shows a directed graph with $\Theta(2^n)$ minsinks – $S + b$ is a minsink for any $S \subseteq [1..n]$. Each of these minsinks determines the same mincut (edge ab). In contrast, Figure 4 shows a directed multigraph with $\Theta((n/\lambda)^\lambda)$ mincuts. In this figure the vertices are arranged in λ "chains", where a chain is a vertical path of $(n-2)/\lambda$ vertices; each edge has multiplicity 1 or λ, and edges of multiplicity λ are so labelled. A minsink consists of vertex b plus the last k vertices of each chain, where k can be chosen arbitrarily and independently for each chain.

We show two compactness properties for directed graphs. We start with a result that is useful for directed graphs and also the rigidity posets.

Consider an undirected multigraph T consisting of k edge-disjoint spanning trees T_i (distinct T_i's can contain distinct copies of the same edge). Analogous to Section 3, say that a set of edges S is *cospanned by* T if for each i, $S \cap T_i$ is a tree spanning vertices $V(S)$. Clearly the family of edge sets cospanned by T is closed under intersection and has a poset representation P. Similar to the minsink poset, the sets of this family are precisely the sets that can be written as $\bigcup\{M(e) \mid e \in R\}$ where the edges R form a subtree of T_1.

P has two compactness properties. First it has $O(n)$ nodes. Second it has the following simple representation. We will describe $\delta([e])$ for a given edge e. Assume that each tree T_i and all partition sets $[x]$ are known. Choose any tree, say T_1. To compute $\delta([e])$, let U be the edge set of the smallest subtree of T_1 that spans $[e]$. Let $U' = U - [e]$. Then

$$\delta([e]) = \{[e][x] \mid x \in U'\}.$$

Lemma 4.1. Let $\mathcal{F}$ be the family of edge sets cospanned by T.

(*i*) $P(\mathcal{F})$ has at most n nodes.

(*ii*) The above formula holds for any set $\delta([e])$.

Proof. (*i*) It suffices to show that for any $e \in T$, $|[e] \cap T|$ is a multiple of k. To do this observe that $M(e)$ forms a subtree in each T_i. The set $M(e) - [e]$ has the same connected components in each T_i, since it is a union of sets $M(f)$. Thus if $M(e) - [e]$ has $c \geq 2$ components in the tree containing e, it has c components in each T_i, and $|[e] \cap T_i| = (c-1)k$.

(*ii*) We must show that $M(e) = [e] \cup \bigcup\{M(x) \mid x \in U'\}$. Let R denote the right-hand side. The definition of U implies that $R \subseteq M(e)$. Thus it suffices to show R is cospanned by T.

For each i let $F_i = R \cap T_i$. Using (*i*), each F_i has the same cardinality, and forms a forest on vertices $V(R)$. F_1 is a tree spanning $V(R)$ by the definition of U. Thus each F_i is a tree spanning

$V(R)$, as desired. ∎

The minsink poset is essentially a generalization of the above poset. (Given the graph T and trees T_i, root each T_i at the same vertex a and direct all edges away from the root. In the resulting directed graph, the a-minsinks are the cospanned a-sets of T.) The minsink poset has similar properties. To state the representation of P, let T be a complete λ_a-intersection for the given directed graph. Define edge sets U and U' exactly as above. Let W be the set of vertices on edges of $[e]$. Let $W' = W - V(U') - [e]$ (recall that $[e]$ contains vertices in addition to edges). Then

$$\delta([e]) = \{[e][x] \mid x \in U' \cup W'\}.$$

(Here x ranges over edges in U' and vertices in W'.)

Theorem 4.1. For a directed graph, let $\mathcal{F}$ be the family of a-minsinks.

(i) $P(\mathcal{F})$ has at most $2n$ nodes.

(ii) The above formula holds for any set $\delta([e])$.

Proof. (i) There are at most n distinct sets $[e]$ for $e \notin T$, since any vertex v has $\rho(v) - T \subseteq [v]$. There are at most n distinct sets $[e]$ for $e \in T$, by the argument of Lemma 4.1(i).

(ii) We must show that $M(e) = (W \cap [e]) \cup \bigcup\{M(x) \mid x \in U' \cup W'\}$. (Note that $M(e)$ is a set of vertices, whereas in Lemma 4.1 it is a set of edges.) Let R denote the right-hand side. The definition of U implies that $R \subseteq M(e)$. Thus it suffices to show R is an a-minsink, i.e., it is cut and cospanned by T. We make a preliminary observation: A vertex on an edge of $[e]$ is either in $W \cap [e]$ or $V(U')$ or W'.

To show R is cut by T, note that $v \in W \cap [e]$ implies $\rho(v) - T \subseteq [e]$; now use the preliminary observation. To show that R is cospanned by T, for each i let F_i denote the edges of T_i that are either in $[e]$ or span the subtrees corresponding to $\bigcup\{M(x) \mid x \in U' \cup W'\}$. The rest of the argument is the same as Lemma 4.1(ii) with one extra note: Each F_i forms a forest on vertices R by the preliminary observation. ∎

We turn to undirected graphs. The main result is illustrated in Figure 1 and rests on this notion: An *m-tree* rooted at R (where R is a sequence of nodes) is a directed graph defined by the following recursive rules. We use this notation: for x a node and S a set of nodes, xS denotes the set of directed edges $\{xs \mid s \in S\}$.

(i) A single node r is an m-tree rooted at r.

(ii) Let T_i be an m-tree rooted at R_i for $i = 1, \dots, h$, $h \geq 1$. Let r be a new node. Then the set of edges $\bigcup_{i=1}^{h}(rR_i \cup T_i)$ is an m-tree rooted at r.

(iii) Let T_i be an m-tree rooted at R_i for $i = 0, \dots, h$, $h \geq 2$. Let r_i, $i = 1, \dots, h$ be new nodes. Then the set of edges $\bigcup_{i=1}^{h}(r_i(R_{i-1} \cup R_i) \cup T_{i-1} \cup T_i)$ is an m-tree rooted at $r_i, i = 1, \dots, h$.

The following related terminology is used in Section 5: For any edge xy of an m-tree, x is a *parent* of y and y is a *child* of x. In (iii) two consecutive nodes r_i and r_{i+1} are *mates.* Thus a node of an m-tree has 0, 1 or 2 parents; in the last case the parents are mates. In (iii) nodes r_i, $i = 1, \dots, h$ form a *clan*, as does the single node r in (ii). Clearly each node in a clan has the same parents. A *child-clan of* p is a clan having p as one of its parents.

We show that P^r is an m-tree; furthermore it represents all mincuts, explicitly. Figure 1 shows a 2-edge-connected graph G and the m-tree of a-minsinks. Each node $[e]$ of the m-tree is labelled with the vertices of $[e]$, or if there are none then the edges of $[e]$; an edge xy of G that does not appear in a label is in $[x]$ or $[y]$, whichever node precedes the other (e.g., edge di is in $[d]$).

Begin by fixing an arbitrary vertex a of the undirected graph G. The poset represents the a-minsinks of G. It is derived from the family of sets $M(e)$, where e ranges over all edges of G. Recall that $D(G)$ is the directed graph that contains each edge of G in both directions. We can view the poset as representing the a-minsinks of $D(G)$.

Lemma 4.2(i) below is well-known. It can be proved using submodularity, or using Lemma 3.1 as follows.

Lemma 4.2. Consider an undirected graph G with a-minsinks R and S that cross.

(i) λ is even, say $\lambda = 2k$, and $|d(R \cap S, R - S)| = k$.

(ii) If $R = M(e)$ then $[e] = d(R \cap S, R - S)$.

Proof. Consider the directed graph $D(G)$. Let T be a complete λ-intersection for a on $D(G)$. Note that $R \cap S$ is a minsink, since the a-minsinks are closed under intersection. Similarly $R - S$ is a minsink.

(i) By Lemma 3.1, R is cospanned by T. Since $R - S$ and $R \cap S$ are both nonempty, each tree of T contains an edge of $D(G)$ joining $R - S$ and $R \cap S$. Similarly each tree contains an edge of $D(G)$ joining $S - R$ and $R \cap S$. This gives 2λ directed edges incident to $R \cap S$. Since $R \cap S$ is a minsink, $|d(R \cap S)| = \lambda$. Thus each edge of $d(R \cap S)$ occurs in both directions in T, and $|d(R \cap S, R - S)| = \lambda/2$.

(ii) Since $R \cap S$ and $R - S$ are a-minsinks, $[e] \subseteq d(R \cap S, R - S)$. Consider any $f \in d(R \cap S, R - S)$. Let e' be either of the two edges corresponding to e in $D(G)$; similarly for f'. For definiteness let $e' \in T_1$. The fundamental cycle of f' in T_1 contains edge e'. Thus $e' \in M(f')$, so $M(e) = M(e') = M(f') = M(f)$. ∎

Now suppose $R = M(e)$ and R crosses an a-minsink S. Lemma 4.2(ii) implies that $R \cap S$ is one of the two connected components of $M(e) - [e]$. Suppose in addition that $S = M(f)$. Lemma 4.2(ii) shows that k of the edges of $d(M(e))$ constitute $[f]$. Thus $M(e)$ crosses at most two sets $M(f)$. If it crosses $M(f)$ and $M(f')$ then the connected components of $M(e) - [e]$ are $M(e) \cap M(f)$ and $M(e) \cap M(f')$. This motivates the next definition.

A *chain* is a sequence of $h \geq 1$ sets $M(e_i)$, $i = 1, \ldots, h$ such that for $i < h$, $M(e_{i+1})$ crosses $M(e_i)$. If $h \geq 2$ the chain is *proper*. We say that a set of vertices S "is a chain" if $S = \bigcup_{i=1}^{h} M(e_i)$.

Consider a proper chain. For $i = 0, \ldots, h$ define sets M_i so that M_{i-1} and M_i are the connected components of $M(e_i) - [e_i]$. Thus $M(e_i) = M_{i-1} \cup M_i$ and each M_i is a minsink. There are precisely

k edges in $d(M_{i-1}, M_i) = [e_i]$. There are precisely $2k$ edges in $d(\bigcup_{i=0}^{h} M_i)$, k in $d(M_0)$ and k in $d(M_h)$. The sets M_i are pairwise-disjoint.

The remarks after Lemma 4.2 show each set $M(e)$ is in a unique maximal chain. Also note this property:

(a) If $e \in M(f)$ then either $M(e) = M(f)$ or $\bigcup_{i=1}^{h} M(e_i) \subseteq M(f)$ where $M(e_i)$, $i = 1, \ldots, h$ is the maximal chain containing $M(e)$.

In proof, suppose $M(e)$ is in a proper chain and $M(f) \neq M(e)$ (otherwise (a) is immediate). Then $M(f)$ doesn't cross any $M(e_i)$, and a simple argument shows that $M(f)$ contains each $M(e_i)$.

Lemma 4.3. An a-minsink S is a chain.

Proof. Consider a chain $M(e_i)$, $i = 1, \ldots, h$ of maximal sets $M(e_i)$ contained in S, with h as large as possible. We show this chain contains every vertex of S by assuming the contrary. Since S is connected there is an edge f joining $M(e_1)$ or $M(e_h)$, say $M(e_h)$, to a vertex of S not in the chain. Clearly $M(f)$ can be appended to give a longer chain. $M(f)$ is maximal in S, by (a). This is the desired contradiction. ∎

Observe that if λ is odd we have already deduced the structure of the poset P^r. Lemma 4.2(i) shows that a-minsinks never cross, i.e., they are disjoint or nest. Thus P^r is a forest. (For convenience, the proof of Theorem 4.1 below modifies P^r to be a tree.) P^r represents all sets $M(e)$, which by Lemma 4.3 are all the a-minsinks.

We return to the case of λ even. Call a set of vertices S *crossed* if there is a set $M(e)$ that crosses S; otherwise S is *uncrossed.* Thus a set $M(e)$ is crossed if and only if it is in a proper chain. Observe that S is an uncrossed chain if and only if S is an uncrossed set $M(e)$ or S is a proper chain of maximal length. (The "if" direction uses (a).)

Let $S(e)$ denote the maximal sets $M(f)$ that are properly contained in $M(e)$. In terms of P^r, $S(e)$ corresponds to the immediate successors of $[e]$. Note this property:

(b) $S(e)$ and $S(f)$ meet only if $M(e)$ and $M(f)$ are identical or cross.

This follows since if $M(e) \subset M(f)$ then a set in $S(e)$ is not maximal in $M(f)$.

Lemma 4.4. For an uncrossed set $M(e)$, $S(e)$ partitions into (zero or more) uncrossed chains. Furthermore $S(e)$ is disjoint from any set $S(f)$, $M(f) \neq M(e)$.

Proof. Consider a maximal chain $M(e_i)$, $i = 1, \ldots, h$ with $M(e_i) \in S(e)$. If the chain is crossed, say some $M(f)$ crosses $M(e_h)$, then $M(f) \subseteq M(e)$ ($M(f)$ doesn't cross $M(e)$) and $M(f) \in S(e)$ (by (a)), a contradiction.

The disjointness of $S(e)$ follows from (b). ∎

Next consider a crossed set $M(e)$. Write it as $M(e_i)$, where i denotes its index in the uncrossed chain containing it. We also refer to sets M_i introduced in the definition of chain.

Lemma 4.5. For a crossed set $M(e_i)$, $S(e_i)$ consists of the maximal sets $M(e)$ comprising M_{i-1} and M_i; each of M_{i-1} and M_i is an uncrossed chain. Furthermore $S(e_i)$ and $S(e_{i+1})$ are the only sets containing a maximal set $M(e) \subseteq M_i$.

Proof. By Lemma 4.2(ii), $S(e_i)$ consists of the maximal sets $M(e)$ comprising M_{i-1} and M_i. For definiteness consider M_i. By Lemma 4.3 M_i is a chain, say starting with $M(f_1)$. Since $M(e_i)$ contains f_1, (a) shows $M(e_i)$ contains the maximal chain containing $M(f_1)$.

For the last part suppose $S(f)$ contains a maximal set $M(e) \subseteq M_i$. It is easy to see that (b) shows $M(f)$ is either $M(e_i)$ or $M(e_{i+1})$. ∎

It is not hard to show that for $\lambda = 2$ each M_i is an uncrossed set $M(e)$. However for larger λ M_i can be a proper chain. This is illustrated in Figure 5: Figure 5.a shows a 4-edge-connected

multigraph, where each edge has multiplicity 1 or 2, and the edges of multiplicity 2 are so labelled. Figure 5.b shows the poset of a-minsinks; the clan fg, gh has two parents.

Now observe that an uncrossed chain $M(e_i)$, $i = 1, \ldots, h$ can be represented by an m-tree rooted at h nodes. This can be proved by induction, since Lemma 4.4 correponds to rule (ii) of the definition of m-tree and Lemma 4.5 correponds to rule (iii).

We construct the poset P^r as follows. Consider an undirected graph G. Recall the multigraph G' of Section 3, which has the same poset as G and each node has a label $[e]$ for an edge e. Modify G' so there are λ copies of edge aa' rather than $\lambda + 1$. Consider the poset P^r for all a'-minsinks. It is an m-tree rooted at one node, corresponding to all vertices v of G that are not in any a-minsink (e.g., vertex a). The children of this root correspond to the maximal uncrossed chains of G. Note that P^r explicitly represents all mincuts of G. For Lemma 4.3 shows that an a-minsink is either a node of P^r or a portion of an uncrossed chain.

Theorem 4.2. For an undirected graph, let $\mathcal{F}$ be the family of a-minsinks. $P^r(\mathcal{F})$ is an m-tree. Furthermore it is a tree if λ is odd. ∎

Now let A be a minimal minsink, i.e., it is a minsink but no proper subset is a minsink. (For instance A can be chosen as a a single vertex of degree λ, or in general, a leaf of an m-tree representation.) Constructing the above poset P^r for any $a \in A$ gives an m-tree whose root is a leaf and corresponds to set A. This m-tree has the property that its leaves are precisely the minimal minsinks. Call this the *m-tree representation of all minsinks.* It is used in Section 6.

Recall that for any integer k, the vertices of an undirected graph G can be partitioned into k-edge-connected components: two vertices are in the same k-edge-connected component if and only if they cannot be separated by deleting fewer than k edges. The partition of $V(G)$ into nodes of P^r gives the $(\lambda + 1)$-edge-connected components of G.

5. Full poset constructions.

This section presents algorithms to construct the complete poset representation for problems involving graphic matroids, specifically the minsink posets and posets for rigid subgraphs.

The algorithm *find_nodes* can be adapted to construct the entire graph P. However a naive approach is ineffficent, because a given edge of P can be discovered many times by the algorithm. Also, the graph P can contain transitive edges. This means that for the undirected minsink poset, P is not the desired tree or m-tree. Furthermore because of transitive edges the depth-first traversal of P (done by *find_nodes*) need not correspond to the tree or m-tree structure.

We begin by constructing the tree or m-tree for the mincuts of an undirected graph. The algorithm is used in Section 6 to augment the connectivity of undirected graphs. It is essentially based on the auxiliary representation of P (Theorem 4.1(ii)).

We begin with the special case of λ odd, i.e. P^r a tree, since it is simpler and illustrates the general algorithm. Use *find_nodes* to compute the nodes of P^r. The edges of P^r are determined using another set merging data structure, on T_1; call this data structure universe U_3. It maintains a subpartition of $V(T_1)$ into the maximal subtrees $M(e)$ of C. The nesting of sets for λ odd implies that universe U_3 consists of disjoint sets. (In fact U_3 is a variant of U_1.) Recall that the strong components algorithm does a depth-first traversal of P. Thus when a new set $[e]$ is output (in its postvisit) all its successors and none of its predecessors have been output. Thus for each child $[c]$ of $[e]$, U_3 contains $M(c)$ as a maximal set. Hence $M(e)$ is the smallest subtree of T_1 that consists of the edges $[e] \cap T_1$ and the sets of universe U_3 that are incident to these edges. Thus the algorithm constructs $M(e)$ in U_3 and outputs the edges from e to its children.

Now consider the case of λ even and P^r an m-tree. We use the following representation for an m-tree: Each clan C has its nodes organized as a doubly-linked list (each node is linked to its mates). In addition each parent of C has a pointer to the first node of C (it may be convenient to make this link bidirectional).

Consider a graph G that is an m-tree. Thus it has no transitive edges but the edges directed from a vertex can be in arbitrary order. The above m-tree data structure can be built in a depth-first search of G as follows. In the postvisit to a node x, the depth-first search determines (i) the pointers joining x with mates y that have been postvisited; (ii) the pointers from x to the first node of each child-clan. (i) can be done using the fact that x and y have a common child. This is easily determined since there are no transitive edges. (ii) can be done since the linked list for each child-clan of x is already determined. The total time for this depth-first search is $O(n)$.

Now we construct P^r by adapting the *find_nodes* routine. We build the above data structure for the m-tree P^r, recording items (i) and (ii) for each postvisited node. In addition there is another set merging data structure on T_1, called universe U_3. To describe it note that a clan C of P^r corresponds to an uncrossed chain, which forms a subtree in T_1; call this subtree $T_1(C)$. Universe U_3 maintains a subpartition of $V(T_1)$ into subtrees $T_1(C)$ for C a clan with a postvisited parent but no postvisited grandparent; equivalently C is a clan with a postvisited parent and $T_1(C)$ maximal. This is a valid subpartition of $V(T_1)$ by the definition of uncrossed chain. Recall the convention that $find_3$ denotes a *find* operation in universe U_3; assume that $find_3(v)$ returns the clan C that gives the partition set $T_1(C)$ containing v.

Let us describe the data structures during the depth-first search of *find_nodes*. Consider a new set $[e]$ that is output in a postvisit. At this time any child of $[e]$, but no predecessor, has been postvisited. In addition $[e]$ has 0, 1 or 2 mates that may have been postvisited. This determines the state of universe U_3: For any clan C of grandchildren of $[e]$, $T_1(C)$ is contained in one set of U_3. If $[e]$ has no postvisited mates then all of these sets are maximal in U_3. If $[e]$ has a postvisited mate $[f]$ then the common child-clan $[e]$ and $[f]$ gives a maximal set in U_3. The clan containing such a postvisited mate $[f]$ is only partially recorded in the data structure, while the clans for grandchildren of $[e]$ are completely recorded.

The *find_nodes* routine is modified so that when $[e]$ is postvisited, it determines a collection of

tentative children of $[e]$. These are the actual children of $[e]$ unless $[e]$ has a postvisited mate. To do this observe that for each vertex v on an edge of $[e] \cap T_1$, the clan $\mathit{find}_3(v)$ has a unique parent $[f]$ in the current data structure. ($[f]$ is unique since a clan with two parents recorded in the data structure has all incident edges postvisited. $[f]$ is known by the definition of U_3.) Let C_f be the clan containing f. Make each such C_f a tentative child-clan of $[e]$.

Lemmas 4.4–4.5 show that as claimed, the tentative children are the actual children of $[e]$ if $[e]$ has no postvisited mate. (Recall that the edges of $M(e)$ in T_1 form a subtree of T_1, consisting of $[e] \cap T_1$ and edges in sets $M(f)$ for f a child of $[e]$.) On the other hand if $[e]$ has a postvisited mate $[f]$ then C_f has been made a tentative child-clan (even though C_f actually consists of mates of $[e]$).

To detect the latter, incorrect situation, we check the following conditions that are necessary for $M(e)$ to cross some set $M(f)$: $[e]$ is a subset of T of cardinality k; $[e]$ has precisely two tentative child-clans, C_f and another, say E; $[f]$ has precisely two child-clans, say F and another, such that for each edge $xy \in [e]$, either $\mathit{find}_3(x)$ or $\mathit{find}_3(y)$ equals F. It is easy to see that together these conditions imply that $M(e)$ crosses $M(f)$: Each edge of $[e]$ joins E and F. Since E and F are both minsinks, $|d(E \cup F)| = 2k$. Thus $E \cup F$ is a minsink crossing $M(f)$. Clearly $M(e) = E \cup F$, as desired. In this case the algorithm revises the set of tentative child-clans to E and F. It also makes $[e]$ and $[f]$ mates.

Now in all cases the tentative child-clans C are the actual children of $[e]$ in P^r. Make each such C a child-clan of $[e]$ in P^r. Also in universe U_3 merge the components of C.

It is easy to see that the time for this procedure to construct P^r is $O(m)$ plus the time to find the complete λ-intersection T. Using the algorithm of [G91] to find T gives this result.

Theorem 5.1. For an undirected graph, the m-tree representing all a-minsinks can be found in time $O(m + \lambda^2 n \log(n/\lambda))$ and space $O(m)$. ∎

The same bounds hold for constructing the cactus representation of all mincuts [DKL] – the m-tree can be converted to the cactus in linear time. As an application, [NGM] shows the edge connectivity augmentation problem, for undirected graphs and $\delta = 1$, can be solved in linear time from the cactus representation. Section 6 shows their algorithm works unchanged for the m-tree representation.

The rest of this section constructs three versions of the poset representation, P, P^r and P^c. Recall that the transitive closure G^c or transitive reduction G^r of a directed acyclic graph G can be found in time $O(nm)$: For G^c find the successors of each node v, by doing a depth-first search from v. For G^r, recall that an edge xy is in G^r if and only if in G, x immediately precedes y but no proper predecessor of y [AGU]. Thus to find G^r, find G^c. Then for each node y, find its immediate predecessors in G^r by scanning the edges of G directed from each vertex x, applying the above criterion.

Consider a directed graph. Recall that P^r can have $\Omega(n^2)$ edges.

Theorem 5.2. For a directed graph, let $\mathcal{F}$ be the family of a-minsinks. $V(P)$ can be found in time $O(\lambda_a m \log(n^2/m))$ and space $O(m)$. P can be found in time $O(\lambda_a m \log(n^2/m) + n^2)$, P^r and P^c in time $O(nm)$, all in space $O(n^2)$.

Proof. First find the complete λ_a-intersection T for a in time $O(\lambda_a m \log(n^2/m))$ [G91]. Use *find_nodes* and operation (i) to find $V(P)$; Corollary 3.1 gives the desired time bound for $V(P)$.

The algorithm to find P is based on the representation of Theorem 4.1(ii), and we use its notation. For a given node $[e]$ of P we compute $\delta([e])$ in time $O(|[e]| + n)$. Note this gives the desired time bound for P, since Theorem 4.1(i) shows these terms sum to $O(m + n^2) = O(n^2)$.

The main task is to find the subtree U of T_1 that spans all edges of $[e]$. To do this root T_1 at a vertex r in some edge of $[e]$. Then mark the vertices of U as follows. Initially each vertex is unmarked. For each edge of $[e]$ mark all ancestors of each end x. To do this efficiently traverse the

path from x to r, marking each vertex, stopping when a previously marked vertex is reached.

To find P^c, first find the nodes of P. Then for each node $[e]$, execute the labelling algorithm to find $M(e)$. The time bound follows from Theorem 4.1(i).

To find P^r, first find P^c. Then for each node $[e]$, find its immediate predecessors in P^r as follows. Let S consist of all proper predecessors of $[e]$. Using operation (iv) of Section 3, find all nodes with no proper successor in S. The immediate predecessors of $[e]$ are the predecessors of $[e]$ that are returned by operation (iv). ∎

We turn to problems involving graph rigidity. A *bar-and-body framework* is a collection of rigid bars attached to rigid bodies by universal joints in some space $\mathbf{R}^d$. Such a framework can be represented by a multigraph G: each vertex corresponds to a body, and each edge corresponds to a bar whose ends are arbitrary points on the corresponding bodies. (Bars can be attached to a given body at different points.) Conversely a multigraph corresponds to an infinite family of bar-and-body frameworks in $\mathbf{R}^d$, each such framework being an assignment of the ends of edges to points in $\mathbf{R}^d$, together with an arbitrary rigid body joining all points that correspond to a vertex of G. G is *generically rigid* [LY] in $\mathbf{R}^d$ if it is rigid (i.e., it admits no instantaneous internal motions) whenever it is placed as a bar-and-body framework in $\mathbf{R}^d$ with the coordinates of all endpoints of all edges algebraically independent. Tay [Tay] shows that G is generically rigid if and only if it contains k edge-disjoint spanning trees, for $k = d(d+1)/2$. White and Whiteley [WW] generalize this to arbitrary k (allowing motions in other spaces). Multigraph G is *k-isostatic* if it is generically rigid for these motions but deleting any edge destroys this property. They show that G is k-isostatic if and only if its edges can be partitioned into k spanning trees. Let G be k-isostatic. It may contain k-isostatic subgraphs, and these graphs form a lattice. In our terminology this is precisely the lattice of cospanned edge sets (the subject of Lemma 4.1). The sets $M(e)$ play a special role in this lattice. They correspond in a precise way to the irreducible factors of the polynomial describing the

rigidity of G ("the pure condition"). White and Whiteley give combinatorial descriptions of certain stresses and motions of a frame, all in terms of the lattice and sets $M(e)$ [WW, pp. 18–23]. These descriptions are easily translated into properties of the poset representation (for some properties this relies on the relation of sets $M(e)$ to arbitrary cospanned sets, given above). Thus the poset representation is useful for analyzing rigidity properties of bar-and-body frameworks.

The poset representation can be computed using Lemma 4.1 and an algorithm similar to Theorem 5.2. This assumes we are given the graph T partitioned into k edge-disjoint spanning trees. Note that such a partition can be found in time $O(kn\sqrt{m + kn\log n}) = O(kn\sqrt{kn\log n})$ [G91].

Theorem 5.3. Consider a k-isostatic graph, given as k edge-disjoint spanning trees. Let $\mathcal{F}$ be the family of k-isostatic subgraphs. $V(P)$ can be found in time and space $O(kn)$. P can be found in time $O(n^2)$, P^c and P^r in time $O(kn^2)$, all in space $O(n^2)$. ∎

A *bar-and-joint framework* is a collection of rigid bars connected with universal joints. The known theory is for frameworks in $\mathbf{R}^2$ (e.g., [LY]). In the *rigidity matroid* $\mathcal{R}$ for a graph G, a set of edges F is independent if any nonempty set $F' \subseteq F$ has $|F'| \leq 2|V(F')| - 3$ [LY]. Laman showed that G is minimally rigid if and only if $m = 2n - 3$ and G is independent in $\mathcal{R}$ [L]. Let $\mathcal{F}$ be the family of rigid subgraphs of a minimally rigid graph G. (Clearly any rigid subgraph of G is minimally rigid.) For any edge e, $\{e\}$ is rigid, i.e., $M(e) = \{e\}$. So the poset representation $P(\mathcal{F})$ provides no information. We now discuss two ways to represent the rigid subgraphs of G.

Fix an edge e. Form the graph G_e by adding a new copy of e, say e', to G. G_e can be partitioned into two edge-disjoint spanning trees, by the definition of $\mathcal{R}$. An edge set F is cospanned by G_e if and only if $F - e'$ is a rigid subgraph containing e. Thus the rigid subgraphs containing e are represented by a poset, say P_e, and Theorem 5.3 (with $k = 2$) gives bounds for constructing P_e. The collection of orders P_e for all edges e of G represent all the rigid subgraphs.

Nakamura proposes a representation that is more succinct, but in some cases is incomplete. Define the edge set $M_2(e)$ as the smallest nonsingleton set of $\mathcal{F}$ containing e, if a unique such set exists; otherwise $M_2(e) = \{e\}$. This gives a poset representation P_2 defined as in Section 1. (Similarly define the transitively closed and reduced versions, P_2^c and P_2^r, respectively.) Nakamura shows this poset represents all sets of $\mathcal{F}$ (any nonsingleton set of $\mathcal{F}$ is an ideal and any connected ideal is in $\mathcal{F}$; however a set of $\mathcal{F}$ may correspond to a disconnected ideal and a disconnected ideal need not be in $\mathcal{F}$) [N]. Note also that a graph G can be tested to be minimally rigid in time $O(n\sqrt{n\log n})$ [GW].

Theorem 5.4. Let $\mathcal{F}$ be the family of rigid subgraphs of a minimally rigid bar-and-joint graph. P_2^c and P_2^r can be found in time and space $O(n^2)$.

Proof. The set $M_2(e)$ can be constructed as follows: The above poset P_e has a unique sink $[e] = \{e, e'\}$. Define set A to be the predecessor of the sink $[e]$ in P_e^r if it is unique, otherwise A is $\emptyset$. Then $M_2(e) = A + e$. This description follows from [N] or alternatively the discussion above.

To implement this construction, first partition G into two disjoint spanning forests (i.e., find a base of $\mathcal{G}^2$, the matroid sum of two copies of the graphic matroid of G). Then do the following for each edge e. Form graph G_e by adding a new copy e' of e to G. Partition G_e into two trees. To do this add e' to the forests of G by finding one augmenting path for $\mathcal{G}^2$ [GW]. Then find the set A of P_e by using operation (iv) with set S containing all nodes of P_e except $[e]$.

The time to find an augmenting path is $O(n)$ [GW]. Thus it is easy to see the total time to find all sets $M_2(e)$ is $O(n^2)$.

It is easy to find P_2^c from the sets $M_2(e)$, using an algorithm for strongly connected components.

Now consider the reduced poset P_2^r. Use a subscript of 2 or e to designate objects in P_2^r or P_e respectively. We describe the set of immediate predecessors of $[e]_2$ in P_2^r. Let S be the set of predecessors $[f]_2$ of $[e]_2$ in P_2. Note that $M_e(f)$ is the smallest rigid subgraph containing e and f.

Clearly for any edge f, $[f]_2 \subseteq [f]_e$. Furthermore for each $[f]_2 \in S$, $[f]_e$ contains no $[f']_2 \in S - [f]_2$. Now it is easy to see that the immediate predecessors of $[e]_2$ in P_2^r are the nodes $[f]_2 \in S$ such that in P_e, $[f]_e$ does not properly precede any $[f']_e$, $[f']_2 \in S$.

This justifies the following algorithm to find the immediate predecessors of $[e]_2$ in P_2^r: Use operation (iv) to find the nodes of P_e that do not properly precede a node of S. ∎

Imai uses network flow techniques to construct similar representations for rigid subgraphs [I]. Corresponding to Theorem 5.3 is a construction of P^c in time $O((kn)^2)$ (use Theorem 3.3 of [I] with values $p, d, m, |A|, |W|$ equal to $k, k, 1, kn, n$ respectively). Corresponding to Theorem 5.4 is a construction of P^c in the same time, $O(n^2)$. Imai's approach can be adapted to compute P^r in the same time using our method.

As a final application of the poset representation, [GW] shows how the "top clump" of a matroid sum determines properties such as winner of the Shannon switching game. The top clump is precisely the maximal ideal of the poset representation. Thus using operation (v) it can be found in time $O(kn)$, given a partition of the graph into edge-disjoint spanning trees. This is a slight improvement over [GW] and [G90]. More importantly the approach does not use algorithms for the dual matroid, so it promises to extend to other matroids.

6. Augmenting undirected edge connectivity.

This section solves the edge connectivity augmentation problem for undirected graphs.

We use two notational conventions. For a function $f : A \to \mathbf{R}$ and a set $S \subseteq A$, define $f(S) = \sum\{f(s) \mid s \in S\}$. We use $d(S)$ as an abbreviation for $|d(S)|$ (there is no need to refer to the set of edges $d(S)$). Furthermore $d(S)$ always refers to the original graph G (even after we have added edges); we refer to other graphs H by writing $d_H(S)$.

Our algorithm, like all others we know of, is based on the following duality relation first proved in [WN]: The minimum number of edges needed to increase the edge connectivity of an undirected

graph to τ is precisely the maximum, over all subpartitions $\mathcal{P}$ of V, of $\lceil \frac{1}{2} \sum\{\tau - d(S) \mid S \in \mathcal{P}\}\rceil$. Our development provides another proof of this relation. It is easy to see that the number of edges is at least as large as the subpartition quantity.

We begin with the case $\delta = 1$. [NGM] shows this problem can be solved in linear time given the cactus representation of G. Their algorithm works unchanged for the m-tree representation. For completeness we include a summary.

Given an undirected graph G, recall that in the m-tree representation of all minsinks, the leaves correspond to the minimal minsinks (see Section 4). Use the algorithm of Section 5 to construct this m-tree P. The algorithm returns a natural representation of P: for each node x, the children of x are stored by clans, with each clan in its natural order. Number the leaves of P, starting with the root and numbering the remaining leaves in left-to-right order (do this by a depth-first search of the given representation of P). Define vertex i to be an arbitrarily chosen vertex of G in the leaf (minsink) numbered i. Let there be ℓ leaves. For $1 \le i \le \lfloor \ell/2 \rfloor$ add a new edge joining vertices i and $i + \lceil \ell/2 \rceil$. Furthermore if ℓ is odd add an edge from $\lceil \ell/2 \rceil$ to one other vertex i.

To show this algorithm is correct, note that $\lceil \ell/2 \rceil$ edges are needed to increase the connectivity, since each leaf is a minsink. The new graph has connectivity $\lambda + 1$, since any minsink of the original graph G contains a set of leaves that are consecutively numbered.

Lemma 6.1. The undirected edge connectivity augmentation problem for $\delta = 1$ can be solved in time $O(m + \lambda^2 n \log(n/\lambda))$ and space $O(m)$. ∎

Now we discuss the general problem. We use the basic notion of [NGM]: An *extreme set* is a set of vertices X such that any nonempty proper subset has larger degree, i.e., $\emptyset \subset Y \subset X$ implies $d(Y) > d(X)$. A *weak extreme set* is the analogous notion using weak inequality, i.e., $\emptyset \subset Y \subset X$ implies $d(Y) \ge d(X)$. It is helpful to bear in mind that the set of vertices V is somewhat anomalous: $d(V) = 0$ while any $X \subset V$ has $d(X) \ge \lambda$. V is an extreme set. Note that an *extreme a-set* is as

expected an a-set that is extreme.

A fundamental fact due to [NGM] is that the extreme sets nest. We prove the following slightly stronger version of this fact.

Lemma 6.2. If an extreme set meets a weak extreme set then the sets nest.

Proof. Recall the submodular identity $d(X) + d(Y) \geq d(X - Y) + d(Y - X)$ for any two vertex sets X, Y. Let X be extreme and Y weak extreme, and suppose the sets meet but do not nest. Then $d(X - Y) > d(X)$ and $d(Y - X) \geq d(Y)$, a contradiction. ∎

The tree corresponding to the nesting of extreme sets is the basis of the algorithm of [NGM]. Rather than construct the entire extreme set tree we define a smaller tree T that we can construct efficiently.

Each node of T is labelled with a vertex x of G; x occurs as at most one label. For convenience we denote a node of T by its label x. Associated with each node x is an extreme set $M(x)$ (we shall see that M stands for maximal) and a nonnegative integer $\overline{d}(x)$.

We start by defining T and M. The root of T is labelled with an arbitrarily chosen vertex r, and $M(r) = V$. Consider any node x of T. Each child of x corresponds to a maximal extreme set N of G contained in $M(x) - x$ and having $d(N) < \tau$. The label for this child is an arbitrary vertex $y \in N$, and $M(y) = N$.

Next we define the function $\overline{d}$. It has domain V; let $\overline{d}(x) = 0$ if x is not a node of T. If x is a node of T, assume $\overline{d}$ is defined for all descendants of x. Let P be the extreme set (of G) with $x \in P \subseteq M(x)$, $P \neq V$, that minimizes $d(P) + \overline{d}(P - x)$. Then $\overline{d}(x) = \max\{\tau - d(P) - \overline{d}(P - x), 0\}$. We occasionally write $P(x)$ to denote the minimizing set P.

T has two important properties. First, it gives the subpartition of V of the fundamental duality relation, as follows. Define a subpartition by choosing the maximal sets $P(x)$ that have

$\overline{d}(x)$ positive. These sets are disjoint (by the nesting of extreme sets). They contain all nodes with positive $\overline{d}$. Each such set $P(x)$ satisfies $d(P(x)) + \overline{d}(P(x)) = \tau$. This implies the total number of edges needed to make G τ-edge-connected is at least $\lceil \overline{d}(V)/2 \rceil$. Our algorithm adds precisely this number of edges, so this is the minimum.

The second property of T is that any extreme set $K \subset V$ has $d(K) + \overline{d}(K) \geq \tau$. In proof, there is a (unique) node x of T such that $x \in K \subseteq M(x)$. Hence the definition of $\overline{d}$ implies $d(K) + \overline{d}(K) \geq \tau$. (Actually any $K \subseteq V$ satisfies the same inequality, since K contains an extreme set of no larger degree; however we do not use this fact directly.)

Our connectivity augmentation algorithm has the same high-level structure as [NGM]. It repeatedly increments the connectivity by one; δ such incrementations achieve the desired connectivity τ. Each connectivity incrementation is done using the algorithm of Lemma 6.1 with one added rule for choosing vertices i: Choose i as a vertex of G in the leaf numbered i where if possible, i is on fewer than $\overline{d}(i)$ new edges. (We shall see this is always possible except perhaps for one vertex on the last edge.)

Lemma 6.3. The algorithm constructs a τ-edge-connected graph, adding the fewest possible number of edges.

Proof. Let G^c denote the graph constructed by the algorithm that achieves connectivity c, for $c = \lambda, \ldots, \tau$. Thus G^λ is the original graph G and G^τ is the final graph. The $\delta = 1$ algorithm shows that if K is an extreme set of G^c then $G^{c+1} - G^c$ contains either one or two edges incident to K.

We first show that for any $c > \lambda$, any extreme set of G^c is extreme in G^{c-1}. If not, choose the smallest c where this fails. Suppose that adding edge xy creates a new extreme set S of G^c. Clearly both x and y are in S. At least one of x and y is in an extreme set of G^{c-1} that has degree c in G^c. Let this be vertex x in set X. Note that X is weak extreme in G^c. Hence Lemma 6.2 shows

that S and X nest. Since $y \notin X$ this implies $X \subseteq S$. But $d_{G^c}(S) \geq c = d_{G^c}(X)$, a contradiction.

An extreme set of degree c in G^c is a maximal extreme set of G^c. Thus no edge added to G^c has both ends in the same extreme set of G^c. Thus if K is extreme in G^c, no edge of $G^c - G$ has both ends in K.

Now consider the connectivity incrementation algorithm when it adds edges to G^c to obtain G^{c+1}. Let K be a leaf of the m-tree for G^c. We show that for each new edge added incident to K, K contains a vertex y on fewer than $\overline{d}(y)$ new edges, except possibly for one end of the last edge added by the algorithm.

Before any edge is added to K, K has degree c. Recall the basic property $d(K) + \overline{d}(K) \geq \tau$. Since no edge of $G^c - G$ has both ends in K, fewer than $\overline{d}(K)$ edges with an end in K have been added. Thus K contains a vertex y on less than $\overline{d}(y)$ new edges if $c \leq \tau - 2$, even if two edges incident to K are added. Similarly if $c = \tau - 1$ the desired vertex y exists for the first edge added to K. Thus the desired y does not exist only if $c = \tau - 1$ and two edges incident to K are added, i.e., one end of the last new edge.

We conclude that the total increase in degree of vertices (due to new edges) is at most $\overline{d}(V)+1$. Thus the number of edges added is at most $\lfloor (\overline{d}(V) + 1)/2 \rfloor = \lceil \overline{d}(V)/2 \rceil$.

The final graph is τ-edge-connected. We have already seen that $\lceil \overline{d}(V)/2 \rceil$ new edges are needed, and the algorithm adds at most this number. The lemma follows. (This also proves the basic duality relation.) ∎

It remains to show how to compute T and the associated functions. Start by setting each value $\overline{d}(x)$ to 0. Then execute the procedure $t(r, V)$ for an arbitrarily chosen vertex r. Here $t(x, X)$ is a recursive procedure, called with X an extreme set containing vertex x. Procedure t constructs the subtree of T rooted at node x with $M(x) = X$, plus all function values for nodes in this subtree. It works in two phases. Phase one finds the entire subtree and all function values except $\overline{d}(x)$. Phase

two finds $\overline{d}(x)$. The main data structure for t is a multigraph H that gets repeatedly modified. The vertices of H consist of the vertices in X plus a new vertex a, that represents $V-X$ if $X \neq V$. When we contract a set of vertices Y in H, $[Y]$ denotes the new vertex; any parallel edges resulting from contraction are included in the new multigraph H.

Procedure $t(x, X)$ starts by creating a new node of T labelled x and setting $M(x) \leftarrow X$. Phase one initializes multigraph H by starting with G and doing the following: If $X \neq V$ then contract $V-X$ into a single vertex a; if $X = V$ then add a new vertex a; in both cases add τ additional edges joining a and x. Then repeat the following steps until phase one is complete:

If $\lambda(H) \geq \tau$ then stop (phase one is complete). Otherwise find all extreme a-sets Y of H that have degree $\lambda(H)$. Do the following for each such Y. Arbitrarily choose a vertex $y \in Y$ and call $t(y, Y)$. This returns a subtree rooted at a node labelled y; make node y a child of x in T. In H contract Y to a single vertex $[Y]$ and add $\min\{\overline{d}(Y), \delta\}$ new edges joining $[Y]$ with a.

Phase two starts by modifying (the current) H as follows. Delete the τ edges joining a and x that phase one added (keep any other edges joining a and x). Furthermore if $X = V$ then for Y_0 an extreme set of smallest degree found in phase one, increase the number of edges joining $[Y_0]$ and a to τ. Finally set $\overline{d}(x) \leftarrow \max\{\tau - \lambda(H), 0\}$. ∎

Lemma 6.4. Procedure t is correct.

Proof. Consider an invocation $t(x, X)$. Assume by induction (on $|X|$) that each recursive call $t(y, Y)$ is correct if $M(y) = Y$.

We start by analyzing phase one. We prove it is correct by induction (on the number of steps) with the help of this auxiliary inductive assertion:

$(*)$ At any time during phase one for any a-set S of H, $d_H(S) = d(S)$ unless S contains x or some contracted vertex $[Y]$, in which case $d_H(S) \geq \tau$.

Clearly $(*)$ holds for the multigraph H constructed in the initialization.

Now consider an iteration of phase one that starts with multigraph H. The smallest degree of an extreme set in H is $\lambda(H)$, and H contains an extreme a-set Y of this degree. $(*)$ implies $Y \subseteq X - x$. $(*)$ also implies that an extreme set of H, contained in $X - x$ and having degree less than τ, is extreme in G. Thus any such set Y corresponds to a child of x in T, and the algorithm correctly constructs the corresponding subtree. Finally note that $(*)$ holds after contracting Y to a vertex, since $d(Y) + \min\{\overline{d}(Y), \delta\} \geq \tau$ and any set $S \neq V$ containing Y has $d(S) \geq d(Y)$.

We conclude that every extreme set found in phase one is correct. Finally note that phase one eventually stops having found all children of x: It suffices to show that $\lambda(H)$ strictly increases each iteration (since then $(*)$ implies all maximal extreme sets of $X - x$ with degree less than τ have been found). This follows since after all extreme sets of degree $\lambda(H)$ have been contracted, $(*)$ implies that any cut has more than $\lambda(H)$ edges.

This shows that phase one is correct. The previous paragraph also proves an important property for the efficiency of t: In phase one $\lambda(H)$ increases every iteration.

Now consider phase two. Recall the definition of $\overline{d}(x)$ involves the quantity $q(P) = d(P) + \overline{d}(P - x)$. The extreme set $P(x)$ minimizes $q(P)$ subject to the constraints $x \in P \subseteq M(x)$ and $P \neq V$ is an extreme set of G. Let H denote the multigraph for phase two. Since the extreme sets of G nest, $P(x)$ corresponds to a set in H (i.e., for any contracted vertex $[Y]$ either $P(x)$ contains Y or is disjoint from it). For any a-set P of H let $\overline{P}$ denote the set of vertices of G obtained by expanding each contracted vertex $[Y] \in P$ to Y.

For any a-set $P \neq V$, $d_H(P)$ and $q(\overline{P})$ are either equal or both quantities are at least τ. This follows since if a contracted vertex $[Y] \in P$ has $\overline{d}(Y) \geq \delta$, both quantities are at least $d(\overline{P}) + \delta \geq \lambda + \delta = \tau$. (This is true even if $X = V$ and P contains $[Y_0]$. This follows since some r-set has degree λ in G, whence $d(Y_0) = \lambda$ and $\overline{d}(Y_0) \geq \delta$.) Thus the minimum value of $q(\overline{P})$ for an a-set P of H, $P \neq V$ either equals $\lambda(H)$ or both quantities are at least τ. (If $X = V$ use the

fact that $d_H(V) \geq \tau$.)

If $\lambda(H) \geq \tau$ then $q(P(x)) \geq \tau$, $\overline{d}(x) = 0$ and phase two is correct. So suppose $\lambda(H) < \tau$. Let P be an extreme a-set of H with $q(\overline{P}) = \lambda(H)$. $\overline{P}$ is an extreme set of G. (To see this start with set P in H. Removing the new edges from contracted vertices to a gives an extreme set of the new graph. Then expanding each contracted vertex gives $\overline{P}$.) Also $\overline{P} \neq V$ (since $d_H(V) \geq \tau$). Suppose for the moment that $x \in \overline{P}$. Then $\overline{P}$ satisfies the constraints for $P(x)$ and $q(\overline{P}) \leq q(P(x))$. Thus $\overline{P} = P(x)$ and $q(\overline{P}) = \lambda(H)$ defines $\overline{d}(x)$, i.e., phase two is correct.

It remains only to show that $x \in \overline{P}$. First observe $\overline{P} \neq Y$ for a contracted vertex $[Y]$ of H, since $d_H([Y]) \geq \tau$. Then since $\overline{P}$ is an extreme set in G, $x \in \overline{P}$, since phase one finds all the children of x. ∎

We turn to efficiency. Procedure t is implemented as follows. Compute $\lambda(H)$ using the round robin connectivity algorithm. In phase one, find the extreme a-sets Y of H with degree $\lambda(H)$ by constructing the m-tree representation of H; the sets Y are the leaves of the m-tree representation that do not contain a.

We shall see that the time for the entire algorithm is dominated by the time for all connectivity computations (in phases one and two). Recall from Section 1 that the round robin connectivity algorithm computes $\lambda(H)$ by finding a complete k-intersection for vertex a, where k takes on successive values $1, \ldots, \lambda(H)$. We speed this up by making each phase one connectivity computation start with a previously computed intersection, as follows.

Consider a call $t(y, Y)$ made from invocation $t(x, X)$. Suppose the current graph H of $t(x, X)$ has $\lambda(H) = c$. Thus the connectivity algorithm has found a complete c-intersection T on H. The subgraph induced by Y is the same in H as in G (clearly Y does not contain any contracted vertex). Thus this subgraph corresponds precisely to the graph H constructed initially in $t(y, Y)$. Since Y is an a-minsink, Lemma 3.1 shows T contains c subtrees that span Y, say Y_i, $i = 1, \ldots, c$ and

T also contains each of the c directed edges $v_i w_i$, $i = 1, \ldots, c$ in set $\rho(Y)$. (T may also contain reversed edges $w_i v_i$; furthermore a tree of T may contain Y_i but no directed edge $v_j w_j$; this causes no harm.) To get a complete c-intersection on graph H of $t(y, Y)$, let the ith spanning tree consist of Y_i plus the edge aw_i corresponding to $v_i w_i$. The invocation $t(y, Y)$ uses this intersection to start its first connectivity computation. When $t(y, Y)$ returns, contract the vertices of Y in T to get a complete intersection for the new graph H. The invocation $t(x, X)$ does this for each Y to get a complete intersection for its next connectivity computation.

Lemma 6.5. Procedure $t(r, V)$ uses time $O(\tau(m + \tau n) \log n)$ and space $O(m + \delta n)$.

Proof. We estimate the time for all connectivity computations, and conclude by showing this dominates the rest of the time. Call a graph processed by an execution of round robin (in any invocation $t(x, X)$) an "r-graph". Each r-graph R has an associated index c, where round robin was run to compute a complete c-intersection on R. (Thus round robin started with $c - 1$ trees and computed a cth tree.) A given graph R may occur for a number of different indices c. We show below that all r-graphs corresponding to a fixed index c contain $O(n)$ vertices and $O(m + \tau n)$ edges. Thus round robin uses total time $O((m + \tau n) \log n)$ on all r-graphs for c, giving time $O(\tau(m + \tau n) \log n)$ for all executions of round robin. This also bounds the time for computing m-tree representations. This proves the time bound of the lemma.

We first bound the number of vertices in r-graphs. For any index c, any vertex v of G is in at most one r-graph for c in each of phases one and two. To show this choose node x of T so that v is in $M(x)$ but not in $M(y)$ for any child y of x. Let $t(x, X)$ denote the invocation of t for node x. In phase one $\lambda(H)$ increases every iteration (see the proof of Lemma 6.4). Thus it is easy to see that before the invocation $t(x, X)$, v is in precisely one phase-one r-graph for each $c \in [1..d(X)]$ (this occurs in invocations for proper ancestors of x in T). In $t(x, X)$, v is in at most one phase-one r-graph for each value $c \in (d(X)..\tau]$. Also in $t(x, X)$, v is in at most one phase-two r-graph for

each $c \in [1, \tau]$. After $t(x, X)$, v is never in an r-graph since any invocation $t(u, U)$, for u a proper ancestor of x, contracts some set containing X. (This argument is valid even if $X = V$.)

We now show that all phase-one r-graphs for a given value c contain $O(n)$ vertices. The same argument applies to phase two, we discuss phase one for definiteness. Recall that T contains at most n nodes. Thus there are at most n r-graphs for c. These r-graphs contain at most n vertices a. There are at most n sets Y that get contracted (one for each node of T). Each is a contracted vertex $[Y]$ in only one invocation $t(x, X)$. Thus the r-graphs for c contain at most n contracted vertices $[Y]$. Adding the vertices of G gives at most $3n$ vertices total.

Next we show there are $O(m + \tau n)$ edges in all phase-one or phase-two r-graphs for c. Consider an edge e of G. The analysis for vertices shows that in each of phases one and two, at most one r-graph for c contains both ends of e as vertices (i.e., neither end is in a contracted vertex). Thus it suffices show that in all r-graphs for c, all vertices a and contracted vertices $[Y]$ are incident to a total of $O(\tau n)$ edges. In phase one the initialization of H creates at most τn edges by contracting $V - X$ into a and adds at most τn additional edges incident to a. At most τn edges join contracted vertices $[Y]$ to vertices of $H - a$, and at most δn new edges joining a to a contracted vertices $[Y]$ are added. The same bounds hold for phase two. This gives the bound for edges.

Now we show that procedure t uses $O(m + \delta n)$ space. Any graph H is derived from the original graph G by contracting a set of vertices to form vertex a and contracting other sets to form vertices $[Y]$. Each vertex v of G has a label. If v is in a contracted vertex $[Y]$ the label indicates $[Y]$. Otherwise the label indicates the deepest level of recursion for which v is a vertex of H (if the current level is greater than this label then v is a vertex of a).

Consider phase one for an invocation $t(x, X)$. When a recursive call $t(y, Y)$ is made, the labels for vertices of Y are increased by one. Also the invocation is given a list of the edges of its graph H and a list of the edges of each tree of its complete intersection (i.e., tree $Y_i + v_i w_i$ described above). An edge on any list is represented by its endpoints in the original graph G. The invocation

uses the lists and vertex labels to construct its local data structure for H and the intersection. The invocation $t(x, X)$ saves edges with at least one end in $X - Y$ in the recursion stack: there is a list of such edges for H and also a list for each tree T_i. (Note that the edges of set $\rho(Y)$ are saved in the stack and also passed to $t(y, Y)$.)

After an iteration of $t(x, X)$ computes $\lambda(H)$ it forms the lists for all recursive calls $t(y, Y)$, plus the remaining edges of its graph and intersection. Then it makes all the recursive calls. Then the vertices in each set Y are relabelled (with label $[Y]$). Using vertex labels and the edges of H and T saved in the recursion stack, the new graph and intersection are constructed. The total time for all processing of these lists is easily seen to be $O(\tau(m + \tau n))$. The space is linear in the total number of edges, $O(m + \tau n)$. (Note that the total number of edges saved in the stack for sets $\rho(Y)$ is at most τn, i.e., at most τ edges for each recursive call.)

The same bound holds for phase two since only one execution of phase two is active at a given time. The desired space bound follows since $\tau n \leq 2m + \delta n$. ∎

The algorithm can be improved on graphs G with more than τn edges. To do this let F be a maximal subgraph of G that consists of τ edge-disjoint subtrees of G. Observe that for any set of new edges N, $G + N$ is τ-edge-connected if and only if $F + N$ is. The "only if" direction follows from the fact that for any cut C of G consisting of c edges, F contains at least $\min\{\tau, c\}$ edges of C.

In summary for any graph G (even if $m < \tau n$) proceed as follows: Find the above subgraph F using the algorithm of [NagI]. Then run our connectivity augmentation algorithm on F.

Theorem 6.1. The edge-connectivity augmentation problem for undirected graphs can be solved in time $O(m + \tau^2 n \log n)$ and space $O(m + \delta n)$.

Proof. The algorithm of [NagI] uses $O(m)$ time. Substituting τn for m in Lemma 6.5 gives the

time bound. This quantity also bounds the time for the δ connectivity incrementations, since this involves just δ iterations of round robin, each using time $O(\tau n \log n)$. ∎

7. Augmenting directed edge connectivity.

This section solves the edge connectivity augmentation problem for directed graphs. The approach is similar to Section 6 although more complicated, e.g., the connectivity does not get incremented from λ to δ.

We use notation similar to Section 6: For a real-valued function f, $f(S) = \sum\{f(s) \mid s \in S\}$. We use $\rho(S)$ as an abbreviation for $|\rho(S)|$; $\rho(S)$ always refers to the original graph G; we refer to other graphs H by writing $\rho_H(S)$. Similarly for $\delta(S)$. Although $\delta(S)$ is the out-degree function and δ represents the desired increase in connectivity, no confusion will be caused by this notation.

The algorithm is based on this duality relation proved in [F]: The minimum number of edges needed to increase the edge connectivity of a directed graph to τ is precisely the maximum, over all subpartitions $\mathcal{P}$ of V, of quantities $\sum\{\tau - \rho(S) \mid S \in \mathcal{P}\}$ and $\sum\{\tau - \delta(S) \mid S \in \mathcal{P}\}$. Our development provides another proof of this relation. It is easy to see that the number of edges is at least as large as the subpartition quantities.

As in Section 6 the notion of extreme set is central. An *in-extreme set* is a set of vertices X such that any nonempty proper subset has larger in-degree, i.e., $\emptyset \subset Y \subset X$ implies $\rho(Y) > \rho(X)$. An *out-extreme set* is the analogous notion using out-degree $\delta(X)$.

In contrast with undirected graphs the in-extreme sets do not nest. They do have the following useful property.

Lemma 7.1. The union of two in-extreme sets that cross is in-extreme. Similarly for out-extreme.

Proof. Consider crossing in-extreme sets X and Y. Let Z be an in-extreme set of smallest in-degree contained in $X \cup Y$. By submodularity $\rho(X)+\rho(Z) \geq \rho(X \cap Z)+\rho(X \cup Z)$. Since $Z \not\subseteq Y$, $X \cap Z \neq \emptyset$;

thus since X is in-extreme, $\rho(X \cap Z) \geq \rho(X)$. The definition of Z implies $\rho(X \cup Z) \geq \rho(Z)$. Adding these two inequalities shows that all three inequalities hold with equality. Thus $\rho(X \cap Z) = \rho(X)$. This implies $X \subseteq Z$. Similarly $Y \subseteq Z$, i.e., $Z = X \cup Y$ as desired. ∎

We move to a related notion. Consider a set of vertices Y containing vertex y. Y is *weakly in-extreme for y* if $\emptyset \subset S \subset Y$ implies $\rho(S) \geq \rho(Y)$, with equality only if $y \in S$. Similarly for *weakly out-extreme for x*. Clearly an in-extreme set is weakly in-extreme for any of its vertices. Below we define critical sets and sets $Y(y)$ and use the fact that both are weakly extreme.

Lemma 7.2. Let X be weakly out-extreme for x, Y weakly in-extreme for y, and $y \in X$. Then X and Y nest.

Proof. Assume the sets do not nest. By submodularity $\delta(X) + \delta(\overline{Y}) \geq \delta(X - Y) + \delta(X \cup \overline{Y})$. Since X is weakly out-extreme and $X - Y \neq \emptyset$, $\delta(X - Y) \geq \delta(X)$. This implies $\rho(Y) = \delta(\overline{Y}) \geq \delta(X \cup \overline{Y}) = \rho(Y - X)$. Since Y is weakly in-extreme and $Y - X \neq \emptyset$ this implies $y \in Y - X$, a contradiction. ∎

The augmentation algorithm has two steps. The Find_degrees Step computes values $\overline{\rho}(x)$ and $\overline{\delta}(x)$ for each vertex x, giving the number of edges directed to and from x, respectively, that must be added to achieve connectivity τ. The Add_edges Step adds the new edges to the graph using a modification of the $\delta = 1$ algorithm of Section 2. We now describe the two steps. The Add_edges Step departs more radically from Section 6. Most parts of the algorithm are done symmetrically for in-degrees and out-degrees; we describe such parts for in-degrees.

The Find_degrees Step constructs $\overline{\rho}$ using a tree T similar to Section 6 (as mentioned, $\overline{\delta}$ is analogous). Each node of T is labelled with a vertex x of G, an in-extreme set $M(x)$ and a nonnegative integer $\overline{\rho}(x)$. The definition is the same as Section 6, changing d to ρ, $\overline{d}$ to $\overline{\rho}$ and extreme to in-extreme.

Observe that T is well-defined in the directed case: For a node x of T, the maximal in-extreme sets of G contained in $M(x) - x$ exist and form a subpartition of $M(x)$, by Lemma 7.1.

We give two properties of T similar to Section 6 using the following terminology. Consider a set S, $\emptyset \subset S \subset V$. S is *complete for* ρ if $\rho(S) + \overline{\rho}(S) \geq \tau$; S is *in-critical* if equality holds [F]. For example if $\overline{\rho}(x)$ is positive then $P(x)$ is in-critical. Note that an in-critical set S is weakly in-extreme for any $x \in S$ with positive $\overline{\rho}(x)$. Also the set V is never in-critical. Function $\overline{\rho}$ is *complete* if every set is complete. Similarly for $\overline{\delta}$ and out-critical sets. S is *critical* if it is in-critical or out-critical.

(i) The function $\overline{\rho}$ defined by T is complete.

(ii) $\max\{\overline{\rho}(V), \overline{\delta}(V)\}$ new edges are needed to achieve τ-edge-connectivity.

To prove (i) consider a set S, $\emptyset \subset S \subset V$. S contains an in-extreme set of in-degree at most $\rho(S)$. Thus it suffices to show that for any in-extreme set S with $\rho(S) < \tau$, some node z of T has $z \in S \subseteq M(z)$. Suppose $S \subseteq M(x)$ (for instance this is true for $x = r$). If $x \notin S$ then S is contained in a maximal in-extreme set of $M(x) - x$ by Lemma 7.1; thus $S \subseteq M(y)$ for some child y of x. Repeating this argument gives the desired node z of T.

Property (ii) follows immediately from [F]. For completeness we repeat the proof, along with a proof of this useful fact [F]:

(a) The union of two in-critical sets that cross is in-critical.

Property (a) follows from a simple submodularity argument (note that $\overline{\rho}$ is modular, i.e., $\overline{\rho}(X) + \overline{\rho}(Y) = \overline{\rho}(X \cup Y) + \overline{\rho}(X \cap Y)$).

Now we prove (ii). By symmetry assume $\overline{\rho}(V) \geq \overline{\delta}(V)$. The argument of Section 6 shows that if V has a subpartition into maximal in-critical sets, containing each vertex with positive $\overline{\rho}$, then $\overline{\rho}(V)$ edges are needed. As noted above each vertex with positive $\overline{\rho}$ is in an in-critical set. So suppose that

two maximal in-critical sets P and Q meet. By property (a), $P \cup Q = V$. Thus $\delta(P-Q) = \rho(Q)$ and $\delta(Q-P) = \rho(P)$. This implies $\overline{\delta}(V) \geq \overline{\delta}(P-Q) + \overline{\delta}(Q-P) \geq \overline{\rho}(Q) + \overline{\rho}(P) \geq \overline{\rho}(V)$. The initial assumption implies equality holds. Thus sets $P-Q$ and $Q-P$ give the desired subpartition. This proves (ii). (For use at the end of Lemma 7.5 note that we have shown $\overline{\rho}(P \cap Q) = 0$, since $\overline{\rho}(Q) + \overline{\rho}(P) = \overline{\rho}(V)$.)

Now we describe the Find_degrees Step. It is a modified version of procedure t of Section 6. The main differences, due to the fact that extreme sets do not nest, are that sets Y do not get contracted, and phase two changes for $X = V$. We state the algorithm in full.

Start by setting each $\overline{\rho}(x)$ to 0. Let Y_0 be a set of in-degree λ. Choose any vertex $r \notin Y_0$ and execute $t(r, V)$. Here $t(x, X)$ is a recursive procedure, called with X an in-extreme set containing vertex x. It constructs the subtree of T rooted at node x with $M(x) = X$, plus all function values for nodes in this subtree. The main data structure for t is a multidigraph H that gets repeatedly modified. The vertices of H consist of X plus a new vertex a. When we contract a set of vertices in H, any parallel edges that result are included in the new multidigraph H. Recall that $\lambda_a(H)$ denotes the smallest number of edges directed to an a-set in H.

Procedure $t(x, X)$ starts by creating a new node of T labelled x and setting $M(x) \leftarrow X$. Phase one initializes multidigraph H by starting with G and doing the following: If $X \neq V$ then contract $V - X$ into a single vertex a; if $X = V$ then add a new vertex a; in both cases add τ (directed) edges ax. Then repeat the following steps until phase one is complete:

If $\lambda_a(H) \geq \tau$ then stop (phase one is complete). Otherwise find all in-extreme a-sets Y of H that have in-degree $\lambda_a(H)$. Do the following for each such Y. Arbitrarily choose a vertex $y \in Y$ and call $t(y, Y)$. This returns a subtree rooted at a node labelled y; make node y a child of x in T. In H add $\overline{\rho}(z)$ new edges az for each $z \in Y$.

Phase two works in two cases. First assume $X \neq V$. In this case modify H by deleting the τ

edges ax that phase one added (keep any other edges ax). Then set $\overline{\rho}(x) \leftarrow \max\{\tau - \lambda_a(H), 0\}$.

Next assume $X = V$. Choose a set of vertices $A \subseteq Y_0$ with $\overline{\rho}(A) \geq \delta$. For each vertex $a \in A$ define a multidigraph H_a by starting with G and adding $\overline{\rho}(z)$ edges az for each vertex $z \neq a$. Set $\overline{\rho}(r) \leftarrow \max\{\tau - \lambda_a(H_a), 0 \mid a \in A\}$. ∎

The proof that this procedure correctly constructs T is a simple version of Lemma 6.4. In phase one observe that an in-extreme a-set of in-degree $\lambda_a(H)$ is a maximal in-extreme set of H. Furthermore an in-extreme a-set Y of H is in-extreme in G (removing the new edges az keeps Y in-extreme). Thus the children y of x are constructed correctly and phase one is correct. Phase two is correct for $X \neq V$ as in Lemma 6.4. Suppose $X = V$. For any set $\emptyset \subset S \subset V$ that is not complete (i.e., $\rho(S) + \overline{\rho}(S) < \tau$) $A - S \neq \emptyset$ and $r \in S$. Thus it is easy to see that $\overline{\rho}(r)$ is set correctly.

Now we show that the total time for $t(r, V)$ is $O(\delta\tau(m + \delta n)\log n)$. First consider phase one. Each invocation $t(x, X)$ executes the round robin connectivity algorithm, building a complete k-intersection for $k = 1, \ldots, \tau$; each execution of round robin begins with the previously computed intersection. For each k the in-extreme a-sets of in-degree k are found using operations $(i) - (ii)$ of Section 2.

Estimate the time using r-graphs as in the proof of Lemma 6.5. Each invocation of t executes round robin τ times, and each r-graph is contained in the final r-graph, for index τ. Thus it suffices to show that in any level of tree T the r-graphs for τ have a total of $O(m + \delta n)$ edges. This follows from the following observations: A vertex of G is in at most one such r-graph. Thus for any edge of G, an end is a vertex in at most two such r-graphs. A given vertex z has $\overline{\rho}(z)$ edges az added in phase one, and $\overline{\rho}(z) \leq \delta$. Finally vertex x in the invocation $t(x, X)$ has τ edges ax (recall $\tau n \leq m + \delta n$). The time bound for phase one follows.

Phase two has the same time bound. The argument for phase one applies to invocations with

$X \neq V$. For $X = V$ note that there are δ connectivity computations, each on a graph with $O(m + \delta n)$ edges.

The space for the algorithm is $O(m + \delta n)$. Implementation details are similar to Section 6: The contracted vertices a are treated using vertex labels. Each invocation $t(y, Y)$ is given a list of the edges of its graph H. The invocation $t(x, X)$ saves edges with at least one end in $X - Y$ in the recursion stack, using a list of such edges for H and also a list for each tree T_i. (The edges of set $\rho(Y)$ are saved in the stack and also passed to $t(y, Y)$.) One extra detail is needed so the invocation $t(x, X)$ can construct the intersection after all recursive calls $t(y, Y)$: In phase two of $t(y, Y)$, note that $\lambda_a(H) > \rho(Y)$. Phase two begins by computing a complete $\rho(Y)$-intersection for the graph H without the edges az added in phase one. This intersection is saved in the recursion stack for use by the calling invocation $t(x, X)$, as mentioned. The rest of the phase two computation for $t(y, Y)$ proceeds with the edges az of phase one added to H. The time and space for this implementation are shown to be as desired, as in Section 6.

The quantities $\overline{\rho}(V)$ and $\overline{\delta}(V)$ as calculated by procedure t may differ. Property (ii) shows the larger equals the number of edges needed to achieve connectivity τ. The Find_degrees Step concludes by making the quantities equal: Choose a vertex x arbitrarily and increase $\overline{\rho}(x)$ or $\overline{\delta}(x)$ to make $\overline{\rho}(V) = \overline{\delta}(V)$.

We now introduce the basic ideas of the Add_edges Step by giving an overview. The Add_edges Step works simultaneously on in-degrees and out-degrees. Most parts of the algorithm are symmetric, in which case we describe the in-degrees case and indicate that out-degrees are symmetric. Functions ρ and $\overline{\rho}$ are maintained to refer to the current graph. More precisely when an edge xy is added, $\rho(y)$ is increased by one and $\overline{\rho}(y)$ is decreased by one. Out-degrees are similar.

For a set of vertices A, an *A-set* is a nonempty set of vertices disjoint from A. The algorithm maintains a sets of vertices A_y. At any time A_y is contained in an in-critical set. λ_y denotes the smallest number of edges entering an A_y-set in the current graph. The algorithm has three main

features. It adds edges so that λ_y increases; concurrently the set A_y gets enlarged; edges may be added that make $\overline{\rho}$ incomplete, but the algorithm backtracks and removes these edges so $\overline{\rho}$ is complete. Symmetrically the algorithm maintains set A_x and value λ_x which behave similarly.

Notions similar to Section 2 are used to add an edge: $\mathcal{Y}$ is a set of vertices containing one vertex with positive $\overline{\rho}$ from each in-extreme A_y-set. (As in Section 2 the in-extreme A_y-sets are disjoint. Unlike Section 2 $\mathcal{Y}$ contains no analog of a_y.) For each vertex $y \in \mathcal{Y}$, $Y(y)$ is the maximal A_y-minsink satisfying $Y(y) \cap \mathcal{Y} = \{y\}$.

Now we establish two basic properties for the algorithm. In the following lemma assume that A_y is an arbitrary set, and $\mathcal{Y}$ and $Y(y)$ are defined as above.

Lemma 7.3. Let C be an in-critical A_y-set containing a vertex $y \in \mathcal{Y}$. Then $C \subseteq Y(y)$.

Proof. By submodularity $\rho(Y(y)) + \rho(C) \geq \rho(Y(y) \cup C) + \rho(Y(y) \cap C)$. Examining λ_y shows $\rho(Y(y) \cup C) \geq \lambda_A = \rho(Y(y))$. Since C is in-critical and meets $Y(y)$, $\rho(Y(y) \cap C) \geq \rho(C)$. Combining the last two inequalities shows all three inequalities hold with equality. Since $\rho(Y(y) \cap C) = \rho(C)$, in-criticality implies $\overline{\rho}(C - Y(y)) = 0$. Since $\rho(Y(y) \cup C) = \lambda_A$, this implies $C \subseteq Y(y)$. ∎

Now suppose $\overline{\rho}$ is complete but adding edge xy makes it incomplete. Thus before xy is added there is an in-critical set C containing x and y. Let C_A denote the in-critical set containing A_y. We shall see that the algorithm chooses y so that $y \in \mathcal{Y}$ and $x \notin Y(y)$.

Lemma 7.4. Let C be an in-critical set containing vertices x and y, where $y \in \mathcal{Y}$ and $x \notin Y(y)$. Then $C_A \cup C$ either equals V or is an in-critical set with $\overline{\rho}(C_A \cup C) > \overline{\rho}(C_A)$.

Proof. Vertex x shows $C \not\subseteq Y(y)$. Thus Lemma 7.3 implies C meets A_y. Hence C_A and C are in-critical sets that meet. Their union is V or in-critical, by (a). The Lemma follows since $y \in C - C_A$.
∎

Now we state the algorithm for the Add_edges Step. It consists of an Initialization Step, followed by a sequence of phases (we shall see there are at most 6δ phases). At the end of the last phase $\overline{\rho}$ and $\overline{\delta}$ are complete and $\overline{\rho}(V) = \overline{\delta}(V) \leq 2\delta$. The Critical_sets Step then completes the algorithm. (This step is essentially Frank's algorithm).

The Initialization Step sets A_y to the vertices with positive $\overline{\rho}$ in some in-critical set. Similarly for A_x.

A phase calculates $\mathcal{X}$ and $\mathcal{Y}$ and then repeatedly adds an edge until $\mathcal{X}$ or $\mathcal{Y}$ becomes a singleton set. We now give the details.

To calculate $\mathcal{Y}$, if $\lambda_y < \tau$ then proceed as above: Choose $\mathcal{Y}$ as a set of vertices containing one vertex with positive $\overline{\rho}$ from each in-extreme A_y-set. For each vertex $y \in \mathcal{Y}$, $Y(y)$ is the maximal A_y-minsink satisfying $Y(y) \cap \mathcal{Y} = \{y\}$. If $\lambda_y \geq \tau$ then let $\mathcal{Y}$ contain every vertex not in A_y having positive $\overline{\rho}$. Each $y \in \mathcal{Y}$ has $Y(y) = \emptyset$. Treat $\mathcal{Y}$ as a multiset with $\overline{\rho}(y)$ copies of y. Thus a vertex y of $\mathcal{Y}$ can be used $\overline{\rho}(y)$ times in Case 1 below. $\mathcal{X}$ and $X(x)$ are symmetric.

Each repetition in a phase executes the appropriate case below.

Case 1: $\min\{|\mathcal{X}|, |\mathcal{Y}|\} \geq 2$. Choose vertices x and y so that $x \in \mathcal{X} - Y(y)$ and $y \in \mathcal{Y} - X(x)$. Add edge xy to the graph. Remove x from $\mathcal{X}$ and y from $\mathcal{Y}$. Continue with the next repetition of the current phase.

Case 2: $\min\{|\mathcal{X}|, |\mathcal{Y}|\} = 1$. Check if $\overline{\rho}$ and $\overline{\delta}$ are complete.

Case 2.1: $\overline{\rho}$ or $\overline{\delta}$ is incomplete. Let $e_1, \ldots, e_k$ be the sequence of edges added in this phase. Find the first edge e_i that makes $\overline{\rho}$ or $\overline{\delta}$ incomplete. Delete edges $e_i, \ldots, e_k$ from the graph. If $\overline{\rho}(V) \leq 2\delta$ then go to the Critical_sets Step. Otherwise enlarge set A_x or A_y (or both): Suppose an in-critical set C contains both ends of e_i. Then add the vertices of C with positive $\overline{\rho}$ to A_y. Similarly for an out-critical set. Start the next phase.

Case 2.2: $\overline{\rho}$ and $\overline{\delta}$ are complete. If $\overline{\rho}(V) \leq 2\delta$ then go to the Critical_sets Step. We state the

remainder of this step for the case $|\mathcal{Y}| = 1$. $|\mathcal{X}| = 1$ is symmetric (if both cardinalities are 1, choose $\mathcal{Y}$). Note that $\lambda_y < \tau$ since otherwise $|\mathcal{Y}| = 1$ implies $\overline{\rho}(V) \leq \delta + 1$.

Let $\mathcal{Y} = \{y\}$. Let D_y be the maximal in-critical set containing A_y.

Case 2.2.1: $y \in D_y$. Add the vertices of D_y with positive $\overline{\rho}$ to A_y. (Any remaining A_y-minsink contained y so this increases λ_y.) Start the next phase.

Case 2.2.2: $y \notin D_y$. Let F be the maximal out-critical set containing y (if such exists). Choose $x \notin F \cup Y(y)$ with $\overline{\delta}(x) > 0$. Add edge xy. This increases λ_y by one. Start the next phase.

This completes the description of a phase.

The Critical_sets Step works by repeating the following until $\overline{\rho}(V) = 0$. Choose a vertex y with positive $\overline{\rho}$. Let C be the maximal in-critical set containing y, D the maximal out-critical set containing y. Choose a vertex $x \notin C \cup D$ with positive $\overline{\delta}$. Add edge xy. ∎

In a phase, note that when a vertex a enters A_y it has positive $\overline{\rho}(a)$. It is possible that during a phase $\overline{\rho}(A_y)$ becomes zero: A vertex x chosen in Case 2.2.2 can be in A_x, and symmetrically Case 2.2.2 can choose a vertex in A_y. We shall see that A_y contains at most 2δ vertices. Symmetric remarks hold for A_x.

Lemma 7.5. The Add_edges Step is correct.

Proof. We first analyze the phase algorithm. We prove it is correct by induction. The inductive assertion includes the fact that $\overline{\rho}$ and $\overline{\delta}$ are complete at the start of each phase.

Consider Case 1. We show that the desired vertices x and y exist: Consider any vertices $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. If $x \notin Y(y)$ and $y \notin X(x)$ then x and y are the desired vertices. Otherwise without loss of generality $X(x) \subseteq Y(y)$. This follows by symmetry and using Lemma 7.2 if $\lambda_x, \lambda_y < \tau$; it is trivial otherwise. Choose x and any vertex $y' \in \mathcal{Y} - y$. As in Section 2, $Y(y')$ and $Y(y)$ are

disjoint, so x and y are the desired vertices. (Note that this argument is correct even if $\overline{\delta}$ or $\overline{\rho}$ is incomplete.)

Next consider Case 2.1. Since $\overline{\rho}$ and $\overline{\delta}$ are complete at the start of the phase, the first edge to make a set incomplete is some e_i, $i \geq 1$. Thus deleting edges $e_i, \ldots, e_k$ makes $\overline{\rho}$ and $\overline{\delta}$ complete for the start of the next phase.

Continuing in Case 2.1, suppose an in-critical set C contains both ends of e_i. We show that the conclusion of Lemma 7.4 holds. This is clear if $\lambda_y < \tau$, since Lemma 7.4 applies directly. Suppose $\lambda_y \geq \tau$. (Now Lemma 7.4 does not apply since sets $Y(y)$ are defined differently.) C meets A_y, since otherwise $\rho(C) \geq \tau$ and C is an in-critical set containing y, a contradiction. Now observe that the conclusion of Lemma 7.4 follows from the property C meets A_y (see the proof).

The conclusion of Lemma 7.4 gives two possible outcomes. The first outcome implies $\overline{\rho}(V) \leq 2\delta$, so the Critical_sets Step is executed next. If the second outcome holds then the new in-critical set shows that A_y is enlarged correctly. We conclude that Case 2.1 is correct.

In Case 2.2 the maximal in-critical set containing A_y is well-defined by property (a) (recall $\overline{\rho}(V) \leq 2\delta$). In Case 2.2.1 clearly A_y gets enlarged since it does not contain y.

Consider Case 2.2.2. First we show that vertex x exists. The maximal critical set F is well-defined as above. Lemma 7.2 shows that sets F and $Y(y)$ nest. If $F \subseteq Y(y)$ then x exists since $\rho(Y(y)) = \lambda_y < \tau$, and $\overline{\delta}$ is complete. If $Y(y) \subseteq F$ then x exists since $\overline{\delta}(V) = \overline{\rho}(V) > \overline{\rho}(V - F) \geq \overline{\delta}(F)$ (the last inequality follows since F is out-critical).

Next we show that in Case 2.2.2 $\overline{\rho}$ and $\overline{\delta}$ are complete at the start of the next phase. It suffices to show that no critical set contains both x and y. The maximal out-critical set containing y is F. Lemma 7.3 shows any in-critical set containing y is a subset of $Y(y)$.

This completes the proof that the phase algorithm is correct. We turn to the Critical_sets Step. It is correct because it is essentially Frank's algorithm. For completeness we prove it correct.

Consider a vertex y with positive $\overline{\rho}$. It is in a unique maximal in-critical set. (In proof, if not

then property (a) shows y is in in-critical sets P and Q with $P \cup Q = V$. The proof of (ii) is still valid and shows that $\overline{\rho}(P \cap Q) = 0$. This is a contradiction.) Now the argument of Case 2.2.2 shows that vertex x of the Critical_sets Step always exists.

The Critical_sets Step maintains completeness of $\overline{\rho}$ and $\overline{\delta}$, since no critical set contains both x and y. Thus the algorithm halts with $\overline{\rho}$ and $\overline{\delta}$ both identically zero and complete, i.e., the graph is τ-edge-connected. ∎

We turn to efficiency of the algorithm.

Lemma 7.6. There are at most 6δ phases.

Proof. A phase other than the last ends in one of four ways. It can enlarge A_y, strictly increasing $\overline{\rho}(A_y)$. It can increase λ_y, for λ_y initially less than τ; this may decrease $\overline{\delta}(A_x)$ by one. The other two possibilities are the symmetric ones for out-degrees. (Recall that in Case 2.2.2 λ_y is initially less than τ.)

Since λ_x never decreases, it increases from a value less than τ at most δ times. This implies that $\overline{\rho}(A_y)$ decreases by at most δ. Thus A_y gets enlarged at most $2\delta - 1$ times (Initially $\overline{\rho}(A_y) \geq 1$. Each enlargement increases $\overline{\rho}(A_y)$. A_y is always contained in an in-critical set, so $\overline{\rho}(A_y) \leq \delta$.)

In summary, at most 2δ nonlast phases increase λ_x or λ_y and at most $4\delta - 2$ nonlast phases enlarge A_x or A_y. ∎

Now we supply the remaining implementation details. Consider Case 1. To find x, y, use E sets if $|\mathcal{X}|, |\mathcal{Y}| \geq 3$, else calculate all $X(x)$ and $Y(y)$ sets, as in Section 2. The time is $O(m + \delta n)$.

The rest of the implementation is done in terms of a representation of all in-critical sets. It allows us to check if $\overline{\rho}$ is complete and find all maximal in-critical sets. Symmetrically for out-degrees. The representation is related to procedure t.

The representation of in-critical sets is defined as follows. For a vertex a, construct a multidigraph H_a by starting with G and adding $\overline{\rho}(x)$ edges ax for each vertex $x \neq a$. We shall define a set of vertices A so the in-critical sets of G are precisely the a-minsinks of H_a, for a ranging over all vertices of A. (It is possible for an in-critical set to be an a-minsink for more than one vertex $a \in A$.) Thus we represent the in-critical sets using the poset representations of the a-minsinks of H_a, $a \in A$. We now give three definitions for A, depending on the relative size of $\overline{\rho}(V)$ and τ. (In the last case we represent the in-critical sets with positive $\overline{\rho}$; this suffices for the algorithm.) We also estimate the resources used by each representation. It is convenient to state resource bounds in terms of the bound on the number of edges in the solution graph,

$$m' = m + \delta n.$$

Consider the case $\overline{\rho}(V) > \tau$. Let a be a new vertex not in $V(G)$. Add a to G as an isolated vertex and construct H_a. Use $A = \{a\}$.

To see this definition is correct note that in H_a, an a-set S has in-degree $\rho(S) + \overline{\rho}(S)$. If $\overline{\rho}(V) > \tau$ the a-set V has in-degree larger than τ. Thus the in-critical sets are precisely the a-minsinks.

To construct this representation we first find a complete τ-intersection for a on H_a. In our applications we only need to find the maximal in-critical sets. Thus we use operation (v) on the poset representation to find the maximal a-minsinks. Thus the representation is constructed in time $O(\tau m' \log n)$. To save the representation we record the complete intersection. This uses $O(\tau n)$ space.

Next suppose $\overline{\rho}(V) > \delta$. Choose A so that $\overline{\rho}(A) > \delta$. This choice is correct because any in-critical set C has some $a \in A - C$, whence C is an a-minsink in H_a.

Proceeding as in the first case, the representation is constructed in time $O(\delta\tau m' \log n)$. This construction uses space for one complete intersection, $O(\tau n)$. To save the entire representation for future use requires space $O(\delta\tau n)$. We will also have occasion to update the representation, after

the algorithm adds a new edge xy. We update each complete intersection by deleting one edge ay and replacing it by xy; the latter is done by finding an augmenting path for xy (see [G91]). The time to compute one augmenting path is $O(m')$. Thus the time to update the representation is $O(\delta m')$ (this includes the time to repeat operation (v) in each poset).

Finally let $\overline{\rho}(V)$ be arbitrary. We represent the in-critical sets with positive $\overline{\rho}$. Let $A = \{a \mid \overline{\delta}(a) > 0\}$. Any in-critical set C with $\overline{\rho}(C) > 0$ has $\rho(C) < \tau$. Since $\overline{\delta}$ is complete, $A - C \neq \emptyset$. Now correctness of the representation follows as in the previous case. We shall use this representation when $\overline{\rho}(V) \le \delta$. Thus it uses the same resources as the previous representation.

We now estimate the efficiency of the augmentation algorithm. There are $O(\delta)$ binary searches. Each search makes $O(\log n)$ probes, since a phase adds at most n edges. Thus there are $O(\delta \log n)$ binary search probes. These dominate the time.

We analyze two implementations of the augmentation algorithm. The first implementation is more space efficient. Execute each binary search probe by constructing the representation of in-critical sets. Use the representation for $\overline{\rho}(V) > \delta$ when it applies, and switch to the representation for arbitrary $\overline{\rho}(V)$ when $\overline{\rho}(V) \le \delta$. Each probe constructs the representation in time $O(\delta \tau m' \log n)$. Hence the total time is $O(\delta^2 \tau m' \log^2 n)$. The space is $O(m + \tau n) = O(m + \delta n)$.

The second implementation is more time efficient. First consider the phases with $\overline{\rho}(V) > \tau$. Use the representation of in-critical sets for $\overline{\rho}(V) > \tau$. Each probe constructs this representation, using time $O(\tau m' \log n)$. Hence the time for these phases is $O(\delta \tau m' \log^2 n)$; the space is $O(\tau n)$.

For the remaining phases, modify the algorithm so it checks completeness after each edge is added; the details of the modification are obvious. The number of such checks is $O(\tau + \delta) = O(\tau)$. Use the representations for $\overline{\rho}(V) > \delta$ and $\overline{\rho}(V) \le \delta$ as appropriate. The representations are stored, and updated after each edge is added. The time to initialize the representation is $O(\delta \tau m' \log n)$. Each edge addition uses $O(\delta m')$ time; since τ edges are added this gives total time $O(\delta \tau m')$. The space to save the representation is $O(\delta \tau n)$.

Finally consider the last phase. It is similar to the previous paragraph, but does only 2δ iterations.

To state the final result, recall that $m' = m + \delta n$.

Theorem 7.1. The edge-connectivity augmentation problem for directed graphs can be solved in time $O(\delta \tau m' \log^2 n)$ and space $O(m + \delta \tau n)$, and also time $O(\delta^2 \tau m' \log^2 n)$ and space $O(m')$. ∎

Acknowledgments. Many thanks to András Frank, Dan Gusfield, Dalit Naor and Éva Tardos for their kind help.

References.

[AGU] A.V. Aho, M.R. Garey and J.D. Ullman, "The transitive reduction of a directed graph", *SIAM J. Comput., 1,* 2, 1972, pp. 131–137.

[CLR] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[DKL] E.A. Dinits, A.V. Karzanov and M.V. Lomonosov, "On the structure of a family of minimal weighted cuts in a graph", in *Studies in Discrete Optimization*, A.A. Fridman (Ed.), Nauka Publ., Moscow, 1976, pp. 290–306.

[DM] A.L. Dulmage and N.S. Mendelsohn, "A structure theory of bipartite graphs of finite exterior dimension", *Trans. Roy. Soc. Canada*, Section III, 53, 1959, pp. 1–13.

[E69] J. Edmonds, "Submodular functions, matroids, and certain polyhedra", *Calgary International Conf. on Combinatorial Structures and their Applications*, Gordon and Breach, New York, 1969, pp. 69–87.

[E72] J. Edmonds, "Edge-disjoint branchings", in *Combinatorial Algorithms*, R. Rustin, Ed., Algorithmics Press, New York, 1972, pp. 91–96.

[F] A. Frank, "Augmenting graphs to meet edge-connectivity requirements", *Proc. 31st Annual Symp. on Found. of Comp. Sci.,* 1990, pp. 708–718.

[G90] H.N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking", *Proc. First Annual ACM-SIAM Symp. on Disc. Algorithms,* 1990, pp. 434–443.

[G91] H.N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences", *Proc. 23rd Annual ACM Symp. on Theory of Comp.,* 1991, pp. 112–122.

[GI] D. Gusfield and R.W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press, Cambridge, MA, 1989.

[GT] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *J. Comp. and System Sci., 30,* 2, 1985, pp. 209–221.

[GW] H.N. Gabow and H.H. Westermann, "Forests, frames and games: Algorithms for matroid sums and applications", *Proc. 20th Annual ACM Symp. on Theory of Comp.,* 1988, pp. 407–421; also *Algorithmica*, to appear.

[I] H. Imai, "Network-flow algorithms for lower-truncated transversal polymatroids", *J. Op. Res. Soc. of Japan, 26*, 3, 1983, pp. 186–210.

[KT] A.V. Karzanov and E.A. Timofeev, "Efficient algorithm for finding all minimal edge cuts of a nonoriented graph", *Kibernetika*, 2, 1986, pp. 8–12; translated in *Cybernetics*, 1986, pp. 156–162.

[L] G. Laman, "On graphs and rigidity of plane skeletal structures", *J. Engineering Math. 4*, 4, 1970, pp. 331–340.

[LY] L. Lovász and Y. Yemini, "On generic rigidity in the plane", *SIAM J. Alg. Disc. Meth., 3,* 1982, pp. 91–98.

[N] M. Nakamura, "On the representation of the rigid sub-systems of a plane link system", *J. Op. Res. Soc. of Japan, 29*, 4, 1986, pp. 305–318.

[NGM] D. Naor, D. Gusfield and C. Martel, "A fast algorithm for optimally increasing the edge-connectivity", *Proc. 31st Annual Symp. on Found. of Comp. Sci.,* 1990, pp. 698–707.

[NagI] H. Nagamochi and T. Ibaraki, "Linear time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph", *Algorithmica*, to appear.

[NakI] M. Nakamura and M. Iri, "Fine structures of matroid intersections and their applications", *Proc. Int. Symp. Circuits and Systems*, 1979, pp. 996–999.

[PQ] J.-C. Picard and M. Queyranne, "On the structure of all minimum cuts in a network and applications", *Math. Prog. Study 13*, 1980, pp. 8–16.

[R] A. Recski, *Matroid Theory and its Applications in Electric Network Theory and in Statics*, Springer-Verlag, New York, 1989.

[RT] J. Roskind and R.E. Tarjan, "A note on finding minimum-cost edge-disjoint spanning trees", *Math. Op. Res. 10*, 4, 1985, pp. 701–708.

[Tar72] R.E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM J. Comput., 1,* 2, 1972, pp. 146–160.

[Tar83] R.E. Tarjan, *Data Structures and Network Algorithms,* SIAM Monograph, Philadelphia, PA, 1983.

[Tay] T-S. Tay, "Rigidity of multi-graphs. I. Linking rigid bodies in n-space", *J. Comb. Th. B, 36,* 1984, pp. 95–112.

[WN] T. Watanabe and A. Nakamura, "Edge-connectivity augmentation problems", *J. Comp. and System Sci., 35,* 1, 1987, pp. 96–144.

[WW] N. White and W. Whiteley, "The algebraic geometry of motions of bar-and-body frameworks", *SIAM J. Alg. Disc. Meth., 8,* 1, 1987, pp. 1–32.

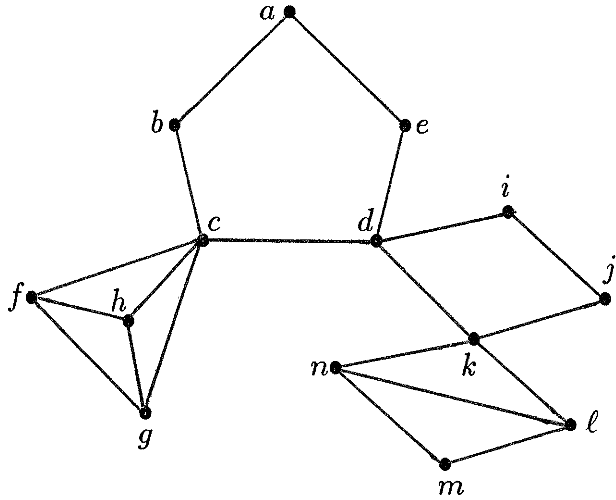


Figure 1.a. 2-edge-connected graph.

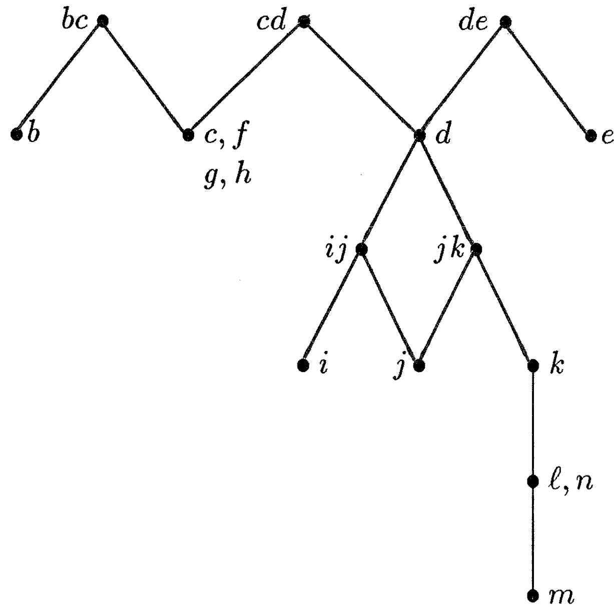


Figure 1.b. Poset of a-minsinks.

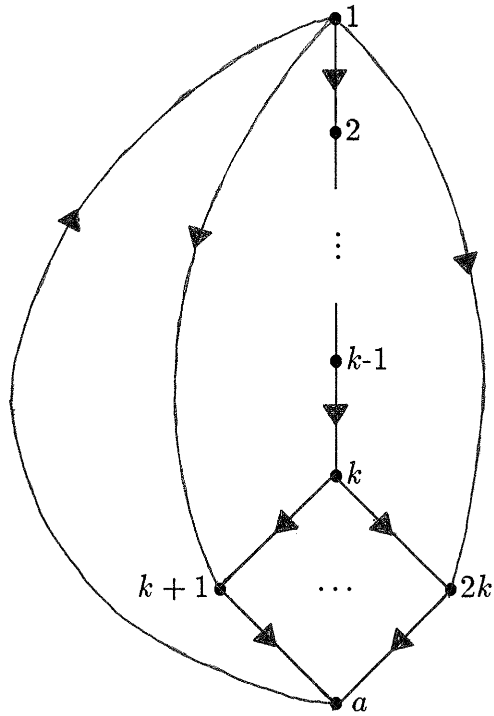


Figure 2.a. Directed graph with large poset.

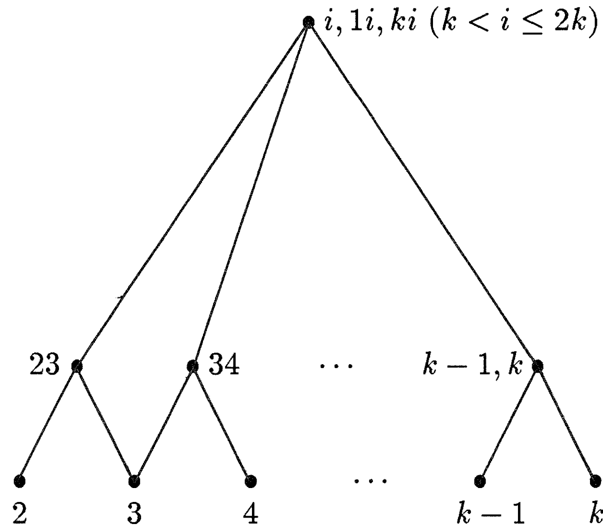


Figure 2.b. The poset.

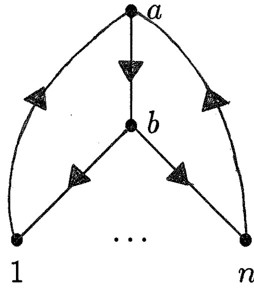


Figure 3. Directed graph with $\Theta(2^n)$ minsinks.

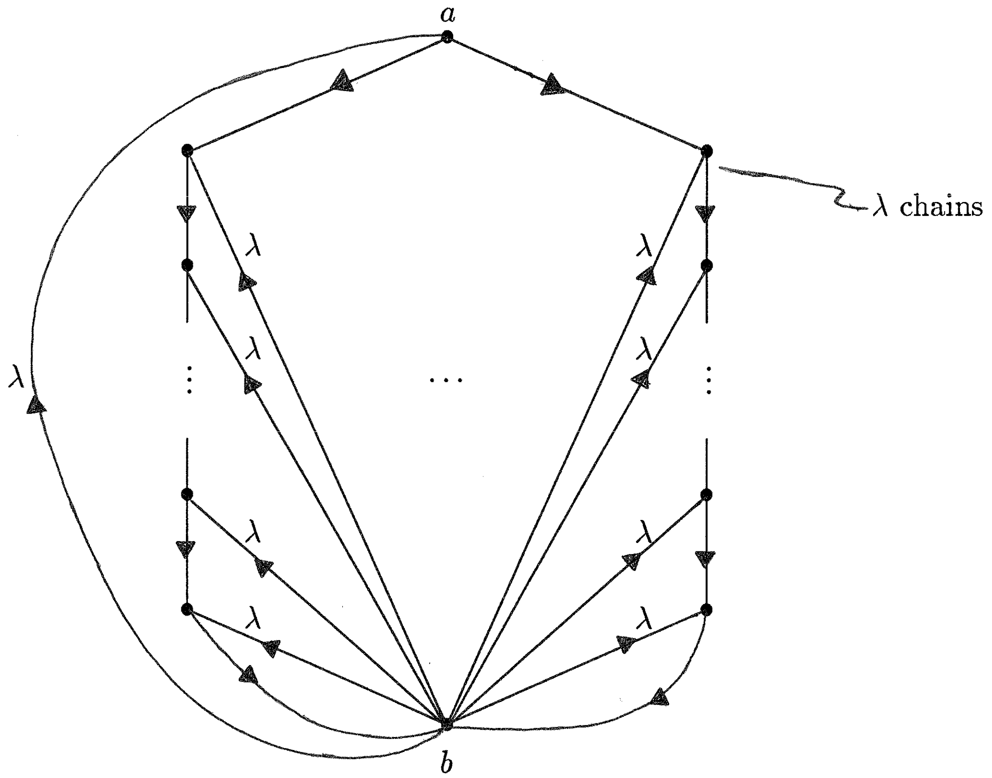


Figure 4. Directed multigraph with $\Theta((n/\lambda)^\lambda)$ mincuts.

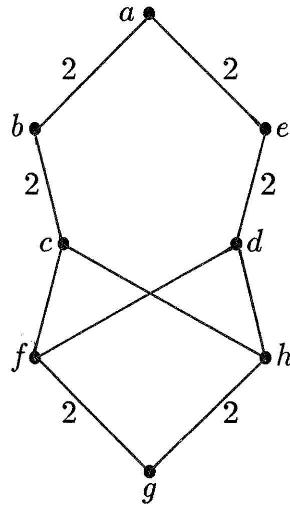


Figure 5.a. 4-edge-connected multigraph.

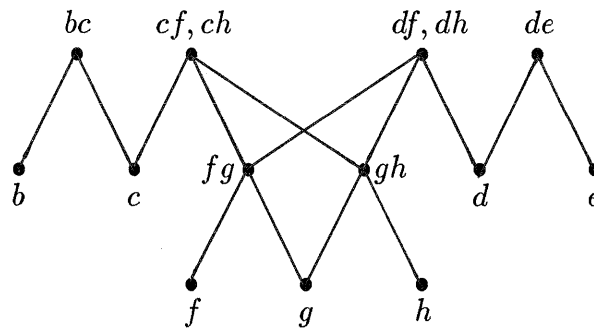


Figure 5.b. Poset of a-minsinks.