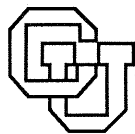
A Matroid Approach to Finding Edge Connectivity and Packing Arborescences

Harold N. Gabow

CU-CS-539-91 August 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Matroid Approach to Finding Edge Connectivity and Packing Arborescences

Harold N. Gabow

CU-CS-539-91 August 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

A Matroid Approach to Finding Edge Connectivity and Packing Arborescences

Harold N. Gabow*

Department of Computer Science

University of Colorado at Boulder

Boulder, CO 80309

hal@cs.colorado.edu

August 9, 1991

Abstract. We present an algorithm that finds the edge connectivity λ of a directed graph in time $O(\lambda m \log(n^2/m))$ and is slightly faster on an undirected graph (n and m denote the number of vertices and edges, respectively). This improves the previous best time bounds, $O(\min\{mn, \lambda^2 n^2\})$ on a directed graph and $O(\lambda n^2)$ on undirected. We present an algorithm that finds k edge-disjoint arborescences on a directed graph in time $O((kn)^2)$. This improves the previous best time bound, $O(kmn + k^3 n^2)$. Unlike previous work, our approach is based on two theorems of Edmonds that link these two problems and show how they can be solved.

---

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

1. Introduction.

The starting point of this work is two characterizations of Edmonds for packing arborescences. Consider a directed graph $G = (V, E)$. An *arborescence* is a directed spanning tree, i.e., an acyclic subgraph where some vertex a has in-degree zero and every other vertex has in-degree one. We sometimes write *a-arborescence* and call a the *root*. The *arborescence packing problem* for vertex a is to construct the greatest possible number of edge-disjoint a-arborescences. An *a-cut* is the set of edges directed from $V - S$ to S, where vertex set S satisfies $\emptyset \neq S \subseteq V - a$. Edmonds' first characterization for packing arborescences is in terms of cuts.

Cut Characterization. In a directed graph the maximum number of edge-disjoint a-arborescences equals the minimum cardinality of an a-cut.

Edmonds gave an algorithmic proof using first principles of graph theory [Ed72] (the algorithm is involved and seems to use exponential time in the worst case). Alternative proofs are in [F, Lo, TL].

The *edge connectivity* of a directed graph G, denoted λ or $\lambda(G)$, is the smallest number of edges whose deletion leaves a graph that is not strongly connected. Such a set of λ edges is a *connectivity cut.* G is *k-edge-connected* if $\lambda \geq k$. Clearly λ is the minimum cardinality of an a-cut in either G or G with all edges reversed. Thus the Cut Characterization gives a way to compute edge connectivity. This observation also holds for undirected graphs G (since the edge connectivity of G equals that of the directed graph where each edge is oriented in both directions).

To state Edmonds' second characterization, recall that a *spanning tree* is a connected acyclic spanning subgraph of an undirected graph. Throughout this paper we refer to a spanning tree (or forest) of a directed graph – for this, simply ignore edge directions.

Matroid Characterization. The edges of a directed graph can be partitioned into k a-arborescences if and only if they can be partitioned into k spanning trees and every vertex except a has in-degree k.

We call this the Matroid Characterization since the condition corresponds to a matroid intersection, as indicated below. For $k = 1$ the Matroid Characterization is simply the definition of arborescence. However for $k \geq 2$ the proof is not easy. Edmonds shows the Matroid Characterization follows from the Cut Characterization [Ed69].

This paper gives two algorithms, corresponding to the two Characterizations. The algorithm for the Matroid Characterization can be used to efficiently compute edge connectivity. The two algorithms together can be used to efficiently pack arborescences. Both algorithms are based on matroid theory. We now discuss the two algorithms and mention some related results. To state resource bounds throughout this paper n and m denote the number of vertices and edges of the given graph, respectively. Note that a directed graph has $m \geq \lambda n$, since every vertex has in-degree at least λ; similarly an undirected graph has $m \geq \lambda n/2$.

The algorithm corresponding to the Matroid Characterization finds k edge-disjoint spanning trees such that every vertex other than a has in-degree k, and k is as large as possible. This can be done using an algorithm for maximum cardinality matroid intersection [Ed65] along with techniques for efficiently implementing matroid intersection and matroid sum algorithms on graphic matroids [GX89b, GW]. However we achieve much better efficiency with a different overall organization which we call round robin. The algorithm using round robin runs in time $O(km \log(n^2/m))$. Round robin is easy to program and uses only simple data structures, arrays and lists.

Round robin can be used to find the edge connectivity, as observed above. The algorithm also produces a connectivity cut. The time is $O(\lambda m \log(n^2/m))$. For undirected graphs a preprocessing step reduces the time to $O(m + \lambda^2 n \log(n/\lambda))$ (since $m = \Omega(\lambda n)$ this is at most the previous bound

and can be smaller). Alternatively round robin can check if a graph is k-edge-connected. The time is given by changing λ to k in the above bounds. For instance round robin can check if an undirected graph is k-edge-connected, for any constant k, in time $O(m + n \log n)$.

Let us summarize previous work on edge connectivity. Early algorithms for computing edge connectivity are given in [ET, Sch]. The best-known time bound for finding the edge connectivity of an undirected graph is $O(\lambda n^2)$; since $m = \Omega(\lambda n)$, this bound is also $O(nm)$. The latter bound was first achieved by Podderyugin [P], the former by Karzanov and Timofeev [KT] (improving [Ti]). Matula [M87] rediscovered both bounds independently. Mansour and Schieber generalized Matula's work to directed graphs, achieving time $O(\min\{mn, \lambda^2 n^2\})$ [MS]. These bounds apply to checking if a graph is k-edge-connected, changing λ to k. An undirected graph can be checked to be 2- or 3-edge-connected in time $O(m)$ ([GI] reduces checking 3-edge-connectedness to checking 3-vertex-connectedness).

Recently Nagamochi and Ibaraki have given a linear-time graph searching algorithm that is helpful in computing edge connectivity of undirected graphs [NI89a]. We use their algorithm in our undirected connectivity algorithm. An application of Matula [M90] is discussed in Section 4. [NI89b] uses it to find a minimum cut on graphs with large capacities (this does not improve any of the above bounds).

All the above general edge connectivity algorithms (but not [NI89a–b]) are based on network flow and Menger's Theorem rather than Edmonds' Characterizations. Recall that Menger's Theorem characterizes the size of a smallest ab-cut (where for given vertices a and b, an ab-cut is a set of edges intersecting every path from a to b) [Ev]. This gives a less direct characterization of edge connectivity, leading to less efficient algorithms. Our time bound is asymptotically never larger than any of the above general bounds and it is smaller for graphs with $m = o(n^2)$ (since $\lambda m \log(n^2/m)$ is $O(\lambda n^2)$ and is $o(\lambda n^2)$ when $m = o(n^2)$).

Our algorithm for the Cut Characterization arranges a set of edges into k edge-disjoint arbores-

cences, if possible. It runs in time $O((kn)^2)$. Thus our two algorithms together find k edge-disjoint arborescences in a given graph in total time $O((kn)^2)$, if they exist. This improves the previous best-known bound $O(kmn + k^3n^2)$ due to Tong and Lawler [TL]. Earlier packing algorithms include [T74, Sh]. For the special case $k = 2$ Tarjan gives an algorithm that runs in time $O(m)$ if a linear-time set merging algorithm is used [T76, GT]. Our two step approach to packing also facilitates finding a minimum cost packing (see Section 4).

The analysis of our algorithms gives a new proof of the two Characterizations, so our development is self-contained. The remainder of this paper is organized as follows. Section 2 presents the round robin algorithm for the Matroid Characterization, and the edge connectivity algorithms. Section 3 presents the arborescence packing algorithm, and proves the two Characterizations. Section 4 states several extensions of the results of this paper. The rest of this section gives notation and reviews matroids. Note that the paper is presented in terms of elementary graph theory. Material on matroids is included to place the results in their proper setting, but our proofs do not logically draw upon matroid theory.

If S is a set and e an element, $S + e$ denotes $S \cup \{e\}$ and $S - e$ denotes $S - \{e\}$. The function $\log n$ denotes logarithm base two.

For any graph G, $V(G)$ and $E(G)$ denote the vertex set and edge set of G, respectively; we use V and E when the graph is understood. 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. A directed edge vw has *tail* v and *head* w.

Consider a directed graph on vertices V and a subset of vertices S. $\rho(S)$ denotes the set of all edges directed from $V - S$ to S; if F is a set of edges then $\rho_F(S)$ denotes $\rho(S) \cap F$. In these notations we abbreviate an argument $\{v\}$ to v, e.g., $\rho_F(v)$; also for a set of edges L we abbreviate an argument $V(L)$ to L, e.g., $\rho_F(L)$. The function δ is analogous to ρ for edges directed from a set; thus $\delta(S) = \rho(V - S)$.

Introductory treatments of matroids are given in [L76, R, W]. A *matroid* consists of a finite set S and a family of subsets of S called *independent sets*, that satisfy simple axioms. This paper works with several specific matroids, so we concentrate on describing them rather than matroids in general. The *graphic matroid* of an undirected graph G, denoted $\mathcal{G}(G)$ (or $\mathcal{G}$ when the graph is understood) has $S = E(G)$ and independent sets the forests of G. A *partition matroid* on a set S is determined by a partition of S into sets S_i and nonnegative integers d_i, $i = 1, \ldots, p$; $A \subseteq S$ is independent if $|A \cap S_i| \leq d_i$ for each i.

For matroids M_i, $i = 1, \ldots, k$ on the same set S, the *matroid sum* $\bigvee_{i=1}^{k} M_i$ is a matroid on S, whose independent sets are all subsets of S that can be partitioned into k subsets, the ith subset independent in M_i. A special case is the *k-fold sum* of a matroid M, $\bigvee_{i=1}^{k} M$, also denoted M^k. For example an independent set of $\mathcal{G}^k$ is a subgraph that can be partitioned into k forests. For matroids M_i, $i = 1, 2$ on S, a (*matroid*) *intersection* is a subset of S that is independent in both M_i; the set of all such intersections is denoted $M_1 \cap M_2$. (This is not a matroid.) Polynomial-time algorithms for finding a maximum cardinality independent set of a matroid sum or a maximum cardinality matroid intersection are due to Edmonds [Ed65]; more efficient algorithms are in [GW, GX89a–b].

Let A be a set of elements of S for an arbitrary matroid. If A is independent and spans an element e, then $C(e, A, M)$ denotes the *fundamental circuit* of e in A. We drop the last argument M when it is clear from context. For example in a graphic matroid with edge e and spanning tree T, $C(e, T)$ (or $C(e, T, \mathcal{G})$) is the fundamental cycle of e in T.

2. The round robin algorithm and edge connectivity.

This section presents the round robin algorithm and uses it to solve the intersection problem for the Matroid Characterization. Then it applies round robin to compute edge connectivity.

The bulk of the section is devoted to round robin and the intersection problem. It is organized

in a top-down fashion as follows. It starts by reviewing facts about matroids, giving a definition of augmenting path and a labelling algorithm to find such a path. Then a basic lemma is proved. This leads to the round robin algorithm. To make round robin efficient a refined labelling algorithm, the "cyclic scanning algorithm", is presented. To make that efficient the "cycle traversal algorithm" is presented, along with the "augmenting algorithm". Thus the final algorithm at the highest level is round robin, with portions implemented by the cyclic scanning algorithm, the cycle traversal algorithm, and the augmenting algorithm. Although the first labelling algorithm is not used we present it to achieve a simple proof of the basic lemma, which is fundamental to our study.

We begin by defining our matroid intersection problem and briefly reviewing matroid intersections and sums. Fix a directed graph G with a distinguished vertex a. Let k be any positive integer. Define $\mathcal{G}^k$ and $\mathcal{D}^k$ to be matroids over the edge set $E(G)$ as follows. $\mathcal{G}^k$ is the k-fold sum of the graphic matroid of G. $\mathcal{D}^k$ is the partition matroid where an independent set contains no edges of $\rho(a)$ and at most k edges of $\rho(v)$ for every $v \neq a$ ($\mathcal{D}^k$ is easily seen to be a k-fold sum). A *k-intersection (for a on G)* is a set in $\mathcal{G}^k \cap \mathcal{D}^k$, i.e., it can be partitioned into k forests and contains at most k edges directed to any vertex but none to a. A k-intersection is *complete* if it contains precisely $k(n-1)$ edges. Thus the Matroid Characterization corresponds to finding a complete k-intersection. We do this by finding a maximum cardinality intersection.

Let us review the algorithm to find a maximum cardinality matroid intersection, specialized to $\mathcal{G}^k \cap \mathcal{D}^k$. (For a complete discussion of the general matroid algorithm see e.g. [L75; L76, Section 8.4; R, Section 13.1].) The algorithm maintains a k-intersection T partitioned into forests T_i, $i = 1, \ldots, k$. (As already mentioned the directions of edges in these forests is irrelevant.) The algorithm repeats the following step: Search for an "augmenting path" P. If P is found then "augment T along P", thereby enlarging T by one edge; then repeat the step. If no augmenting path exists then halt with the maximum cardinality intersection T.

Before defining the notion of augmenting path we describe how T is enlarged. Let z be a vertex

with $|\rho_T(z)| < k$. We wish to enlarge T by one edge, increasing the in-degree of z by one. To do this an edge $e_1 \in \rho(z) - T$ is added to some forest T_i. If this results in a new forest we are done. Otherwise adding e_1 creates a cycle, which is broken by removing some edge e_2 from T_i. Edge e_2 is treated in one of two ways. It can be added to another forest T_j; this is done using the above procedure for adding e_1. Alternatively it can be deleted from T; if e_2 is directed to vertex y, this decreases the in-degree of y; this is remedied by adding an edge $e_3 \in \rho(y) - T$, using the above procedure for adding e_1. This pattern continues until some edge e_ℓ is added to a forest without creating a cycle. In this case the sequence $e_1, \ldots, e_\ell$ is an augmenting path P; the process just described is called augmenting T along P.

Now we give the precise meaning of these terms. Let z be a vertex with $|\rho_T(z)| < k$. A *partial augmenting path P from z* is a sequence of edges e_i, $i = 1, \ldots, \ell$ satisfying the following conditions.

(*i*) $e_1 \in \rho(z) - T$.

(*ii*) For each $i < \ell$ either

(*a*) $e_{i+1} \in C(e_i, T_j, \mathcal{G})$, where T_j contains e_{i+1} but not e_i, or

(*b*) e_{i+1} is directed to the same vertex as e_i, where T contains e_i but not e_{i+1}.

(*iii*) A pair e_i, e_{i+1} as in (*ii.a*) is called a *swap*; to *execute the swap* is to replace e_{i+1} in T_j by e_i. Then executing each swap of P gives a new collection of forests.

Note that in condition (*ii.a*) it is implicit that the fundamental cycle $C(e_i, T_j, \mathcal{G})$ exists. (If it does not then P is an augmenting path, defined below.) It is easy to see that each edge of P is in one or two swaps. Also in (*iii*) each of the new forests T_i spans precisely the same vertices as its original counterpart.

Condition (*iii*) is essential and is not implied by the others. It can be guaranteed by any of several rules, e.g., path P has no "shortcuts" (i.e., no subsequence of P satisfies the definition) or P is formed using the cyclic scanning rule [RT, GW, and the references of RT] or the topological

numbering rule [GX89a–b]. This paper uses the first two rules.

An edge is *joining (for T_h)* if its ends are in different subtrees of forest T_h. An *augmenting path P from z* is a partial augmenting path from z containing precisely one joining edge, the last edge e_ℓ. To *augment T along P* is to execute each swap of P and add e_ℓ to T_h, where e_ℓ is joining for T_h. Clearly augmenting T along P makes T to a new k-intersection T'. T' contains one more edge than T since the in-degree of z increases by one and no other in-degree changes.

The reader familiar with matroids will observe that in our definition of augmenting path, a maximal subsequence of edges e_i, $i = r, \ldots, s$ with each consecutive pair forming a swap corresponds to the condition $e_s \in C(e_r, T, \mathcal{G}^k)$ in the matroid definition of augmenting path (so edges $e_{r+1}, \ldots, e_{s-1}$ do not appear in the matroid definition). Also condition $(ii.b)$ essentially corresponds to the condition $e_i \in C(e_{i+1}, T, \mathcal{D}^k)$ in the matroid definition of augmenting path. A slight difference is that if e_i and e_{i+1} are directed to x, the matroid definition requires that $|\rho_T(x)| = k$, which we do not require. This relaxation is for convenience and causes no harm: if $|\rho_T(x)| < k$ then the augmenting path could actually begin with edge e_{i+1}. (Our definition simplifies the statement of Lemma 2.1.)

The definition of augmenting path indicates how to search for such a path. We give a high-level description of the search for P. (A more efficient algorithm, cyclic scanning, is given below.) The data structure consists of a label for each edge: if edge f has the label e then there is a partial augmenting path ending with edges e, f. The labels allow the augmenting path P to be traced out, starting from its last edge. We now sketch an algorithm that specifies L, the set of edges that get labelled in this search. It is a simple matter to assign appropriate labels.

The *labelling algorithm* works as follows. Consider a k-intersection T partitioned into forests T_i, $i = 1, \ldots, k$. Let z be a vertex having $|\rho_T(z)| < k$. To search for an augmenting path from z, initialize a set of edges L to $\rho(z) - T$. Then repeat the following step until it halts:

L Step. If L contains a joining edge then halt (the search is *successful*). Otherwise add $\bigcup\{C(e,T_i,\mathcal{G}) \mid e \in L - T_i\} - L$ to L. If no edge has been added to L then halt (the search is *unsuccessful*). Otherwise add $\bigcup\{\rho(v) - T \mid \rho_{T\cap L}(v) \neq \emptyset\} - L$ to L. ∎

We now give the properties of this algorithm. Recall that in any graph, a collection of trees is *cospanning (for S)* if each tree has the same vertex set S. Let L denote the set of the labelling algorithm when it halts.

(i) In a successful search L contains an augmenting path from z.

(ii) In an unsuccessful search $L \cap T$ is a set of k subtrees of T that are cospanning for $V(L)$. Furthermore vertex v has $\rho(v) - T \subseteq L$ if $v = z$ or $\rho_{L\cap T}(v) \neq \emptyset$.

Property (ii) is proved as follows. First observe that for each i, $L \cap T_i$ is connected, i.e., $L \cap T_i$ is a subtree. This follows by a simple induction (note that each $L \cap T_i$ spans vertex z). When the algorithm halts unsuccessfully any edge of L is spanned by any subtree $L \cap T_i$, i.e., the k subtrees $L \cap T_i$ are cospanning for $V(L)$. The second part of (ii) is obvious.

Property (ii) can be used to show that in an unsuccessful search there is no augmenting path from z. We do not use this fact; instead we prove the stronger Lemma 2.1 below.

Next consider property (i). Let e be the first joining edge to enter L and let P be the sequence of edges corresponding to the label of e (assume the algorithm generates appropriate labels). It is clear that P satisfies all the defining conditions of an augmenting path except possibly (iii). Condition (iii) will be clear to readers familiar with the matroid intersection algorithm (P has no shortcuts if one views the matroid $\mathcal{G}^k$ as a direct sum of k copies of $\mathcal{G}$ and $\mathcal{D}^k$ as a direct sum of a truncation of a partition matroid for each vertex). For completeness we provide a proof, based on a notion that is useful later on.

Let P satisfy all the conditions for an augmenting path except possibly (iii). In addition suppose P is *$\mathcal{G}$-shortcut free*, i.e., for any swap g, h in P with $h \in T_i$, no edge following h in P is in

$C(g, T_i)$. Then P satisfies (iii) and is a valid augmenting path. We give a simple proof, similar to [L76]. Note this completes the proof of property (i) for the labelling algorithm, since its paths are $\mathcal{G}$-shortcut-free if labels are assigned in the natural way. (Most treatments of matroid intersection use a different notion that might be termed "$\mathcal{G}^k$-shortcut free", i.e., if g, h is a swap then no edge following h in P is in $C(g, T, \mathcal{G}^k)$.)

Given P that is $\mathcal{G}$-shortcut free, augment along P in reverse, i.e., first add the joining edge of P to the appropriate T_h and then execute the swaps of P in reverse of their order along P. We prove by induction that each change to T gives a new k-intersection, thus establishing (iii). (Note that when swaps are executed in reverse order an edge e_i in two swaps of P will have two copies in the intersection after swap e_i, e_{i+1} is executed but before swap e_{i-1}, e_i is executed. This causes no harm.)

For the base case of the induction note that adding the joining edge of P gives a new k-intersection. For the inductive step consider a swap g, h of P. Immediately before we execute this swap $C(g, T_i)$ is the same as it was before the augment began. This follows since P is $\mathcal{G}$-shortcut-free. Thus executing swap g, h keeps T_i a valid forest. We conclude that P is a valid augmenting path.

This completes our review of matroid theory. Now we give the basic lemma. Let G be a directed graph with distinguished vertex a and let T be a k-intersection for a. Use the labelling algorithm to search for an augmenting path from a vertex $z \neq a$ with $|\rho_T(z)| < k$. Let L denote the set of the labelling algorithm when it halts.

Lemma 2.1. In an unsuccessful search, an a-cut with less than k edges is formed by $\rho(L)$ if $L \neq \emptyset$, or $\rho(z)$ if $L = \emptyset$.

Proof. If $L = \emptyset$ then z has in-degree less than k, by property (ii) of the labelling algorithm. Thus $\rho(z)$ is the desired a-cut.

Now suppose $L \neq \emptyset$. Let $\ell = |V(L)|$. By property (ii), $L \cap T$ forms k subtrees of T that are cospanning for $V(L)$. These subtrees contain $k(\ell - 1)$ edges, each of which is in $\rho_{L \cap T}(x)$ for some $x \in V(L)$. Strictly more than $k(\ell - 2)$ of these edges are in $\rho_{L \cap T}(x)$ for some $x \in V(L) - z$. Since $|\rho_{L \cap T}(x)| \leq k$, all $\ell - 1$ vertices $x \in V(L) - z$ have $\rho_{L \cap T}(x) \neq \emptyset$.

We draw two consequences from this fact. First the fact clearly implies that $a \notin V(L)$. Thus $\rho(L)$ is an a-cut. Second with property (ii) the fact implies that every $x \in V(L)$, including z, has $\rho(x) - T \subseteq L$. Thus $\rho(L) = \rho_T(L)$. Since $z \in V(L)$, $|\rho_T(L)| < k\ell - k(\ell - 1) = k$. In other words $\rho(L)$ is an a-cut containing less than k edges. ∎

The following result is an obvious consequence of Edmonds' Characterizations. We give a short proof (Section 3 then uses it prove Edmonds' Characterizations). The proof also gives a simple version of our algorithm.

Corollary 2.1. A directed graph G contains a complete k-intersection for a if and only if any a-cut contains k or more edges.

Proof. Consider the "only if" direction. Let T be a complete k-intersection and let the vertex set S determine an a-cut $\rho(S)$; let $s = |S|$. Precisely ks edges of T have their head in S. At most $s - 1$ edges of each tree of T have both ends in S. Thus $|\rho(S)| \geq ks - k(s - 1) = k$ as desired.

For the "if" direction consider the following algorithm to find a complete k-intersection. Let T be a k-intersection. Stop if it is complete. Otherwise choose a vertex $z \neq a$ with $|\rho_T(z)| < k$. Use the labelling algorithm to search for an augmenting path from z. If successful then augment the intersection and repeat. If unsuccessful then halt.

If any a-cut contains k or more edges the algorithm finds a complete k-intersection, by Lemma 2.1. This proves the "if" direction. ∎

We now present the round robin algorithm. It is an efficient implementation of the algorithm

of Corollary 2.1. 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. Thus to find a complete k-intersection from scratch, start with $k = 1$ and repeatedly execute round robin until the desired value of k is reached.

Round robin maintains a k-intersection T partitioned into spanning trees T_i, $i = 1, \ldots, k-1$ and a spanning forest T_k. The goal is to enlarge T_k to a spanning tree. An *f-tree* is a subtree of forest T_k. (Thus an edge is joining if and only if its ends are in different f-trees.) Two invariants on the in-degrees are maintained: Each vertex $v \neq a$ has $|\rho_T(v)|$ equal to $k-1$ or k; each f-tree has precisely one vertex z with $|\rho_T(z)| < k$. The notation F_z denotes the f-tree having z as its unique vertex of in-degree less than k. Note that z is either a or a vertex of in-degree $k-1$.

The algorithm is organized as a sequence of "rounds". At the start of a round every f-tree except F_a is *active*; an f-tree created during the round is not active. The following version of round robin is efficient for graphs that are not extremely dense. (Below we give another version of round robin with better performance on dense graphs.)

Round robin initializes T to contain the given $(k-1)$-intersection plus T_k, a forest consisting of all the vertices and no edges. Then it repeats the Round Step until the algorithm halts.

Round Step. If T_k is a spanning tree then halt (T is a complete k-intersection). Otherwise make all f-trees except F_a active. Then repeat the Augment Step until no f-tree is active or it halts.

Augment Step. Choose an active f-tree, say F_z. Search for an augmenting path P from z. If P does not exist then halt (no complete k-intersection exists). Otherwise augment intersection T along P. ∎

We comment briefly on the Augment Step. The last edge e of P joins F_z to another f-tree F_y; it can be in $\rho(F_z)$ or $\delta(F_z)$. The augment adds e to T_k, forming a new f-tree. The edges of F_z may

change in the augment but its vertices do not, so the new f-tree has vertices $V(F_z) \cup V(F_y)$. The new f-tree is inactive.

In an augment $|\rho_T(x)|$ changes only for $x = z$, for which $|\rho_T(z)|$ becomes k. Thus the new f-tree has a unique vertex with in-degree less than k, preserving the in-degree invariants. Since round robin implements the algorithm of Corollary 2.1 it is correct, i.e., it finds a complete k-intersection if one exists.

The second version of round robin achieves better performance on dense graphs by making the following change to the Round Step: At the start of the rth round make only f-trees with at most 2^r vertices active. Clearly this version is also correct.

It remains to provide an efficient implementation of the Augment Step. The basic idea is simple: A search for an augmenting path stops as soon as a joining edge is encountered. Thus an edge is labelled only once per round, since it has an end in F_z. Each edge is labelled in $O(1)$ time. This gives time $O(m)$ per round.

Before providing the details let us analyze the efficiency of round robin. As mentioned we will show that the time for the Augment Step amounts to $O(m)$ per round for either version; also for the second version the time is $O(m_r + 2^r n)$ for the rth round, where m_r is the number of edges labelled in the rth round. Now we show these bounds imply the first version of round robin runs in time $O(m \log n)$, the second in time $O(m \log (n^2/m))$.

There are at most $\lceil \log n \rceil$ rounds in either version of round robin. This follows since in the rth round any f-tree other than F_a has at least 2^{r-1} vertices (in either version of round robin). This gives the desired time bound for the first version of round robin since each round uses $O(m)$ time.

To analyze the second version, observe that at most n joining edges get labelled in the entire algorithm. In addition for any vertex x, the rth round labels at most 2^r nonjoining edges in $\rho(x)$ and 2^r nonjoining edges in $\delta(x)$. This holds because each edge of $\rho(x)$ comes from a different vertex, and similarly for $\delta(x)$. Thus $m_r = O(2^r n)$ and the time for the rth round is $O(2^r n)$. Hence

the first $\lceil \log (m/n) \rceil$ rounds use total time $O(m)$. The number of remaining rounds is at most $\lceil \log n \rceil - \lceil \log (m/n) \rceil \leq 1 + \log (n^2/m)$. Since each such round uses time $O(m)$ the total time is $O(m \log (n^2/m))$ as desired.

We turn to implementing the Augment Step. The main task is implementing the labelling algorithm. We use two ideas. The first is cyclic scanning, a labelling method used in [RT, GW] for matroid sum algorithms. (Our cyclic scanning algorithm differs slightly because of additional constraints for efficiency.) The second idea is a fact about augmenting paths that allows them to be found more efficiently. We first give the cyclic scanning algorithm, with some details at a high level. We then use the second idea to implement cyclic scanning and achieve the desired efficiency.

The cyclic scanning algorithm implements the labelling algorithm by searching for a $\mathcal{G}$-shortcut-free augmenting path. The idea is to gain efficiency by propogating labels from T_i only to T_{i+1}. It uses the following data structures. Each edge of G has a label which, as in the labelling algorithm, indicates how to construct the augmenting path; the label also indicates if the edge is labelled or not. The algorithm maintains an invariant that for each $i \leq k$, the labelled edges of T_i form a subtree. There is a queue Q that contains labelled edges. In addition one element of Q may be a "flag" (it separates the current group of edges from the next). A variable i indicates the current tree T_i for computing fundamental cycles.

The *cyclic scanning algorithm* works as follows. In an Augment Step to search for an augmenting path P from vertex z, initialize set A to the edges of $\rho(z) - T$, initialize Q to empty, set $e \leftarrow \Lambda$ and $i \leftarrow 1$. Execute the Label Step. If this step does not stop (it places the edges of $\rho(z) - T$ in Q) then repeatedly execute the Cyclic_scanning Step until the algorithm stops.

Cyclic_scanning Step. If Q is empty then stop (the search is unsuccessful). Otherwise remove the first element e from Q. If e is a flag, add a new flag to the end of Q, set $i \leftarrow (i \bmod k) + 1$ and continue with the next Cyclic_scanning Step. If e is an edge then go to the Fundamental_cycle

Step.

Fundamental_cycle Step. Let u be the end of e such that $u \neq z$ and u is not on a labelled edge of T_i. If u does not exist then continue with the next Cyclic_scanning Step. Otherwise (u exists) let A be the path of unlabelled edges of T_i from z or a labelled edge to u. (We show below that u is unique if it exists, in which case A is unique.)

Label Step. Examine the edges f of A in order, doing the following for each f. Assign the label e to f. If f is a joining edge then make it the last edge of P and stop (the search is successful). Otherwise add f to the end of Q; furthermore if $f \in \rho(y)$ and no edge of $\rho(y) - T$ is labelled then do the following: For each edge $g \in \rho(y) - T$, assign the label f to g; if g is a joining edge then make it the last edge of P and stop (the search is successful); otherwise add g to the end of Q. ∎

The correctness of the cyclic scanning algorithm essentially follows from [RT, GW]. For completeness we outline the argument. The basic fact is that A contains precisely the unlabelled edges in $C(e, T_i)$, where $i = (j \bmod k) + 1$ and T_j contains either e, or if $e \notin T$, the label of e. Prove this by induction, with the help of the following inductive assertions: For each $i = 1, \ldots, k$ define a set of vertices L_i to consist of z plus every vertex on a labelled edge of T_i. (For convenience allow subscripts $i > k$; such an i corresponds to the value $1 + (i-1) \bmod k$.) Then at all times each L_i is a subtree of T_i. Furthermore whenever the first element of Q is a flag f (i.e., i is about to advance) these two properties hold: each edge of Q contains a vertex in L_{i+1} or in a previous edge of Q;

$$L_{i+1} \subseteq L_{i+2} \subseteq \ldots \subseteq L_{i+k-1} \subseteq L_i \subseteq L_{i+1} \cup V(Q - f).$$

The argument also shows that as mentioned, u is unique in the Fundamental_cycle Step. Also since each L_i is a subtree, the path A is unique.

Now we show the cyclic scanning algorithm satisfies properties (i) – (ii) of the labelling algorithm. Property (i) follows because P is $\mathcal{G}$-shortcut-free (since every edge of A gets labelled). For

property (ii) consider an unsuccessful search, i.e., the Cyclic_scanning Step stops with Q empty. Let L denote the set of labelled edges. The above set inclusion shows that sets L_j, $j = 1, \ldots, k$ are equal, i.e., the labelled edges $L \cap T_i$ form k cospanning subtrees. The second part of (ii) is clear. This completes the analysis of the cyclic scanning algorithm.

To achieve the desired efficiency we implement the Fundamental_cycle Step so it traverses only edges of $C(e, T_i)$ having a vertex in F_z (traversing the entire fundamental cycle can be too time-consuming if it contains a joining edge.) This also allows us to delay executing all augmenting paths until the end of the round. (This simplifies the presentation.) We now present the cycle traversal algorithm, which implements the Fundamental_cycle Step, and the augmenting algorithm, which executes all augmenting paths.

The cycle traversal algorithm uses the following data structures. All trees of T are rooted as follows. For $i < k$ the root of T_i is vertex a. For each f-tree F_z, add a new vertex z' with edge zz' and make z' the root. (This handles boundary conditions for terminating the traversal.) Each vertex in each T_i, $i \le k$ has a parent pointer. For each edge e that is in a tree T_i, $i \le k$, $d(e)$ (its "depth") is the length of the path beginning with e and ending at the root (a or z'). The vertices of each f-tree F_z are labelled with the value z. (None of these values change even as the algorithm discovers augmenting paths, since all augments are executed at the end of the round.) For each $i \le k$, r_i is the root of the subtree containing z and the labelled edges in T_i.

In the algorithm below references to the Label Step mean the step of the cyclic scanning algorithm, which does not change. Assume the above data structure is correct on entry. (Initialization of data structures is discussed below.)

The *cycle traversal algorithm* implements the Fundamental_cycle Step as follows. It works in tree T_i, $i \le k$ so terms like "tree path" and "ancestor" refer to T_i. First determine vertex u for edge e (see the Fundamental_cycle Step). Suppose u exists (else stop). Traverse the tree paths

from r_i and u to the root concurrently; always traverse the edge with greater depth $d(e)$ or in case of a tie, traverse both edges. Stop as soon as an edge that is common, joining or labelled is traversed. (Common means common to both paths. The deeper vertex of this first common edge is the nearest common ancestor c of r_i and u. c is never the root a or z'. When both paths are at the same depth check for a common edge before checking the other conditions.)

Suppose we stop because a joining edge f is traversed. Set $A \leftarrow \{f\}$ and go to the Label Step (it stops successfully).

Suppose we stop because a common or labelled edge is traversed. (Every unlabelled edge in $C(e,T_i)$ has been traversed, and there is no joining edge.) Set r_i to the nearest common ancestor of r_i and u. Set A to the tree path of unlabelled edges from z or a labelled edge to u. Go to the Label Step. (If a common edge has been traversed, with deeper end c the nearest common ancestor of r_i and u, r_i is set to c and path A goes from r_i to c to u. If a labelled edge f has been traversed, r_i does not change and A goes from the deeper end of f to u.) ∎

It is not hard to see that the cycle traversal algorithm correctly implements the Fundamental_cycle Step. If we stop because a joining edge f is traversed then $f \in C(e,T_i)$ (the algorithm has verified that f is on only one path). Note for later use that any other joining edge $g \in C(e,T_i)$ has $d(g) \leq d(f)$. If we stop because of the other conditions, every unlabelled edge of $C(e,T_i)$ gets placed in A. (The end of e opposite u, say w, is in the subtree of labelled edges of T_i. Hence the path from w to r_i can be ignored.) We conclude that the cycle traversal algorithm is correct. This finishes the implementation of the labelling algorithm.

Now we describe how the augmenting algorithm updates the intersection at the end of the round. To represent the new intersection we must update the parent pointers of T and the depth function d. Assume each successful search ends by recording the last edge of the augmenting path.

The *augmenting algorithm* proceeds as follows. For each $i \leq k$ compute T_i', the set of edges in

T_i after all augments have been done. Do this by using edge labels to trace out each augmenting path and transfer edges appropriately. Then do the following for each T_i' that has changed from T_i: Place the edges of T_i' in an adjacency structure and do a depth-first search to determine new parent pointers and d values for T_i' (start the search from root a for $i < k$ and roots z' for $i = k$). At this point it is also convenient to erase all edge labels, and in each (new) f-tree F_z, set all vertex labels to z, in preparation for the next round. ∎

It is not entirely obvious that this approach is correct: Augmenting a path changes the intersection, and conceivably this can invalidate swaps of subsequent augmenting paths. However we show this does not occur.

Lemma 2.2. The augmenting algorithm constructs a new k-intersection.

Proof. We begin with two definitions. For each index $i < k$ define a tree J_i by contracting each nonjoining edge of T_i. Thus J_i is a rooted tree; each of its edges is a joining edge; each of its vertices corresponds to a set of vertices of G contained in one f-tree. For a rooted tree R and a real-valued function d on $E(R)$, say *d is a valid depth function for R* if d decreases strictly along any path to the root (i.e., if edge e is followed by e' then $d(e) > d(e')$).

Augment the paths P one by one; to augment P, execute its swaps in reverse of their order along P, and then add the joining edge of P. At any time in this process let T' be the current intersection, with forests T_i', $i \leq k$; define J_i' similar to J_i, for $i < k$. We prove by induction that each change makes the new T' a valid intersection. In addition we prove two auxiliary assertions concerning depths in J_i': d is a valid depth function on J_i', and the deeper end of an edge in J_i' contains its deeper end in J_i. (For the second assertion note that any edge in J_i' is in J_i and its ends are sets of vertices of G. The main significance of this assertion is that the deeper and shallower ends of an edge never get interchanged; in general a swap interchanges these ends for some edges.)

Consider an augmenting path P from z. Let e, f be a swap of P with $f \in T_i$, $i \le k$. First suppose that f is the joining edge of P. (Such a swap is last in P. It may not exist since the last two edges of P may satisfy condition $(ii.b)$ of the definition of augmenting path.) Let e be edge x_1x_2. For $j = 1, 2$ let x_j be in the deeper end of edge f_j in J_i. We can assume $d(f_1) \ge d(f_2)$ and $f = f_1 \ne f_2$ by the cycle traversal algorithm (see the proof of correctness). By induction x_j is in the deeper end of f_j in $J_i^!$ (note that each f_j is in $J_i^!$). Since d is a valid depth function for $J_i^!$ this implies $f = f_1 \in C(e, J_i^!)$. Thus $f \in C(e, T_i^!)$. In other words e, f is a valid swap.

For $J_i^!$ the effect of executing swap e, f is that edge $f = f_1$ is deleted and the vertices containing x_1 and x_2 get contracted together. Since $d(f_1) \ge d(f_2)$, d remains a valid depth function. Since x_1 is in the deeper end of f_1, no shallower end of an edge becomes the deeper end, so the second assertion on $J_i^!$ holds.

Next suppose that edge f of swap e, f is not joining. Thus all edges of $C(e, T_i)$ have both vertices in F_z (by the cycle traversal algorithm). Clearly fundamental cycle $C(e, T_i)$ is not changed by any preceding augmenting path. It is not changed by a preceding swap of P, which is $\mathcal{G}$-shortcut-free. Thus swap e, f is valid. Since all edges of $C(e, T_i)$ are nonjoining, the swap does not change $J_i^!$.

Finally consider adding the joining edge of P to $T_k^!$. Clearly this does not create a cycle in $T_k^!$. The effect on $J_i^!$ is that some edges become nonjoining and so get contracted. This preserves the two properties for $J_i^!$. This completes the induction. ∎

Clearly the augmenting algorithm enlarges the intersection by one edge for each augmenting path. Thus it is correct.

The data structures of the algorithm are initialized as follows. To prepare for the first round initialization similar to the augmenting algorithm is done at the start of round robin. When the Augment Step begins a search for an augmenting path from vertex z, it sets each $r_i \leftarrow z$, $i \le k$.

This concludes the detailed implementation of round robin. (Another version of round robin augments each path upon discovery. When programmed carefully the efficiency of this version is the same. A similar analysis applies.)

We now justify the timing analysis already given for round robin. We must show that the total time for round robin amounts to $O(m)$ per round for either version; also for the second version, time $O(m_r + 2^r n)$ for the rth round, where m_r is the number of edges labelled in the rth round. (The timing analysis already given also assumes that an edge is labelled only once per round. We have already mentioned that the cycle traversal algorithm achieves this since a labelled edge has its deeper end in F_z.)

To compute the time first consider initializations. At the start of round robin initializing parent pointers, d values and vertex labels uses time $O(kn)$. At the start of a search for an augmenting path initializing r_i values uses time $O(k)$, giving total time $O(kn)$. Thus initializations amount to $O(kn)$ time total. This contribution can be absorbed in the bound for the last round, since $O(kn) = O(m) = O(n^2)$.

We now show that the time to search for all augmenting paths in the rth round is $O(m_r + n)$, which is also $O(m)$. The Label Step uses time $O(m_r)$. The time for the cycle traversal algorithm is proportional to the number of edges that get placed in A plus additional edges of F_z traversed when a joining edge is found. There are at most m_r edges of the first type, since each gets labelled. There are at most n edges of the second type, since each has its deeper end in F_z.

Finally consider the time for the augmenting algorithm. The new edge sets T_i' are constructed in time $O(m_r)$. Suppose that c edge sets T_i' differ from T_i ($c \geq 1$ since T_k changes). Then the time to construct the new data structure is $O(cn)$. Since $c \leq k$, $O(cn) = O(m)$. This implies the first bound for the time of a round. Thus it remains only to show that $c = O(2^r)$ for the second version of round robin.

Recall from the analysis of cyclic scanning that at all times the labelled edges in T_i form a

subtree of T_i. Each such subtree contains an edge of $\rho(z) \cup \delta(z)$. The analysis of the second version of round robin shows that at most $2^{r+1}+1$ such edges are labelled in the search for an augmenting path from z (the term of 1 allows for a joining edge). Thus $c = O(2^r)$.

Theorem 2.1. A complete k-intersection, if it exists, can be found in time $O(m \log(n^2/m))$ given a complete $(k-1)$-intersection or time $O(km \log(n^2/m))$ starting from scratch. The space is $O(m)$. ∎

Although round robin is most efficient on graphs it extends to multigraphs. Represent a multigraph by associating a multiplicity with each edge. The first version of round robin works correctly. It is implemented by representing each tree T_i explicitly and each edge of $G-T$ by storing its multiplicity. In each round an edge of $G-T$ is examined only once. Thus the time to find a complete k-intersection is $O(k(m+kn)\log n)$. The space is $O(m+kn)$. The second version of round robin speeds up the processing of edges of $G-T$ but not T. Thus it improves the time to $O(k(m\log(n^2/m)+kn\log n))$. If each multiplicity is at most d then the $\log n$ factor decreases to $\log(dn/k)$ (since the ith round spends $O(2^i dn)$ time on edges of T, whence the first $\lceil \log(k/d) \rceil$ rounds spend $O(kn)$ time on T).

We also mention a type of multigraph used extensively in [G]. Suppose every edge with multiplicity greater than one is incident to the distinguished vertex a. Then the time is $O(k(m\log(n^2/m)+kn))$. This follows since any edge from a is joining.

We turn to edge connectivity. We first compute the edge connectivity of a directed graph G. Choose an arbitrary vertex a; let G^r denote graph G with the direction of each edge reversed. For k starting at one and assuming successively larger values, execute round robin on both G and G^r. Stop upon finding the smallest value k such that G or G^r does not have a complete k-intersection. Then $\lambda = k-1$ by Corollary 2.1. Furthermore Lemma 2.1 shows that the edges labelled in the unsuccessful search determine a cut of λ edges, i.e., a connectivity cut. Clearly the time for the

entire procedure is $O(\lambda m \log(n^2/m))$.

Next consider an undirected graph G. As noted in Section 1 the above procedure correctly computes $\lambda(G)$ and a connectivity cut for G when applied to the directed graph where each edge has both orientations. An obvious practical simplification is that the reverse graph G^r need not be processed. More importantly we can use the following lemma of [NI89a, Th]: Partition $E(G)$ into a sequence of spanning forests F_i, $i = 1, \ldots, n$; do this by letting F_1 be a maximal spanning forest of G and proceeding recursively on $G - F_1$. It is easy to see that G is k-edge-connected (i.e., $\lambda(G) \geq k$) if and only if $\bigcup_{i=1}^{k} F_i$ is k-edge-connected.

This implies we can compute the edge connectivity of an undirected graph G as follows. First find the spanning forests F_i. Initialize a directed graph D to contain vertices $V(G)$ and no edges, and an intersection T to contain no edges. Then for k starting at one and assuming successively larger values do the following: Add each edge of $E(F_k)$ in both orientations to D; execute round robin to enlarge T to a complete k-intersection for D. Stop when the complete k-intersection does not exist, setting $\lambda = k - 1$.

The algorithm of [NI89a] finds all sets F_i, $i = 1, \ldots, n$ in total time $O(m)$. The kth run of round robin uses time $O(kn \log(n/k))$. A simple calculation shows that λ runs use time $O(\lambda^2 n \log(n/\lambda))$ (the runs for $k \in (2^{j-1}..2^j]$ use time proportional to $2^{2j}n((\log n) - j)$).

Theorem 2.2. The edge connectivity and a connectivity cut can be found in time $O(\lambda m \log(n^2/m))$ for a directed graph, $O(m + \lambda^2 n \log(n/\lambda))$ for an undirected graph. The space is $O(m)$ for both. ∎

3. Packing arborescences.

This section begins with a high-level version of our packing algorithm. The correctness proof gives a new proof of Edmonds' Characterizations. Then an efficient implementation of the packing algorithm is presented.

For convenience we summarize some properties of the labelling algorithm. Let L be the set of labelled edges when the algorithm halts; assume $L \neq \emptyset$.

(iii) In an unsuccessful search, $\rho(L) = \rho_T(L)$ is an a-cut of less than k edges; the edges with heads in $V(L)$ are $L \cup \rho_T(L)$.

The first half is proved in Lemma 2.1. For the second half recall that $L \cap T$ forms k cospanning trees (property *(ii)* of the labelling algorithm). Hence all edges of T with both ends in $V(L)$ are in L. For edges not in T, the proof of Lemma 2.1 shows that every $x \in V(L)$ has $\rho(x) - T \subseteq L$.

Consider a directed graph G that is a complete k-intersection for distinguished vertex a. We wish to partition the edges of G into k a-arborescences. We do this by maintaining two subgraphs: A is an a-arborescence spanning a subset of the vertices of G; call such a subgraph a *partial arborescence.* T is a complete $(k-1)$-intersection that is edge-disjoint from A.

An *enlarging path (for e)* consists of an edge $e \in \delta(A)$, plus if $e \in T$, an augmenting path P for the $(k-1)$-intersection $T - e$ such that $E(P) \subseteq E(G) - E(A) - e$. (Note that P starts with the unique edge of $\rho(z) - T$ where $e \in \rho(z)$.) An enlarging path allows us to extend A by one edge as follows: If $e \in T$ then modify T by removing e and augmenting along P, obtaining a new complete $(k-1)$-intersection T. Now in either case $e \notin T$. Add e to A. Clearly we have enlarged A by an edge and maintained the defining properties of A and T.

The following lemma allows us to use enlarging paths as the main tool of the algorithm. Let G be a directed graph that is a complete k-intersection for a. Let A and T be subgraphs as defined above.

Lemma 3.1. $V(A) \neq V(G)$ implies there is an enlarging path.

Proof. Choose any edge $e \in \delta(A)$ (e exists by Corollary 2.1). Suppose $e \in T$, else e forms the desired enlarging path. Let $e \in \rho(z)$. Search for an augmenting path from z for the $(k-1)$-

intersection $T - e$ by executing the labelling algorithm. If the search is successful we have the desired enlarging path. We first show that if the search halts unsuccessfully, having labelled a set of edges L, then $\delta_L(A) \neq \emptyset$.

Let $R = V(L) - V(A)$. Note that $\rho(R)$ contains e but no edge of A; the remaining edges of $\rho(R)$ are in the graph searched by the labelling algorithm. Thus (iii) shows $\rho(R) \subseteq e + \rho_L(R) \cup \rho_{T-e}(L)$, and further $|\rho_{T-e}(L)| < k-1$. Since G has a complete k-intersection and $a \notin R$, $|\rho(R)| \geq k$. Thus $\rho_L(R) \neq \emptyset$; since $\rho_L(R) = \delta_L(A)$ we have $\delta_L(A) \neq \emptyset$ as desired.

Now consider an edge $e \in \delta(A)$ whose search is unsuccessful and has $|L|$ minimum. As just shown there is an edge $e' \in \delta_L(A)$. We prove there is an enlarging path for e'. It suffices to show that searching for an augmenting path in $T - e'$ is successful. For the sake of contradiction suppose the search is unsuccessful. Let it label a set of edges L'. Then $L' \subseteq L$, since any edge labelled in the search for e' could be similarly labelled in the search for e. (It is not significant that the searches for e and e' are executed on different sets $T - e$ and $T - e'$: Since each search is unsuccessful the same edges are labelled if it is executed on T.) By definition $e' \notin L'$. Thus $|L'| < |L|$, the desired contradiction. ∎

The lemma can be used to partition G into k disjoint a-arborescences as follows. Initialize T to a complete $(k-1)$-intersection. Initialize A to a partial arborescence containing vertex a and no edges. Then repeatedly find an enlarging path and use it to add an edge to A, until $V(A) = V(G)$. Output the arborescence A. If $k = 1$ then halt, else decrease k by one and repeat the entire procedure on graph $G - A$.

The correctness of this procedure follows from Lemma 3.1 and the fact that a complete $(k-1)$-intersection on G exists. The latter is a consequence of Corollary 2.1: Since G is a complete k-intersection, any a-cut has at least k edges, whence G contains a complete $(k-1)$-intersection.

This proves the Matroid Characterization. The Cut Characterization also follows, using Corol-

lary 2.1.

We turn to an efficient implementation of this procedure. The main difficulty is that the edge e that has an enlarging path is unknown. A simple strategy like trying all possibilities is inefficient (it increases the time by a factor of kn). We avoid this inefficiency using the following observations.

Suppose edge $e \in \delta(A)$ gives an unsuccessful search for an augmenting path for $T - e$. Let L be the set of edges labelled in the search. $L \cap T$ consists of $k-1$ trees that are cospanning for $V(L)$ (property (ii) of the labelling algorithm). Furthermore $|\rho_T(L)| = k-1$. This follows since property (iii) shows $|\rho_{T-e}(L)| < k-1$, i.e., $|\rho_T(L)| \leq k-1$. The last inequality holds with equality since T is a complete $(k-1)$-intersection.

Now let B be an a-arborescence containing A and edge-disjoint from a complete $(k-1)$-intersection U. (The discussion following Lemma 3.1 shows B and U exist.) Let the trees of U be U_i, $i = 1, \ldots, k-1$.

Lemma 3.2. $L \cap U$ consists of $k-1$ trees that are cospanning for $V(L)$. Furthermore $\rho_T(L) \subseteq U$.

Proof. As in Lemma 3.1 let $R = V(L) - V(A)$; let $\ell = |V(L)|$ and $r = |R|$. The definitions imply that $|B \cap L| \leq r$ and for each i, $|L \cap U_i| \leq \ell - 1$. Furthermore $|L| \geq (k-1)(\ell-1) + r$, since $L \cap T$ forms $k-1$ cospanning trees and each $x \in R$ has the edge of $\rho(x) - T$ contained in L (by (iii)). Combining these inequalities show they all hold with equality.

Since $L \cap U_i$ contains $\ell - 1$ edges, it is a subtree spanning $V(L)$. This gives the first part of the lemma. $B \cap L$ contains r edges, each with a head in R. Thus the edges of B with heads in L are in $A \cup L$. Thus $\rho_T(L) \subseteq U$ as desired. ∎

Based on these observations we construct an algorithm that processes L recursively. (A possible organization is to find all k arborescences for L recursively; for notational simplicity our algorithm finds only one arborescence recursively.) It is convenient to now let G_0 denote the given graph, with

distinguished vertex a_0. We give a recursive procedure $p(G, a, A, T)$. Argument G is a graph that is a complete k-intersection for a; arguments A and T are as above, i.e., A is a partial arborescence and T is a complete $(k-1)$-intersection edge-disjoint from A; in addition T is partitioned into $k-1$ spanning trees. Procedure p modifies A and T but maintains these defining properties. It returns with A an a-arborescence for G that is edge-disjoint from some complete $(k-1)$-intersection. The initial call to procedure p is $p(G_0, a_0, A, T)$, where A contains vertex a_0 and no edges, and T is a complete $(k-1)$-intersection; it constructs an a_0-arborescence, edge-disjoint from a complete $(k-1)$-intersection.

Procedure $p(G, a, A, T)$ returns if $V(A) = V(G)$. Otherwise choose an edge $e \in \delta(A)$, where if possible $e \notin \delta(a)$. Execute the case below that applies and then return A.

Case (i) An enlarging path for e exists. Find such a path. Use it to enlarge A. Then call $p(G, a, A, T)$.

Case (ii) No enlarging path for e exists. Let L be the set of edges labelled in the search for an augmenting path for $T - e$. Denote the $k-1$ subtrees comprising $L \cap T$ as L_i; denote the $k-1$ edges of $\rho_T(L)$ as e_i; here $i = 1, \ldots, k-1$ but the indexing is arbitrary. (The discussion preceding Lemma 3.2 shows L_i and e_i exist. Note that a tree of T need not contain any edge e_i.)

Modify graph G to G' by deleting all edges with head not in L and contracting all vertices not in L into a single vertex a'. Construct subgraphs A' and T' of G' as follows. A' contains the edges of A with heads in L. T' is the complete $(k-1)$-intersection for a' on G', whose ith tree consists of L_i plus e_i. Call $p(G', a', A', T')$. Add the edges of $A' - A$ to A.

Modify graph G to G' by contracting $V(L)$ into a single vertex L'. Let A' be $A - \rho_A(L) + f$ where f is an edge in $\rho_A(L)$ of minimum depth. Modify T to a $(k-1)$-intersection on G' by contracting each subtree L_i to vertex L'. Call $p(G', a, A', T)$. Add the edges of $A' - A$ to A. ∎

We prove that procedure p is correct in two steps. The first step is to show by induction

that for each invocation $p(G,a,A,T)$, the arguments satisfy their defining properties; furthermore if the invocation returns, A is an a-arborescence for G, edge-disjoint from some complete $(k-1)$-intersection. The argument is as follows.

The assertions are obvious for Case (i), so assume Case (ii) is executed. Consider the first recursive call. To show the argument G' has a complete k-intersection, observe that Lemma 3.2 implies G' consists of k spanning trees. (The first tree consists of the edges B with heads in L. The other $k-1$ trees are the trees of $L \cap U$ with an edge of $\rho_T(L)$ added to each one.) This implies G' is a complete k-intersection. Argument T' is correct by the discussion preceding Lemma 3.2.

Next observe that when the first recursive call returns, adding the edges of $A' - A$ to A gives a new partial arborescence (containing $V(L)$ in its span). To prove this we must show that no e_i is in A' (adding an e_i to A need not give a partial arborescence, since the tail of e_i may not be in A). By induction when p returns A' is edge-disjoint from some complete $(k-1)$-intersection of G'. Note that $\delta(a')$ consists of edges in A and the edges $\rho_T(L)$, i.e., the edges e_i. Since there are precisely $k-1$ edges e_i, each tree of the intersection contains exactly one of them.

Consider the second recursive call in Case (ii). Argument G' is a complete k-intersection by Lemma 3.2. (Note that $B - \rho_B(L) + f$, for f an edge in $\rho_B(L)$ of minimum depth, is an arborescence on G'.) Argument T' is correct by the discussion preceding Lemma 3.2. Now it is easy to see that Case (ii) returns with A the desired a-arborescence. This completes the induction.

The second step is to prove that all recursive calls terminate. This follows from the fact that in Case (ii) each graph G' has fewer vertices than G. This is clear for the second graph G' ($L \neq \emptyset$ so $|V(L)| \geq 2$). Consider the first graph G'. Let $e = xz$. Since the search is unsuccessful neither a nor x is in $V(L)$. Thus it suffices to show that $x \neq a$. If $x = a$ the choice of e implies that $\delta(A) \subseteq \delta(a)$; thus $\delta_L(A) = \emptyset$; but as shown in Lemma 3.1 an unsuccessful search for an enlarging path has $\delta_L(A) \neq \emptyset$.

We conclude that procedure p is correct. To analyze its efficiency we add one implementation

detail. Search for an augmenting path for $T - e$ using the cyclic scanning and cycle traversal algorithms of Section 2. Thus a search uses time proportional to the number of edges of G (recall G is the current graph). In fact the entire time for any invocation of p, excluding recursive invocations, is proportional to the number of edges of G.

Now we show that the total time for an initial call $p(G_0, a_0, A, T)$ (including all recursive invocations) is $O(kn^2)$. Since each invocation (excluding recursive invocations) uses time $O(kn)$, it suffices to show there are $O(n)$ executions of Cases (i) and (ii). Clearly Case (i) is executed at most n times (it is easy to see that any edge added to A in Case (i) is in the arborescence returned by the initial call).

For Case (ii) define $t(n)$ to be the greatest number of executions of Case (ii) that can be done (in all recursive invocations) for a graph of n vertices. We show that for $c = 2$, $t(n)$ satisfies the recurrence

$$t(n) \leq 1 + \max\{t(n_1) + t(n_2) \mid n_1, n_2 \leq n - 1,\ n_1 + n_2 \leq n + c\}.$$

As already noted, each recursive call is on a graph with fewer vertices. The total number of vertices in graphs of recursive calls is precisely $n + 2$ since the first graph has vertices $V(L) + a'$, the second $(V(G) - V(L)) + L'$. The recurrence implies $t(n) = O(n)$. (Setting $T = \max\{t(i) \mid i \leq c\}$, $r = 2T+2$ and $s = rc + 1$, it is easy to show $t(n) \leq rn - s$ for $n > c$.) This completes the proof that the total time for p is $O(kn^2)$.

The space needed by the algorithm is $O(kn)$. This can be achieved by standard techniques in a variety of ways. We sketch one approach. First observe that for any recursive call the graph G' is obtained from G by contracting a set of vertices. There is a forest F that represents the structure of the currently contracted sets. The leaves of F are the vertices of G_0; each internal node of F represents a contracted set formed from its children; F has $O(n)$ nodes, since each internal node has at least two children. The algorithm maintains F.

In addition when a recursive call is made, it is given a list of the edges of G and a list of the

edges of each tree T_i of T. The lists contain edges of G_0; we must determine the vertex $[x]$ of G that contains a given vertex x of G_0. To do this use F to construct a table giving the value $[x]$ for each vertex x of G_0. The table is constructed in $O(n)$ time. This allows the given lists of edges to be translated into the desired representation of G and T in time $O(kn)$. (Specifically the desired representation is that required by the cyclic scanning and cycle traversal algorithms to search for an augmenting path, i.e., an adjacency structure for G and parent pointers for T.)

We note two more implementation details. First the argument A for procedure p is not needed. Instead we maintain one global partial arborescence for G_0; the algorithm adds all edges directly to it. Each recursive invocation only needs to know the vertices in its local partial arborescence. This is easily determined from the global structure. (Our presentation of procedure p maintains the partial arborescence A for conceptual purposes, to demonstrate the correctness of p.) Second, when the algorithm makes the first recursive call in Case (ii), it stores in the recursion stack a list of the edges of $T_i - L$, for each i. Using these lists it reconstructs T when the first recursive call returns. Note that the edges of $\rho_T(L)$ are stored in the recursion stack and are also in G'. Since there are at most n levels of recursion this uses $O(kn)$ extra space. From these remarks it is not hard to see that a detailed implementation uses $O(kn)$ space total.

Now we solve the arborescence packing problem. Recall that we are given a directed graph G with distinguished vertex a and wish to construct the greatest possible number of edge-disjoint a-arborescences. Start by using the round robin algorithm to find the largest value k such that G has a complete k-intersection T. Then set G_0 to the subgraph with edges $E(T)$ and execute the following procedure.

Use round robin to find a complete $(k-1)$-intersection T. Initialize A to the vertex a. Execute $p(G_0, a, A, T)$ and output the a-arborescence A. If $k = 1$ then halt, else decrease k by one and repeat the entire procedure on graph $G_0 - A$.

The time for this procedure is $O((kn)^2)$. To show this observe that the bound for round robin

to find a complete k-intersection is $O(km \log(n^2/m)) = O(kn^2)$. The bound for one execution of p is the same. There are $k+1$ executions of the round robin procedure and k executions of p.

Theorem 3.1. In a directed graph, the greatest number k of edge-disjoint a-arborescences can be found in time $O((kn)^2)$ and space $O(m)$. ∎

Clearly the same bounds apply to the problem where k is given and we wish to find k edge-disjoint a-arborescences, if they exist.

A similar algorithm applies to multigraphs, using $O((kn)^2)$ time and $O(m+kn)$ space. We must proceed more carefully since k can be larger than n. The idea is to execute round robin only once, to determine the value of k. The second version of round robin does this in time $O(k(m \log(n^2/m) + kn \log n)) = O(kn^2 + k^2 n \log n) = O((kn)^2)$. We change the rest of the procedure in the following two ways.

Procedure p is modified to return, in addition to A, a complete $(k-1)$-intersection T that is edge-disjoint from A. (Doing this is straightforward.) This implies that at the start of each iteration we have a complete k-intersection T, which we wish to convert to a complete $(k-1)$-intersection. To do this first make T a valid $(k-1)$-intersection: remove tree T_k from T; then for each vertex x still having $|\rho_T(x)| = k$, delete an arbitrary edge of $\rho_T(x)$. This gives a $(k-1)$-intersection at most n edges short of being complete. Now repeatedly find an augmenting path for T and augment, until T is complete.

The time to find each augmenting path is $O(kn)$ if we use the cyclic scanning and cycle traversal algorithms. (A slight difference from Section 2 is that more than one T_i can be a forest rather than a tree. It is easy to adapt cyclic scanning so it works for such a T: when an edge gets labelled, check if it is joining in the forest after the current one, $T_{(i \bmod k)+1}$. It is easy to see the new version of cyclic scanning is valid and runs in the same time bound.) Thus the time to find the complete $(k-1)$-intersection is $O(kn^2)$. This implies the desired bound of $O((kn)^2)$ time for the

arborescence packing problem on a directed multigraph. In practice it may be desirable to use this modification on graphs as well as multigraphs.

4. Related results.

This section briefly summarizes some extensions of the results of this paper.

The algorithm of [KT] for undirected edge connectivity finds all connectivity cuts, not just one. In [G] we show that the output of the round robin connectivity algorithm (i.e., a complete λ-intersection) can be processed in $O(m)$ time to find a succinct representation of all connectivity cuts. (The representation is similar to the one of [KT].) [G] uses this representation and round robin to develop efficient algorithms that increase the edge connectivity of a graph to a given target value, adding the fewest edges possible.

Consider an undirected graph with minimum degree δ. Choose any $\epsilon > 0$. Matula shows that if $\lambda \leq (\frac{1}{2} - \epsilon)\delta$ the edge connectivity can be found in time $O(m + n^2/\delta)$ [M90]. Incorporating round robin into his algorithm improves the time to $O(m \log n)$ or better ($O(m)$ when $\delta = \Omega(\sqrt{n})$, as in [M90]).

A problem in cluster analysis is, find a subgraph of maximum possible edge-connectivity. Matula accomplishes this in time $O(n^3)$ [M87]. Repeated applications of round robin achieves time $O(m^2 \log(n^2/m))$, an improvement for sparse graphs.

Next consider the problem of packing weighted arborescences. In a directed graph with edge costs, we seek a set of k edge-disjoint a-arborescences of minimum total cost. (Equivalently we seek a minimum cost subgraph that has k edge-disjoint paths from a to any vertex v, plus the paths for all v.) The edges of the desired arborescences can be found in time $O(kn(m + n \log n) \log n)$; alternatively using cost-scaling the time is $O(k\sqrt{n \log n}(m + kn \log n) \log(nN))$ for integral costs at most N. Then the desired arborescences are constructed using the packing algorithm of Section 3.

We close by mentioning the more general version of Edmonds' Characterizations. In a directed graph G, if A is a nonempty set of vertices then an *arborescence rooted at* A is a subgraph such that for every vertex v there is exactly one directed path from a vertex of A to v. Consider a family of k nonempty sets of vertices A_i, $i = 1, \ldots, k$. The Cut Characterization states that G contains k edge-disjoint arborescences, the ith one rooted at A_i, if and only if for any nonempty set of vertices S, $|\rho(S)| \geq \varphi(S)$. Here $\varphi(S) = |\{i \mid A_i \cap S = \emptyset\}|$. The Matroid Characterization states that the edges of G can be partitioned into k arborescences, the ith one rooted at A_i, if and only if they can be partitioned into k spanning forests, the ith one rooted at A_i, and each vertex v has in-degree $k - a(v)$. Here $a(v) = |\{i \mid v \in A_i\}|$ and a *spanning tree rooted at* A is the undirected version of an arborescence rooted at A.

The results of this paper extend to the general case. To state the basic observation, consider an undirected graph. Let H be a subgraph whose edges can be partitioned into k forests, the ith one containing no path between any two vertices of A_i. Then for $S = V(H)$, $|E(H)| \leq k|S| - \varphi(S) - \sum\{a(v) \mid v \in S\}$. In the labelling algorithm an unsuccessful search for an augmenting path labels a subgraph that satisfies this inequality with equality.

Lemmas 2.1 and 3.1 for the general case are proved analogously. The main change in the algorithms is that in round robin, cyclic scanning is no longer valid. However a generalization of cyclic scanning achieves the same efficiency. In summary the results of Theorems 2.1 and 3.1 apply to the general case.

Acknowledgments. Thanks to Zvi Galil, David Matula, Vijaya Ramachandran, Éva Tardos and Ramki Thurimella for sharing their ideas.

References.

[Ed65] J. Edmonds, "Minimum partition of a matroid into independent subsets", *J. Res. National Bureau of Standards 69B*, 1965, pp. 67–72.

[Ed69] 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.

[Ed72] J. Edmonds, "Edge-disjoint branchings", in *Combinatorial Algorithms*, R. Rustin, Ed., Algorithmics Press, New York, 1972, pp. 91–96.

[Ev] S. Even, *Graph Algorithms,* Computer Science Press, Potomac, MD, 1979.

[ET] S. Even and R.E. Tarjan, "Network flow and testing graph connectivity", *SIAM J. Comput., 4,* 4, 1975, pp. 507–518.

[F] A. Frank, "Kernel systems of directed graphs", *Acta Sci. Math., 41*, 1979, pp. 63–76.

[G] H.N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity", *Proc. 32nd Annual Symp. on Found. of Comp. Sci.,* 1991, to appear.

[GI] Z. Galil and G.F. Italiano, "Reducing edge connectivity to vertex connectivity", manuscript.

[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.

[GX89a] H.N. Gabow and Y. Xu, "Efficient theoretic and practical algorithms for linear matroid intersection problems", Technical Rept. CU-CS-424-89, Comp. Sci. Dept., Univ.

Colorado, Boulder, CO, 1989; submitted for publication.

[GX89b] H.N. Gabow and Y. Xu, "Efficient algorithms for independent assignment on graphic and linear matroids", *Proc. 30th Annual Symp. on Found. of Comp. Sci.,* 1989, pp. 106–111.

[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.

[L75] E.L. Lawler, "Matroid intersection algorithms", *Math. Programming, 9*, 1975, pp. 31–56.

[L76] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Rhinehart and Winston, New York, 1976.

[Lo] L. Lovász, "On two minimax theorems in graph theory", *J. Comb. Theory, B, 21*, 1976, pp. 96–103.

[M87] D.W. Matula, "Determining edge connectivity in $O(nm)$", *Proc. 28th Annual Symp. on Found. of Comp. Sci.,* 1987, pp. 249–251.

[M90] D.W. Matula, "A linear time $2 + \epsilon$ approximation algorithm for edge connectivity", preprint.

[MS] Y. Mansour and B. Schieber, "Finding the edge connectivity of directed graphs", *J. Algorithms, 10,* 1, 1989, pp. 76–85.

[NI89a] H. Nagamochi and T. Ibaraki, "Linear time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph", *Algorithmica*, to appear.

[NI89b] H. Nagamochi and T. Ibaraki, "Computing edge-connectivity in multiple and capacitated graphs", Technical Rept. #89009, Dept. of Applied Math. and Physics, Kyoto Univ., 1989.

[P] V.D. Podderyugin, "An algorithm for finding the edge connectivity of graphs," *Vopr. Kibern., 2*, 1973, p. 136.

[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.

[Sch] C.P. Schnorr, "Bottlenecks and edge connectivity in unsymmetrical networks", *SIAM J. Comput., 8,* 2, 1979, pp. 265–274.

[Sh] Y. Shiloach, "Edge-disjoint branchings in directed multigraphs", *Inf. Proc. Letters, 8,* 2, 1979, pp. 24–27.

[T74] R.E. Tarjan, "A good algorithm for edge-disjoint branching", *Inf. Proc. Letters, 3,* 2, 1974, pp. 51–53.

[T76] R.E. Tarjan, "Edge-disjoint spanning trees and depth-first search", *Acta Informatica 6*, 1976, pp. 171–185.

[T83] R.E. Tarjan, *Data Structures and Network Algorithms,* SIAM Monograph, Philadelphia, PA, 1983.

[Th] R. Thurimella, "Finding a sparse k-edge connected spanning subgraph of a k-edge connected graph", preprint.

[Ti] E.A. Timofeev, "An algorithm for constructing minimax k-connected oriented graphs", *Kibernetika, 2*, 1982, p. 109.

[TL] P. Tong and E.L. Lawler, "A faster algorithm for finding edge-disjoint branchings", *Inf. Proc. Letters, 17,* 2, 1983, pp. 73–76.

[W] D.J.A. Welsh, *Matroid Theory,* Academic Press, New York, 1976.

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