Packing Algorithms for Arborescences (and Spanning Trees) in Capacitated Graphs

Harold N. Gabow* and K.S. Manu*

Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309

Abstract. In a digraph with real-valued edge capacities, we pack the greatest number of arborescences in time $O(n^3 m \log(n^2/m))$; the packing uses at most m distinct arborescences. Here n and m denote the number of vertices and edges in the given graph, respectively. Similar results hold for integral packing: we pack the greatest number of arborescences in time $O(\min\{n, \log N\} n^2 m \log(n^2/m))$ using at most $m + n - 2$ distinct arborescences; here N denotes the largest (integral) capacity of an edge. These results improve the best previous bounds for capacitated digraphs. The algorithm extends to several related problems, including packing spanning trees in an undirected capacitated graph, and covering such graphs by forests. The algorithm provides a new proof of Edmonds' theorem for arborescence packing, for both integral and real capacities, based on a laminar family of sets derived from the packing.

Keywords: packing, covering, arborescences, spanning trees, laminar family, Newton's method.

1 Introduction

Packing arborescences and spanning trees are among the most basic packing problems. They probably provide the best-known examples of the so-called strong integrality and integer rounding properties of packing, respectively [Tro]. (Suppose a packing problem, or more generally a linear program, is specified by integral data. Strong integrality guarantees the problem has an integral optimum solution. Integer rounding guarantees the value of an integral optimum solution equals the rounded value of an optimum solution.) Many efficient algorithms for both problems have been presented for graphs with unit capacity edges and also for graphs with arbitrary capacity functions. This paper presents the most efficient algorithms known for graphs with arbitrary capacity functions.

We now state our results and compare them to previous work. In the time bounds throughout this paper, n and m denote the number of vertices and edges of the given graph, respectively. The parameter N represents the largest capacity. A bound involving N assumes that all given capacities are integers (between 0 and N) and so is a weak-polynomial bound.

The bulk of this paper is devoted to packing arborescences (more precisely a-arborescences). Edmonds proved the strong integrality property for arborescence packing [E72] (stated precisely

* Research supported in part by NSF Grant No. CCR-9215199.

at the end of this section). Other proofs are in [F79, G95a, L, Mad, TL]. Algorithms for packing arborescences in (unit capacity) graphs are presented in [Ta, S, TL]. The most efficient algorithm for general unit capacity graphs finds k edge-disjoint arborescences in time $O((kn)^2)$ [G95a]. Algorithms for packing arborescences in capacitated graphs are given in [GLS, Mad]. The most efficient algorithm for capacitated graphs is due to Pevzner, using the time for $O(n^3)$ maximum flow computations.

This paper presents two related algorithms for packing arborescences in capacitated graphs. The first finds a maximum packing in a digraph with arbitrary real-valued capacities. We call this a fractional packing – a given arborescence can be used α times, for any nonnegative real value α. (This type of packing is needed for the problems of [PW], see below.) The second algorithm finds a maximum integral packing in a digraph with integral capacities. The two algorithms provide independent proofs of the real and integral cases of Edmonds' theorem respectively [E72]. The proof differs from previous ones in that it is based on maintaining a laminar family of sets.

Our fractional packing algorithm runs in time $O(n^3 m \log(n^2/m))$. Our integral packing algorithm has a slightly better time bound, $O(\min\{n, \log N\} n^2 m \log(n^2/m))$. This improves [P] by a factor $\max\{1, n/\log N\} n/\log(n^2/m)$. (Here we assume the algorithm of [KRT] is used to compute a maximum flow in [P] and runs in time $O(nm)$. This is true if $m = \Omega(n^{1+\epsilon})$ for some positive constant ϵ. The maximum flow time bound is slightly more for sparser graphs, so in this case our improvement factor is slightly more.) Our algorithms find packings with essentially the fewest possible number of distinct arborescences: we use at most m distinct arborescences for fractional packing and $m+n-2$ distinct arborescences for integral packing. The algorithm of [P] uses $O(nm)$ distinct arborescences. (We note that Mader's approach to packing by splitting off edges [Mad] can be implemented efficiently using the ideas of this paper. However the best bound on the number of distinct arborescences that we can prove for this approach is $O(n^2)$.)

We present four applications of the packing algorithms. Each runs in the above time bounds (for fractional and integral arborescence packing) and each finds a packing or covering, the number of whose distinct members is close to the above number of distinct arborescences. Two applications are for capacitated digraphs. The first is finding a minimum (fractional or integral) covering of a digraph by branchings. The second is finding a maximum root-constrained packing of arborescences. In this problem lower and upper bounds are given for the number of times each vertex can be the root of a spanning arborescence. Minimax formulas for root-constrained packing are given in [F78, Mao].

The two other applications are for capacitated undirected graphs. They are finding a maximum packing of spanning trees, and finding a minimum covering by forests. Edmonds gave the first polynomial-time algorithms for these problems, establishing the integer rounding property [E65]. Efficient algorithms are given in [G95b, GW, I, RT] for (unit capacity) graphs and [GLS, PW] for capacitated graphs. The most efficient algorithms are as follows. Trubin [Tru91] gives algorithms for both packing and covering in capacitated graphs, running in the time for $O(n^3)$ network flow computations. We improve this by the same factor as given above for Pevzner's algorithm. Barahona [B] gives an algorithm for packing in capacitated graphs, running in the time for $O(n^2)$ network flow computations. In the algorithm as presented, these computations are on graphs that can grow to $\Theta(n^2)$ distinct edges, so we improve the time by a factor $\max\{1, n/\log N\} n^2/(m \log (n^2/m))$. Also we improve the number of distinct spanning trees from $O(n^2)$ to $O(m)$. Our algorithms for undirected problems are based on the reduction in [G95b] of undirected packing and covering to the directed case.

The paper is organized as follows. Section 2 discusses the greedy algorithm for fractional arborescence packing. It proves the fractional case of Edmonds' Arborescence Packing Theorem. Section 3 provides the last part of our fractional packing algorithm, an efficient method to compute the capacity of an arborescence. Section 4 presents an integral packing algorithm that finds arborescences one-by-one. It proves the integral case of Edmonds' theorem. Section 5 extends this to our efficient integral packing algorithm. Section 6 discusses root-constrained arborescence packing. Section 7 presents the applications to digraph covering and undirected packing and covering. The rest of this section gives notation and definitions.

$\mathbf{R}^+$ denotes the set of positive real numbers.

Consider a universe V containing elements s, t and subsets S, T. $\overline{S}$ denotes the complement $V - S$. We use both set containment $S \subseteq T$ and proper set containment $S \subset T$. We often denote singleton sets by omitting set braces, e.g., $S \cup s$. An *$s\bar{t}$-set* contains s but not t; a *$\bar{t}$-set* is a nonempty set not containing t. Sets S and T are *intersecting* if $S \cap T$, $S - T$ and $T - S$ are all nonempty. A family of subsets of V is *laminar* if every two sets in the family are either disjoint or one contains the other.

Consider a directed or undirected graph $G = (V, E)$. If A is a set of edges $V(A)$ denotes the set of all vertices on these edges. A *capacity function* is a mapping $c : E \to \mathbf{R}^+$. By definition any edge has positive capacity. If an edge has zero capacity we always delete it from E.

An edge uv *enters* any $v\bar{u}$-set and *leaves* any $u\bar{v}$-set. For a set of vertices W, the *in-degree* $\rho_G(W)$ equals the total capacity of all edges entering W. Any graph in this paper will have a capacity function, so this notation always makes sense. On the other hand if A is a set of edges, we shall implicitly assume that each edge has unit capacity in A. Thus $\rho_A(W)$ denotes the number of edges entering W. This convention will be used frequently, but to prevent confusion the set of edges will always be named A when the convention is used.

The following submodular-type identity is well-known and easy to verify. Let $d_G(S,T)$ denote the total capacity of all edges having one end in $S-T$ and the other in $T-S$. Then for any sets $S,T \subseteq V$, $\rho(S)+\rho(T)=\rho(S\cup T)+\rho(S\cap T)+d(S,T)$. Here we abbreviate ρ_G to ρ and d_G to d.

A *branching* in a digraph is a set of edges A such that any vertex v has $\rho_A(v)\le 1$, and ignoring edge directions A is a forest. Branching A is an *arborescence* if $V(A)$ contains a unique vertex a with $\rho_A(a)=0$; in this case A is an *a-arborescence*. A is *spanning* if $V(A)=V$.

Consider a digraph $G=(V,E)$ with distinguished vertex a, and a capacity function $c: E\to \mathbf{R}^+$. A *fractional packing of a-arborescences* is a family $\mathcal{A}$ of spanning a-arborescences A, each with an associated multiplicity $\alpha(A)\in\mathbf{R}^+$ such that for any edge e, $\sum\{\alpha(A) : e\in A\in\mathcal{A}\}\le c(e)$. The *total multiplicity* of $\mathcal{A}$ is $\sum\{\alpha(A) : A\in\mathcal{A}\}$. An *integral packing* has all multiplicities $\alpha(A)$ integral. A *maximum packing* has maximum total multiplicity. The main problem of this paper is to find maximum fractional and integral packings of a-arborescences.

Let $\lambda(G)$ denote the minimum in-degree of an $\overline{a}$-set, i.e., $\lambda(G)=\min\{\rho_G(X) : \emptyset\ne X\subseteq V-a\}$. Edmonds' Arborescence Packing Theorem states that in any digraph with integral capacities, a maximum integral packing $\mathcal{A}$ has total multiplicity $\lambda(G)$ [E72]. (When we write $\lambda(G)$ the choice of root vertex a will be clear from context. Usually the root is named a.)

2 Greedy Fractional Packing

This section discusses the greedy approach to fractional packing. It proves the fractional case of Edmonds' Arborescence Packing Theorem and gives our efficient algorithm for fractional packing.

Consider an arbitrary set of edges A (the following notions will be used mainly when A is an arborescence). Define

$$\mu(A)=\min\{c(e) : e\in A\}.$$

When the graph G (with associated capacity function c) is not clear we write $\mu_G(A)$. For any α, $0\le\alpha\le\mu(A)$, let $G-\alpha A$ denote the graph G with the given capacity function c decreased by α on the edges of A. (As mentioned in Section 1, in $G-\alpha A$ we delete any edge that has zero capacity.) As an example when all capacities are one, $G-A$ is the graph with the edges A removed.

The *capacity of* A is the largest value $\alpha \le \mu_G(A)$ having $\lambda(G - \alpha A) = \lambda(G) - \alpha$. The capacity always exists, although it may be 0. We sometimes use the notation $\alpha(A)$ to denote the capacity of A.

To motivate this definition observe that since a spanning a-arborescence A enters any $\overline{a}$-set, $\lambda(G - \alpha A) \le \lambda(G) - \alpha$ for any $\alpha \ge 0$. Thus the capacity of A is an upper bound on the multiplicity of A in an a-arborescence packing of total multiplicity $\lambda(G)$, and $\lambda(G)$ is an upper bound on the total multiplicity of an a-arborescence packing. Edmonds' Arborescence Packing Theorem shows that in fact both these upper bounds are tight.

We will not invoke this theorem, but instead rely on the following simple characterization of capacity. If a set of edges A has $\mu(A) > 0$ yet $\alpha(A) = 0$ then some $\overline{a}$-set U has

$$\rho_G(U) = \lambda(G) \text{ and } \rho_A(U) > 1.$$

In proof suppose the opposite, i.e., every $\overline{a}$-set U has $\rho_G(U) > \lambda(G)$ or $\rho_A(U) = 1$. Then $\rho_{G-\alpha A}(U) = \rho_G(U) - \alpha\rho_A(U) \ge \lambda(G) - \alpha$ holds for small enough positive α in the first case and for any α in the second case (since $\rho_G(U) \ge \lambda(G)$). Thus $\alpha(A) > 0$ as desired.

Now consider the following *greedy algorithm* for fractional packing: Choose any spanning a-arborescence A of G that has positive capacity α. Place A in the packing with multiplicity α and replace G by $G - \alpha A$. If $\lambda(G)$ is still positive then find the rest of the packing recursively.

We will prove that the greedy algorithm works (in particular A exists) and does essentially the best job possible: It constructs a maximum packing, using at most m distinct arborescences. The main idea of the proof is to use the above sets U from capacity computations to build a laminar family of sets $\mathcal{F}$, which guides the choice of arborescences. We will use different versions of $\mathcal{F}$ for fractional and integral packing. We begin with remarks about $\mathcal{F}$ that hold for both versions.

$\mathcal{F}$ is a laminar family of nonsingleton $\overline{a}$-sets. Thus $\mathcal{F}$ contains at most $n-2$ sets (since $\mathcal{F}$ plus all singleton $\overline{a}$-sets and $\emptyset$ is a laminar family on $n-1$ elements). We always initialize $\mathcal{F}$ to $\emptyset$. We only change $\mathcal{F}$ using the following operation, defined for an $\overline{a}$-set U. To *uncross* $\mathcal{F} \cup \{U\}$ means to execute the following procedure:

$YI \leftarrow$ the family of maximal sets of $\mathcal{F}$ that are intersecting with U;
$Z \leftarrow U \cup \bigcup\{Y : Y \in YI\}$;
add Z to $\mathcal{F}$;

The family YI can be empty. If U is a nonsingleton set (as will always be the case) then so is Z. The new family $\mathcal{F}$ is laminar. (In proof let $\mathcal{F}$ denote the family before uncrossing. There is a unique minimal set $X \in \mathcal{F} \cup \{V\}$ that contains U. Any set of $\mathcal{F}$ properly contained in X is either disjoint from Z or contained in U or contained in some $Y \in YI$. Hence $\mathcal{F} \cup \{Z\}$ is laminar.)

The uncrossing operation may not change $\mathcal{F}$ since $\mathcal{F}$ may have already contained Z. We will always prove that when we uncross, $\mathcal{F}$ actually gets enlarged.

Now we return to the specific case of fractional packing. The complete specification of $\mathcal{F}$ is that it consists of nonsingleton $\overline{a}$-sets X that satisfy

$$\rho_G(X) = \lambda(G).$$

Recall that our algorithm changes the graph G as it constructs the packing; in this equation G refers to the current graph. Let us summarize the properties of various sets of the uncrossing operation in fractional packing.

Lemma 2.1. *Suppose we uncross $\mathcal{F} \cup \{U\}$ for an $\overline{a}$-set U that has $\rho_G(U) = \lambda(G)$. Let Y and Y' be any sets in YI. Then $\rho_G(Z) = \rho_G(Y \cap U) = \lambda(G)$ and $d_G(Y, U) = d_G(Y - U, Y' - U) = 0$.*

Proof. In this argument we omit the subscript from ρ_G and d_G. The submodular identity $\rho(Y) + \rho(U) = \rho(Y \cap U) + \rho(Y \cup U) + d(Y, U)$ implies $\rho(Y \cap U) = \rho(Y \cup U) = \lambda(G)$ and $d(Y, U) = 0$. A similar argument applied to $Y \cup U$ and $Y' \cup U$ shows $d(Y \cup U, Y' \cup U) = d(Y - U, Y' - U) = 0$. Finally repeated applications of the argument, using $Z = \bigcup\{Y \cup U : Y \in YI \cup \{\emptyset\}\}$, show $\rho(Z) = \lambda(G)$. □

It is convenient to work with a more detailed version of the greedy algorithm. (We relate this version to the original greedy algorithm after Theorem 2.1.) The following version of the greedy fractional packing algorithm uses $\mathcal{F}$ to choose an arborescence A. (Unlike the greedy algorithm, A can have capacity 0.)

procedure *fractional_pack*; **begin**
$\mathcal{F} \leftarrow \emptyset$;
while $\lambda(G) > 0$ **do begin**
  $A \leftarrow$ a spanning a-arborescence of G with $\rho_A(X) = 1$ for every $X \in \mathcal{F}$;
  $\alpha \leftarrow$ the capacity of A; /* possibly $\alpha = 0$ */
  $G \leftarrow G - \alpha A$;
  if $\mu_G(A) > 0$ **then begin** /* no edge was voided in the previous step */
    $U \leftarrow$ a minimal $\overline{a}$-set with $\rho_G(U) = \lambda(G)$ and $\rho_A(U) > 1$;
    uncross $\mathcal{F} \cup \{U\}$; **end**; **end**; **end**;

In the statement of this algorithm we implicitly assume that each arborescence A with positive capacity α gets added to the packing (with multiplicity α).

The existence of the set U was shown above (since A has capacity 0 after αA is deleted). U is not a singleton set since $\rho_A(U) > 1$. To show *fractional_pack* is well-defined we must prove that the spanning a-arborescence A exists. We begin with a key fact.

Fix any set $X \in \mathcal{F} \cup \{V\}$. Define G_X as the subgraph of G induced by X, with every maximal set of $\mathcal{F}$ that is properly contained in X contracted. Choose a vertex b in G_X as follows: If $X = V$ then $b = a$. If $X \neq V$ then choose b so that some edge of G entering X is directed to b.

Lemma 2.2. *G_X has a spanning b-arborescence.*

Proof. The lemma is equivalent to $\lambda(G_X) > 0$. Thus it suffices to show that for every $\overline{b}$-set S in G_X, some edge of G goes from $X - S$ to S.

If $X = V$ this follows from $\lambda(G) > 0$. Suppose $X \neq V$. The edge defining b contributes to $\rho_G(X)$ but not $\rho_G(S)$. Yet since $X \in \mathcal{F}$, $\rho_G(X) = \lambda(G) \leq \rho_G(S)$. Thus some edge contributes to $\rho_G(S)$ but not $\rho_G(X)$, i.e., it enters S but not X. This is the desired edge. □

Now we show the arborescence A of *fractional_pack* exists by giving an algorithm to construct it. Initialize A to a spanning a-arborescence of G_V. Then repeat the following step until A spans G: Choose a set $X \in \mathcal{F}$ such that A contains an edge directed to a vertex b of G_X but A does not span G_X. Enlarge A by adding a spanning b-arborescence of G_X.

Lemma 2.2 shows the spanning a- and b-arborescences exist. A simple induction shows that A is an arborescence. Each $X \in \mathcal{F}$ has $\rho_A(X) = 1$ (by construction). Thus the algorithm constructs A as specified in *fractional_pack*.

Theorem 2.1. *Procedure* fractional_pack *finds a fractional packing with total multiplicity $\lambda(G)$, containing at most m distinct arborescences.*

Proof. The definition of capacity implies that any iteration decreases $\lambda(G)$ by α and increases the multiplicity of the packing by α. Thus if *fractional_pack* halts (necessarily with $\lambda(G) = 0$) the packing has the desired multiplicity. So to prove the theorem we need only show that *fractional_pack* halts having found at most m distinct arborescences.

We will show that the family $\mathcal{F}$ gets enlarged by any uncrossing operation. This implies the theorem. To see this note that $\mathcal{F}$ gets enlarged at most $n-2$ times. An iteration that does not enlarge $\mathcal{F}$ voids an edge of G when αA is deleted. Before the last iteration this occurs at most $m-n+1$ times (since the last arborescence contains $n-1$ edges). Thus the number of iterations, and hence the number of distinct arborescences, is at most $(m-n+1)+1+(n-2)=m$.

To complete the proof consider an iteration that does an uncrossing operation. We wish to show that uncrossing $\mathcal{F}\cup\{U\}$ enlarges $\mathcal{F}$, i.e., the set Z of the uncrossing procedure was not already in $\mathcal{F}$. We claim

$$\rho_A(Z) \ge \rho_A(U).$$

Since $\rho_A(U) > 1$ (by the choice of U) the claim implies $\rho_A(Z) > 1$. Hence $Z \notin \mathcal{F}$ before uncrossing (by the choice of A). Thus the claim suffices to prove the theorem.

To prove the claim we first note two relations. Let the iteration end with graph G. Any set $Y \in YI$ has

$$\rho_A(Y \cap U) = 1. \tag{1}$$

This follows since Lemma 2.1 shows $\rho_G(Y\cap U)=\lambda(G)$, so if (1) fails then $Y \cap U$ contradicts the minimality of U. (This is the only place the proof uses the minimality of U. This fact will be used below.)

The second relation is that any sets $Y, Y' \in YI$ have

$$d_A(Y,U) = d_A(Y-U, Y'-U) = 0. \tag{2}$$

In proof, Lemma 2.1 shows (2) for the function d_G. Since an uncrossing operation is done, no edge of A was voided, i.e., every edge of A has positive capacity in G. This implies (2) holds for d_A as well as d_G.

Now we prove the claim by exhibiting an injection from the edges of A that enter U to edges of A that enter Z. Let edge $e \in A$ enter U. If e also enters Z it is mapped to itself. Suppose e does not enter Z. Thus for some $Y \in YI$, e leaves $Y-U$. Now (2) shows e goes from $Y-U$ to $Y\cap U$. Thus (1) shows e is unique for the set Y. Since some edge $f\in A$ enters Y, (1) shows f goes from $\overline{Y}$ to $Y-U$. Now (2) shows f goes from $\overline{Z}$ to $Y-U$. Since f enters Z we can map e to f. It is clear that e is the only edge that gets mapped to f. $\square$

The theorem gives an independent proof of the real-valued case of Edmonds' Arborescence Packing Theorem [E72]: Any digraph with real capacities has a fractional packing of a-arborescences of total multiplicity $\lambda(G)$.

The theorem also implies our claim for the greedy algorithm, i.e., it finds a maximum packing, and this packing contains at most m distinct arborescences A. This follows because any arborescence A chosen by the greedy algorithm can be chosen by *fractional_pack*. (The greedy algorithm chooses A so its capacity is positive. This implies $\rho_A(X) = 1$ for each $X \in \mathcal{F}$ by the definition of $\mathcal{F}$.) Hence *fractional_pack* can construct the same packing as the greedy algorithm.

We note an extension of the theorem used in Section 6: If $\mathcal{F}$ is initialized to a laminar family of f nonsingleton $\overline{a}$-sets of in-degree $\lambda(G)$, *fractional_pack* obtains a maximum fractional packing containing at most $m - f$ distinct arborescences. This follows easily from the proof of the theorem.

We turn to implementing *fractional_pack* efficiently. Section 3 gives an algorithm to compute the capacity of an arborescence A. The algorithm also computes a set $\widehat{U}$ that satisfies all the properties of U in *fractional_pack* except minimality. To use $\widehat{U}$ instead of U in *fractional_pack* replace the line that computes U by the following two lines:

$\widehat{U} \leftarrow$ an (arbitrary) $\overline{a}$-set with $\rho_G(\widehat{U}) = \lambda(G)$ and $\rho_A(\widehat{U}) > 1$;
$U \leftarrow \widehat{U} \cap X$ for X a minimal set of $\mathcal{F} \cup \{V\}$ having $\rho_A(\widehat{U} \cap X) > 1$;

Let us prove Theorem 2.1 holds for this implementation of *fractional_pack*. It suffices to show that the set U constructed above has all the necessary properties of U in the original algorithm. The first property is that $\rho_G(U) = \lambda(G)$. This translates to $\rho_G(\widehat{U} \cap X) = \lambda(G)$, which is true by the submodularity argument of Lemma 2.1. The second property of U is $\rho_A(U) > 1$. This translates to $\rho_A(\widehat{U} \cap X) > 1$, which holds by definition. The final property of U is its minimality, but recall this is only used to establish (1). Since in (1) Y is intersecting with $U = \widehat{U} \cap X$, $Y \subset X$. Thus $Y \cap U = Y \cap \widehat{U}$. The minimality of X shows $\rho_A(Y \cap \widehat{U}) = 1$, i.e., (1) holds.

Now we show that assuming the results of the next section, this implementation of *fractional_pack* achieves our time bound for fractional packing.

Corollary 2.1. *Procedure* fractional_pack *(as implemented above) runs in time $O(n^3 m \log(n^2/m))$.*

Proof. We find the arborescence A in time $O(m)$ using the procedure stated after Lemma 2.2. We now consider two cases depending on whether or not α, the capacity of A, equals $\mu(A)$, the capacity of the smallest edge in A.

Suppose $\alpha = \mu(A)$ (i.e., deleting αA voids an edge). In this case Lemma 3.1 shows the capacity is found in time $O(nm \log(n^2/m))$. Since there are $\leq m \leq n^2$ iterations of *fractional_pack*, the total time for these iterations is within the desired time bound.

Next suppose $\alpha < \mu(A)$. In this case the capacity is found in time $O(n^2 m \log(n^2/m))$, as well as the set $\widehat{U}$ of the implementation (by Lemma 3.1 and the remark following it). Sets X and U are found in a straightforward way in time $O(n^2)$. (Find X by testing each set of $\mathcal{F}$.) The uncrossing operation can also be done in this time bound. There are $\leq n-2$ iterations with $\alpha < \mu(A)$ since each enlarges $\mathcal{F}$. This gives the desired time bound. $\square$

3 Capacity Computation

Let A be an arbitrary set of edges. This section presents algorithms for computing the capacity $\alpha(A)$. The main method amounts to an implementation of Newton's method for finding a root (although we will not rely on this correspondence).

We begin by reviewing the definition of capacity. An equivalent version of the definition is that the capacity is the largest value $\alpha \leq \mu_G(A)$ such that for each $\overline{a}$-set U, $\rho_{G-\alpha A}(U) = \rho_G(U) - \alpha\rho_A(U) \geq \lambda(G) - \alpha$. If $\rho_A(U) \leq 1$ the desired inequality holds for any α, since $\rho_G(U) \geq \lambda(G)$. If $\rho_A(U) > 1$ then the desired inequality holds if and only if $\alpha \leq \alpha_U$ where we define

$$\alpha_U \equiv \frac{\rho_G(U) - \lambda(G)}{\rho_A(U) - 1}.$$

It is possible that $\alpha_U = 0$.

This characterization of α motivates the following algorithm to compute the capacity.

> $\alpha \leftarrow \mu_G(A)$;
> **while** $\lambda(G - \alpha A) < \lambda(G) - \alpha$ **do begin**
> $U \leftarrow$ an $\overline{a}$-set of minimum in-degree in $G - \alpha A$;
> $\alpha \leftarrow \alpha_U$; **end**;

To verify this algorithm is correct we must check two facts. First the algorithm never divides by 0 in computing α_U (since it is easy to see that $\rho_A(U) > 1$). The second fact is that the algorithm halts. We will prove this by showing there are at most $|A|$ iterations.

First observe that the sequence of values of α is strictly decreasing. In proof, recall that U has in-degree $\geq \lambda(G) - \alpha$ exactly when $\alpha \leq \alpha_U$. Thus each iteration changes α from a value larger than α_U to α_U.

Next we claim that if an iteration computes the set U, then any $\overline{a}$-set X with $\rho_A(X) \geq \rho_A(U)$ has $\alpha_X \geq \alpha_U$. Observe that the claim implies the desired bound: It implies the sequence $\rho_A(U)$ is strictly decreasing. Since $\rho_A(U) \leq |A|$ the desired bound of at most $|A|$ iterations follows.

To prove the claim let α equal its value when the iteration begins and U is computed. The definition of U implies the inequality $\rho_G(X) - \alpha\rho_A(X) \geq \rho_G(U) - \alpha\rho_A(U)$. The inequality $(\alpha - \alpha_U)\rho_A(X) \geq (\alpha - \alpha_U)\rho_A(U)$ follows from $\alpha_U < \alpha$ and the assumption $\rho_A(X) \geq \rho_A(U)$. Adding the two inequalities gives $\rho_G(X) - \alpha_U\rho_A(X) \geq \rho_G(U) - \alpha_U\rho_A(U) = \lambda(G) - \alpha_U$. This implies the claim $\alpha_X \geq \alpha_U$.

Lemma 3.1. *The capacity $\alpha(A)$ of an arborescence A can be computed in time $O(n^2 m \log(n^2/m))$. If $\alpha(A) = \mu_G(A)$ the time is $O(nm \log(n^2/m))$.*

Proof. We check the termination condition of the loop and find the set U using the global minimum cut algorithm of Hao and Orlin [HO]. This gives time $O(nm \log(n^2/m))$ for one iteration.

As shown above there are at most $|A| = n - 1$ iterations. This implies the first time bound. If $\alpha(A) = \mu(A)$ there is only one iteration. This implies the second time bound. $\square$

Observe that if $\alpha(A) < \mu(A)$, the last set U found in the algorithm has $\rho_A(U) > 1$ and $\alpha(A) = \alpha_U$. Writing $\alpha = \alpha(A)$ we have $\rho_{G-\alpha A}(U) = \lambda(G) - \alpha = \lambda(G - \alpha A)$. Hence this set U can be returned as the set $\widehat{U}$ in the efficient implementation of procedure *fractional_pack*.

We close this section by discussing integral packing. The integral packing algorithm of Section 5 computes the floor of the capacity (i.e., $\lfloor \alpha(A) \rfloor$). It is easy to see that our algorithm works if we change the last line to $\alpha \leftarrow \lfloor \alpha_U \rfloor$. Lemma 3.1 continues to hold. (All arguments are the same, even to show $\rho_A(U)$ is decreasing.) The advantage of this version is that all computations preserve integrality.

In addition to the time bounds of Lemma 3.1, this algorithm runs in time $O((\log(nN))\, nm \log(n^2/m))$. We will not use this fact but we summarize the proof: We need only show there are $O(\log(nN))$ iterations. But this bound holds for any implementation of the Newton-Dinkelbach method for finding the minimum of a function $u(x)/v(x)$ [R]. It is straightforward to check that our algorithm is an implementation of this method.

A similar (slightly better) time bound can be achieved by binary search. By definition $\lfloor \alpha(A) \rfloor$ is the largest integer value $\alpha \leq \mu_G(A)$ having $\lambda(G - \alpha A) = \lambda(G) - \alpha$. This largest value can be found by a binary search. Each probe computes $\lambda(G - \alpha A)$ in time $O(nm \log(n^2/m))$ using the Hao-Orlin algorithm. There are $O(\log N)$ probes to find the desired value.

Corollary 3.1. *If all capacities are integers then* $\lfloor \alpha(A) \rfloor$ *can be computed in time* $O(\min\{n, \log N\}$ $nm \log(n^2/m))$. $\square$

4 One-by-one Packing

This section presents an integral packing algorithm. The algorithm finds a packing where every arborescence has multiplicity one (so it is not efficient when capacities are large). The algorithm gives an independent proof of the integral case of Edmonds' Arborescence Packing Theorem, and is used in our efficient integral packing algorithm (in Section 5).

Consider a digraph $G = (V, E)$ with distinguished vertex a. G may have an integral capacity function, in which case we consider each edge to have multiplicity equal to its capacity. Graph G does not change in this section, so we write $\lambda = \lambda(G)$.

We will grow an a-arborescence A of the packing edge-by-edge. Initially A contains one edge. We maintain A as an a-arborescence that in addition satisfies the invariant

$$\lambda(G - A) \geq \lambda - 1. \tag{3}$$

When A becomes a spanning arborescence, still satisfying (3), (3) must hold with equality. At that point we add A to the packing and recursively pack the graph $G - A$.

To guide the construction of A we maintain a laminar family $\mathcal{F}$ of $\overline{a}$-sets X, each of which has

$$\rho_{G-A}(X) = \lambda - 1 \text{ or } \rho_G(X) = \lambda. \tag{4}$$

Observe that condition (4) does not change if we replace the second alternative $\rho_G(X) = \lambda$ by $\rho_G(X) = \lambda$ and $\rho_A(X) = 0$. (If $\rho_G(X) = \lambda$ and $\rho_A(X) > 0$ then (3) implies the first alternative holds, $\rho_{G-A}(X) = \lambda - 1$.)

When the packing algorithm begins to grow the very first arborescence, $\mathcal{F}$ is initialized to $\emptyset$. Thereafter the only operation on $\mathcal{F}$ is to uncross $\mathcal{F} \cup \{U\}$, where U is a (nonsingleton) $\overline{a}$-set with $\rho_{G-A}(U) = \lambda - 1$. Now we summarize the properties of various sets when we uncross $\mathcal{F} \cup \{U\}$. (Recall the definitions of YI and Z in the uncrossing operation.)

Lemma 4.1. *Suppose we uncross* $\mathcal{F} \cup \{U\}$ *for an* $\overline{a}$*-set* U *that has* $\rho_{G-A}(U) = \lambda - 1$. *Then* $\rho_{G-A}(Z) = \lambda - 1$. *For any set* $Y \in YI$, $\rho_{G-A}(Y) = \rho_{G-A}(Y \cap U)$ *and* $d_{G-A}(Y, U) = 0$.

Proof. In this argument ρ and d refer to the graph $G - A$. We first prove the two claims involving Y, as well as the relation $\rho(Y \cup U) = \lambda - 1$. Consider the submodular identity $\rho(Y) + \rho(U) =$

$\rho(Y \cap U) + \rho(Y \cup U) + d(Y, U)$. Using (3), $\rho(Y \cup U) \geq \lambda - 1 = \rho(U)$. Thus to show the three desired equations it suffices to prove $\rho(Y \cap U) \geq \rho(Y)$.

If $\rho(Y) = \lambda - 1$ then (3) shows $\rho(Y \cap U) \geq \rho(Y)$ as desired. Suppose $\rho(Y) > \lambda - 1$. Since $Y \in \mathcal{F}$, (4) shows $\rho(Y) = \rho_G(Y) = \lambda$ and $\rho_A(Y) = 0$. Since A is an arborescence the latter implies $\rho_A(Y \cap U) = 0$. The definition of λ now implies $\rho(Y \cap U) \geq \lambda$ as desired.

To complete the proof it remains to show that $\rho(Z) = \lambda - 1$. Since $Z = \bigcup\{Y \cup U : Y \in YI \cup \{\emptyset\}\}$, repeated applications of the submodular identity, using (3) and the relations $\rho(Y \cup U) = \lambda - 1$, give the desired equation. $\square$

The following algorithm constructs one spanning a-arborescence A for the packing.

procedure *grow*; **begin**

$A \leftarrow$ any edge leaving a;

while A is not spanning **do begin**

  let e be an edge leaving $V(A)$ that does not enter any set $X \in \mathcal{F}$ with $\rho_{G-A}(X) = \lambda - 1$;

  if $A \cup e$ satisfies (3) **then** add e to A **else begin**

    $U \leftarrow$ the minimal set entered by e and having $\rho_{G-A}(U) = \lambda - 1$;

    uncross $\mathcal{F} \cup \{U\}$; **end**; **end**; **end**;

To show this algorithm is well-defined we must prove the edge e exists. This is done right after Lemma 4.2 below. It is clear that there is a unique minimal set U. (In proof write $e = vw$. Then U is a $w\overline{a}$-set of minimum in-degree by (3). The minimal such set is unique, a well-known consequence of the submodularity of ρ.)

Now we will prove that the edge e exists. For $X \in \mathcal{F} \cup \{V\}$ recall the definition of G_X from Section 2. Choose any minimal set $X \in \mathcal{F} \cup \{V\}$ where A contains at least one but not every vertex of G_X (strictly speaking we do not mean A here, but rather the arborescence corresponding to A in the contracted graph G_X).

Lemma 4.2. *G_X contains an edge leaving $V(A)$.*

Proof. Let S be the set of vertices of G_X not contained in A. A contains every vertex of G that is in $X - S$ or in a contracted vertex in $X - S$ (by the minimality of X). Thus it suffices to show that some edge of G goes from $X - S$ to S. This is clear for $X = V$ since $\lambda > 0$. For $X \neq V$ the fact that A enters X but not S shows that $\rho_{G-A}(S) \geq \lambda = \rho_{G-A}(X) + 1$. Thus $G - A$ contains an edge entering S but not X. This is the desired edge. $\square$

It is easy to check that the edge of the lemma can be used for e in *grow*.

We will show below that *grow* actually halts, finding a spanning a-arborescence A. Assuming that, consider the following packing algorithm, *one-by-one_pack*: Initialize $\mathcal{F}$ to $\emptyset$. Execute *grow* to find a spanning arborescence A. Then add A to the packing (with multiplicity one) and replace G by $G - A$. If $\lambda(G) > 0$ then repeat the process.

Theorem 4.1. *Procedure* one-by-one_pack *finds an integral packing of* $\lambda(G)$ *arborescences.*

Proof. Assume temporarily that every iteration of *grow* either adds an edge to A or enlarges $\mathcal{F}$. The latter can occur at most $n - 2$ times. Thus each execution of *grow* eventually returns a spanning arborescence A. Adding A to the packing increases its multiplicity by 1 while deleting A from G decreases λ by 1 (by (3) which holds with equality). Furthermore deleting A preserves (4), since immediately before deleting each $X \in \mathcal{F}$ has $\rho_{G-A}(X) = \lambda - 1$. Thus it is clear that the desired packing of $\lambda(G)$ arborescences is eventually obtained.

It remains to prove the assumption. Suppose an iteration does not enlarge A, i.e., $A \cup e$ does not satisfy (3). The uncrossing operation adds a set Z to $\mathcal{F}$. Lemma 4.1 shows $\rho_{G-A}(Z) = \lambda - 1$, i.e., (4) holds. U is not a singleton (if $e = vw$, $\rho_{G-A}(w) \geq \lambda$). We claim that edge e enters Z. This shows $Z \notin \mathcal{F}$ before uncrossing (by the choice of e). Thus $\mathcal{F}$ is actually enlarged as desired.

To prove e enters Z we assume the contrary and derive a contradiction. Let $e = vw$. Since e enters U but not Z, $v \in Y - U$ for some $Y \in YI$. This implies $w \in Y \cap U$, since $e \in G - A$ and $d_{G-A}(Y, U) = 0$ (Lemma 4.1). Thus e enters $Y \cap U$. Furthermore $\rho_{G-A}(Y \cap U) = \lambda - 1$. (To see this note that $\rho_A(Y) > 0$. Since $Y \in \mathcal{F}$ this implies $\rho_{G-A}(Y) = \lambda - 1$ by (4). Now Lemma 4.1 gives the desired equation.) The last two properties of $Y \cap U$ contradict the minimality of U. $\square$

The theorem gives an independent proof of Edmonds' Arborescence Packing Theorem [E72], i.e., any digraph with integral capacities has a packing of $\lambda(G)$ a-arborescences.

Procedure *one-by-one_pack* can be efficiently implemented on a graph where all edges have unit capacity (parallel edges are allowed): We use the matroid techniques of [G95a] to check condition (3) and to find the set U when (3) fails. This implementation of *one-by-one_pack* achieves the same asymptotic time bound as the packing algorithm of [G95a]: if G has kn edges and $\lambda(G) = k$, it finds a packing of k arborescences in time $O((kn)^2)$. In practice *one-by-one_pack* may be faster than

[G95a], since it may be easier to maintain $\mathcal{F}$ than to do the graph contraction operations needed in the packing algorithm of [G95a].

We conclude this section by presenting the more efficient version of procedure *grow* that we use in the algorithms of this paper (mainly the integral packing algorithm *integral_pack* of Section 5). The idea is to gain efficiency by not checking condition (3) for every edge e. Instead we guess a spanning arborescence, and then search it to find the first edge whose addition violates (3).

The following procedure *fast_grow* is called with argument A a spanning a-arborescence. Also when *fast_grow* is called every $X \in \mathcal{F}$ satisfies both alternatives of (4), i.e., $\rho_{G-A}(X) = \lambda - 1$ and $\rho_G(X) = \lambda$. However A need not satisfy (3). The *fast_grow* procedure returns a spanning a-arborescence A that satisfies (3); furthermore all the entrance conditions hold. For $e \in A$ let A/e denote the set of all edges added to A strictly before e.

procedure *fast_grow*(A); **begin**
while A does not satisfy (3) **do begin**
    $e \leftarrow$ the last edge added to A with A/e satisfying (3);
    $A \leftarrow A/e$;   /* $A/e \cup e$ violates (3) */
    $U \leftarrow$ the minimal set entered by e having $\rho_{G-A}(U) = \lambda - 1$;
    uncross $\mathcal{F} \cup \{U\}$;
    repeat   /* "grow loop" */
        add an edge to A that leaves $V(A)$
            and does not enter any set $X \in \mathcal{F}$ with $\rho_{G-A}(X) = \lambda - 1$;
    until A is spanning;
    end;
return A; **end** ;

In the first iteration of *fast_grow* we need a convention so the notation A/e makes sense. Assume the edges of A have been added in an order consistent with the grow loop, i.e., for each e, A/e is an a-arborescence.

The procedure *fast_grow* is correct because it is essentially the same as *grow*. More precisely we can prove by induction that immediately after each uncrossing operation in *fast_grow*, A, e and $\mathcal{F}$ are exactly the same as they are in an execution of *grow* that has constructed the arborescence A, attempted to add e, and instead done the same uncrossing operation as *fast_grow*. (The only difference between procedures is that *fast_grow* adds more tentative edges to A in its grow loop. This loop works correctly, i.e., A eventually becomes a spanning arborescence, by repeated application of Lemma 4.2. After that the statement $A \leftarrow A/e$ prunes A back to another arborescence, say A_0. Note that A_0 is an arborescence by the grow loop and the definition of $/$. A_0 could have been constructed by *grow*, by the statement of the grow loop (or by the entrance conditions if A is the input arborescence). Also A_0 contains the previous A_0, since both satisfy (3).)

It remains to estimate the efficiency of *fast_grow*. We do this in the context of the integral packing algorithm *integral_pack* of Section 5. This procedure calls *fast_grow*(A) with an arborescence A that does not satisfy (3) (but does satisfy all entrance conditions). We now give the implementation details of *fast_grow*, and show that the total time for all calls to *fast_grow* from *integral_pack* is

$$O(n^2 m \log(n^2/m) \log n).$$

We check that A satisfies (3) using the minimum cut algorithm of Hao and Orlin [HO]. This uses time $O(nm \log(n^2/m))$. If (3) fails we find the desired edge e using a binary search. The search maintains an interval of edges that were added consecutively to A and contains e. Each probe chooses an appropriate edge $f \in A$ and checks if A/f satisfies (3), again using the Hao-Orlin algorithm. The last execution of the Hao-Orlin algorithm finds e as well as the desired minimal set U. The entire binary search uses time $O(nm \log(n^2/m) \log n)$.

The algorithm maintains the forest representing $\mathcal{F}$. This allows us to uncross $\mathcal{F} \cup \{U\}$ in time $O(n)$. Also the grow loop constructs each arborescence A in $O(m)$ time.

If an iteration of (the while loop of) *fast_grow* starts with A not satisfying (3), the uncrossing operation enlarges $\mathcal{F}$. This follows from the proof of Theorem 4.1 and the correspondence between *fast_grow* and *grow*. Since each call *fast_grow*(A) in *integral_pack* has A not satisfying (3), we conclude there are a total of $O(n)$ iterations in all calls to *fast_grow* from *integral_pack*. This implies the desired time bound.

5 Integral Packing

This section presents our algorithm to find a maximum integral packing that uses at most $m+n-2$ distinct arborescences.

Let $G = (V, E)$ be a digraph with a distinguished vertex a and an integral capacity function c. For a set of edges A, the *integral capacity of* A is the largest integral value $\alpha \leq \mu(A)$ having $\lambda(G - \alpha A) = \lambda(G) - \alpha$. The integral capacity equals the floor of the capacity defined in Section 2. Our basic approach is to pack arborescences A using their integral capacity as their multiplicity in the packing.

We add two ideas to make this approach efficient. When we pack an arborescence A that has integral capacity $< \mu(A)$, we also pack another arborescence (of multiplicity 1) that enlarges $\mathcal{F}$. This idea allows us to achieve the desired number of distinct arborescences in the packing.

Another consequence of this idea is that relatively few arborescences in the packing have capacity $< \mu(A)$. This allows us to avoid capacity computations that have the trivial answer $\mu(A)$, as follows: We pack arborescences in batches, finding a sequence of n arborescences to add to the packing. We tentatively pack each of these arborescences A with multiplicity $\mu(A)$. Then we determine the first arborescence B of this sequence that actually has capacity $< \mu(B)$. We need to compute the capacity of B, but we avoid computing the trivial capacities of the arborescences preceding B. Now we present the packing algorithm in detail.

Let $\mathcal{A}$ be a family of a-arborescences A, each with an associated integral multiplicity $\tau(A)$. $G - \mathcal{A}$ denotes the graph obtained by starting with G and deleting $\tau(A)$ copies of each $A \in \mathcal{A}$. $\mathcal{A}$ is *valid* if each $A \in \mathcal{A}$ is a spanning a-arborescence and for $\tau(\mathcal{A})$ the total multiplicity of $\mathcal{A}$, $\lambda(G - \mathcal{A}) = \lambda(G) - \tau(\mathcal{A})$. For $A \in \mathcal{A}$, $\mathcal{A}/A$ denotes the family of all arborescences added to $\mathcal{A}$ strictly before A. In the following integral packing algorithm we make the usual assumption that whenever we delete arborescences from G we add these arborescences to the packing, with the corresponding multiplicities.

procedure *integral_pack*; **begin**

$\mathcal{F} \leftarrow \emptyset$;

while $\lambda(G) > 0$ **do begin**

    $\mathcal{A} \leftarrow \emptyset$;

    repeat

        $A \leftarrow$ a maximal a-arborescence in $G - \mathcal{A}$ with $\rho_A(X) \leq 1$ for every $X \in \mathcal{F}$;

        add A with multiplicity $\mu_{G-\mathcal{A}}(A)$ to $\mathcal{A}$;

    until A is not spanning or $|\mathcal{A}| = n$;

    if $\mathcal{A}$ is valid **then** $G \leftarrow G - \mathcal{A}$

    else begin

        $B \leftarrow$ the last arborescence added to $\mathcal{A}$ with $\mathcal{A}/B$ valid;

        $G \leftarrow G - \mathcal{A}/B$;

        if $\lambda(G) > 0$ **then begin**

            $\beta \leftarrow$ the integral capacity of B (in the current graph G);

            $G \leftarrow G - \beta B$;

$A \leftarrow$ *fast_grow*(B);   /∗ *fast_grow* maintains the laminar family $\mathcal{F}$ ∗/

$G \leftarrow G - A$; **end**; **end**; **end**; **end**;

We remark that B can be the first arborescence of $\mathcal{A}$. Also B can have capacity 0.

We now show that *integral_pack* finds an integral packing of total multiplicity $\lambda(G)$. The main step is to show that every family of arborscences deleted from G (including the singleton family A in the last line) is valid. We prove this by induction. In addition we inductively assert that throughout the execution of *integral_pack* except during the execution of *fast_grow*, every set $X \in \mathcal{F}$ has $\rho_G(X) = \lambda(G)$. Note that this equation is preserved when we delete a valid family of arborescences (every arborescence in a valid family enters every such X exactly once).

Consider the family $\mathcal{A}$ constructed in the repeat loop. If $\mathcal{A}$ is valid it is clear that when *integral_pack* deletes $\mathcal{A}$ the inductive assertions are preserved.

Suppose $\mathcal{A}$ is invalid. Deleting the valid family $\mathcal{A}/B$ preserves the inductive assertions. Assume the resulting graph has $\lambda(G) > 0$ (else we are done). We first claim this implies arborescence B is spanning. In proof, when B was added to $\mathcal{A}$ (in the repeat loop) the graph $G - \mathcal{A}$ was precisely the current graph G. In general during the repeat loop constructing $\mathcal{A}$, the arborescence A is spanning if $\mathcal{A}$ is valid and $\lambda(G - \mathcal{A}) > 0$. This follows from Lemma 4.2. The claim follows.

Now βB is a valid family, so deleting it preserves the inductive assertions. After this deletion each edge of B still has positive capacity and B is a spanning arborescence with integral capacity 0, i.e., $\lambda(G - B) < \lambda(G) - 1$. Thus in the call *fast_grow*(B), B does not satisfy (3). Furthermore all the entrance conditions of *fast_grow* are satisfied (every $X \in \mathcal{F}$ satisfies both alternatives of (4) since $\rho_G(X) = \lambda(G)$ and $\rho_B(X) = 1$). Hence *fast_grow* returns an arborescence A that is valid (i.e., A satisfies (3) with equality). Thus deleting A preserves the inductive assertions. (For the second assertion, i.e., every set $X \in \mathcal{F}$ has $\rho_G(X) = \lambda(G)$, note that *fast_grow* enlarges the family $\mathcal{F}$. Deleting A reestablishes the second assertion, by (4) and (3).) This completes the induction.

Since each iteration of *integral_pack* adds at least one arborescence to the packing, the procedure eventually halts with a packing of total multiplicity $\lambda(G)$.

Theorem 5.1. *Procedure* integral_pack *finds an integral packing with total multiplicity* $\lambda(G)$*, containing at most* $m + n - 2$ *distinct arborescences.*

Proof. Let $\mathcal{A}$ be the packing obtained for the original input graph. It remains only to check the number of distinct arborescences in $\mathcal{A}$. At most $m - n + 2$ arborescences A have multiplicity $\mu(A)$

in $\mathcal{A}$. This follows since deleting such an A voids an edge, and the last arborescence added to $\mathcal{A}$ contains $n-1$ edges. The remaining arborescences of $\mathcal{A}$ come in pairs $B, A = \mathit{fast\_grow}(B)$. Section 4 shows that each execution of *fast_grow* does an uncrossing operation that enlarges $\mathcal{F}$ (since as noted above, B does not satisfy (3)). Hence there are at most $n-2$ such calls. Thus $|\mathcal{A}| \le (m-n+2) + 2(n-2) = m+n-2$. □

As in fractional packing we have an extension of the theorem used in Section 6: If $\mathcal{F}$ is initialized to a laminar family of f nonsingleton $\overline{a}$-sets of in-degree $\lambda(G)$, *integral_pack* obtains a maximum fractional packing containing at most $m+n-2(f+1)$ distinct arborescences. This follows from the proof of the theorem.

We conclude this section by giving the remaining implementation details of *integral_pack* and computing the time bound.

Corollary 5.1. *Procedure* integral_pack *runs in time* $O(\min\{n, \log N\} n^2 m \log(n^2/m))$.

Proof. Consider an iteration of the main while loop of *integral_pack*. It starts by finding the family of arborescences $\mathcal{A}$. Since $|\mathcal{A}| \le n$ the time to construct $\mathcal{A}$ (in the repeat loop) is $O(nm)$. We now consider two cases depending on whether or not $\mathcal{A}$ is valid.

Consider the iterations that have $\mathcal{A}$ valid. Validity is checked using the Hao-Orlin minimum cut algorithm [HO]. Thus the time for such an iteration is $O(nm \log(n^2/m))$. Hence it suffices to show there are $\le n$ such iterations.

An iteration with $\mathcal{A}$ valid has $|\mathcal{A}| = n$. (If $|\mathcal{A}| < n$ the last arborescence of $\mathcal{A}$ is not spanning and $\mathcal{A}$ is invalid.) Each arborescence $A \in \mathcal{A}$ has multiplicity $\mu(A)$, so deleting it from G voids an edge. Thus $\le m$ arborescences are found in all these iterations. We conclude that $\le m/n \le n$ iterations have $\mathcal{A}$ valid, as desired.

Now consider the iterations of the main while loop that have $\mathcal{A}$ invalid. The arborescence B is found by a binary search in time $O(nm \log(n^2/m) \log n)$. Corollary 3.1 shows that the integral capacity of B is found in time $O(\min\{n, \log N\}\ nm \log(n^2/m))$. Since each execution of *fast_grow* enlarges $\mathcal{F}$, at most n iterations have $\mathcal{A}$ invalid. (This includes the last iteration, where *fast_grow* is not called.) This contributes total time equal to the desired bound. Also the total time for all executions of *fast_grow* (computed at the end of Section 4) is within the desired bound. □

6 Root-constrained Packing

Consider a packing of spanning arborescences where each arborescence is rooted at an arbitrary vertex v. Let $r(v)$ denote the total multiplicity of all arborescences rooted at v. In a *maximum root-constrained packing* we seek a packing of spanning arborescences with the largest total multiplicity subject to the constraint that each value $r(v)$ lies in a given interval $[\ell(v), u(v)]$. Clearly we can assume $u(v)$ is at most the out-degree of v. If $\ell(v) = u(v) = 0$ then v is never a root. This section gives an algorithm that computes the above maximum multiplicity, as well as each value $r(v)$ in the desired packing (for both fractional and integral packing). It is then a simple matter to find the desired root-constrained packing using our a-arborescence packing algorithm.

Our algorithm is based on minimax formulas due to Frank [F78] (which treats the case of upper bounds only) and Mao-Cheng [Mao] (which treats the case of lower bounds only). In fact our algorithm is similar to the proof of [F78]. Frank and Tardos [FT] showed that these two minimax formulas can be combined to give the following theorem: A digraph has k edge-disjoint spanning arborescences such that $\ell(v) \le r(v) \le u(v)$ for every vertex v if and only if these two conditions hold:

$$\rho_G(S) + u(S) \ge k \quad \text{for every nonempty set } S \subseteq V \tag{5a}$$

$$\sum\{\rho_G(S_i) : 1 \le i \le t\} \ge \ell(S_0) + k(t-1) \quad \text{for every partition of } V \text{ into sets } S_i,\ 0 \le i \le t, \tag{5b}$$
$$\text{where } t \ge 0 \text{ and } S_i \text{ is nonempty for } 1 \le i \le t$$

Here and throughout this section if S is a set of vertices then $u(S)$ denotes $\sum\{u(v) : v \in S\}$, and similarly for $\ell(S)$. Our development gives another proof of this result, for both the fractional and integral cases. We discuss the fractional case as the default, noting the minor modifications needed for the integral case.

Let us first verify that (5) is a necessary condition for existence of the packing. Suppose G has a root-constrained packing of multiplicity k. (5a) with $S = V$ states $u(V) \ge k$, true since every arborescence has a root. If $\emptyset \ne S \subseteq V$ then (5a) states $\rho_G(S) \ge k - u(S)$, true since at least $k - u(S)$ arborescences enter S.

For (5b), the case $t = 0$ states $k \ge \ell(V)$, true since every arborescence has one root. Suppose $t \ge 1$. Consider a spanning arborescence A of the packing, rooted at a vertex v. Any set S_i, $i \ge 1$ not containing v has $\rho_A(S_i) \ge 1$. Thus A contributes $t-1$ to the left-hand side of (5b) if $v \notin S_0$ and t if $v \in S_0$. (5b) follows.

Our algorithm decreases u values until u equals the correct multiplicity for each root. Inequalities (5) guide the decrease. We first describe the data structures of the algorithm.

The algorithm uses a digraph $\overline{G}$ which is formed from G by adding a new vertex a. Every vertex $v \in V$ has a new edge av of capacity $u(v)$. This capacity, which we always denote $u(v)$, gets decreased during the algorithm as noted above. We use two in-degree functions: ρ and $\overline{\rho}$ denote ρ_G and $\rho_{\overline{G}}$ respectively ($\overline{\rho}$ changes as $\overline{G}$ changes). Observe that $\lambda(\overline{G}) = u(V)$ if and only if G has a packing of total multiplicity $u(V)$, with each vertex v the root of exactly $u(v)$ arborescences. (As usual $\lambda(\overline{G})$ is defined using a as the root.) Hence we decrease u values until $\lambda(\overline{G}) = u(V)$.

The algorithm maintains a variable k that decreases to the maximum multiplicity of a root-constrained packing (if such exists). The algorithm returns the final values of k and u, where k is the maximum multiplicity of a root-constrained packing $\mathcal{A}$, and each vertex v is the root of arborescences of total multiplicity $u(v)$ in $\mathcal{A}$. (The packing itself can be found using our a-arborescence packing algorithm on graph $\overline{G}$.)

procedure *find_roots*; **begin**
$k \leftarrow \lambda(\overline{G})$;
while true do begin
  decrease values $u(v), v \in V$ as much as possible
    maintaining $u(v) \geq \ell(v)$ and $\lambda(\overline{G}) \geq k$;   /* "decrease step" */
  if $u(V) = k$ **then return** k, u;
  $\mathcal{T} \leftarrow \{S : S$ is a maximal $\overline{a}$-set with $\overline{\rho}(S) = k\}$; $t \leftarrow |\mathcal{T}|$;
  if $t \leq 1$ **then return** "no root-constrained packing exists";
  $S_0 \leftarrow V - \bigcup\{S : S \in \mathcal{T}\}$;
  $k \leftarrow (\sum\{\rho(S) : S \in \mathcal{T}\} - \ell(S_0))/(t-1)$;
  /* for integral packing set k to the floor of this value */
end; **end**;

Now we prove that *find_roots* returns the correct answer. The argument also proves that (5) is necessary and sufficient for root-constrained packing. (Specifically we show that the algorithm returns a value of k that has a root-constrained packing and (5) is violated for any larger k, or no packing exists because (5) is violated for any k.)

Lemma 6.1. *Procedure* find_roots *is correct.*

Proof. We begin by showing that k is nonincreasing. Note this implies the algorithm maintains the invariant $\lambda(\overline{G}) \geq k$. It also implies that the sets of $\mathcal{T}$ are pairwise disjoint. To prove the latter suppose $\mathcal{T}$ is nonempty, so $\lambda(\overline{G}) = k$. A set of $\mathcal{T}$ containing a vertex v is thus a $v\overline{a}$-set of minimum in-degree. The maximal such set is unique by submodularity of $\overline{\rho}$. Disjointness follows.

Let an iteration start with the value k and end with the value k'. We will show $k \ge k'$. The definitions imply $\sum\{\rho(S) : S \in \mathcal{T}\} = \sum\{\overline{\rho}(S) - u(S) : S \in \mathcal{T}\} = kt - u(V) + u(S_0)$. Since we can assume that $\lambda(\overline{G}) \ge k$ after the decrease step,

$$u(V) = \overline{\rho}(V) \ge k. \tag{6}$$

If $u(v) > \ell(v)$ then v is in a set of $\mathcal{T}$. Thus $u(S_0) = \ell(S_0)$. Combining these relations shows

$$\sum\{\rho(S) : S \in \mathcal{T}\} - \ell(S_0) = kt - u(V) \le k(t-1). \tag{7}$$

Dividing shows $k \ge k'$.

Now we can show that $k = k'$ implies the next iteration halts. If $k = k'$ then equality holds in (7) (even in the integral case). Thus equality holds in (6). Hence $u(V) = k = k'$ so the next iteration of *find_roots* halts (returning k, u).

This implies that *find_roots* eventually halts. In proof suppose the current iteration computes a family of sets $\mathcal{T}$ that was computed in some previous iteration. Hence the current iteration computes k' which is the same as the value computed in that previous iteration. Since k is nonincreasing, the current k also equals k'. Thus the next iteration halts.

It remains to show that *find_roots* returns the correct answer. Observe that any value of k computed by the algorithm satisfies some constraint of (5) with equality: The first value of k is defined so some set S satisfies (5a) with equality (using the given values u). Subsequent values of k are defined so there is a family $\mathcal{T}$ satisfying (5b) with equality. (In the integral case equality need not hold but k is the largest integer satisfying (5b).)

If *find_roots* returns with $u(V) = k$ the definition of $\overline{G}$ shows G has a root-constrained packing with total multiplicity k. Furthermore (5) is violated for any larger k, by the equality in (5) just observed. Hence *find_roots* is correct.

Next suppose *find_roots* returns with $t \le 1$. From the algorithm we see $u(V) \ne k$, i.e., $u(V) > k$. If $t = 0$ then $S_0 = V$. As noted previously we always have $u(S_0) = \ell(S_0)$. Thus $\ell(V) = u(V) > k$. This implies any value of k violates an inequality of (5): values at most the current value of k violate $k \ge \ell(V)$ (which is (5b) with $t = 0$) and larger values violate the equality in (5) for the current k. Hence no root-constrained packing exists.

The remaining possibility is $t = 1$, i.e., $\mathcal{T}$ consists of the single set S. In this case $u(V) > k = \overline{\rho}(S)$. Subtracting $u(S)$ shows $u(S_0) = \ell(S_0) > \rho(S)$. This violates (5b) (with $t = 1$) for any k. Hence no root-constrained packing exists. $\square$

We now prove the key fact for the efficiency of *find_roots*. Define $\mathcal{F}$ to be the family of all sets S such that at some point in the algorithm S is a maximal $\overline{a}$-set with $\overline{\rho}(S) = k$. A set enters $\mathcal{F}$ when the decrease step lowers u values. Clearly $\mathcal{F}$ contains any set in any family $\mathcal{T}$ computed by *find_roots*. But a set of $\mathcal{F}$ need not be in any family $\mathcal{T}$ (the set can enter $\mathcal{F}$ during a decrease step and get absorbed in a larger set of $\mathcal{F}$ during the same decrease step, never entering $\mathcal{T}$). This will be important when we implement the decrease step.

Lemma 6.2. *$\mathcal{F}$ is a laminar family. Procedure* find_roots *performs $O(n)$ iterations.*

Proof. Call a set $S \subseteq V$ *extreme* if any nonempty subset $T \subseteq S$ has $\overline{\rho}(T) \geq \overline{\rho}(S)$. (This is a slight abuse of terminology since the usual definition of extreme set requires strict inequality.) At any point in the algorithm every set $S \in \mathcal{F}$ is extreme. This follows because S is extreme when it enters $\mathcal{F}$, since at that time any set $T \subseteq S$ has $\overline{\rho}(T) \geq \lambda(\overline{G}) \geq k = \overline{\rho}(S)$. After that S remains extreme, since the decrease step preserves extreme sets ($\overline{\rho}(S)$ decreases at least as much as $\overline{\rho}(T)$ if $T \subseteq S$).

Now we show the first part of the lemma, $\mathcal{F}$ is always laminar. After k is initialized in *find_roots* $\mathcal{F}$ is a family of disjoint sets (the maximal $\overline{a}$-sets of in-degree $k = \lambda(\overline{G})$). Suppose a set S enters $\mathcal{F}$, i.e., S is a newly-created maximal $\overline{a}$-set with $\overline{\rho}(S) = k$. Take any set $T \in \mathcal{F}$ that is not disjoint from S. The submodular inequality $\overline{\rho}(S) + \overline{\rho}(T) \geq \overline{\rho}(S \cap T) + \overline{\rho}(S \cup T)$ with $\overline{\rho}(S \cap T) \geq \overline{\rho}(T)$ (since T is currently extreme) shows $k = \overline{\rho}(S) \geq \overline{\rho}(S \cup T)$. Hence $k = \overline{\rho}(S \cup T)$ and by maximality $S = S \cup T$, i.e., $T \subseteq S$ as desired.

We claim that when $\mathcal{T}$ is computed, it consists of the maximal sets $S \in \mathcal{F}$ that in addition have $\overline{\rho}(S) = k$. To prove this first observe that a set $S \in \mathcal{T}$ is maximal in $\mathcal{F}$. This follows since $S \subseteq T \in \mathcal{F}$ implies $\overline{\rho}(T) = k$ since T is extreme. Hence the definition of $\mathcal{T}$ implies $S = T$. Next observe that if S is a maximal set of $\mathcal{F}$ and $\overline{\rho}(S) = k$ then no larger set T has $\overline{\rho}(T) = k$ (since T would be in $\mathcal{F}$). Hence $S \in \mathcal{T}$.

This characterization of $\mathcal{T}$ shows that if a set S ever enters $\mathcal{T}$ and then leaves it (in a subsequent iteration) then S never reenters $\mathcal{T}$. To prove this note that S drops out of $\mathcal{T}$ when it is no longer maximal in $\mathcal{F}$ or it is maximal in $\mathcal{F}$ but $\overline{\rho}(S) > k$. In the first case clearly S never reenters $\mathcal{T}$. In the second case $u(S) = \ell(S)$ (by the decrease step). Hence the inequality $\overline{\rho}(S) > k$ holds throughout the rest of the algorithm, and S never reenters $\mathcal{T}$.

Now we can prove that *find_roots* has $O(n)$ iterations. Consider any iteration that is not the last or next-to-last. As shown in Lemma 6.1 the iteration changes the family $\mathcal{T}$. Thus some set S

has either entered or left $\mathcal{T}$. Each $S \in \mathcal{F}$ enters or leaves $\mathcal{T}$ at most once. Since $|\mathcal{F}| = O(n)$ (by laminarity) this occurs $O(n)$ times. $\square$

Our efficient implementation of *find_roots* uses the parametric s-set algorithm of [G95b]. This algorithm is also used in Section 7. We now summarize the parametric s-set problem, the algorithm of [G95b] that solves it, and all properties of this algorithm that are needed in Sections 6 and 7.

We use graphs that have a parametrized capacity function. This means that the given graph has a capacity function c_p, where the parameter p can take on any value in some interval $[d, \infty)$. The capacity of an edge $c_p(e)$ can change with p. For any value of p, the given graph with capacity function c_p is denoted G_p. The function ρ_p denotes in-degree in the graph G_p.

The *parametric s-set problem* is defined by a digraph G with distinguished vertex a, capacity function c_p parameterized by $p \geq d$, and "target function" $\tau(p)$. The problem is to find the smallest value $p \geq d$ such that $\lambda(G_p) \geq \tau(p)$ (or else determine that no such p exists).

We make several assumptions to ensure this problem is well-defined and tractable:

(i) Any $\overline{a}$-set S has an associated value p_S (possibly infinite) such that $\rho_p(S) \geq \tau(p)$ exactly when $p \geq p_S$.

(ii) Any edge e leaving vertex a has $c_p(e)$ a nondecreasing function of p, and any other edge e has $c_p(e)$ constant.

(iii) Given S we can compute p_S in time $O(m)$. Given p and an edge e we can compute $c_p(e)$ in $O(1)$ time.

In all other respects the capacities can take on arbitrary nonnegative real values.

Under these assumptions the algorithm of [G95b] solves the parametric s-set problem in time $O(nm \log(n^2/m))$ and space $O(m)$. The overall organization of the algorithm is to increase the parameter p, starting at $p = d$ and stopping when p reaches its desired value.

In more detail the algorithm initializes p to d. In general if the current value of p satisfies $\lambda(G_p) \geq \tau(p)$ then the algorithm returns this p as the desired value. In the opposite case the algorithm finds a set S with $\rho_p(S) < \tau(p)$, and it increases p to p_S. (This ensures $\rho_p(S) \geq \tau(p)$.) This process is repeated until the desired value of p is found.

An important property of this algorithm is that the values p_S and $c_p(e)$ can be supplied by an oracle. The oracle supplies the function c_d at initialization; each time p is increased to a new value the oracle supplies the corresponding value p_S and its associated capacity function c_p. As suggested by the above description, these values need not be fixed when the algorithm starts. Instead the new values of p_S and $c_p(e)$ can be chosen by the oracle when it is called. The only constraint is that the values satisfy conditions (i) – (iii) above.

Now we return to the task of implementing *find_roots*. We begin with the decrease step. We implement this step in a number of iterations. The net result of each iteration is that values $u(v)$ get decreased, maintaining $u(v) \geq \ell(v)$ and $\lambda(\overline{G}) \geq k$, and a new set S with $\overline{\rho}(S) = k$ gets created. (Since the decrease in u need not be maximal, several iterations may be needed.) The parametric s-set algorithm is used to accomplish this. (During the execution of the parametric s-set algorithm values $u(v)$ get increased, but this does not invalidate the above description of an iteration.) The details of an iteration of the decrease step are as follows.

To start an iteration of the decrease step, set $u_0(v)$ to $u(v)$ for each vertex v. Then compute the family $\mathcal{T}$ using the Hao-Orlin algorithm. For each $S \in \mathcal{T}$ decrease values $u(v), v \in S$ as much as possible maintaining $u(v) \geq \ell(v)$ and $\overline{\rho}(S) \geq k$. Subject to these constraints, the choice of vertices to decrease is arbitrary. If now $\lambda(\overline{G}) \geq k$ the decrease step is complete (test this condition using the Hao-Orlin algorithm).

On the other hand if $\lambda(\overline{G}) < k$ we will increase $u(v)$ values to restore $\lambda(\overline{G}) = k$. We do this by solving a parametric s-set problem. The problem is on graph $\overline{G}$ with distinguished vertex a. The target function $\tau(p)$ equals the constant k. It remains to describe the parametric capacity function c_p and the values p_S. We do this by specifying an oracle that computes these values when they are needed.

The parametric s-set algorithm begins with all edges of $\overline{G}$ at their current capacity (in this iteration of the decrease step). Suppose the parametric s-set algorithm finds a set S with current in-degree $\overline{\rho}(S)$ less than the target k. The oracle calculates a new capacity function achieving $\overline{\rho}(S) = k$ using this rule: Increase values $u(v), v \in S$, maintaining $u(v) \leq u_0(v)$, until $\overline{\rho}(S) = k$. Note that the rule eventually achieves the condition $\overline{\rho}(S) = k$ since the values $u_0(v)$ make $\lambda(\overline{G}) \geq k$. The new capacity function corresponds to the new parameter value p_S. It is clear that conditions (i) – (iii) of the parametric s-set problem are satisfied by this oracle.

The parametric s-set algorithm repeats this process as necessary. It eventually returns with each new value $u(v) \leq u_0(v)$, $\lambda(\overline{G}) = k$ and some set S having $\overline{\rho}(S) = k$ but $\overline{\rho}(S) < k$ immediately after u values were decreased in the current iteration of the decrease step. This completes the iteration of the decrease step.

The implementation of the rest of *find_roots* is straightforward. As above use the Hao-Orlin algorithm to compute $\mathcal{T}$.

Theorem 6.1. *Procedure* find_roots *computes the maximum total multiplicity of a fractional or integral root-constrained packing, along with the multiplicity of each root vertex, in time* $O(n^2 m \log(n^2/m))$*. A corresponding fractional packing is found in the time bound of Corollary 2.1, using at most* $m+n-1$ *distinct arborescences. For integral packing the time is the same as Corollary 5.1, using at most* $m+2n-3$ *distinct arborescences.*

Proof. First consider the efficiency of *find_roots*. Each execution of the parametric s-set algorithm uses time $O(nm \log(n^2/m)$ [G95b] and enlarges the family $\mathcal{F}$. Since $|\mathcal{F}| = O(n)$ (Lemma 6.2) the parametric s-set algorithm is executed $O(n)$ times, as desired. The remaining time amounts to $O(1)$ executions of the Hao-Orlin algorithm per iteration of *find_roots*. There are $O(n)$ iterations (Lemma 6.2). The desired time bound follows.

We find the desired root-constrained packing by executing our a-arborescence packing algorithm on $\overline{G}$. The time is given by Corollary 2.1 or 5.1.

We complete the proof by bounding the number of distinct arborescences in the packing. Graph $\overline{G}$ has $n+1$ vertices and $m+n$ edges. Hence Theorems 2.1 and 5.1 give $m+n$ and $m+2n-1$ distinct arborescences respectively. We decrease these bounds by one and two respectively, by initializing the laminar family $\mathcal{F}$ of the packing algorithms to contain one set, V. Note that *find_roots* returns with $u(V) = k = \lambda(\overline{G})$. Thus V is a set of in-degree $\lambda(\overline{G})$ in $\overline{G}$. Now the extensions noted after our two theorems show the bounds on distinct arborescences decrease appropriately. □

Note that if the given root constraints allow only r vertices to be roots, the bounds on the number of distinct arborescences become $m+r-1$ for fractional packing and $m+n+r-3$ for integral packing. Setting $r=1$ returns us to the bounds of Theorems 2.1 and 5.1.

7 Applications of Packing

This section applies our packing algorithms to related problems, specifically, covering digraphs by branchings, and packing and covering in undirected graphs.

We first consider the problem of covering a digraph by branchings. To define this problem precisely let G be a digraph with capacity function c. We use terminology analogous to packing. The problem is to find a family of branchings, each with an associated multiplicity, such that each edge e occurs in branchings of total multiplicity equal to $c(e)$ and the family has the minimum possible total multiplicity. This minimum multiplicity is the *branching cover number*, and the corresponding family of branchings is a *minimum branching cover*. We consider both fractional and integral covering; correspondingly we have fractional and integral branching cover numbers, etc.

We characterize the branching cover number as follows. Form the digraph $\overline{G}$ from G by adding a new vertex a and an edge av for every vertex $v \in V$. Let $d = \max\{\rho_G(v) : v \in V\}$. For $p \geq d$ define a parameterized capacity function c_p on $\overline{G}$ as follows: If edge e leaves vertex a, say $e = av$, then $c_p(e) = p - \rho_G(v)$; otherwise $c_p(e) = c(e)$. For any p, $\lambda(\overline{G}_p) \leq p$ since each vertex $\neq a$ has in-degree p in $\overline{G}_p$. (As usual $\lambda(\overline{G}_p)$ is defined using a as the root vertex.)

Lemma 7.1. *The (integral or fractional) branching cover number of G is the smallest (integral or fractional) value p such that $\lambda(\overline{G}_p) = p$.*

Proof. Let b be the (integral or fractional) branching cover number of G. Let p be the smallest (integral or fractional) value as specified in the lemma. We wish to show $b = p$, which we do by proving $b \geq p$ and $b \leq p$.

We first show $b \geq p$. We convert a cover of G having multiplicity b to a packing of a-arboresences in $\overline{G}_b$ having total multiplicity b, as follows: Add edge av to each branching that has v as a root, for each $v \in V$. This gives a packing of arborescences, in which each edge av has multiplicity $b - \rho_G(v) = c_b(av)$. Hence this is a valid packing in $\overline{G}_b$. (Note this argument also ensures the existence of the quantity p.)

Next we show $p \geq b$. We convert a packing $\mathcal{A}$ of a-arborescences in $\overline{G}_p$ having total multiplicity p to a cover of G having total multiplicity p, as follows: Delete all occurrences of edges av from $\mathcal{A}$. Note that $\mathcal{A}$ contains every edge of $\overline{G}_p$ with multiplicity equal to its capacity (since as already noted each vertex $\neq a$ has in-degree p). Thus the resulting family of branchings is a cover of G. $\square$

We calculate the quantity of the lemma using the parametric s-set algorithm as follows. The parametric s-set problem uses graph $\overline{G}$ with distinguished vertex a, capacity function c_p as specified before Lemma 7.1 and target function $\tau(p) \equiv p$. The parameter p lies in $[d, \infty)$. Lemma 7.1 shows the branching cover number equals the answer to the parametric s-set problem.

Let us check that the assumptions of the parametric s-set problem hold. For (i), the values p_S exist by the following formulas: Any set S having $\rho_d(S) \geq d$ for capacity function c_d has $p_S = d$ (for example this includes any singleton set S). Any other set S has $p_S = d + (d - \rho_d(S))/(|S| - 1)$. (For fractional covering this formula follows easily from the definitions. For integral covering use the ceiling of this value.) Assumptions (ii) – (iii) follow.

Theorem 7.1. *The (integral or fractional) branching cover number of a digraph can be found in time $O(nm \log(n^2/m))$. A minimum branching cover containing $m + O(n)$ distinct branchings can be found in the time bound of Corollary 2.1 for fractional covers and Corollary 5.1 for integral covers.*

Proof. We have shown how to calculate the branching covering number b in the desired time. A corresponding minimum cover can be found by executing our arborescence packing algorithm on $\overline{G}_b$. This graph has $n+1$ vertices and $m+n$ edges. Now the bounds on time and number of distinct branchings in the cover follow from Theorems 2.1, 5.1 and their Corollaries. $\square$

We turn to problems on undirected graphs. Let G be an undirected graph with capacity function c. The *spanning tree packing number* is the largest possible total multiplicity of a family of spanning trees, each with its own multiplicity, such that each edge e occurs in trees of total multiplicity at most $c(e)$. The corresponding family of trees is a *maximum spanning tree packing.* We also discuss the problem of covering the edges by forests. The *forest cover number* and a *minimum forest cover* are defined analogously. We consider the fractional and integral versions of the packing and covering problems.

Our approach is based on a reduction presented in [G95b]. We summarize the resulting procedure here and refer the reader to that paper for more details of the reduction.

The main step of the reduction is as follows. Consider a graph H having a capacity function, that can be partitioned into k forests. (For fractional packing and covering introduce multiplicities in the obvious way.) [G95b] shows how to construct a digraph D from H for which a maximum packing of a-arborescences has total multiplicity k, and furthermore such a packing is easily converted to a partition of H into k forests. (We sketch the construction below.)

Our covering algorithm works as follows. First calculate the (integral or fractional) forest cover number k, using the algorithm given in [G95b] for this task. Form the corresponding digraph D (using the graph G and the value k). Use the algorithm of Section 2 or 5 to find a maximum packing of a-arborescences in D. Finally convert this packing to the desired minimum forest cover.

Our packing algorithm works as follows. As before start by calculating the (integral or fractional) spanning tree packing number k, using the algorithm of [G95b]. The next step is to find a subgraph H of G that is the union of k disjoint spanning trees, i.e., we wish to calculate the capacities of the edges of G in a maximum spanning tree packing. A base b of the graphic polymatroid $P(kr)$ (where r denotes the graphic matroid rank function, see [G95b] for definitions) gives these capacities, i.e., $b(e)$ is the desired capacity of edge e of G. Use the algorithm of [G95b] to find a base of $P(kr)$. Now proceed as in our covering algorithm, using these capacities to form D, finding a maximum packing of a-arborescences, and converting it to a maximum spanning tree packing.

We next sketch how the digraph D is constructed from H. We give enough detail to calculate the size of D. D is called the k-equivalent graph in [G95b].

To form D replace each edge vw of H by directed edges vw and wv, and add a new vertex a plus an edge av for every $v \in V$. The capacity function on D is derived from a maximum flow computation (on another graph, called G^* in [G95b]). Corollary 3.1 of [G95b] proves that in D, a maximum packing of a-arborescences has total multiplicity k; furthermore deleting a from these arborescences and ignoring edge directions gives the set of k forests that partition the edges of H.

We make two remarks on this construction. First for integral covering and packing on G, D has integral capacities, and we use an integral arborescence packing on D. Second, the algorithms of [G95b] used in our covering algorithm take time $O(nm \log (n^2/m))$, and those used in our packing algorithm take time $O(n^2 m \log (n^2/m))$.

Theorem 7.2. *Consider a capacitated undirected graph. A minimum forest cover containing $O(m)$ distinct forests can be found in the time bound of Corollary 2.1 for fractional covers and Corollary 5.1 for integral covers. A maximum spanning tree packing containing $O(m)$ distinct trees can be found in the time bound of Corollary 2.1 for fractional packing and Corollary 5.1 for integral packing.*

Proof. Our description of D shows it has $n+1$ vertices and at most $2m+n$ edges. Now apply Theorems 2.1, 5.1 and their Corollaries. Note there are at most $2m+n$ $(2m+2n-1)$ distinct forests or spanning trees in our fractional (integral) cover or packing. Note also that the time for the algorithms of [G95b] is dominated by the time bounds of the theorem. □

The bounds on number of distinct forests and trees can be reduced to $m + O(n)$. This results from the fact that the capacity function on D is derived from a maximum flow computation, and we can choose the maximum flow to limit the number of edges in D. The details depend on the exact definition of D; see [G95b].

We also note that when the given undirected graph has an edge cost function, a maximum spanning tree packing having smallest possible cost can be found in the bounds of Theorem 7.2.

This holds because the algorithm of [G95b] finds the subgraph H consisting of the edges in the desired packing, in the same time bound.

Acknowledgments The first author wishes to pay tribute to Gene Lawler, who provided help and inspiration for many years.

References

[B] F. Barahona: Packing spanning trees. Mathematics of Operations Research **20**,1, 1995, pp. 104–115

[E65] J. Edmonds: Minimum partition of a matroid into independent subsets. Journal of Research of the National Bureau of Standards **69B**, 1965, pp. 67–72

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

[F78] A. Frank: On disjoint trees and arborescences. In: Algebraic Methods in Graph Theory, L. Lovász and V.T. Sós, Eds., Colloquia Mathematica Societatis János Bolyai **18**, North-Holland, New York, 1978, pp. 159–169

[F79] A. Frank: Kernel systems of directed graphs. Acta Scientiarum Mathematicarum (Szeged) **41**, 1979, pp. 63–76

[FT] A. Frank, É. Tardos: Generalized polymatroids and submodular flows. Mathematical Programming, 42, 1988, pp. 489–563

[G95a] H.N. Gabow: A matroid approach to finding edge connectivity and packing arborescences. Journal of Computer and System Sciences **50**, 2, 1995, pp. 259–273

[G95b] H.N. Gabow: Algorithms for graphic polymatroids and parametric s-sets. Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 88–97; Journal of Algorithms, to appear.

[GLS] M. Grötschel, L. Lovász, A. Schrijver: Geometric Algorithms and Combinatorial Optimization, Springer-Verlag, New York, 1988

[GW] H.N. Gabow, H.H. Westermann: Forests, frames and games: Algorithms for matroid sums and applications. Algorithmica **7**, 1992, pp. 465–497

[HO] J. Hao, J.B. Orlin: A faster algorithm for finding the minimum cut in a directed graph. Journal of Algorithms **17**, 3, 1994, pp. 424–446

[I] H. Imai: Network-flow algorithms for lower-truncated transversal polymatroids. Journal of the Operations Research Society of Japan **26**, 3, 1983, pp. 186–210

[KRT] V. King, S. Rao, R. Tarjan: A faster deterministic maximum flow algorithm. Journal of Algorithms **17**, 3, 1994, pp. 447–474

[L] L. Lovász: On two minimax theorems in graph theory. Journal of Combinatorial Theory, B, **21**, 1976, pp. 96–103

[Mad] W. Mader: On n-edge-connected digraphs. Annals of Discrete Mathematics **17**, 1983, pp. 439–441

[Mao] C. Mao-cheng: Arc-disjoint arborescences of digraphs. Journal of Graph Theory **7**, 1983, pp. 235–240

[P] P.A. Pevzner: Branching packing in weighted graphs. American Mathematical Society Translations **158**, 2, 1994, pp. 185–200

[PW] M.W. Padberg, L.A. Wolsey: Fractional covers for forests and matchings. Mathematical Programming **29**, 1984, pp. 1–14

[R] T. Radzik: Newton's method for fractional combinatorial optimization. Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, 1992, pp. 659–669

[RT] J. Roskind, R.E. Tarjan: A note on finding minimum-cost edge-disjoint spanning trees. Mathematics of Operations Research **10**, 4, 1985, pp. 701–708

[S] Y. Shiloach: Edge-disjoint branchings in directed multigraphs. Information Processing Letters **8**, 2, 1979, pp. 24–27

[Ta] R.E. Tarjan: A good algorithm for edge-disjoint branching. Information Processing Letters **3**, 2, 1974, pp. 51–53

[Tro] L.E. Trotter, Jr.: Discrete packing and covering. In: Combinatorial Optimization: Annotated Bibliographies, M. O'hEigeartaigh, J.K. Lenstra and A.H.G. Rinnooy Kan, Eds., John Wiley, New York, 1985, pp. 21–31

[Tru91] V.A. Trubin: Strength and reinforcement of a network and tree packing. Kibernetika **2**, 1991, pp. 67–75

[Tru93] V.A. Trubin: Strength of a graph and packing of trees and branchings. Kibernetika i Sistemnyi Analiz **3**, 1993, pp. 94–99

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