Augmenting Undirected Edge Connectivity in $\tilde{O}(n^2)$ Time

András A. Benczúr* David R. Karger†

June 7, 2000

Abstract

We give improved randomized (Monte Carlo) algorithms for undirected edge splitting and edge connectivity augmentation problems. Our algorithms run in time $\tilde{O}(n^2)$ on n-vertex graphs, making them an $\tilde{\Omega}(m/n)$ factor faster than the best known deterministic ones on m-edge graphs.

1 Introduction

Edge augmentation and edge splitting problems are special network design problems [GGP+94, GW97] in which a graph must be modified achieve specified edge connectivity properties while minimizing the total weight of edges used.

In the *edge augmentation problem*, one wants to add an (integer weighted) edge set of minimum total weight such that the input graph becomes k-edge-connected (by weight). The value k is called the *target connectivity*.

Edge splitting is a tool widely used to solve connectivity-related problems. The goal is to cut a given vertex s out of the graph without decreasing the connectivity of the rest of the graph. We do so by repeated shortcutting, *splitting* pairs of edges (u, s) and (v, s) to a new edge (u, v). By arranging for the graph created by splitting-off to preserve the connectivity properties of the original graph, we can use the construction in inductive proofs of various connectivity theorems [Fra93]. These inductive proofs can be turned into efficient recursive algorithms via splitting algorithms based on flows [Gab94] or the Nagamochi–Ibaraki algorithm [NI96].

Unlike generic network design problems which are NP-complete, edge augmentation is tractable because each edge's cost is equal to its weight. This was first shown by Watanabe and Nakamura [WN87] for the unweighted case; the first strongly polynomial algorithm was given by Frank [Fra92]. Edge splitting can also be solved in polynomial time. Progress has been made on improved time bounds for both problems, but prior to this paper the best known bounds [NI96] were $\tilde{O}(nm)$.[1]

1.1 Our results

In this paper, we present new, faster algorithms for the edge augmentation and splitting off problems. We give randomized algorithms that solve both of these problems (with high probability) in $\tilde{O}(n^2)$ time. With minor changes, our algorithms also solve (or show unsolvable) the *degree constrained* augmentation problem, a common generalization of both problems in which there is an upper bound on the total weight of edges that may be added incident to each vertex.

---

*Computer and Automation Institute, Hungarian Academy of Sciences. Also supported at the Department of Operations Research, Eötvös University, Budapest. Supported from grants OTKA T-30132 and T-29772, FKFP 0206/1997 and AKP 98-19.
email: `benczur@@cs.elte.hu`

†MIT Laboratory for Computer Science, Cambridge, MA 02139. Supported by NSF contract CCR-9624239 and an Alfred P. Sloane Foundation Fellowship.
email: `karger@@lcs.mit.edu`.
URL: `http://theory.lcs.mit.edu/~karger`

[1] The notation $\tilde{O}(f)$ denotes $O(f \operatorname{polylog} n)$.

Our algorithms are Monte Carlo: although the are guaranteed to run fast, they have a small chance of giving the wrong answer. However, they give the right answer *with high probability*—that is, with probability $1 - 1/n^d$ for some fixed $d > 1$ on problems of size n. (In fact, d can be made arbitrarily large without affecting the asymptotic running time by repeating the algorithm a fixed number of times and taking the best answer.) At present, Monte Carlo algorithms are the only way even to *test* whether a graph is k-connected in $o(nm)$ time [KS96, Kar96].

Like many augmentation algorithms [WN87, NGM90, Gab91a, Ben94], ours manipulates the *extreme sets* of a graph. A set X is *extreme* if its *degree* (outgoing edge weight) $d(X) < d(W)$ for all proper subsets $W \subset X$. It thus has the "dominant" demand for additional outgoing edges in the augmentation or splitting process. Our improved running time arises from two contributions.

- We give a faster algorithm for finding extreme sets.
- We give a faster algorithm for using the extreme sets to perform edge augmentation and splitting.

The extreme set algorithm is based on the Recursive Contraction Algorithm of Karger and Stein [KS96] for finding minimum cuts and some degree-testing algorithms used by Karger [Kar96]. The algorithm for augmentation and splitting given the extreme sets is based on work of Benczur [Ben94], and reflects the first time that extreme sets are used directly for splitting.

Our algorithms run in $O(n^2 \log^5 n)$ time, which reflects the time needed to find the extreme set system. Given this system our algorithm requires only an $O(n^2)$-time deterministic computation and an $O(n^2 \log^3 n)$-time randomized minimum cut computation to perform the augmentation.

1.2 Related Work

Algorithms for solving the edge augmentation problem traditionally take two different approaches, one using edge splitting [CS89, Fra92, Gab94, NI96] and another increasing connectivity one unit at a time [WN87, NGM90, Gab91a]. The idea of applying edge splitting in connectivity augmentation was first explored by Cai and Sun [CS89]. Frank [Fra92] used this approach in the first strongly polynomial augmentation algorithm, which solves the weighted case in $O(n^5)$ time. Gabow [Gab94] improved Frank's running time to $\tilde{O}(n^2 m)$. In these algorithms, edge splitting was the computational bottleneck. Faster edge splitting algorithms were later given; one running in $\tilde{O}(n^3)$ time, followed by one running in $\tilde{O}(nm)$ time [NI96]. But $\tilde{O}(mn)$ remained the best bound achieved for splitting-based edge augmentation.

Edge augmentation is slightly easier than splitting—augmentation is relatively easy given a splitting algorithm but not vice versa. Several papers [WN87, NGM90, Gab91a] devise augmentation algorithms that do not use splitting. These algoritms are not strongly polynomial and are therefore efficient for small connectivity values and unweighted graphs only. The best known such algorithm, with runtime $\tilde{O}(nk^2)$ for target connectivity k, is due to Gabow [Gab91b]. Although it uses splitting, our edge augmentation algorithm is based on ideas from another non-splitting augmentation algorithm [Ben94]. That Monte Carlo algorithm, in contrast to earlier algorithms using the same approach, has a strongly polynomial $\tilde{O}(n^3)$ running time.

Algorithms for finding extreme sets have been less well studied, perhaps because our augmentation algorithm is the first where finding extreme sets is the computational bottleneck. Gabow [Gab91a] describes an efficient algorithm for unweighted graphs. However the only previously known weighted-graph algorithm, that of Naor et al. [NGM90], builds a Gomory–Hu tree [GH61] and thus runs in $\tilde{O}(n^2 m)$ time. Ours runs in $\tilde{O}(n^2)$ time.

2 Problems and Definitions

In this section, we formally define the problems that we will be solving and show how they are related to one another. We also provide additional definitions we will need later. Given a vertex set U, we say that edge e *crosses* U if exactly one endpoint of e is in U. For a vertex set U, let the *degree* $d(U)$ be the weight of edges of G crossing U. More generally, given any set of edges E, let $d_E(U)$ be the weight of edges of E crossing U.

2.1 The Basic Problems

Definition 2.1 (Edge augmentation). Let $G = (V, E)$ be an undirected graph with connectivity c. Given a *target connectivity* k the *edge augmentation problem* is to find an edge set of minimum total weight whose addition to G increases its connectivity to k.

For reasons that will shortly become clear, we also refer to this problem as *internal* edge augmentation because all edges are added inside the graph. Without loss of generality, we assume the target $k > 1$. When $k = 1$, we simply want to add the minimum total edge weight necessary to connect the graph, which is trivial.

Definition 2.2 (Edge splitting). Let G be a weighted undirected graph with a distinguished vertex s. Given a pair of edges (u, s) and (v, s) incident on s, we say that we *split edges* (u, s) *and* (v, s) *by weight* w if we decrease the weights of these edges by w and increase that of edge (u, v) by w. We say that we *split* s *off* if, by a sequence of splits, we isolate s from the graph.

In this paper we concentrate on the following theorem concerning edge splitting.

Theorem 2.3 (Lovász [Lov93, problem 6.53]). *Let G be a weighted undirected graph with a distinguished vertex s. Let k be the minimum value among the cuts of G other than $(\{s\}, V - s)$. Then it is possible to split s off, such that the resulting graph has connectivity at least k. Furthermore if all edge weights are integers, $k \geq 2$, and the total weight of edges incident to s is even, then it is possible to give a sequence of splits by integer weights.*

Edge splitting is possible under stronger requirements as well [Mad78]; however that stronger edge splitting task is algorithmically much harder [Gab94].

2.2 Degree-constrained augmentation: a common generalization

Although not explicit in the literature, it is known that the edge splitting and edge augmentation problems share a common generalization to a *degree-constrained* augmentation problem defined next. In fact, the problem is a slight generalization of the edge augmentation problem but is essentially equivalent to the edge splitting problem.

Definition 2.4. Given a non-negative integer weight $w_0(v)$ on the vertex set V and a *target connectivity* k, the *degree-constrained edge augmentation problem* is to find an edge set E' of minimum total weight whose addition to G increases its connectivity to k and satisfies $d_{E'}(v) \leq w_0(v)$ for all $v \in V$.

(Recall that $d_{E'}(U)$ is the weight of edges of E' with exactly one endpoint in U.) Here and throughout the paper we define $w(U) = \sum_{v \in U} w(v)$ for a weight function w on the vertex set V (note that this notation does not apply to the degree function $d(U)$). The above degree-constrained augmentation problem is solvable only if $d(U) + w_0(U) \geq k$ for all $U \subset V$, so throughout the paper we assume this inequality holds.

The edge augmentation problem reduces to the degree constrained one by setting $w_0(v) = \infty$ for all vertices. The edge splitting problem can be reduced similarly by setting $w_0(v) = d(s, v)$; after solving the degree-constrained augmentation problem, any unused weight can be paired to edges arbitrarily to obtain an edge splitting at s.

We will show below that our algorithms can be modified to solve (or show unsolvable) the degree constrained version of edge augmentation with the same $\tilde{O}(n^2)$ time bound.

2.3 External Augmentation: An intermediate step

Our algorithm (and others' [Fra92]) for solving internal edge augmentation divides naturally into two stages. In the first, we aim to solve an easier "external" version of the edge augmentation problem, in which all the new edges are incident on a special new vertex s. In the second stage, we split s off to transform this external solution into an internal one.

Definition 2.5 (External edge augmentation). Let $G = (V, E)$ be an undirected graph with connectivity c. Given a a *target connectivity* $k = c + \tau$, the *external edge augmentation problem* is to add an edge set of minimum total weight connecting vertices of G to some new external vertex s, such that every cut in G other than $(\{s\}, V - s)$ has value at least k.

In other words, we want the connectivity of the new graph $G \cup \{s\}$ to be k, with the possible exception of the trivial cut around s. There is an obvious degree-constrained variant of the problem.

We can also formulate the external augmentation problem as follows: given a graph G, give non-negative integer weights w to the vertices, minimizing the total weight $w(V)$, such that the weight $w(U)$ of any set U satisfies $d(U) + w(U) \geq k$. The weight of vertex v represents the weight of edges between v and s.

The following well-known lemma shows the close connection between the internal and external augmentation problems.

Lemma 2.6 (cf. [Fra92]). *Let w be the minimum weight of edges needed to solve the internal edge augmentation problem, and let w' be the minimum weight of edges needed to solve the external edge augmentation problem. Then $w = \lceil w'/2 \rceil$. The same holds for the degree-constrained problem variants.*

Proof. We show that one can go back and forth between external and internal augmentation solutions, doubling or halving the total weight of edges used.

Suppose we have a set of edges that solves the internal augmentation problem. To solve the external augmentation problem to a vertex s, replace each added edge (u, v) with two edges (s, u) and (s, v). This doubles the weight used but preserves the desired connectivity.

Suppose we have a set of edges that solves the external augmentation problem to a vertex s. If the set is odd, add one edge from s to an arbitrary vertex. According to Lemma 2.3 we can split off s, halving the weight of edges used while preserving the desired connectivity.

For the degree constrained versions, simply note that our two-way transformation above does not change the degree of any vertex other than s. □

Our algorithms will use this internal-external transformation, solving the external augmentation problem and then splitting it off; the above lemma shows this solution is optimal.

2.4 Extreme Sets

Both the external augmentation and splitting off stages of our algorithm rely on a concept of *extreme sets* developed by Watanabe and Nakamura [WN87]. Suppose we wish to find a weight assignment w that solves the external augmentation problem. Consider a set X with $d(X) < k$. Provided X has a subset U with $d(X) \geq d(U)$, the augmentation condition imposed on X is automatically satisfied if it is satisfied for U. This motivates the following definition.

Definition 2.7. A set X is d-*extreme* if $d(X) = d$ and no proper subset $U \subset X$ has $d(U) \leq d$. For convenience, we also declare the vertex set V to be extreme.

Extreme sets have a special structure that we will exploit frequently. To describe it, we need the following definitions. We say that two sets *meet* if their intersection is nonempty. They *nest* if one is contained in the other. Two sets C and D are called *overlapping* if $C \cap D$, $C - D$ and $D - C$ are all nonempty—that is, they meet but do not nest. A set system is *laminar* if it contains no overlapping pair—in other words, if any two sets that meet also nest.

Lemma 2.8 (cf. [WN87]). *No two extreme sets overlap, so the extreme sets form a laminar system.*

Proof. The proof of the laminar property uses the *submodularity* of the function d over vertex subsets. We say that a function d on subsets of V (the vertex set) is *submodular* if, for all pairs of sets X and Y,

$$d(X) + d(Y) \geq d(X \cap Y) + d(X \cup Y) .$$

The cut value function d is submodular. For undirected graphs, the cut value function d is also symmetric, i.e. $d(X) = d(V - X)$. Hence the submodular inequality also holds in a different form:

$$d(X) + d(Y) \geq d(X - Y) + d(Y - X) .$$

In particular, it follows that either $d(X - Y) \leq d(X)$ or $d(Y - X) \leq d(Y)$.

Now suppose that X and Y are extreme but overlapping. Then by submodularity, without loss of generality $d(X - Y) \leq d(X)$. If X and Y overlap, then $X - Y$ is a nonempty proper subset of X; since $d(X - Y) \leq d(X)$ it must be that X is not extreme. $\square$

Sets of the laminar extreme set system $\mathcal{F}$ can be viewed as nodes of a tree where the children of $Y \in \mathcal{F}$ are the maximal extreme subsets of Y—that is, $X \in \mathcal{F}$ is a child of $Y \in \mathcal{F}$ if $X \subset Y$ but there is no $Z \in \mathcal{F}$ with $X \subset Z \subset Y$. We will refer to this tree as the *extreme set tree*. We call the sets in the tree *nodes* in order to distinguish them from the vertices of G. Note that children must have degree greater than their parents' (else they would make the parent non-extreme). Since all individual vertices are vacuously extreme, the individual vertices form the n leaves of the extreme set tree. Since the children of a set partition the set, no set of the tree has less than 2 children. Thus the tree has $O(n)$ nodes and gives a size-$O(n)$ representation for the extreme sets, even though explicitly listing them can take $\Theta(n^2)$ space.

2.5 Overview of Solution

With these definitions completed, we can outline the course of our algorithm for edge augmentation. In a first step, we find the extreme set tree for the input graph. This step is randomized and takes $O(n^2 \log^5 n)$ time. Using the extreme set tree, we give a trivial $O(n)$-time greedy algorithm for the external augmentation problem. Once the external augmentation problem is solved, we use a splitting off algorithm to turn the solution into an internal augmentation solution. Splitting off again makes use of the extreme set tree, and involves an $O(n^2)$ time deterministic algorithm plus a randomized minimum cut/cactus computation (which requires $O(n^2 \log^3 n)$ time). Finding the extreme sets is the bottleneck in our solution, and along with minimum cut computations is the only part that uses randomization.

We use the same approach to solve the degree constrained version of our problem, solving a degree constrained external problem and then using splitting off (which preserves vertex degrees) to transform the external into an internal version.

We begin in Section 3 by presenting the simplest part of our approach, the greedy external augmentation algorithm. We then describe the splitting off algorithm in Section 4, and finally the extreme sets algorithm in Section 5.

3 An External Augmentation Algorithm

Naor, Gusfield and Martel [NGM90] use the extreme set tree to greedily solve the external augmentation problem. We outline their approach here. As above, we formulate the problem as assigning weights w to vertices so that $w(U) + d(U) \geq k$ for all U. Define the *k-demand* of a set U to be $\text{dem}(U, k) = \max\{0, k - d(U)\}$; this is the minimum weight $w(U)$ needed for set U in order to satisfy the external augmentation objective (the k-demand of V is 0). We omit k when it is clear from context.

3.1 Focusing on Extreme Sets

The following two lemmas formalize the argument that extreme sets are the only important sets for the augmentation problem.

Lemma 3.1. *Any vertex set U contains an extreme set $X \subseteq U$ with $d(X) \leq d(U)$.*

Proof. Since a set U is either extreme or has a subset X with $d(X) \leq d(U)$, there is no minimal counterexample to the lemma. $\square$

Lemma 3.2. *Let a weight function w on the vertex set be such that $w(X) \geq \text{dem}(X)$ for every $X \in \mathcal{F}$, where $\mathcal{F}$ is the system of extreme sets in G. Then w satisfies the external augmentation problem.*

Proof. For every $U \subset V$ choose an extreme set $X \in \mathcal{F}$ such that $X \subseteq U$ and $d(X) \leq d(U)$. Then

$$w(U) \geq w(X) \geq \text{dem}(X) \geq \text{dem}(U)$$

which proves the claim. $\square$

Thus, to solve the external augmentation problem, we need only assign weights that satisfy the demands of the extreme sets. Each extreme set therefore determines a lower bound on the total weight of the optimal solution.

3.2 Recursive demands and a compact min-max formula

To solve the external augmentation problem it helps to work with a somewhat stronger lower bound.

Definition 3.3. In the laminar extreme set tree $\mathcal{F}$, let the *recursive demand* $\text{rdem}(X,k)$ be $\text{dem}(X,k)$ for all leaf nodes $\{v\} \in \mathcal{F}$. Given $\text{rdem}(X_i,k)$ for all children $X_i \in \mathcal{F}$ of X, let

$$\text{rdem}(X,k) = \max\{\text{dem}(X,k), \sum_i \text{rdem}(X_i,k)\} .$$

Lemma 3.4. *Any external augmentation to value k uses weight at least* $\text{rdem}(V,k)$.

Proof. Let $w(\cdot)$ be a solution (assignment of weights) for the external augmentation problem. We prove by induction on the laminar family that for every extreme set X, $w(X) \geq \text{rdem}(X,k)$ for every extreme set X. The claim follows by taking $X = V$. The leaf nodes of the tree (corresponding to vertices of G) clearly must have $w(v) \geq \text{dem}(v) = \text{rdem}(v)$. For any non-leaf node X with children X_i, we must have $w(X) \geq \text{dem}(X)$. We must also have

$$\begin{aligned} w(X) &\geq \sum w(X_i) \\ &\geq \sum \text{rdem}(X_i) \end{aligned}$$

where the second step is by induction on the X_i. Thus $w(X) \geq \max(\text{dem}(X), \sum \text{rdem}(X_i)) = \text{rdem}(X)$. □

3.3 The Algorithm

We now give an algorithm that solves the external augmentation problem using total weight $\text{rdem}(V,k)$. By Lemma 3.4, such an assignment is optimal.

Algorithm 3.5. Initially let $w(v) = 0$ for all vertices $v \in V$. Process the laminar set tree in a postorder traversal, so all children are processed before their parent. Consider all sets $X \in \mathcal{F}$, from the minimal sets of $\mathcal{F}$ up to the extreme set tree root V. If when X is considered we find $w(X) < \text{dem}(X)$, then choose any vertex in X and increase its weight until $w(X) = \text{dem}(X)$.

The above algorithm clearly assigns weights satisfying the demands of all extreme sets; we need only bound the total weight assigned.

Lemma 3.6. *The total weight assigned by Algorithm 3.5 is* $\text{rdem}(V,k)$.

Proof. We show by induction up the extreme set tree $\mathcal{F}$ that the amount of weight assigned to extreme set X and its children is $\text{rdem}(X,k)$. This is clear for the leaves of $\mathcal{F}$ (vertices of G). For set X with children X_i, when we process X_i we will (by induction) have assigned total weight $\sum \text{rdem}(X_i)$ to X. If $\text{rdem}(X) = \sum \text{rdem}(X_i)$ then we assign no further weight to X and the inductive step is done. If $\text{rdem}(X) = \text{dem}(X)$, then we will increase the total weight assigned to X, raising it to $\text{dem}(X) = \text{rdem}(X)$. □

It follows that Algorithm 3.5 yields an optimal solution. The algorithm can be implemented trivially in $O(n)$ time by walking up the extreme set tree.

3.4 Degree Constraints

The above algorithm generalizes to the degree constrained external augmentation problem. As we are adding weight to sets to meet recursive demands, we only add to vertices v whose upper bounds $w_0(v)$ have not yet been met. The assumption that $d(U) + w_0(U) \geq k$ ensures that if we need to add weight to a set there will be a vertex in it whose upper bound has not yet been met. For if $d(U) + w(U) < k$ (which is the only time we need to add weight) then $w_0(U) - w(U) > 0$, meaning some vertex in U has residual weight available.

4 A Splitting Off Algorithm

As was discussed in Lemma 2.6, the optimal external augmentation just found can be turned into an optimal internal augmentation by *splitting off*. Recall that the weight $w(v)$ added in the external augmentation can be thought of as the weight of edges connecting v to s in a splitting off problem (if $d(s)$ is odd, we add one more edge to s to make it even). Previously, the best splitting off algorithm ran in $\tilde{O}(mn)$ time [NI96]. In this section, we show how the extreme set system can be used to solve the splitting off problem in $O(n^2)$ time followed by an $O(n^2 \log^3 n)$-time minimum cut computation. We take

$$k = \min_{\emptyset \subset U \subset V} (d(U) + w(U))$$

to be the goal-connectivity of the split-off graph (guaranteed to be achievable by Lemma 2.3). This goal connectivity may be part of the problem instance (as in our problem) or can be found by a minimum cut computation. Note that this turns splitting off into a kind of internal augmentation problem, but with the added constraint that each vertex v have added degree exactly $w(v)$ at the end. Having these exact degree constraints makes the augmentation problem easier to solve. We continue to refer to the demand of a set U—it is the difference between the goal connectivity k and the current degree of U in G (not including edges to s). By the assumption of our splitting off problem we have $\text{dem}(U) + w(U) \geq k$ for every $U \subset V$. As before, for any edge set E', we define $d_{E'}(U)$ to be weight of edges of E' crossing U.

Our algorithm is incremental in nature. It runs in phases. In each phase, it adds edges to reduce the demand of certain extreme sets. The challenge is finding the right set of edges to add. Care must be taken because it is possible to find edge sets that optimally reduce the demand part-way but cannot be extended to optimal solutions to the entire problem. In the following sections, we give an $O(n)$-time deterministic algorithm to find a safe increment towards the optimal solution. Our algorithm can be applied whenever the maximum remaining demand exceeds 1. We show that $O(n)$ applications of our algorithm, taking total time $O(n^2)$, suffice to reduce the maximum demand to 1. At this point, we can use an algorithm of Naor, Gusfield and Martel [NGM90] that optimally increases connectivity by one to finish solving the problem. Their algorithm runs in $O(n)$ time given the *cactus representation* of all minimum cuts of a graph, which we can find with high probability in $O(n^2 \log^3 n)$ time [KS96].

4.1 A partial split criterion

In the next lemma we give a sufficient condition for a "partial" splitting off to be continued to a valid solution.

Lemma 4.1. *Let an edge set E' satisfy that*

1. $d_{E'}(v) \leq w(v)$ *for all* $v \in V$;
2. *no edge of E' has both endpoints in any one extreme set; and*
3. *any extreme set of $G + E'$ is extreme in G as well.*

Then there exists an edge set E'' such that $E' + E''$ is a legal splitting-off of s. The set E'' can be obtained by edge splitting recursively, using the updated values

$$w'(v) = w(v) - d_{E'}(v) \quad \textit{for } v \in V\ .$$

Remark. Condition 1 is forced upon us by the problem input and Condition 2 is a natural requirement that we not "waste" an augmentation edge by putting it inside an extreme set. Condition 3 is also natural; it says that no new extreme set arises to derail our progress towards an optimal solution. Conditions 1 and 2 are necessary for an optimum solution while 3 is merely useful.

Proof. Observe that E' denotes a set of edges that, by Condition 1, can legally be split from the external vertex s. Thus, we only need to show that after splitting E' the criteria of the splitting lemma (2.3) remain true—that is, that for every set $U \subseteq V$, we have $d_{G\cup E'}(U) + w'(U) \geq k$. As before, it suffices to prove this holds for every *extreme* set X of $G \cup E'$, since every set contains an extreme set of no greater degree (by definition) and no greater total w' (since all w' are positive). By Condition 3, this extreme set X is also an extreme set of G.

So consider any extreme set X of G. The difference $w(X) - w'(X)$ is equal to the number of endpoints of edges of E' in X. By Condition 2, no edge of E' has both endpoints in X. Thus $w(X) - w'(X)$ is equal to the number of edges of E' with an endpoint in X. In other words, $w(X) = w'(X) + d_{E'}(X)$. By assumption we had $w(X) + d_G(X) \geq k$. It follows that

$$\begin{aligned} w'(X) + d_{E'\cup G}(X) &= w'(X) + d_{E'}(X) + d_G(X) \\ &= w(X) + d_G(X) \\ &\geq k \end{aligned}$$

□

4.2 A particular good partial split

We now give a rule for selecting a particular set of edges E' satisfying the criteria of Lemma 4.1. This rule applies whenever some extreme set has demand at least 2. Select all inclusion-wise maximal extreme sets of the current graph with k-demand at least 2. Number these sets X_i for $i \leq \ell$ so that X_1 and X_ℓ have the smallest (and we will soon see equal) degrees: that is,

$$d(X_\ell) = d(X_1) \leq d(X_2), \ldots, d(X_{\ell-1}).$$

Connect each X_i to X_{i+1} by an edge for $i < \ell$. Doing so requires that we find one vertex in each of X_1 and X_ℓ that we can use as an endpoint, and 2 vertices in each other X_i. These are guaranteed to exist since $w(X_i) \geq \text{dem}(X_i) \geq 2$. Note that we might use the same vertex (at most) twice to receive both edges incident on some X_i. But no $d_{E'}(v) > 2$, so E' is a collection of vertex-disjoint paths.

To see why the choice $d(X_1) = d(X_\ell)$ is possible, notice that the minimum degree $d(X_1)$ of an extreme set is equal to the (current) connectivity c of G. If we consider the two sides of a minimum cut, both sides must contain c-extreme sets (which are clearly maximal extreme sets of minimum degree) and hence we may choose these sets as X_1 and X_ℓ.

It is easy to see that this construction satisfies two of our requirements:

Lemma 4.2. *The edge set E' just described satisfies Conditions 1 and 2 of Lemma 4.1.*

Proof. We have already argued that since $w(X_i) \geq \text{dem}(X_i) \geq 2$ by choice of the X_i, we can choose the endpoints of the set E' to satisfy Criterion 1. Since each edge connects two distinct maximal extreme sets, no edge has both of its endpoints in the same extreme set, as required in Criterion 2. □

Proving that Condition 3 is satisfied is harder. We assume that there is some set U that violates it, being extreme in $G' = G \cup E'$ but not in G, and derive a contradiction. We begin with some lemmas showing that the set of edges E' essentially can be thought of as a path with respect to the hypothetical set U; then we show such a path must cross U several times, raising its degree and making it non-extreme.

Lemma 4.3. *If U overlaps an extreme set X of G then $d_{E'}(X) = d_{E'}(X \cap U, U - X) = 2$.*

Proof. By the definition of extreme sets, $d_G(X - U) \geq d_G(X) + 1$. This implies by submodularity that $d_G(U - X) \leq d_G(U) - 1$. On the other hand, since U is extreme in G', we have $d_{G'}(U - X) \geq d_{G'}(U) + 1$. Combining the two latter inequalities, we get that $d_{E'}(U - X) \geq d_{E'}(U) + 2$. Since all edges not counted in $d_{E'}(U)$ but counted in $d_{E'}(U - X)$ connect $U \cap X$ and $U - X$, there are at least two such edges; furthermore $d_{E'}(X) \geq 2$. Since $d_{E'}(X) \leq 2$ by the construction of E', equality must hold everywhere, proving the claims. □

Corollary 4.4. *X_1 and X_ℓ do not overlap U.*

Proof. $d_{E'}(X_1) = d_{E'}(X_\ell) = 1$, while any overlapping X_i must have $d_{E'}(X_i) = 2$ by the previous lemma. □

Lemma 4.5. *U contains (both endpoints of) an edge of E'.*

Proof. Since U is not extreme in G, by definition there is some (extreme in G) set $X \subset U$ such that $d_G(X) \leq d_G(U)$. However, U is extreme in G', which means that E' must increase the degree of X more than that of U. This implies that some edge e of E' crosses X but not U. Since one endpoint of e is inside $X \subset U$, both endpoints must be in U. $\square$

Corollary 4.6. *Write $E' = \{(a_i, b_i)\}$ where $a_i \in X_i$ and $b_i \in X_{i+1}$. Then $b_{i-1} \in U$ if and only if $a_i \in U$.*

Proof. By Lemma 4.5, set U contains an edge of E'. This edge is not contained in any X_i so U is not contained in any X_i. So U contains, is disjoint from, or overlaps every X_i. Both a_i and b_{i-1} are in X_i. If X_i is contained in or disjoint from U the claim is clear. If X_i overlaps U then (by Lemma 4.3) both a_i and b_{i-1} are contained in $X_i \cap U$ and thus in U. $\square$

The previous corollary says that if we move along the sequence $a_1, b_1, a_2, b_2, \ldots$, we will only cross into or out of U by traversing one of the edges of E'. So we can think of E' as a path with respect to U. We now show that this path must cross U many times, raising its degree and making it nonextreme.

We argued above that X_1 and X_ℓ do not overlap or contain U; thus each is entirely inside our outside of U. We consider two cases:

At least one of X_1 or X_ℓ (say X_1) is inside U. Note that $d_{G'}(X_1) = c + 1$, so the only way U can be extreme is if $d_{G'}(U) = d_G(U) = c$, meaning no edge of E' crosses U. If $X_1 \subset U$ then $a_1 \in U$. Since no edge of E' can cross U, we must have $b_1 \in U$. Applying Corollary 4.6, we find that $a_2 \in U$. Continuing inductively, we deduce that all a_i and b_i are inside U, meaning no endpoint of E' is in $\overline{U}$.

But if $d_G(U) = c$, then $d_G(\overline{U}) = c$ as well, which means $\overline{U}$ must contain or be a c-extreme set, which is a maximal extreme set (with demand at least 2). So some edge of E' has an endpoint in $\overline{U}$. This is a contradiction.

Both X_1 and X_ℓ are outside U. We have already argued that some edge of E' is in U. Thus the sequence $a_1, b_1, a_2, b_2, \ldots$ must start in X_1 (so outside U), then cross into and back out of U to reach X_ℓ. But Corollary 4.6 tells us that each of these crossings must involve an (a_i, b_i) pair in the sequence rather than a (b_i, a_{i+1}) pair. In other words, $d_{E'}(U) \geq 2$.

On the other hand, U was originally not extreme, so it contained some extreme X of no greater degree. Our choice of E' ensures that no extreme set's degree increases by more than 2. So $d_{G'}(X) \leq d_G(X) + 2 \leq d_G(U) + 2 \leq d_{G'}(U)$. So U is not extreme in G'

We arrived at a contradiction in both cases, so U cannot be extreme.

4.2.1 Motivation

While our splitting rule may seem a bit strange it is actually quite naturally motivated. A natural way to guarantee that no end-vertices of some edge of E' belong to the same extreme set is to connect distinct maximal extreme sets. Some previous edge augmentation algorithms [NGM90, Ben94] are also based on this simple idea. The following argument leads to our path-like construction.

We consider all minimum cuts first. Since the maximal (by containment) extreme sets include all minimal (by containment) min-cut sides, at least one edge will be added across each minimal min-cut side. As a result, some non-minimal min-cut sides will become minimal and thus extreme, unless we also add edges crossing these minimum cuts. In other words, we must increase the connectivity of G by at least one to avoid creating new extreme sets.

The next natural idea is that a cycle connecting either all maximal extreme sets or all minimal min-cut sides automatically adds *two* edges to each minimum cut. This is the main idea of Benczúr's algorithm [Ben94]. Unfortunately the addition of cycles may create new extreme sets if, for example, there are exactly three $(c+1)$-extreme subsets X_1, X_2 and X_3, each of which is the subset of the same c-extreme set X. Then without loss of generality edges of E' may be added to X_1 and X_2 but not to X_3. Hence X_3 becomes the only extreme set of G with $d_{G+E'}(X_3) = c + 1$. But then the complement of X_3 must contain another (new) $(c + 1)$-extreme set in $G + E'$.

The above example suggests the form of our algorithm: if we remove an edge from the cycle as above that connects two c-extreme sets, these sets become $(c+1)$-extreme in $G+E'$. They prevent the formation of new $(c+1)$-extreme sets.

4.3 An Algorithm

We now use our partial split rule repeatedly to get a splitting-off algorithm. Our algorithm actually settles for correctly reducing the maximum extreme set demand to 1. At this point the graph $G-s$ must be $(k-1)$-connected, so the remaining splitting off need only optimally increase the connectivity of $G-s$ by one. This can be done using an algorithm of Naor et al. [NGM90]. Their algorithm runs in $O(n)$ time given the *cactus representation* of G, which can be constructed in $O(n^2 \log^3 n)$ time by a Monte Carlo algorithm [KS96].

While the graph has demands exceeding 1, we move incrementally towards the optimal solution. Since some demands are at least 2, we can use an edge set E' of the kind described in Section 4.2. Such an edge set is easy to find using the extreme set tree. The children of the root form the maximal extreme sets, so we can easily find all maximal sets X_i with demand at least 2 in $O(n)$ time. Within each X_i, any vertices with positive weight can serve as endpoints for the edges of E' (of course, if 2 edges of E' share a vertex, that vertex needs to have weight 2). Such vertices are always guaranteed to exist since (as shown in the previous section) the splits we perform preserve the solvability of the problem (that is, $\sum_{v\in X_i} w(v) \geq \text{dem}(X_i)$). They can also be found in $O(n)$ time.

Once we have E', we can split it off. But in order to make the algorithm efficient, we try to avoid computing a new edge set by reusing the same one many times. The same edge set E' can be used again if after the split

- the updated weights w' satisfy $w'(v) \geq d_{E'}(v)$ for all $v \in V$,
- all X_i still have $\text{dem}(X_i) > 1$, and
- all X_i remain extreme.

(Note that the relation $d(X_\ell) = d(X_1) \leq d(X_2), \ldots, d(X_{\ell-1})$ remains valid after the splitting of E' since X_1 and X_ℓ start with the smallest degrees and have them increased more slowly than the other X_i.) We compute the maximum number of times t that E' can be reused without violating these constraints.

As a first step, we compute the quantities $d_{E'(X)}$ for every extreme set X. Since no edge has both endpoints in any one X, this is just the total weight of edges of E' with endpoints in each extreme set. This can be computed for all X in $O(n)$ time by working up the extreme set tree.

Given the quantities $d_{E'}$, to meet the first constraint, we compute $t_1 = \min_v \lfloor w(v)/d_{E'}(v) \rfloor$, We can use t_1 copies of E' without dropping any $w(v)$ below 0. The same approach works for the second constraint.

For the third constraint, that all X_i remain extreme, we need to identify the smallest t for which splitting t copies of E' makes some X_i non-extreme. This happens when the demand of X_i drops to meet the demand of some (extreme) set Y contained in X_i. Since no extreme sets are created as we add copies of E' (by induction on the number of copies), Y must be a descendant of X_i in the starting extreme set tree. For each such Y descended from X_i the number of copies at which $\text{dem}(Y)$ overtakes $\text{dem}(X_i)$ is

$$\left\lfloor \frac{\text{dem}(X_i) - \text{dem}(Y)}{d_{E'}(X_i) - d_{E'}(Y)} \right\rfloor,$$

since for such a t we know that after $t-1$ splits of E', the set Y still has demand less than that of X_i (note that if the denominator is 0 then $\text{dem}(Y)$ will never overtake $\text{dem}(X_i)$ so we can ignore it). Since the X_i are disjoint, we compare each Y against exactly one X_i for a total of $O(n)$ comparisons, so finding the limiting set takes $O(n)$ time.

Thus, in $O(n)$ time, we can compute the maximum number of times t it is safe to use E'. Once we have done so, we can split off t copies of E' in $O(n)$ time by updating vertex weights and extreme set demands and removing all sets that become non-extreme as a result.

Finally, we bound the number of times we need to find a new set E'. It is the 3 constraints above that prevent us from splitting another copy of E', so one of the constraints must be "tight" for the t we used. If the first constraint is tight, it is because some $w(v) < d_{E'}(v)$ when we finish. Since $\max_v d_{E'}(v) \leq 2$, this means that $w(v)$ has dropped to

1 or to 0. Similarly, the second constrain is tight only when the demand of some X_i drops to 1 or 0. The third constraint is tight if one of our extreme sets is made non-extreme. Each of these events—a weight drop, a demand drop, or the disappearance of an extreme set—happens $O(n)$ times. Thus we need to compute a new set E' and corresponding t only $O(n)$ times. Since identifying E' and updating the data after splitting it takes $O(n)$ time, we have shown:

Theorem 4.7. *Given the extreme set system for G, in $O(n^2)$ time we can deterministically carry out edge splitting to reduce the maximum set demand in G to 1, at which point we can finish splitting (with high probability) in $O(n^2 \log^3 n)$ time using cactus-based algorithms [NGM90, KS96].*

Oddly, only the last unit of splitting involves randomization.

5 Extreme set algorithms

The two previous sections assumed that we had the extreme set system available. In this section we give Monte Carlo algorithms that find this system with high probability. To present our techniques one by one, we describe three increasingly powerful extreme set algorithms. The first algorithm (Section 5.4) is based solely on finding minimum cuts and serves as an illustration of the main ideas in our extreme set algorithm. Its running time is proportional to the maximum vertex degree, which is $O(nW_{\max})$ where $W_{\max}$ is the largest edge weight in the input graph. The second algorithm (Sections 5.5 and 5.6) uses *near*-minimum cuts: extreme sets are repeatedly extracted from all cuts of value between $\delta^{i-1} \cdot c$ and $\delta^i \cdot c$, for $i = 1, 2$, etc. The running time of this algorithm is thus dependent on $\log(W_{\max}/c)$. Our final algorithm (Section 5.7) achieves a strongly polynomial $\tilde{O}(n^2)$ running time by dividing the edges according to their weight into windows within which $W_{\max}/c$ becomes polynomial in n. The $\tilde{O}(n^2)$ runtime follows from the fact that each edge occurs in only a constant number of windows. A similar windowing scheme was used in other cut algorithms [BK96].

5.1 The Recursive Contraction Algorithm

Our extreme set algorithms use the *Recursive Contraction Algorithm* (RCA) [KS96]. This is an algorithm for finding all minimum cuts in a graph. It is based on contraction of graph edges. Contracting an edge causes its two endpoints to be merged into a single "metavertex." At any time, each existing metavertex represents a set of original graph vertices that have been contracted into it. If metavertex v represents a set of original vertices S, then the degree of v in the contracted graph is equal to the value of the cut $(S, \overline{S})$ in the original graph.

We need two definitions. We say that a cut *survives the contraction* of a set of graph edges if no edge connecting the two cut sides gets contracted. Similarly we say that a set X *survives the contraction* if the same holds for the cut $(X, \overline{X})$. If this occurs, then there is a set of metavertices Y containing exactly the vertices of X; we say that X *is contracted* to Y.

In rough outline, RCA has the following form:

Algorithm RCA(G, n)

input: an n vertex graph G

if $n \leq 7$ **then**
  Use brute force enumeration to find all cuts
else repeat twice
  Contract randomly chosen edges of G until we get G' with $1 + n/\sqrt{2}$ vertices
  RCA$(G', 1 + n/\sqrt{2})$

The contractions at a given stage can be implemented in $O(n^2)$ time on an n-vertex graph [KS96], so RCA satisfies the running time recurrence

$$\begin{aligned} T(n) &= 2T(1 + n/\sqrt{2}) + O(n^2) \\ &= O(n^2 \log n) \end{aligned}$$

We can also prove that RCA "encounters" any minimum cut of G (by contracting all vertices on one side of it into a metavertex whose degree is the minimum cut) with probability $\Omega(1/\log n)$. More precisely, for any particular cut, with probability $\Omega(1/\log n)$ the cut survives the contractions to a 7-vertex graph along at least one of the execution paths of the recursive algorithm, at which point it is found by the brute force enumeration of the base case. Thus $O(\log^2 n)$ iterations of the above algorithm suffice to encounter all minimum cuts with high probability. We will refer to these iterations of RCA as Algorithm iRCA. Since we can track the degrees of sets we encounter [KS96], we can recognize a minimum cut when we encounter it. Thus the time to find all minimum cuts with high probability using iRCA is $O(n^2 \log^3 n)$.

5.2 Finding Extreme Sets—Basic Approach

We will modify iRCA to identify extreme sets rather than minimum cuts. A key observation is that, like minimum cuts, extreme sets are likely to survive contraction by RCA to single metavertices. Unfortunately, unlike minimum cuts, we have no trivial method (e.g. degree tests) for deciding whether a given metavertex represents an extreme set. We therefore need to add a "verification" step that decides which of our candidate extreme set is truly extreme. We use the following modification ES (Extreme Sets) of RCA:

Algorithm ES(G, n)

input: an n vertex graph G
output: laminar family $\mathcal{F}$ containing some extreme sets of G

if $n \leq 8$ **then**
    Use brute force enumeration to find and return all extreme sets
else
    repeat twice (letting $i = 1, 2$)
        Contract random edges of G to get G_i with $2 + n/\sqrt{2}$ vertices
        $\mathcal{F}_i \leftarrow$ES$(G_i, 2 + n/\sqrt{2})$
        expand the extreme sets of G_i in $\mathcal{F}_i$ to G
            (by uncontracting the metavertices of G_i to vertex sets of G)
    $\mathcal{F} \leftarrow \mathcal{F}_1 \cup \mathcal{F}_2$
    add to $\mathcal{F}$ all vertices of G as singleton extreme sets
    cull some non-extreme sets from $\mathcal{F}$ to make $\mathcal{F}$ laminar
    Return the resulting set system

Since all singleton vertices are returned, and since culling only removes nonextreme sets, any extreme sets "encountered" by ES (that is, contracted to single metavertices at some point), will be returned by ES.

We will actually need to call ES $O(\log^2 n)$ times, like iRCA, to have a high probability of finding all extreme sets. But we can cull the $O(\log^2 n)$ resulting set systems by repeatedly merging and culling pairs of them. We will end up with a laminar family that contains all extreme sets with high probability. We refer to this iteration of ES as Algorithm iES.

5.3 An Easy Case

As a first demonstration of the above process, we find all *c-extreme sets*—that is, extreme sets that are also minimum cuts. We can use the fact that if U and X are (sides of) two different minimum cuts that overlap, then $U - X$ and $X - U$ are minimum cuts (this follows from the submodularity of the function $d(U)$). Thus, neither U nor X is c-extreme and both can be discarded. It follows that the c-extreme sets of G are actually disjoint, forming a *subpartition* of the vertex set—that is, a collection of disjoint subsets of V.

Our culling procedure for the subpartitions $\mathcal{F}_1$ and $\mathcal{F}_2$ returned by the recursive calls is therefore quite simple. We represent a subpartition $\{X_1, \ldots, X_r\}$ by a vector $(x_1, x_2, \ldots)$ where $x_i = j$ if the i-th vertex of G is contained in X_j. We use a special null symbol for a vertex not in any of the sets. By referring to the vector representing $\mathcal{F}_2$, we can decide in $O(n)$ time whether a given set $C \in \mathcal{F}_1$ overlaps or contains some element of $\mathcal{F}_2$. If it does, we can discard C since it is not extreme. Since $\mathcal{F}_1$ has $O(n)$ sets, checking all $C \in \mathcal{F}_1$ takes $O(n^2)$ time as desired. We then repeat the procedure, exchanging the roles of $\mathcal{F}_1$ and $\mathcal{F}_2$, to cull non-extreme sets from $\mathcal{F}_2$.

Since culling takes $O(n^2)$ time, it is dominated by the other operations of `iRCA` at each recursion node; thus `iES` has the same running time as `iRCA`. The $O(\log^2 n)$ culling calls needed to merge the families produced by the $O(\log^2 n)$ iterations of `ES` take an additional $O(n^2 \log^2 n)$ time, which is also dominated by `iRCA`.

Since `iRCA` (and thus `iES`) contracts all minimum cuts (and thus all c-extreme sets) to metavertices with high probability at some point in the recursion, all extreme sets will be introduced as singleton vertices at some point in the recursion. Clearly they will never be culled. Thus the algorithm will output all extreme sets.

In fact, this algorithm may also output certain non-extreme sets (since it never checks their degrees), but it is guaranteed that the output sets will be disjoint. Thus it takes only $O(m)$ time to compute the degrees of all output sets. We can discard all that have degree exceeding c.

5.4 A First Algorithm

We now build upon the above idea to find all extreme sets. Suppose that we have found all d'-extreme sets $\{X_i\}$ for $d' < d$, and wish to find all d-extreme sets. Since extreme sets are laminar, no d-extreme set can contain or overlap any X_i. So define M_i to be the set of vertices contained in X_i but not in any child of (i.e. d'-extreme set contained in) X_i. Also let M_0 be the set of vertices not in any X_i. Then any d-extreme set is contained in some M_i. Note also that the M_i form a partition of the vertex set V. Thus, if for each M_i we find all d-extreme sets strictly contained in it in $\tilde{O}(|M_i|^2)$ time then we will have found all d-extreme sets in $\tilde{O}(\sum |M_i|^2) = \tilde{O}(n^2)$ time (since $\sum |M_i| = n$).

To process a single set M_i in $\tilde{O}(|M_i|^2)$ time, we can contract $V - M_i$ to a single vertex q. Any extreme set in M_i will still be extreme in the contracted graph. But all extreme sets not in M_i have been contracted. In particular, no set inside M_i is d'-extreme for any $d' < d$. We might therefore aim to apply our previous algorithm for finding c-extreme sets to M_i with $c = d$. One small problem is that potentially $d(M_i) < d$ (this can happen if M_i is itself an extreme set). This would make the cut $(M_i, \{q\})$ into the unique minimum cut of the contracted graph. Fortunately, the existence of one such unusually small cut does not affect anything:

Lemma 5.1. *Suppose that a graph G has a unique minimum cut and let c be the next smallest cut value in G. Then with high probability* `iRCA` *encounters (contracts to a metavertex) all cuts of value c in G, and* `iES` *encounters all c-extreme sets of G.*

Proof. The proof goes much as the proof that algorithm `RCA` is correct [KS96]. We lower bound the probability that none of the edges crossing the extreme set get contracted. If this happens, the extreme set will eventually be noticed when it gets contracted to a single vertex or the recursion bottoms out.

We begin by analyzing a sequence of random edge contractions from n down to $2 + n/\sqrt{2}$ vertices. Consider a particular c-extreme set. Suppose we have performed contractions until G has r vertices remaining. There is only one cut of value less than c, so in particular every vertex but one has degree at least c. Thus there are at least $(r-1)c/2$ edges in the graph, which means that the probability we pick an edge crossing the particular c-extreme set we want to find is only $2/(r-1)$. It follows that as we contract from n to $2 + n/\sqrt{2}$ vertices, the probability we never pick a bad edge is (c.f. [KS96]):

$$(1 - \frac{2}{n-1})(1 - \frac{2}{n-2}) \cdots (1 - \frac{2}{n/\sqrt{2}+1}) \geq 1/2.$$

It follows that the probability $P(n)$ that algorithm `ES` finds a particular c-extreme set in the n-vertex graph satisfies the recurrence

$$P(n) = 1 - (1 - \frac{1}{2} \cdot P(2 + n/\sqrt{2}))^2 = \Omega(1/\log n).$$

The recurrence arises from the fact that for `ES` to fail to find the set on n vertices, it must fail on both of its two independent subproblems (thus the squaring in the recurrence). But it succeeds on a subproblem if the the particular

c-extreme set survives the contractions to $2 + n/\sqrt{2}$ vertices (probability 1/2) and then the recursive call finds it (probability $P(2 + n/\sqrt{2})$). This recurrence is solved in the earlier paper [KS96] to yield the claimed bound.

Since a single iteration encounters any particular set with probability $\Omega(1/\log n)$, we can run $O(\log^2 n)$ iterations of ES to reduce the probability of missing the particular set to $O(1/n^3)$. Since there are $O(n)$ extreme sets, we encounter *all* of them with probability $1 - O(1/n^2)$ as claimed. □

The above lemma shows that iES will still encounter (contract to a metavertex) all d-extreme sets with high probability; it remains to show how to cull them. We can use the same technique as in Section 5.3. Suppose two second-minimum cuts Y and Z overlap. By submodularity, $d(Y - Z) + d(Z - Y) \leq d(Y) + d(Z)$. Since Y and Z are second minimum cuts, this inequality must in fact be an equality, and both Y and Z non-extreme as a result, unless one of $Y - Z$ or $Z - Y$ is the unique minimum cut. So we cull our family in two steps: first, in $O(n)$ time, we delete all sets that contain the unique minimum cut $\{q\}$. Then we use the $O(n^2)$ time algorithm of Section 5.3 to eliminate any pairs of sets that overlap. We omit additional details since we will shortly give a more powerful algorithm.

It follows that with high probability[2] we will indeed encounter all d-extreme cuts in the contracted graph $M_i \cup \{q\}$. This gives a simple algorithm for finding extreme sets: starting with $d = c$ and incrementing d each time, find all minimum-degree extreme sets in each set of the M_i partitions defined above. The running time of this algorithm can be bounded two ways. For augmentation to connectivity $k = c + \tau$, we need only find extreme sets of degree up to $c + \tau$; thus we need only τ iterations for an overall running time of $O(n^2 \tau \log^3 n)$. On the other hand, regardless of τ there are $O(n)$ extreme sets, and a new one is found in each iteration. So there will be $O(n)$ iterations for a running time of $O(n^3 \log^3 n)$.

5.5 Geometric Growth

The scheme described above will, in the worst case, add 1 to to the degree d of extreme sets detected in each iteration. We now show how instead we can multiply d by some quantity exceeding 1 in each iteration. Thus will reduce the number of iterations needed to reach connectivity k from $k - c$ to $\log(k/c)$.

Lemma 5.2 ([KS96]). *If, instead of contracting to $n/\sqrt{2}$ vertices, the recursion in* RCA *contracts to $n/2^{1/2\alpha}$ vertices, then* iRCA *runs in $O(n^{2\alpha} \log^3 n)$ time and, with high probability, encounters all cuts of value less then αc in a graph with minimum cut c.*

The above lemma generalizes in a straightforward fashion to the case where we are looking for extreme sets using ES and the graph has a unique minimum cut:

Corollary 5.3. *If, instead of contracting to $n/\sqrt{2}$ vertices, the recursion in* ES *contracts to $n/2^{1/2\alpha}$ vertices, then (aside from culling)* iES *runs in $O(n^{2\alpha} \log^3 n)$ time and, with high probability, encounters all extreme sets of degree less than αc in a graph with minimum cut c. The result holds even if there is a unique cut of value less than c.*

Proof. Again, the proof matches that of the previous paper [KS96]. The running time obeys

$$T(n) = 2T(n/\sqrt[\alpha]{2}) + O(n^2) = O(n^{2\alpha} \log n).$$

Arguing as in Lemma 5.1, the probability that we never contract an edge crossing a particular αc-extreme set as we reduce from n to $n/\sqrt[\alpha]{2}$ vertices is at least

$$(1 - \frac{2\alpha}{n-1})(1 - \frac{2\alpha}{n-2}) \cdots (1 - \frac{2\alpha}{n/\sqrt[\alpha]{2} + 1}) \geq 1/2.$$

We continue exactly as before to deduce an $\Omega(1/\log n)$ probability of encountering a particular extreme set in one iteration of ES, and a high probability of success over $O(\log^2 n)$ iterations. □

[2] Note that regardless of the size n' of M_i, we will iterate ES $O(\log n \log n')$ times, rather than $O(\log^2 n')$ times, in order to keep the probability of failure below $1/n^2$ instead of $1/n'^2$, but this is already accounted for in our time bounds.

Now suppose that we choose $\alpha = 1+1/\log n$. Then the running time of `iES` (aside from culling) is $O(n^{2\alpha}\log^3 n) = O(n^2 \log^3 n)$. Once we have found all extreme sets of degree less than αc, we can partition the vertices into sets M_i as we did in the previous section and recursively find all extreme sets of degree exceeding αc separately in each M_i. Since the second-minimum cut value increases by a factor of at least $1 + 1/\log n$ when we recurse, and since the maximum extreme set degree is at most $nW_{\max}$ (the maximum vertex degree) for maximum edge weight $W_{\max}$, the recursion will find all extreme sets after $(\log n)\log(nW_{\max}/c)$ iterations.

Shortly we will develop a culling algorithm with running time $O(n^2)$ that we can use in Algorithm `ES`. This time is asymptotically dominated by the other time `ES` spends at each recursion node and thus does not affect the overall running time of `ES`.

Combining these arguments leads to the following:

Lemma 5.4. *In a graph with minimum cut c and maximum edge weight $W_{\max}$, all extreme sets can be found with high probability in $O(n^2\log^4 n\log(nW_{\max}/c))$ time (aside from culling). In particular, when $W_{\max} = n^{O(1)}c$, the time needed is $O(n^2\log^5 n)$.*

5.6 A general culling algorithm

It remains to describe a culling scheme that we can apply in our extreme set algorithm. The most obvious approach is to work as before: whenever two degree-c sets overlap, we know that neither is extreme. But this idea only applies when all candidate extreme sets being examined have the same degree; in the new approach, we might simultaneously find sets with many different degrees. When two such sets overlap, *one* of the overlapping sets might be extreme, and we have no obvious way to tell which. So we take a more complicated approach. We exploit the fact that our culling input is the union of two laminar families of sets. We will give a culling algorithm that runs in $O(n^2)$ time on a pair of laminar families over an n vertex graph. Culling is therefore not the bottleneck in `iES`, so that algorithm's running time remains as claimed before.

Given that we use the extreme-set finding algorithm above, our culling task is as follows. We are given two laminar set systems $\mathcal{S}$ and $\mathcal{T}$ (the results from the two recursive calls in `ES`). We wish to build a new, laminar system that contains (at least) all extreme sets in $\mathcal{S}\cup\mathcal{T}$. We will do so by discarding certain non-extreme sets from $\mathcal{S}$ or $\mathcal{T}$. The challenge is deciding which sets are not extreme. Recall that a family is non-laminar if and only if two sets in it overlap. So suppose two sets X and Y in the (merged) family $\mathcal{S}\cup\mathcal{T}$ do overlap. As was argued before in Lemma 2.8, the submodularity of the degree function tells us that

$$d(X-Y)+d(Y-X) \le d(X)+d(Y).$$

It follows that either $d(X-Y)\le d(X)$ or $d(Y-X)\le d(Y)$. So one of X or Y is not extreme and can be discarded from the family.

The above discussion reveals our plan for culling the input set family. For every overlapping pair of sets X and Y (where without loss of generality $X\in\mathcal{S}$ and $Y\in T$, since $\mathcal{S}$ and $\mathcal{T}$ are separately laminar) we compute $d(X-Y)$ and compare it to $d(X)$ to see if we can discard X. We then do the same symmetrically to identify discardable sets in $\mathcal{T}$.

As a first step, we show how to quickly compute $d(X)$ for every $X\in\mathcal{S}$. We use a method similar to one used for finding minimum cuts [Kar96]. Recall that the laminar set system $\mathcal{S}$ corresponds to a tree whose leaf nodes are the vertices of G and whose (leaf and internal) nodes each correspond to a set of $\mathcal{S}$. For clarity we will always refer to nodes of the tree (laminar family) versus vertices of the graph G. We begin with some definitions.

Definition 5.5. $X^{\downarrow}$ is the set of nodes that are descendants of node X, including X.

Definition 5.6. Given a function f on the nodes of a tree, the *treefix sum* of f, denoted $f^{\downarrow}$, is the function

$$f^{\downarrow}(X) = \sum_{U\in X^{\downarrow}} f(U).$$

Lemma 5.7. *Given the values of a function f at the tree nodes, all values of $f^{\downarrow}$ can be computed in $O(n)$ time.*

Proof. Perform a postorder traversal of the nodes. When we visit a node X we already will have computed (by induction) the values at each of its children. Adding these values takes time proportional to the number of children of X; adding in $f(X)$ gives us $f^{\downarrow}(X)$. Thus, the overall computation time is proportional to the total number of children in the tree, which is one less than the total number of nodes, so $O(n)$. □

We can compute the degrees $d(X)$ via treefix sums. We define some functions on the nodes of the laminar set tree whose treefix sums we will use. First, let $\delta(\{v\})$ be the (weighted) degree of vertex v for each singleton set (leaf node) $\{v\}$ of the laminar family, and let $\delta(X) = 0$ for all other sets. Then $\delta^{\downarrow}(X)$ is the sum of degrees of vertices in X. Next, let $A(X)$ denote the set of edges whose endpoints' least common ancestor in $\mathcal{S}$ is X. Let $\rho(X)$ denote the total weight of edges in $A(X)$. Then $\rho^{\downarrow}(X)$ is the total weight of edges with both endpoints in X.

Lemma 5.8. $d(X) = \delta^{\downarrow}(X) - 2\rho^{\downarrow}(X)$.

Proof. The term $\delta^{\downarrow}(X)$ counts all the edges with endpoints in X. This correctly counts each edge crossing the cut defined by X, but also double-counts all edges with both endpoints inside X. But an edge has both endpoints inside X if and only if its least common ancestor is in $X^{\downarrow}$. Thus the total weight of such edges is $\rho^{\downarrow}(X)$. We "uncount" both endpoints of these edges. □

Since treefix sums take $O(n)$ time, it follows that the values $d(X)$ for all sets $X \in \mathcal{S}$ can be computed in $O(n)$ time given the functions δ and ρ. But both δ and ρ can be computed in $O(m)$ time. To compute δ, scan the edges once and accumulate their weights into their endpoints. Computing $A(X)$, and from it the function ρ, is equally easy if we know the least common ancestor (in $\mathcal{S}$) of each edge; these can be determined in $O(m)$ time [GT85, BV93, SV88]. We summarize our argument in the following lemma:

Lemma 5.9. *Given a set of n vertices, a laminar family $\mathcal{S}$ of sets of these vertices, and a collection of m edges on these vertices, in $O(m + n)$ time we can compute:*

- *The degree sum of vertices in each set $X \in \mathcal{S}$*
- *The set of edges with both endpoint in each set $X \in \mathcal{S}$.*

and from these two quantities, $d(X)$ for each $X \in \mathcal{S}$ using lemma 5.8.

We now extend this approach to calculate the quantities $d(X - Y)$ for all X and Y. For now we assume that Y is fixed and compute $d(X - Y)$ (as a function of X) for each $X \in \mathcal{S}$. That is, we compute degrees for the sets in the family

$$\{X - Y \mid X \in \mathcal{S}\}.$$

Noting that this is again a laminar family, we can apply the same procedure as before. First, let $\delta_Y(\{v\})$ equal $d(v)$ if $v \notin Y$, and 0 otherwise. Then $\delta_Y^{\downarrow}(X)$ is the sum of degrees of vertices in $X - Y$. Next, let $\rho_Y(X)$ be the weight of edges in $A(X)$ that have neither endpoint in Y. It follows that $\rho_Y^{\downarrow}(X)$ is the weight of edges with both endpoints in $X - Y$. Therefore, $d(X - Y) = \delta_Y^{\downarrow}(X) - 2\rho_Y^{\downarrow}(X)$. So given $\delta_Y(X)$ and $\rho_Y(X)$ for all $X \in \mathcal{S}$, we can compute the desired values $d(X - Y)$ for all X in $O(n)$ time.

It remains to show how to compute the functions δ_Y and ρ_Y for each set $Y \in \mathcal{T}$. We assume that we have already computed $d(v)$ for each v and $A(X)$ and $\rho(X)$ for each X (this takes $O(m)$ time by Lemma 5.9). Given the degrees $d(v)$ for each vertex (leaf node) v, we compute $\delta_Y(\{v\})$ for all v in $O(n)$ time by checking if $v \in Y$ and setting $\delta_Y(\{v\})$ to be 0 or $d(v)$ accordingly. On non-leaf nodes X, $\delta_Y(X)$ is 0 as is $\delta(X)$. Thus for a given set Y, we can compute the function δ_Y in $O(n)$ time.

Computing $\rho_Y(X)$ is somewhat trickier. Recall that this is the weight of edges in $A(X)$ that have neither endpoint in Y. Since we previously computed $\rho(X)$, it suffices to compute the complementary weight of edges in $A(X)$ that have at least one endpoint in Y. But this is just the sum of degrees (in $A(X)$) of vertices in Y, minus the weight of edges with both endpoints in Y (since these are double-counted in the degree sum). Now note that these two quantities are precisely the quantities listed in Lemma 5.9 if we consider the laminar family to be $\mathcal{T}$ and the set of edges to be

$A(X)$. It follows from Lemma 5.9 that we can compute the desired quantities for X in $O(n+\|A(X)\|)$ time. Carrying out this computation for every $X \in T$ requires

$$\sum_X O(n + \|A(X)\|) = O(n^2 + m) = O(n^2)$$

time since $\sum \|A(X)\| = O(m)$.

Compute $A(X)$ for every node $X \in \mathcal{S}$ in $O(m)$ time using LCA computations.
Compute $\rho(X)$ and $\delta(X)$ for each X in $O(m)$ time
Compute $d(X)$ using treefix computations on the above quantities in $O(n)$ time

for each node $X \in \mathcal{S}$
    Select edge set $A(X)$ and laminar family $\mathcal{T}$
    Use Lemma 5.9 to compute, simultaneously for every $Y \in \mathcal{T}$ in $O(n + \|A(X)\|)$ time,
        (i) the weight of edges of $A(X)$ with both ends in Y and
        (ii) the sum of degrees of vertices in Y
    From the above quantities, determine $\rho_Y(X)$ in constant time for each node Y (total time $O(n)$).
for each $Y \in \mathcal{T}$
    compute treefix sums over $X \in \mathcal{S}$ of $\delta_Y(X)$ and $\rho_Y(X)$ in $O(n)$ time
    use them to determine $d(X - Y)$ for all $X \in \mathcal{S}$ in $O(n)$ time.

Given all $O(n^2)$ values $d(X - Y)$, cull any set X for which $d(X - Y) \leq d(X)$ for any Y that meets X

Figure 1: The culling algorithm

Now that we have worked backwards to a full solution, we can restate it in execution order in Figure 1. The sets that survive this culling algorithm will form a laminar family—if two overlap, then by submodularity one will fail the degree test in our culling algorithm. This completes the culling algorithm and shows:

Theorem 5.10. *Given two laminar families, a subset of their union containing all the extreme sets they contain can be constructed deterministically in $O(n^2)$ time.*

If we use this culling algorithm in our extreme set algorithm, we will at the end output a laminar family containing all extreme sets of G that we have encountered. Our implementation of `iES` makes it highly probable (Corollary 5.3 that this will be all extreme sets of G. We can quickly remove non-extreme sets from this laminar family since each contains an extreme set—we compute the degrees of all sets in the family (as described in our culling algorithm) and discard any set that contains a set of smaller degree. Since our culling algorithm takes $O(n^2)$ time on n vertices, using it does not affect the $O(n^2 \log^3 n)$ running time of the extreme set algorithm `iES` described in Corollary 5.3. This fills in the final piece of Lemma 5.4.

5.7 The strongly polynomial $\tilde{O}(n^2)$-time algorithm

In a final step, we eliminate the $\log(k/c)$ factor from the running time of our extreme sets algorithm. We use a *windowing scheme* (similar to one used previously [BK96]) that restricts the search for d-extreme sets only to a small "window" of relevant edge weights in the range $[d/n^3, d]$. Once all edge weights are in this range, the range of extreme set values that exist in the graph becomes polynomial, so we can replace $\log(k/c)$ by $\log n$ in the statement of Lemma 5.4. The final running time of our algorithm thus becomes $O(n^2 \log^5 n)$.

5.7.1 The Core Idea

We define the following W*-windowing* procedure. Let $\mathcal{T}$ be a maximum weight spanning tree of G. Suppose that we delete all edges of $\mathcal{T}$ with weight less than W/n^3 and then contract all edges of $\mathcal{T}$ with weight W or more. This partitions $\mathcal{T}$ into subtrees, each containing some metavertices of G. The metavertices in each tree of the forest define a vertex-induced subgraph on the contraction of G that we call a *pane*.

Lemma 5.11. *Let X be any d-extreme set with $W/n \leq d < W$. Then W-windowing contracts X to some set Y that is an extreme set in one of the panes.*

Proof. Every edge crossing X has weight at most $d(X) < W$, so no edge crossing X is contracted. This proves that X is contracted to some set of metavertices Y. We need only show that Y is entirely contained in one of the panes—that is, that the partition defined by the subtrees of $\mathcal{T}$ does not split X.

Recall the following fact about maximum spanning trees [Tar83]: if e is an edge of weight w, then the path in the maximum spanning tree connecting the endpoints of e is made up entirely of edges with weight at least w. Suppose X is split into multiple pieces in different components of the forest. If X is in at least two pieces, some piece, say $X_1 \subset X$, must have edge-weight at most $d(X)/2$ crossing from it to $\overline{X}$. At the same time, there is no edge of weight at least W/n^3 connecting X_1 and $X_2 = X - X_1$ (if there were, then $\mathcal{T}$ would have to contain a path of edges of weight at least W/n^3 connecting X_1 to X_2, so they would not be split). Thus, the total weight connecting X_1 to X_2 is at most $\binom{n}{2}(W/n^3) < W/2n$. Thus,

$$d(X_1) = d(X_1, X_2) + d(X_1, \overline{X}) \leq d/2 + W/2n \leq d,$$

contradicting the assumption that X is d-extreme. □

We will use the above theorem in a strongly polynomial extreme set algorithm. For each integer value of i, we separately seek all d-extreme sets for $n^{i-1} \leq d \leq n^i$ by looking at the panes in the window at $W = n^i$. Each such pane is spanned by a tree of MST edges of weight at least $W/n^3 \geq d/n^3$, so the minimum cut of the pane is at least this large. So the ratio of extreme set value to minimum cut is polynomial in n. Thus within the pane the weakly polynomial algorithm given in Lemma 5.4 has a strongly polynomial running time.

We will shortly argue that total size $\sum n_j$ of panes in windows that we need to solve is $O(n)$ metavertices; it follows that we can find the extreme sets in all the windows with high probability in $\tilde{O}(\sum n_j^2) = \tilde{O}(n^2)$ time.

At the conclusion of this windowing procedure, we will have a collection of $O(n)$ laminar families $\mathcal{F}_1, \ldots, \mathcal{F}_r$, one for each value of W we used. These families together contain all the extreme sets of G, but may also contain some other sets that are extreme in some pane but not in G. We need to merge these families and remove the non-extreme sets.

In the following subsections, we will fill in three details of this algorithmic outline:

- We need to show that the total number of metavertices in our windows is $O(n)$,
- We need to efficiently construct the various windows, and
- We need to efficiently merge the laminar families from the various windows.

We will discuss an $O(n^2)$-time implementation that meets these goals. While we are confident that an $O(m)$-time algorithm is possible, the $O(n^2)$ bound is already dominated by the time to find extreme sets. Settling for the slower bound lets us give simpler algorithms.

5.7.2 An Evolutionary Model

To analyze and implement our algorithm we imagine an evolutionary model of our graph. Recall the parameter $W = n^k$ from our windowing procedure, and consider what happens if k starts at ∞ and decreases through the integers to $-\infty$. Initially, all maximum spanning tree edges are too small to be in the window so the panes are just the singleton vertices. As k decreases, certain MST edges "arrive" in the window. This causes certain panes to merge. Later, MST edges that entered the window get contracted, creating new metavertices.

This arrival and contraction of MST edges affects the other edges of G. Edges move through 3 stages. Initially, an edge connects two separate panes; we say this edge is *pending.* At some time, the endpoints of the edge are connected into the same pane; we say the edge is *active.* Finally, the endpoints of this edge are contracted into a single metavertex, turning the edge into an irrelevant loop; we say the edge is *finished.* Note that these definitions also apply consistently to the MST edges themselves. Note also that an edge's state is determined not by its own weight, but by the weight of the MST edges connecting its endpoints.

5.7.3 Size Analysis

We use our evolutionary model to bound the total size (in number of metavertices) of the panes we need to analyze. We distinguish two kinds of pane: a *trivial* pane is a single metavertex, while a *nontrivial* pane is made up of more than one metavertex. An isolated metavertex is extreme by definition, so we need spend no time analyzing it. We only need to bound the total size of the nontrivial panes. Each nontrivial pane is a component spanned by some active MST edges; the number of metavertices in the pane is at most twice the number of MST edges in it. Thus, the total size (in number of metavertices) of the panes for a given window value W is proportional to the number of active MST edges—that is, the number of MST edges with weight between W and W/n^3. An MST edge of weight w contributes to this count only when $W/n^3 \leq w \leq W$, which happens for at most 4 values of $W = n^k$. Thus, over all W, the total size (in metavertices) of problems solved is $O(n)$.

We can similarly bound the total number of edges in all panes. The endpoints of an edge end up in the same pane, activating the edge, when all edges on the MST path connecting them have arrived; the last such edge to arrive is the lightest edge on the MST path. The endpoints of the edge are contracted, finishing the edge, when all edges on this MST path are contracted; again the last edge to be contracted is the lightest one on the MST path. There are only four phases between the arrival and contraction of this lightest MST edge, so each edge is active for only 4 phases. Thus, over all phases, the total number of active edges is $O(m)$.

In summary, we have panes j with $O(n_j)$ metavertices and $O(m_j)$ edges, such that $\sum n_j = O(n)$ and $\sum m_j = O(m)$. Some of the panes might have multiple edges with the same endpoints, which could invalidate our $\tilde{O}(n^2)$ time bound analysis for extreme sets; however, we can merge all parallel edges of pane j in $O(m_j)$ time, taking a total of $O(\sum m_j) = O(m)$ time over all panes. It follows that the total time spent to find extreme sets in *all* panes using the algorithm of the previous section is $O(\sum n_j^2 \log^5 n) = O(n^2 \log^5 n)$ as claimed.

5.7.4 Building the Windows

Of course, we must actually generate the windows whose panes are passed to our extreme set algorithm. We implement the evolutionary model just introduced, in which MST edges arrive and get contracted over time.

As was discussed above, the activation and finishing time of an edge are determined by the lightest edge on the MST path between its endpoints. Thus, to determine the evolution of every edge we need merely find, for every edge, the weight of the lightest edge on the MST path between its endpoints. This is the MST verification problem, which can be solved in $O(m)$ time [DRT92], or in $O(m \log n^2/m)$ time by more practical algorithms [Tar83]. Note that since the MST path between an edge's endpoints, rather than the edge's own weight, determines its arrival time, edges' activation order may be quite different from their order by weight.

We would like to output the panes that arise as the graph evolves. Of course, the graph evolves through infinitely many phases, but we only need to consider those phases in which some MST edge is active, as these are the only phases for which some nontrivial (with more than one metavertex) pane exists. There are $O(n)$ such relevant phases, and we can determine them from the list of MST edge weights. If we sort the MST edges by weight, we can run through the phases in temporal order.

We consider the graph at a certain phase of its evolution, and show how to output the panes from that phase. As was discussed above, the total size (in vertices and edges) of nontrivial panes is $O(m)$. The total size of trivial panes is $O(n^2)$ (at most n vertices in each of $O(n)$ phases). Thus if our algorithm outputs each pane in time proportional to its size, the overall time to generate the panes will be $O(n^2)$.

We create a collection of buckets corresponding to the phases in which something happens; into each bucket we place copies of the edges that first become active in that phase. Next we start moving through the phases chronologically. We describe how to transform the graph from one phase to the next. First, we contract all the currently active edges that finish in the phase; this takes time proportional to the number of active edges. The contraction creates new metavertices; by traversing the current panes we can relabel every vertex with the identity of its new metavertex in

$O(n)$ time (a union-find data structure could be used instead, of course). Next, we add all the edges that become active in the new phase. We compute the connected components induced by these and the other still-active edges to identify the new panes; this takes time proportional to the size of the new panes.

It follows that the time to move from one phase to the next is proportional to the size of the two windows involved; thus the total time spent is proportional to the totals size of windows, which we have seen is $O(n^2)$.

5.7.5 Merging the Extreme Sets

We have now shown how to build a set of windows which we have argued contains as extreme sets all the extreme sets of G. Since these windows have total vertex count $O(n)$, the time to find all extreme sets in them is $O(n^2)$. It remains to merge the resulting extreme set families $\mathcal{F}_i$ into the extreme set family $\mathcal{F}$ for G. The problem is that although every extreme set in G is extreme in some $\mathcal{F}_i$, the converse might not be true. We need to cull non-extreme sets as we merge.

We first resolve a minor technical problem. Our definitions of panes involved "cutting out" a pane from the remainder of the graph and computing extreme sets in it. This cutting process, which removes some edges of G, changes the degrees of sets in the pane. We will find it more convenient to have these degrees unchanged. Thus, before finding extreme sets in a pane P, we add a new metavertex s representing the (contraction of) the remainder of the graph $G - P$. We compute the weights of edges incident on s from every metavertex in P. This is just the total weight of pending edges incident on each metavertex in P. This quantity is easily maintained: initially all edges are pending, and as they become active we can subtract their weights from the degrees of their endpoints. Representing s adds at most n edges and 1 vertex to each pane, which does not affect our time bounds.

In $P \cup \{s\}$, the degree of any set is equal to the degree of the corresponding (uncontracted) set in G. While finding the extreme set families $\mathcal{F}_i$, we already compute the degrees (in G) of all the sets in $\mathcal{F}_i$. So we continue to work with these quantities.

First consider a window at weight W_i and its resulting laminar family $\mathcal{F}_i$. All we rely on in arguing that the windows together contain all of $\mathcal{F}$ is that the W_i-window contains all extreme sets of degree between W_i/n and W_i. So we can delete from $\mathcal{F}_i$ all extreme sets whose degree is not in this range. This ensures that if the windows we considered were $W_1 < W_2 \cdots < W_{O(n)}$, then all sets in $\mathcal{F}_i$ have degree less than all sets in $\mathcal{F}_j$ for $i < j$.

To merge these windows, we rely on the following observation: any non-extreme (in G) set in $\mathcal{F}_j$ contains some extreme (in G) set of lower or equal degree, which must therefore appear in some $\mathcal{F}_i$ with $i < j$. So we build our family $\mathcal{F}$ according to the following algorithm. Start with $\mathcal{F}$ empty. Working in increasing order of i, merge $\mathcal{F}_i$ into $\mathcal{F}$. For each set in $\mathcal{F}_i$ (again considered in increasing order of degree), add it to $\mathcal{F}$ if it does not contain any set already in $\mathcal{F}$. By induction, when we add $\mathcal{F}_i$, set $\mathcal{F}$ will contain all G-extreme sets with degrees less than those in $\mathcal{F}_i$. By our statement at the start of the paragraph, this means that only the G-extreme sets in $\mathcal{F}_i$ will be added to $\mathcal{F}$. So at the end, $\mathcal{F}$ will contain all extreme sets of G.

To implement the containment check efficiently, maintain $\mathcal{F}$ as a laminar family. Given a set $X \in \mathcal{F}_i$, use $O(n)$ time to see whether X contains any set in the laminar family (by working up from the leaves of the family). If X does not contain any such set, use $O(n)$ time to add it to the laminar family.

Since the total metavertex count of the windows is $O(n)$, the total number of sets in the laminar families on the windows is also $O(n)$. Since we use $O(n)$ time to merge each set, the total time needed to carry out the mergers is $O(n^2)$.

This completes our discussion of the strongly polynomial algorithm. We have shown how to build windows containing all G-extreme sets in $O(n)$ time, and have shown how to merge the extreme set families of these windows to generate the extreme set family for G in $O(n^2)$ time. It follows that the dominant factor in the time to find extreme sets for G is the time spent finding extreme sets in the windows, which we have already argued is $O(n^2 \log^5 n)$.

6 Conclusion

We have proposed randomized $\tilde{O}(n^2)$-time edge augmentation and extreme sets algorithms. Our edge augmentation algorithm runs faster than the best known deterministic one [NI96] by a factor of $\tilde{\Omega}(m/n)$. While the previous best extreme sets algorithm [NGM90] (in the weighted graph case) finds extreme sets as certain sets naturally defined by a Gomory–Hu tree [GH61], our results show that it is most likely easier to find extreme sets than a Gomory–Hu tree (and seems even easier than finding a single max-flow). Our algorithm also solves the degree-constrained version of the augmentation problem.

An obvious question is whether our extreme sets algorithm, which is the bottleneck both computationally and in terms of descriptive complexity, can be simplified. Though the ideas of the algorithm are relatively simple, the implementation is quite baroque. While some logarithmic factors in the running time of our algorithm may be unnecessary, it appears unlikely that major improvements are possible over the $\tilde{O}(n^2)$ time bound for this algorithm. However, a different approach might work. Karger [Kar96] describes an $\tilde{O}(m)$ time algorithm for finding a minimum cut, but it is not even guaranteed that this single minimum cut is a c-extreme set. Perhaps the algorithm could be modified for extreme sets. Another question is whether finding extreme sets can be done quickly and deterministically. The same holds for augmentation given the extreme sets: it seems odd that only the very last unit of augmentation requires randomization. On the other hand, at present simply checking the connectivity of a graph in $o(mn)$ time requires randomization.

7 Acknowledgment

Thanks to the referees for their very careful reading and helpful comments.

References

[ACM94] ACM. *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing*. ACM Press, May 1994.

[ACM96] ACM. *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing*. ACM Press, May 1996.

[Ben94] András A. Benczúr. Augmenting undirected connectivity in $\mathcal{RNC}$ and in randomized $\tilde{O}(n^3)$ time. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing* [ACM94], pages 658–667. Journal version in preparation.

[BK96] András A. Benczúr and David R. Karger. Approximate s–t min-cuts in $\tilde{O}(n^2)$ time. In *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing* [ACM96], pages 47–55.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, April 1993.

[CS89] G-P. Cai and Y-G. Sun. The minimum augmentation of any graph to a k-edge-connected graph. *Networks*, 19:151–172, 1989.

[DRT92] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[Fra92] Andras Frank. Augmenting graphs to meet edge connectivity requirements. *SIAM Journal on Discrete Mathematics*, 5(1):25–53, 1992. A preliminary version appeared in Proceedings of the 31^{st} Annual Symposium on the Foundations of Computer Science.

[Fra93] Andras Frank. Applications of submodular functions. In K. Walker, editor, *Surveys in Combinatorics*, number 187 in London Math. Society Lecture Notes, pages 85–36. Cambridge, 1993.

[Gab91a] Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the* 32^{nd} *Annual Symposium on the Foundations of Computer Science* [IEE91], pages 812–821.

[Gab91b] Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. Technical Report CU–CS–545–91, University of Colorado Department of Computer Science, 1991.

[Gab94] Harold N. Gabow. Efficient splitting off algorithms for graphs. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing* [ACM94], pages 696–705.

[GGP+94] Michel X. Goemans, Andrew Goldberg, Serge Plotkin, David Shmoys, Éva Tardos, and David Williamson. Improved approximation algorithms for network design problems. In Sleator [Sle94], pages 223–232.

[GH61] Ralph E. Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society of Industrial and Applied Mathematics*, 9(4):551–570, December 1961.

[GT85] Harold N. Gabow and Robert E. Tarjan. A linear time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[GW97] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, 1997.

[IEE90] IEEE. *Proceedings of the* 31^{st} *Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, October 1990.

[IEE91] IEEE. *Proceedings of the* 32^{nd} *Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, October 1991.

[Kar96] David R. Karger. Minimum cuts in near-linear time. In *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing* [ACM96], pages 56–63.

[KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, July 1996. Preliminary portions appeared in SODA 1992 and STOC 1993.

[Lov93] Làszló Lovász. *Combinatorial Problems and Exercises*. North-Holland, Amsterdam, 2nd edition, 1993.

[Mad78] W. Mader. A reduction method for edge-connectivity in graphs. *Annales Discr. Math.*, 3:145–164, 1978.

[NGM90] Dalit Naor, Dan Gusfield, and Charles Martel. A fast algorithm for optimally increasing the edge connectivity. In *Proceedings of the* 31^{st} *Annual Symposium on the Foundations of Computer Science* [IEE90], pages 698–707.

[NI96] H. Nagamochi and T. Ibaraki. Deterministic $\tilde{O}(nm)$-time edge splitting in undirected graphs. In *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing* [ACM96], pages 64–73.

[Sle94] Daniel D. Sleator, editor. *Proceedings of the* 5^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, January 1994.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, December 1988.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[WN87] T. Watanabe and A. Nakamura. Edge connectivity augmentation problems. *Journal of Computer and System Sciences*, 53:96–144, 1987.