An Efficient Approximation Algorithm for the Survivable Network Design Problem*

Harold N. Gabow † University of Colorado

Michel X. Goemans ‡ M.I.T.

David P. Williamson § IBM Watson

Abstract

The survivable network design problem is to construct a minimum-cost subgraph satisfying certain given edge-connectivity requirements. The first polynomial-time approximation algorithm was given by Williamson et al. [20]. This paper gives an improved version that is more efficient.

Consider a graph of n vertices and connectivity requirements that are at most k. Both algorithms find a solution that is within a factor $2k-1$ of optimal for $k \geq 2$ and a factor 2 of optimal for $k=1$. Our algorithm improves the time from $O(k^3n^4)$ to $O(k^2n^2 + kn^2\sqrt{\log\log n})$. Our algorithm shares features with [20] but also differs from it at a high level, necessitating a different analysis of correctness and accuracy; our analysis is based on a combinatorial characterization of the "redundant" edges.

Several other ideas are introduced to gain efficiency. These include a generalization of Padberg and Rao's characterization of minimum odd cuts, use of a representation of all minimum (s,t) cuts in a network, and a new priority queue system. The latter also improves the efficiency of the approximation algorithm of Goemans and Williamson [8] for constrained forest problems such as minimum-weight matching, generalized Steiner trees and others.

Key words: network design, approximation algorithm, Steiner tree

1 Introduction

In the *survivable network design problem (SNDP)* we are given an undirected graph $G=(V,E)$, a non-negative cost c_e for every edge e, and a non-negative *connectivity requirement* r_{ij} for every (unordered) pair of vertices i,j. We must find a minimum-cost subgraph in which each pair of vertices i,j is joined by at least r_{ij} edge-disjoint paths. SNDP is NP-complete since the Steiner tree problem is a special case. An important practical application arises in the design of fiber-optic telecommunication networks. In that context the most interesting case is when r_{ij} is of the form $\min(r_i, r_j)$ for some vector $(r_i)_{i\in V}$ of connectivity types, with each $r_i \in \{0,1,2\}$. Vertices with $r_i = 1$ represent customers; vertices with $r_i = 2$

*A preliminary version of this paper has appeared in the *Proceedings of the Third Mathematical Programming Society Conference on Integer Programming and Combinatorial Optimization*, pages 57-74, 1993.

†Address: University of Colorado, Campus Box 430, Boulder, CO 80309. Email: hal@cs.colorado.edu. Research supported in part by NSF Grant No. CCR-9215199 and AT&T Bell Laboratories.

‡Address: M.I.T. Room 2-382, 77 Massachusetts Ave., Cambridge, MA, 02139. Email: goemans@math.mit.edu. Research supported in part by Air Force contracts AFOSR-89-0271 and F49620-92-J-0125 and DARPA contracts N00014-89-J-1988 and N00014-92-J-1799.

§Address: IBM TJ Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. Email: dpw@watson.ibm.com. This research was performed while the author was a graduate student at MIT. Research supported by an NSF Graduate Fellowship, Air Force contract F49620-92-J-0125, DARPA contracts N00014-89-J-1988 and N00014-92-J-1799, and AT&T Bell Laboratories.

Problem	requirements	performance guarantee	time	reference
Steiner tree	$r_i \in \{0,1\}$	2 11/6 16/9	$O(n^2)$ $O(n^{3.5})$ $O(n^5)$	Mehlhorn [15] Zelikovsky [21] Berman and Ramaiyer [2]
Generalized Steiner tree	$r_{ij} \in \{0,1\}$	2	$O(n^2 \log n)$	Agrawal et al. [1] Goemans and Williamson [8]
2-edge connected subgraph	$r_i = 2$	3	$O(n^2)$	Frederickson and Ja'Ja' [3]
k-edge connected subgraph	$r_i = k$	2	$O(kn^3 \log n)$	Khuller and Vishkin [12]
Generalized Steiner 2-edge-connected subgraph	$r_{ij} \in \{0,2\}$	3	$O(n^2 \log n)$	Klein and Ravi [13]

Table 1: Previous work on special cases of SNDP

represent switching stations that need to be protected from single edge failures, while vertices with $r_i = 0$ are optional sites. For a thorough discussion of the problem and a survey of existing results, the reader is referred to the survey paper by Grötschel et al. [10].

A heuristic that gives a solution guaranteed to be within a factor $\alpha \geq 1$ of optimal has a *performance guarantee* of α. If in addition the heuristic runs in polynomial time it is called an *(α)-approximation algorithm.* The first approximation algorithm for the general SNDP was developed by Williamson, Goemans, Mihail, and Vazirani [20]. Before this no approximation algorithm was known even for the case $r_{ij} = \min(r_i, r_j)$, $r_i \in \{0,1,2\}$. The previously known special-case SNDP approximation algorithms are listed in Table 1. Throughout this paper n and m denote the number of vertices and edges of the given graph. For simplicity the time bounds in the table assume dense graphs, $m = \Theta(n^2)$. Agrawal et al. [1] and Goemans and Williamson [8], building on the work of Goemans and Bertsimas [7], have described an approximation algorithm for a variant of SNDP in which an edge can be selected several times. This version of the problem, however, appears to be easier to approximate.

This paper presents an improved version of the approximation algorithm of [20]. Both algorithms apply to a family of integer programs defined by *proper* functions f. A function $f : 2^V \rightarrow N$ is called *proper* if

- [Symmetry] $f(S) = f(V - S)$ for all $S \subseteq V$; and
- [Maximality] If A and B are disjoint, then $f(A \cup B) \leq \max\{f(A), f(B)\}$.

We will also assume that $f(\emptyset) = 0$ for all proper functions f. Given an undirected graph $G = (V, E)$, the family of integer programs that can be approximated is formulated by

$$
\begin{array}{llll}
 & \text{Min} & \sum_{e \in E} c_e x_e & \\
 & \text{subject to:} & & \\
(IP) & & x(\delta(S)) \geq f(S) & S \subset V \\
 & & x_e \in \{0,1\} & e \in E
\end{array}
$$

where $\delta(S)$ denotes the *coboundary* of S (the set of edges with exactly one endpoint in S) and $x(F) = \sum_{e \in F} x_e$. SNDP is given by the proper function $f(S) = \max_{i \in S, j \notin S} r_{ij}$. The algorithms of Williamson et al. [20] and this paper both achieve this result:

Theorem 1.1 If the proper function f takes only l non-zero distinct values $0 = \rho_0 < \rho_1 < \rho_2 < \ldots < \rho_l$, then there is an approximation algorithm for (IP) with performance guarantee at most

$$2\sum_{i=1}^{l} \mathcal{H}(\rho_i - \rho_{i-1}),$$

where $\mathcal{H}$ is the harmonic function $\mathcal{H}(k) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$. If $\rho_1 = 1$ and $l \geq 2$ then the performance guarantee improves by one unit.

In the rest of this paper, $f_{\max}$ denotes the largest requirement, $f_{\max} = \max_S f(S)$. Thus the performance guarantee of Theorem 1.1 is at most $2f_{\max}$ or, more precisely, $2f_{\max} - 1$ whenever $f_{\max} \geq 2$. Furthermore, both algorithms prove that the factor $2f_{\max} - 1$ (or 2 if $f_{\max} = 1$) bounds the gap between (IP) and its linear programming relaxation. We will assume that we are dealing with simple graphs so that $f_{\max} \leq n$, although our results can be easily extended to the case of multigraphs.

The main result of this paper is an algorithm that achieves the accuracy of Theorem 1.1 efficiently: the time bound of Williamson et al. [20] for SNDP, $O(f_{\max}^3 n^4)$, is improved to $O(f_{\max}^2 n^2 + f_{\max} n^2 \sqrt{\log\log n})$).

To state our specific contributions, we first briefly sketch the approximation algorithm. It proceeds in $f_{\max}$ phases. Each phase finds a low-cost augmentation to the current solution. Let F_{p-1} denote the edges chosen in the first $p-1$ phases, and let $\delta_A(S)$ denote $A \cap \delta(S)$ for $A \subseteq E$. Then in phase p we find a set of edges $F \subseteq E - F_{p-1}$ such that whenever $f(S) \geq p$ and $|\delta_{F_{p-1}}(S)| = p-1$, then $|\delta_F(S)| \geq 1$. We then set $F_p = F_{p-1} \cup F$ so that we maintain the invariant that $|\delta_{F_p}(S)| \geq \min(f(S), p)$ for all subsets S. Thus the final set $F_{f_{\max}}$ is a feasible solution to the integer program (IP).

The augmentation in each phase can be viewed as finding a low-cost solution to the integer program

$$\begin{array}{llll} & \text{Min} & \sum_{e \in E_h} c_e x_e & \\ & \text{subject to:} & & \\ (IP_h) & & x(\delta(S)) \geq h(S) & S \subset V \\ & & x_e \in \{0,1\} & e \in E_h, \end{array}$$

where $E_h = E - F_{p-1}$ and $h(S) = 1$ iff $f(S) \geq p$ and $|\delta_{F_{p-1}}(S)| = p-1$, and $h(S) = 0$ otherwise. Williamson et al. [20] showed how to find such a solution if the function h is *uncrossable*; i.e., if $h(A) = h(B) = 1$, then either $h(A-B) = h(B-A) = 1$, or $h(A \cup B) = h(A \cap B) = 1$. They then showed that the function h given above is uncrossable. Their algorithms for finding such a solution works in two steps. The first step uses a primal-dual algorithm which constructs a set of edges F that is a feasible solution to (IP_h) while simultaneously constructing a feasible solution to the dual of the linear programming relaxation of (IP_h). The second step of the algorithm is a "clean-up step." It removes certain unnecessary edges from F.

We introduce an alternate algorithm to find low-cost solutions to (IP_h) for uncrossable functions. The algorithms of Williamson et al. [20] and this paper use the same procedure for the first step to initialize F, but differ in the clean-up step. The clean-up step is crucial, as no finite performance guarantee can be achieved without a clean-up step. For example, on the shortest s-t path problem their algorithm emulates Dijkstra's algorithm, and the edges of the shortest path tree not on the s-t path must be removed to guarantee low cost. In [20] both steps of the algorithm use the same amount of time. The clean-up step is the bottleneck against speeding this up; it checks the feasibility of $O(n)$ edge sets. We circumvent this problem by giving a combinatorial characterization of a set of edges which may be safely removed. These edges can be easily identified by gathering information in the first step of the algorithm. The characterization leads to a new, more efficient clean-up step and a different proof of the performance guarantee of the algorithm. Our clean-up step is no longer the bottleneck even in the improved algorithm: it uses $O(n)$ time per phase.

The remaining contributions of this paper concern the efficient implementation of the first step of a phase. There are several new ideas. To decide which edges to add to F requires identifying certain "active sets." The high-level algorithm does not indicate how to do this in polynomial time. Williamson et al. [20] show how to find the active sets by solving $O(n^2)$ network flow problems. We identify active sets more efficiently using two ideas from flow theory. First we show the Gomory-Hu cut tree gives a characterization of a feasible solution to (IP). This generalizes Padberg and Rao's characterization of a minimum T-cut in terms of the Gomory-Hu tree [16], since T-cuts correspond to a proper function. Next we combine the Gomory-Hu tree with the representation of Picard and Queyranne for all minimum (s,t) cuts of a network [17]. This allows efficient identification of the active sets. Combining these two ideas has been previously suggested by Gusfield and Naor [11]. We gain further efficiency by showing that the special structure of SNDP allows faster location of the active sets in the representation.

As the last ingredient in an efficient algorithm, we improve the implementation of the rule for selecting the next edge to add to F. This edge-choice rule (also used in [20]) is similar to the rule in the algorithm of Goemans and Williamson [8]. Goemans and Williamson present approximation algorithms for minimum-weight matching (with the triangle inequality), T-joins, Steiner trees and generalized Steiner trees, and a number of other problems. All these algorithms have performance guarantee 2 and run in time $O(n^2 \log n)$.Our implementation improves this time bound to $O(n(n + \sqrt{m \log\log n}))$. The idea of the implementation is to avoid work on irrelevant edges. Independently, Klein [14] gives a $O(n\sqrt{m}\log n)$ time implementation using a new data structure.

Putting the pieces together gives the following results. For SNDP with requirements $r_{ij} \leq f_{\max}$ the performance guarantee is $2f_{\max} - 1$ for $f_{\max} \geq 2$ and 2 for $f_{\max} = 1$. For $f_{\max} = O(1)$ the running time is $O(n(n + \sqrt{m \log\log n}))$; more generally the time is $O(f_{\max}^2 n^2 + f_{\max} n\sqrt{m \log\log n})$. We also give a time bound for general proper functions f (assuming an oracle for f); this bound is $O(f_{\max}^2 n^3 + f_{\max} n^2 \rho)$ where ρ is the time taken by an oracle that computes the function f. More precise time bounds are given in Section 5.

The rest of this paper is organized as follows. Section 2 presents the high-level algorithm. Section 3 proves it finds a feasible solution and Section 4 proves the performance guarantee. Section 5 gives the implementation details, in a number of subsections.

2 The Main Algorithm

This section summarizes the overall algorithm, and our new algorithm for finding low-cost solutions to (IP_h) for uncrossable functions; the algorithms are given in Figures 1 and 2 respectively. We call the algorithm for uncrossable functions UNCROSSABLE. To make the algorithm somewhat simpler to understand, a simulated run of UNCROSSABLE is represented in Figure 5 with comments later in the text.

Recall the overall structure of the algorithm, as given in Figure 1: there are $f_{\max} = \max_S f(S)$ *phases* indexed by $p = 1, \ldots, f_{\max}$. Phase p produces a set F_p such that $|\delta_{F_p}(S)| \geq \min(f(S), p)$. $F_{f_{\max}}$ is feasible for (IP). In phase p we call UNCROSSABLE with the edge set $E_h = E - F_{p-1}$ and with the function $h(S) = 1$ iff $|\delta_{F_{p-1}}(S)| = p - 1$ and $f(S) \geq p$. Recall that we call a function h uncrossable if whenever $h(A) = h(B) = 1$, then either $h(A - B) = h(B - A) = 1$, or $h(A \cup B) = h(A \cap B) = 1$. This function was

Input: An undirected graph $G = (V, E)$, edge costs $c_e \geq 0$, a proper function f, and $f_{\max} = \max_S f(S)$
Output: A set of edges $F_{f_{\max}}$ feasible for (IP)

$F_0 \leftarrow \emptyset$
For $p \leftarrow 1$ to $f_{\max}$
    Comment: Begin phase p.
    $h_p(S) \leftarrow \begin{cases} 1 & \text{if } f(S) \geq p \text{ and } |\delta_{F_{p-1}}(S)| = p-1 \\ 0 & \text{otherwise} \end{cases}$
    $E_p \leftarrow E - F_{p-1}$
    $F' \leftarrow \textsc{Uncrossable}(V, E_p, c, h_p)$
    $F_p \leftarrow F_{p-1} \cup F'$
    Comment: End phase p.
Output $F_{f_{\max}}$

Figure 1: The main algorithm.

proven to be uncrossable by Williamson et al. [20].

For the rest of this paper, we will assume that the uncrossable functions in UNCROSSABLE are symmetric: that is, $h(S) = h(V - S)$ for all $S \subseteq V$. Since proper functions f are symmetric, the uncrossable function used in phase p is always symmetric. Details about using UNCROSSABLE with functions that are not symmetric can be found in Williamson [19]; some proofs and arguments are slightly different.

As discussed in the introduction, the algorithm UNCROSSABLE has two steps. The first step produces a set of edges F and consists of a number of *iterations*, each iteration consisting of an *edge selection* and *edge addition* step. The second step is the *clean-up* step. It removes edges from F. We now describe the two steps in detail. The first step initializes F to be empty. At any point, a set S is called *violated* if $|\delta_F(S)| < h(S)$; that is, if $h(S) = 1$ but $\delta_F(S) = \emptyset$. A set S is *active* if it is a minimal violated set (minimal with respect to inclusion). The first step maintains the family of active sets $\mathcal{C}$. Note that in terms of our algorithm for proper functions, a set S is violated in the call to UNCROSSABLE in phase p if $|\delta_{F_{p-1} \cup F}(S)| = p - 1$ and $f(S) \geq p$.

The violated sets have the following property, which was proven in [20].

Lemma 2.1 [Williamson et al. [20]] If A and B are violated sets at any point in UNCROSSABLE, then either $A \cap B$ and $A \cup B$ are violated or $A - B$ and $B - A$ are violated.

Proof: Since A and B are violated, we know that $h(A) = h(B) = 1$. By the properties of uncrossable functions, either $h(A \cup B) = h(A \cap B) = 1$ or $h(A - B) = h(B - A) = 1$. Suppose the former is true. By the submodularity of δ, we know that $|\delta_F(A)| + |\delta_F(B)| \geq |\delta_F(A \cap B)| + |\delta_F(A \cup B)|$. Since $|\delta_F(A)| = |\delta_F(B)| = 0$, it must be the case that $|\delta_F(A \cap B)| = |\delta_F(A \cup B)| = 0$, and $A \cap B$ and $A \cup B$ are violated.

If the latter case is true, then a similar argument follows since it is also the case that $|\delta_F(A)| + |\delta_F(B)| \geq |\delta_F(A - B)| + |\delta_F(B - A)|$. ■

An immediate corollary of this lemma is that all active sets are disjoint. Another corollary is the following. We say a set A *crosses* B if $A \cap B \neq \emptyset$, but $A \not\subseteq B$ and $B \not\subseteq A$.

Corollary 2.2 No violated set crosses any active set.

In each iteration, an edge $e \in E_h$ is selected from the coboundary of some currently active set C and e is added to F. The edge e can be in the coboundary of either 1 or 2 active sets; e is a *1-edge* in the

Input: An undirected graph $G = (V, E_h)$, edge costs $c_e \geq 0$, and an uncrossable function h
Output: A set of edges F' feasible for (IP_h)

$F \leftarrow \emptyset$
$i \leftarrow 0$
$d(v) \leftarrow 0$ for all $v \in V$
Comment: Implicitly set $y_S \leftarrow 0$ *for all* $S \subset V$
$\mathcal{C} \leftarrow$ all active sets C (minimal violated sets).
While $|\mathcal{C}| > 0$
    $i \leftarrow i + 1$
    Comment: Begin iteration i.
    Comment: Edge selection step.
    For all $v \in C \in \mathcal{C}$, increase $d(v)$ uniformly by ϵ until some edge $e_i = (u, v) \in E_h$ satisfies $d(u) + d(v) = c_{uv}$ for $e_i \in \delta(C)$ of some $C \in \mathcal{C}$.
    Comment: Implicitly set $y_C \leftarrow y_C + \epsilon$ *for all* $C \in \mathcal{C}$.
    $F \leftarrow F \cup \{e_i\}$
    Comment: Edge addition step.
    Update $\mathcal{C}$
    Comment: End iteration i.
Comment: Edge clean-up step.
$F' \leftarrow F$
Let all sets that were active during this phase be unmarked. Mark the set V.
For $j \leftarrow i$ downto 1
    Comment: $C(e), A(e)$, *and special edges are defined in the text.*
    If e_j is special and $C(e_j)$ is unmarked and $\delta_{F'}(A(e_j)) \supseteq \delta_{F'}(C(e_j))$ then
        $F' \leftarrow F' - \{e_j\}$
        Mark $C(e_j)$
Output F'

Figure 2: The UNCROSSABLE algorithm.

first case and a *2-edge* in the second case. The edge selection step chooses e, based on values of certain dual variables. The edge addition step adds e to F. This necessitates updating the family of active sets $\mathcal{C}$, since the active set(s) having e in their coboundary are no longer violated. Moreover a new active set may be created. Such a new active set must contain both ends of e. The first step terminates when no active sets remain.

Another consequence of Lemma 2.1 is the following. A *laminar* family of sets $\mathcal{S}$ is one such that if $A, B \in \mathcal{S}$ and $A \cap B \neq \emptyset$, then either $A \subseteq B$ or $B \subseteq A$; that is, if $A, B \in \mathcal{S}$, then A and B do not cross.

Corollary 2.3 Let $\cup\mathcal{C}$ denote the set of all active sets formed over all iterations. Then $\cup\mathcal{C}$ is a laminar family.

The idea behind the edge selection step is to implicitly maintain a feasible solution y to the dual of the linear programming relaxation of (IP_h) formed by replacing the constraints $x_e \in \{0, 1\}$ with $x_e \geq 0$. This dual is as follows:

$$\text{Max} \quad \sum_{S \subset V} h(S) y_S$$

subject to:

$$(D_h) \qquad \sum_{S: e \in \delta(S)} y_S \leq c_e \qquad e \in E_h,$$

$$y_S \geq 0 \qquad\qquad S \subset V.$$

Initially $y = 0$. Each iteration of the algorithm implicitly increases y_C for each active set $C \in \mathcal{C}$ by a value ϵ which is as large as possible without violating the inequality $\sum y_S \leq c_e$ for any edge $e \in E_h$. This makes an inequality tight for some edge $e \in E_h$ in the coboundary of some active set; this edge e is then chosen to be added to F. This feasible dual solution is used to prove the performance guarantee of the algorithm.

Instead of keeping track of the dual solution y, our algorithm only maintains a variable $d(u) = \sum_{S:u\in S} y_S$ for each $u \in V$. Then increasing y_C for each active set $C \in \mathcal{C}$ by the largest ϵ possible without violating $\sum y_S \leq c_e$ for any edge $e \in E_h$ becomes equivalent to increasing the variables $d(u)$ for $u \in C \in \mathcal{C}$ by the largest ϵ possible without violating $d(u) + d(v) \leq c_{uv}$ for $e = (u,v) \in E_h$, e in the coboundary of some active C. Thus if $a(u) = 1$ if u is in an active set, $a(u) = 0$ otherwise, then

$$\epsilon = \min_{(i,j)\in E_h : a(i)+a(j)>0} \frac{c_{ij} - d(i) - d(j)}{a(i) + a(j)}.$$

Before we define the new edge clean-up stage, we define some notation and we try to provide an intuitive feel for the concepts involved. Suppose for a moment that the uncrossable function h under consideration is such that no new active sets are ever created: we have some initial collection $\mathcal{C}$ of active sets, and in each iteration we select some edge in the coboundary of at least one, and at most two, active sets from $\mathcal{C}$. Once an edge in the coboundary of an active C is selected, then C is no longer violated, and hence no longer active. Thus in this case, there will be at most $|\mathcal{C}|$ iterations of the edge addition step. Let F be the set of edges added during these iterations.

Still assuming that no new active sets are created, F is a forest in the graph with each active set contracted to a single node plus some other nodes (see Figure 3). Consider also how each tree of this forest "grows" as edges of F are added. Each currently existing tree adds a new node by adding a 1-edge which has one endpoint in a currently active set (the new node) and the other in a set that was active in some previous iteration (a node in the growing component); see Figure 4, in which edge e_i was added in the ith iteration. Notice that we never add edges whose endpoints are both in previously active sets, and thus we never link two trees. From this observation, it is not hard to see that 2-edges must always start growing a new tree (e.g. e_1). A new tree can also start growing by a 1-edge which has one endpoint in a currently active set and the other endpoint which is not in any current or previously active set (e.g. e_3).

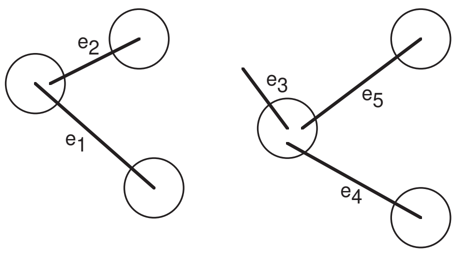

Figure 3: A forest on active sets. Each circle represents an active set before any edge was added.

The edges that particularly interest us for the new edge clean-up step are 1-edges e such that the edges added after e form a tree spanning precisely the sets active immediately before the addition of e. We call these edges *special edges*, and we define them more formally below. In Figure 3, e_3 and e_5 are special edges. The other edges are not special edges: for example, e_4 is not a special edge because e_5 contains

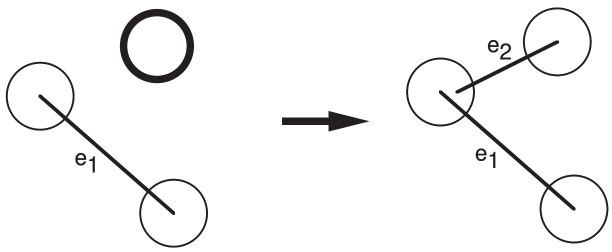

Figure 4: Growing a tree of the forest. The thin circles represent previously active sets; the thick circle represents an active set.

an endpoint in a set that is not active just before e_4 is added. We will show that special edges have a nice combinatorial structure such that we can remove some of them and simultaneously ensure a good performance guarantee and maintain feasibility.

To generalize this concept to the case where new active sets are formed, we partition the edges and the active sets. Let $\cup\mathcal{C}$ be the collection of all active sets formed over all iterations of the algorithm, plus the set V. Recall from Corollary 2.3 that $\cup\mathcal{C}$ is laminar. We define a tree $\mathcal{T}$ based on $\cup\mathcal{C}$, with one vertex v_C for each $C \in \cup\mathcal{C}$. Thus we make v_C a parent of v_D in the tree $\mathcal{T}$ if C is the smallest set in $\cup\mathcal{C}$ that properly contains D. Let $\mathcal{D}(C)$ denote the collection of sets corresponding to the children of v_C in $\mathcal{T}$. The collection $\mathcal{D}(C)$ can be thought of as an equivalence class of active sets. For edge e, let $C(e)$ denote the smallest set $C \in \cup\mathcal{C}$ that contains both endpoints of e. The set of all edges of F for which $C(e) = C$ is denoted F_C. The edges in F_C can be thought of as an equivalence class of the edges of F. The behavior of the edges F_C on the active sets in $\mathcal{D}(C)$ will now be as in the case above in which no new active sets are formed.

Let $\mathcal{A}(e)$ denote the sets in $\mathcal{D}(C(e))$ that are active just before edge e is selected. We make the following observations.

Observation 2.4 For any vertex v_C in $\mathcal{T}$, all edges in F_C must be selected before any edge in $\delta(C)$ is selected.

Proof: Suppose not, and at some iteration an edge in $\delta(C)$ is selected before some edge in F_C. Since not all edges in F_C have been selected, some $C' \subset C$ must be active. But since an edge in $\delta(C)$ is selected, C will never become an active set, a contradiction. ■

Observation 2.5 For the first edge e in F_C selected, $\mathcal{A}(e) = \mathcal{D}(C(e))$.

Proof: If some set $S \in \mathcal{D}(C(e))$ is not active, then some edge e' has been selected from $\delta(S)$ prior to edge e. If $C(e') \neq C(e)$, then it must be the case that $e' \in \delta(C(e))$, which contradicts the observation above. ■

We can now formally define the special edges. Say that an edge set H forms a spanning tree on a family of disjoint vertex sets $\{C_i\}_{i=1}^k$ if each endpoint of each edge in H lies in one of the C_i and H forms a spanning tree on the graph with each C_i contracted to a node. Then e is *special* if it is a 1-edge, and the edges added to $F_{C(e)}$ after e form a spanning tree on the sets in $\mathcal{A}(e)$.

We illustrate the concept in its full generality in Figure 5. Frames 1–9 correspond to the edge addition stage, frames 13–16 to the clean-up step, while the final solution is depicted in frame 11. In frames 1–9, the edges correspond to the edges of F while the rounded boxes represent the active sets. In the figure, only edges e_4 and e_8 are 2-edges, the others being 1-edges. The special edges in the figure are e_1, e_5, e_6 and e_7. The edge e_2 is not special because e_4 is a 2-edge (and hence forms a new tree on the sets of $\mathcal{A}(e_2)$). The edge e_3 is not special because e_5 has an endpoint not in one of the sets of $\mathcal{A}(e_3)$.

The edge clean-up step is given in the algorithm in Figure 2. It scans the edges of F in the reverse order of their selection in the edge addition stage. Let $A(e)$ denote the union of the sets in $\mathcal{A}(e)$; that is, $A(e) = \bigcup_{C \in \mathcal{A}(e)} C$. The edge clean-up step removes edge e from F if e is special, no other edge of $F_{C(e)}$ has already been removed, and all remaining edges of F in $\delta(C(e))$ are also in $\delta(A(e))$. It does not remove e if $C(e) = V$. To keep track of whether an edge of $F_{C(e)}$ has already been removed, the algorithm marks $C(e)$ if edge e is removed. We illustrate the clean-up step in frames 13–16 of Figure 5. Recall that the special edges are e_1, e_5, e_6 and e_7. Frames 13, 14, 15 and 16 correspond to the situation just before the possible removal of edge e_7, e_6, e_5 and e_1 respectively. In the frame corresponding to e_i, the vertices in $A(e_i)$ are represented in white. Edge e_7 is removed since $e_8 \in \delta(A(e_7))$. Edge e_6 is not removed since $e_8 \notin \delta(A(e_6))$. Although $e_7 \notin \delta(A(e_5))$, edge e_5 is removed since e_7 was previously removed. Edge e_1 is not removed since $e_5 \in C(e_1)$ has already been removed. The resulting forest is represented in frame 11.

3 Correctness

This section proves the set F' returned by UNCROSSABLE is feasible for (IP_h).

Theorem 3.1 The edge set F' remaining after the edge clean-up step is a feasible solution for (IP_h).

Proof: The edge addition stage terminates with no active sets, and thus no violated sets. So we need only prove that the clean-up step maintains feasibility. Assume it does not. Suppose F' is feasible for (IP_h) but $F' - e$ is not, and the clean-up step removes e from F'. The situation just before the removal of e is illustrated in Figure 6. Let $C = C(e)$. By the definition of the clean-up step, $C \neq V$. Let S be a set violated by $F' - e$. Let I_e be the iteration in which edge e is added. The set S was violated in iteration I_e because of the ordering of the clean-up step.

Notice that all sets in $\mathcal{A}(e)$ must also be violated in iteration I_e, by the definition of $\mathcal{A}(e)$. By Corollary 2.2, S does not cross any set in $\mathcal{A}(e)$. In fact we can show that either $A(e) \subseteq S$ or $A(e) \cap S = \emptyset$: because no edge of F_C has been removed so far in the clean-up process, the edges of F_C added after e form a spanning tree on the family $\mathcal{A}(e)$. Thus if S crosses $A(e)$ but not any set in $\mathcal{A}(e)$, it would intersect an edge of F_C added after e, contradicting the fact that S is violated.

We now assume that $A(e) \subseteq S$: if $A(e) \cap S = \emptyset$, then by the symmetry of the uncrossable function h, $V - S$ is also violated and we replace S with $V - S$. Since $C \neq V$, the set C, as well as S, is violated in iteration I_e. By Lemma 2.1, either $C - S$ and $S - C$ are violated or $C \cap S$ and $C \cup S$ are violated in iteration I_e. However $C - S$ cannot contain an active set, so it is not violated. Thus $C \cup S$ is violated in iteration I_e. Since it was not violated before e was removed, F' contains an edge $\tilde{e}$ with exactly one endpoint in $V - (C \cup S)$. The edge $\tilde{e}$ is not in the coboundary of S since $\delta_{F'}(S) = \{e\}$ and $e \neq \tilde{e}$ (since e has both endpoints in C). Thus the other endpoint of $\tilde{e}$ is in $C - S$. On the one hand this implies $\tilde{e} \in \delta(C(e))$; on the other it implies $\tilde{e} \notin \delta(A(e))$. But this contradicts the removal of e in the clean-up step. ■

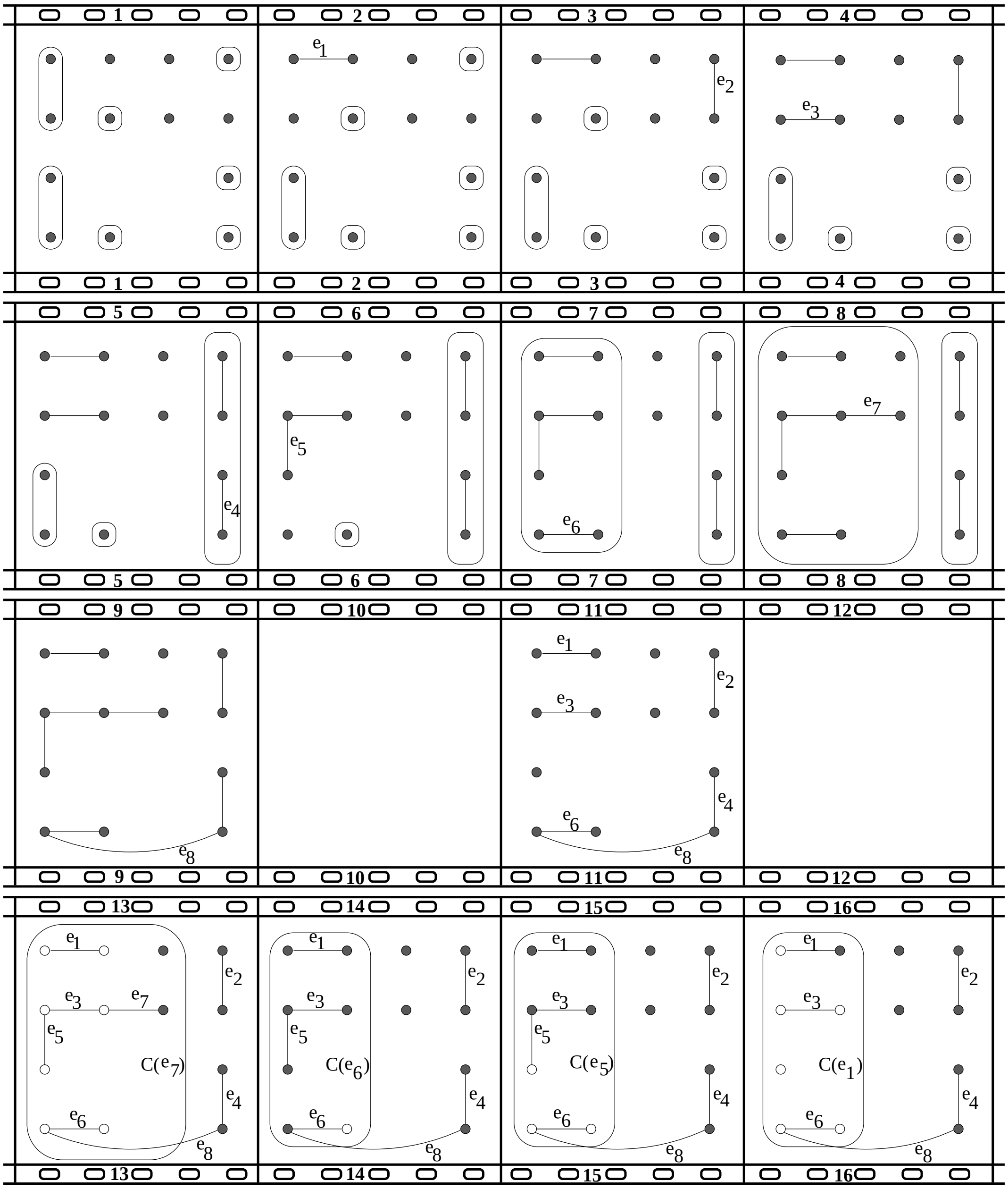

Figure 5: Simulation run of the algorithm.

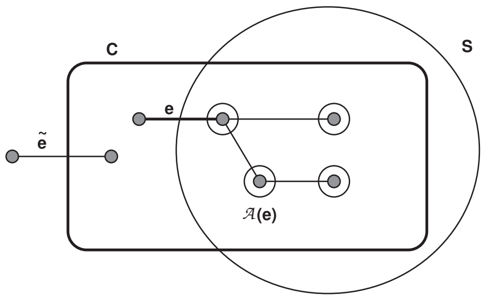

Figure 6: Notation used in the proof of Theorem 3.1.

4 Proof of the Performance Guarantee

Williamson et al. [20] show that the proof of the performance guarantee (Theorem 1.1) reduces to the proof of a particular inequality. We first explain how the inequality implies a performance guarantee (complete details are in [20]), then we show that our new clean-up step also implies the same inequality.

Let F' denote the edges returned by a call to UNCROSSABLE. The desired inequality is that at the start of any iteration of UNCROSSABLE, for $\mathcal{C}$ the family of (currently) active sets,

$$\sum_{C\in\mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 2. \tag{1}$$

Williamson et al. prove that the inequality implies that

$$\sum_S y_S|\delta_{F'}(S)| \leq 2\sum_S y_S,$$

by induction over all the iterations of UNCROSSABLE. They observe that by construction of the dual solution, for any $e \in F'$, $c_e = \sum_{S:e\in\delta(S)} y_S$, so that

$$\sum_{e\in F'} c_e = \sum_{e\in F'}\sum_{S:e\in\delta(S)} y_S = \sum_S y_S|\delta_{F'}(S)| \leq 2\sum_S y_S.$$

Consider the dual of the linear programming relaxation of (IP):

$$\begin{array}{llll}
 & \text{Max} & \displaystyle\sum_{S\subset V} f(S)y_S - \sum_{e\in E} z_e & \\
 & \text{subject to:} & & \\
(D) & & \displaystyle\sum_{S:e\in\delta(S)} y_S \leq c_e + z_e & e\in E, \\
 & & y_S \geq 0 & S\subset V, \\
 & & z_e \geq 0 & e\in E.
\end{array}$$

Given the dual variables y constructed by UNCROSSABLE in phase p, define $z_e = \sum_{S:e\in\delta(S)} y_S$ for all $e\in F_{p-1}$, and $z_e = 0$ otherwise. This gives a feasible solution to (D). Notice that by this definition,

$$\sum_{e\in E} z_e = \sum_{e\in F_{p-1}}\sum_{S:e\in\delta(S)} y_S = \sum_S |\delta_{F_{p-1}}(S)| y_S.$$

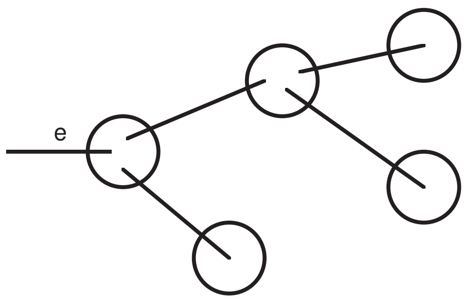

Figure 7: A bad case for the performance guarantee. Circles represent sets active just before edge e is added. The edge e is special.

Since $y_S > 0$ only when $h(S) = 1$, and $h(S) = 1$ iff $f(S) \geq p$ and $|\delta_{F_{p-1}}(S)| = p - 1$, then

$$Z^*_{IP} \geq \sum_S f(S) y_S - \sum_e z_e \geq \sum_S p y_S - (p-1) \sum_S y_S = \sum_S y_S,$$

where Z^*_{IP} is the cost of an optimal solution to (IP). Thus the cost of the edges in F' is no more than twice Z^*_{IP}. Over all $f_{\max}$ phases, then, the cost of the generated solution is no more than $2f_{\max} \cdot Z^*_{IP}$. The exact bound given in Theorem 1.1 is obtained by a slightly more careful analysis.

We now turn to the proof of inequality (1). Our proof strategy is to show that the special edges are exactly those edges whose removal can ensure the inequality. Suppose for a moment, as we did earlier, that no new active sets are created in subsequent iterations. Let e be the edge chosen in the current iteration, and suppose that e is a special edge. Recall that we informally defined a 1-edge e as special if the edges added to F after e form a tree on the sets active just before e is added. If e is added in the current iteration and is special, then $\sum_{C \in \mathcal{C}} |\delta_F(C)| = 2|\mathcal{C}| - 1$, but removing e causes the inequality (1) to be satisfied (see Figure 7). This is the central intuition of the proof; we employ it recursively in order to handle the more general case.

Theorem 4.1 Given the set of edges F' output by UNCROSSABLE, inequality (1) holds at the start of any iteration.

Proof: Recall the definition of the tree $\mathcal{T}$ from Section 2: we create a node of the tree for V and for each set active at some point during the algorithm. A node v_C corresponding to a set C is a child of v_D if D is the smallest set of the collection properly containing C. The sets corresponding to the children of v_C are denoted $\mathcal{D}(C)$. We now define the subtree $\mathcal{T}'$ of the tree $\mathcal{T}$ to contain all the nodes corresponding to sets that will be active in or after the current iteration. Thus the leaves of the tree correspond exactly to the sets in $\mathcal{C}$ in the current iteration. Define $\mathcal{E}(C)$ to be the sets corresponding to the children of an internal node v_C of $\mathcal{T}'$; that is, $\mathcal{E}(C)$ contains the sets of $\mathcal{D}(C)$ that are active in or after the current iteration. Let Y be the set of edges selected in or after the current iteration. Let $Y' = Y \cap F'$.

We will prove inequality (1) by showing for each internal node v_C of the tree $\mathcal{T}'$ that

$$\left(\sum_{S \in \mathcal{E}(C)} |\delta_{Y'}(S)| \right) - |\delta_{Y'}(C)| \leq 2|\mathcal{E}(C)| - 2. \tag{2}$$

In effect, we prove a version of the inequality for each "equivalence class" C, subtracting off the contribution to the total degree made by edges with only one endpoint in C (see Figure 8). Given that $|\delta_{Y'}(V)| = 0$,

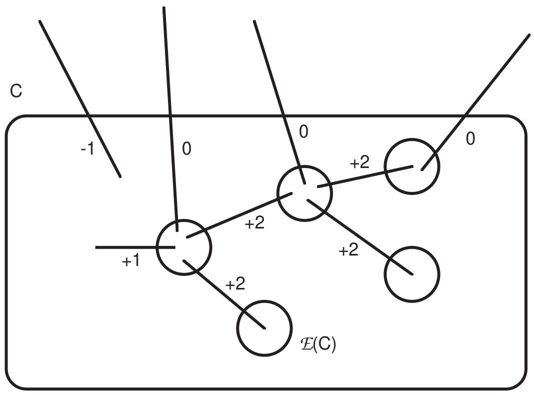

Figure 8: An illustration of inequality (2). Circles represent sets in $\mathcal{E}(C)$. Numbers are the coefficient of the edge in the left-hand side of inequality (2).

by summing this inequality over all internal nodes v_C of the $\mathcal{T}'$, we will obtain

$$\sum_{C\in\mathcal{C}} |\delta_{Y'}(C)| \leq 2|\mathcal{C}| - 2. \tag{3}$$

To see this, observe that on the left-hand side, the negative term $-|\delta_{Y'}(C)|$ for an internal node v_C is cancelled by the positive term in the inequality of the parent of v_C, leaving only the positive terms corresponding to the leaves. Similarly, on the right-hand side, the contribution of -2 for each internal node v_C is cancelled by a contribution of 2 by the parent of v_C, leaving a positive contribution of 2 for each leaf and a contribution of -2 by the node v_V. Inequality (3) implies (1) since $\delta_{Y'}(C) = \delta_{F'}(C)$ for any active set $C \in \mathcal{C}$; that is, no edge of F' in the coboundary of an active C could have been added before the current iteration.

Now we must prove inequality (2) on each internal node v_C of the tree. Let $k = |\mathcal{E}(C)|$. Let $I = F_C \cap Y$, let $\Phi = \sum_{S\in\mathcal{E}(C)} |\delta_I(S)|$, and let J be the subset of edges in Y' with one endpoint in $V - C$ and one endpoint in $C - \bigcup_{S\in\mathcal{E}(C)} S$. Then the inequality on the internal node v_C is implied by

$$\Phi - |J| \leq 2k - 2. \tag{4}$$

The idea behind proving this inequality is that we will always be able to show that $\Phi \leq 2k-1$, and we will be able to show that $\Phi \leq 2k - 2$ when I contains a 2-edge or has more than one "connected component" on the sets $\mathcal{E}(C)$. Thus the bad case is exactly when there is a special edge e, no other edges in I have been removed, and $J = \emptyset$, which is precisely when the clean-up step will remove edge e.

In any iteration in which an edge of I is selected, we must make an active set $S \in \mathcal{E}(C)$ inactive. Thus $|I| \leq k$. Each edge in I contributes at most 2 to Φ, so that we have $\Phi \leq 2k$. If an edge in I is a 2-edge, then it must make 2 active sets in $\mathcal{E}(C)$ inactive while contributing at most 2 to Φ, proving that $\Phi \leq 2k - 2$, which implies the inequality. Note that if $C = V$ then the final edge in I must be a 2-edge between the final two active sets. There must be two final active sets by the symmetry of h_p.

So assume I consists of 1-edges (and thus $C \neq V$). Let e be the first edge of I that was selected; i.e., other edges in I were selected in iterations after e was selected. Notice that e can only contribute 1 to Φ, since it is a 1-edge. Thus $\Phi \leq 2k - 1$. Since e is the first edge of I selected, it must be the case that $\mathcal{A}(e) = \mathcal{E}(C(e))$ by a straightforward modification of Observation 2.5. If e is not special, then I contains an edge e' with an endpoint not in any $S \in \mathcal{E}(C)$. The edge e' contributes 1 to Φ, giving $\Phi \leq 2k - 2$.

Now suppose e is special. If some edge of I is deleted in the final edge set F', then $\sum_{S \in \mathcal{E}(C)} |\delta_{F_C \cap Y'}(S)| \leq 2k-2$, since the edge must have contributed at least 1 to $\Phi = \sum_{S \in \mathcal{E}(C)} |\delta_I(S)|$; note that this implies the desired inequality (2). The remaining possibility is that e is special, was not deleted, and $C \neq V$. By the properties of the clean-up step it must be the case that $J \neq \emptyset$; hence inequality (4) must hold. ■

5 Implementation

In this section, we show how to implement efficiently the various steps of the main algorithm for proper functions. For a general (IP) corresponding to a proper function f, we obtain a running time of $O(f_{\max} n^2 m' + f_{\max} n^2 \rho)$, where $m' = \min(f_{\max} n, m)$ and ρ is the time taken by an oracle that computes the proper function f. Usually $\rho = O(n)$ so in practical cases, where $f_{\max} = O(1)$, the running time is $O(n^3)$. For SNDP we improve the time to $O(f_{\max} n m' + f_{\max} n \sqrt{m \log\log n}) = O(f_{\max}^2 n^2 + f_{\max} n \sqrt{m \log\log n})$.

Section 5.1 shows how to check if a solution is feasible. The remaining sections discuss the problems of implementation: how to initialize and update the active sets (edge addition step); how to find the next edge to be added (edge selection step); and how to perform the clean-up step. Notice that in every phase p of the algorithm, $|F| \leq n-1$, whence $|F_p| \leq m'$.

5.1 Feasibility

In this section, we show how to check whether a set of edges is a feasible solution to (IP) for a given proper function f. The section introduces some ideas which we use in the following sections on finding and updating the active sets.

We begin by considering the *separation problem* associated with the constraints $x(\delta(S)) \geq f(S)$, where f is proper: given an arbitrary rational vector x, find the most violated inequality $x(\delta(S)) \geq f(S)$ or decide that no such inequality is violated. We show that this problem can be solved using the Gomory-Hu cut tree [9]. Given the graph $G = (V, E)$ with edge capacities x_e, the Gomory-Hu procedure returns a tree H on the vertices V with values w_e on its edges such that the value of the minimum cut between any two vertices s and t is given by the smallest value w on the unique path in H between s and t. Let S_e and $V - S_e$ be the partition of the vertex set induced when e is removed. The tree H also has the property that $w_e = x(\delta(S_e))$.

Given a subset S of vertices, let $\gamma(S)$ denote the edges of H (i.e. of the cut tree) with exactly one endpoint in S (i.e. in the coboundary $\delta(S)$).

Lemma 5.1 Let $S \subset V$. Then

$$x(\delta(S)) \geq \max_{e \in \gamma(S)} w_e.$$

Proof: We show that for any $e = (i,j) \in \gamma(S)$ we have $x(\delta(S)) \geq w_e = |x(\delta(S_e))|$. This is immediate since, by definition of the cut tree, S_e is a minimum cut (or simply *mincut*) separating i and j and therefore has value no greater than the value of any other cut separating i and j. But $\delta(S)$ is precisely such a cut. ■

Lemma 5.2 Let f be a proper function. Then, for any $S \subset V$, we have

$$f(S) \leq \max_{e \in \gamma(S)} f(S_e).$$

Proof: Let $(V_1, \ldots, V_k)$ be the vertex sets of the components of the cut tree after removing the vertices in S. By definition, since $V - S = V_1 \cup \ldots \cup V_k$, we have

$$f(S) = f(V - S) \le \max(f(V_1), f(V_2), \ldots, f(V_k)). \tag{5}$$

Consider any V_i. To simplify notation, assume that S_e is disjoint from V_i for $e \in \gamma(V_i)$ (otherwise, replace S_e by $V - S_e$). Notice that

$$V - V_i = \cup_{e \in \gamma(V_i)} S_e,$$

where the sets appearing in the union are disjoint. Hence,

$$f(V - V_i) = f(V_i) \le \max_{e \in \gamma(V_i)} f(S_e).$$

But, by definition of V_i, we must have $\gamma(V_i) \subseteq \gamma(S)$. Therefore,

$$f(V_i) \le \max_{e \in \gamma(S)} f(S_e). \tag{6}$$

Combining (5) and (6), we obtain the desired result. ■

From these two lemmas, we can easily derive the following.

Theorem 5.3 $\max_S \{f(S) - x(\delta(S))\} = \max_{e \in H} \{f(S_e) - x(\delta(S_e))\}$.

Proof: For any given set S, Lemmas 5.1 and 5.2 imply that

$$f(S) - x(\delta(S)) \le \max_{e \in \gamma(S)} f(S_e) - \max_{e \in \gamma(S)} x(\delta(S_e)) \le \max_{e \in \gamma(S)} (f(S_e) - x(\delta(S_e))),$$

so that

$$\max_S \{f(S) - x(\delta(S))\} \le \max_{e \in H} \{f(S_e) - x(\delta(S_e))\}.$$

Obviously the inequality must in fact be an equality. ■

The theorem allows us to solve the problem of finding a set S such that $f(S) - x(\delta(S)) = \max_T \{f(T) - x(\delta(T))\}$ (and hence allows us to solve the separation problem) by solving $n-1$ maximum flow problems and restricting attention to the $n-1$ cuts defined by the cut tree. In addition, this theorem generalizes a result of Padberg and Rao [16] for T-cuts (cuts S for which $|S \cap T|$ is odd) or odd cuts (for which $|S|$ is odd). Their result states that the minimum T-cut or odd cut is among the cuts of the Gomory-Hu tree. To derive this result from Theorem 5.3, we set

$$f(S) = \begin{cases} M & \text{if } |S \cap T| \text{ odd} \\ 0 & \text{otherwise,} \end{cases}$$

where $M > x(E)$, and note that f is a proper function (assuming that $|T|$ is even). Using similar logic, given a $\{0,1\}$ proper function, our theorem shows that the minimum cut over all S such that $f(S) = 1$ is among the cuts of the Gomory-Hu tree (using the function Mf). Ravi and Klein [18] independently showed that Padberg and Rao's result could be generalized to proper functions f with range $\{0,1\}$.

We now describe how to check whether a set of edges generated by the main algorithm is a feasible solution for (IP) with a given proper function. This subroutine was needed in the edge clean-up step of the algorithm of Williamson et al. [20]; they showed how to implement it using $O(n^2)$ max-flow computations. We show how the test can be implemented in $O(f_{\max} nm' + n\rho)$ time. We do not need this subroutine for our version of UNCROSSABLE; however, some of the concepts here will be useful later. Notice first that the main algorithm selects at most m' edges. Moreover, in the construction of the Gomory-Hu cut tree, we need not solve the maximum flow problems to optimality. We can stop as soon as the flow has value $f_{\max} = \max_S f(S)$, as is justified below. Such a flow can be obtained in $O(f_{\max} m')$ time by locating up to $f_{\max}$ augmenting paths. For small values of $f_{\max}$, we can construct the Gomory-Hu cut tree without using a maximum flow subroutine. For example, the case $f_{\max} = 1$ reduces to finding connected components while the case $f_{\max} = 2$ reduces to computing the 2-edge-connected components of a graph (which can be done in linear time).

To avoid computing maximum flows to optimality, we modify the procedure for constructing the cut tree, since whenever the maximum flow has value greater or equal to $f_{\max}$ we cannot use information from a mincut. First recall the classical procedure of Gomory and Hu [9]. The procedure starts from one supervertex containing all vertices of the graph. At any stage of the construction, there is a partial tree whose (super)vertices form a partition of the vertex set. The procedure selects two vertices u and v within a supervertex A, shrinks the vertices in each connected component resulting from the removal of A from the cut tree, and computes the maximum flow and minimum cut between u and v in the resulting shrunk graph. The supervertex A is split into two supervertices linked by an edge, in such a way that the removal of this edge induces the computed mincut. The new edge of the cut tree gets labeled with the value of the maximum flow between u and v. The algorithm terminates when each supervertex consists of a single vertex of the graph, and its output is the tree with the labels produced.

Our modified procedure is similar to the classical procedure, but it also maintains a forest for each supervertex of the cut tree. The forest for a supervertex is defined on the vertices contained in the supervertex. In the modified procedure, we select two vertices, say u and v, of two different components of a forest of the same supervertex, say A. We compute the maximum flow (up to the value $f_{\max}$) between u and v in the shrunk graph, as in the classical procedure. If the maximum flow value is at least $f_{\max}$, we add the edge (u, v) to the forest of the supervertex A and label the edge "$\geq f_{\max}$". Otherwise, we split A (and its forest) as in the classical algorithm; by submodularity and the definition of $f_{\max}$, one can easily show that each connected component of the forest defined on A will end up entirely within one of the two new supervertices. The modified algorithm terminates when each supervertex is either a single vertex or the forest for the supervertex is a tree on its vertices. We replace every supervertex by its associated tree and output the resulting tree as the modified Gomory-Hu subtree.

By the same argument as in [9], for any vertices s and t, the value of a maximum flow from s to t and a corresponding mincut can be obtained from the modified cut tree, provided that this maximum flow value is at most $f_{\max} - 1$. We should also point out that Lemma 5.1 is still valid for this modified cut tree. Thus, for the incidence vector of a graph with at most m' edges, the modified cut tree can be constructed in $O(f_{\max} n m')$ time, since we solve n flow problems by finding up to $f_{\max}$ augmenting paths in a graph of m' edges. The separation problem can be solved in $O(f_{\max} n m' + n\rho)$ time by constructing the modified cut tree and examining each edge.

5.2 Active sets

This section shows how to find and update the active sets for calls to UNCROSSABLE generated by our main algorithm. When sets are violated, the Gomory-Hu cut tree does not immediately give the *minimal* violated sets. However we can use the lemma below. Let F be the set of edges selected so far by UNCROSSABLE. Let H be the Gomory-Hu cut tree corresponding to the graph with edge set F_{p-1}. Recall that set S is violated in the call to UNCROSSABLE in phase p of the main algorithm if $|\delta_{F_{p-1} \cup F}(S)| = p - 1$

and $f(S) \geq p$.

Lemma 5.4 *Any* violated set at any point in the call to UNCROSSABLE in phase p is an (s,t) mincut for some edge $e = (s,t) \in H$ satisfying $w_e = p - 1$.

Proof: Let S be any violated set at any point in the call to UNCROSSABLE in phase p; i.e., $f(S) \geq p$ and $|\delta_{F_{p-1} \cup F}(S)| = p - 1$. Since we know $|\delta_{F_{p-1}}(S)| \geq \min(f(S), p-1)$, it must be the case that $|\delta_{F_{p-1}}(S)| = p - 1$. From Lemma 5.2, there must exist an edge $e = (s,t)$ in $\gamma(S)$ with $f(S_e) \geq p$. Since $|\delta_{F_{p-1}}(S_e)| \geq \min(f(S_e), p-1)$, we must have that $w_e = |\delta_{F_{p-1}}(S_e)| \geq p - 1$. Since $|\delta_{F_{p-1}}(S)| = p - 1$, Lemma 5.1 implies that $w_e = p - 1$. Hence, S defines a cut of minimum value between s and t. ■

When we initially call UNCROSSABLE in phase p, we form the Gomory-Hu cut tree G corresponding to the graph with the edge set F_{p-1}. Because of the lemma above, in the call to UNCROSSABLE in phase p we shall keep track of *all* (s,t) mincuts for each edge $e = (s,t) \in H$ with $w_e = p - 1$. We use the compact representation of all (s,t) mincuts due to Picard and Queyranne [17]. Given a maximum flow from s to t for an edge $e = (s,t) \in H$, form a directed acyclic graph G_e from the residual graph of the flow by contracting each strongly connected component, as well as the set of all vertices reachable from s, and the set of all vertices that can reach t. For notational simplicity, let S and T denote the supervertices of G_e containing s and t respectively. Picard and Queyranne observe that there is a 1-1 correspondence between the (s,t) mincuts and the (T,S) dicuts of G_e, where a (T,S) dicut is a cut with all arcs directed from the side of the cut containing T to the side containing S. Given a maximum flow, the digraph G_e can be computed in $O(m')$ time since the residual graph contains $O(m')$ edges. As UNCROSSABLE adds edges to F, we will update the graphs G_e so that the (T,S) dicuts reflect the minimum (s,t) cuts of value $p-1$ in $F_{p-1} \cup F$; we will explain how to do this later in the section.

Consider a topological ordering of G_e (i.e., a numbering of the supervertices so that any edge of G_e goes from a lower-numbered supervertex to a higher-numbered supervertex). Because we contracted into the supervertex T all vertices in the residual graph that can reach t, the first supervertex in the ordering can be assumed to be T. Similarly, since all vertices reachable from s have been contracted, we can assume that S is the last supervertex in the ordering. By definition, all the supervertices that are predecessors of some supervertex A in the ordering must induce a (T,S) dicut and hence an (s,t) mincut but, clearly, not all (s,t) mincuts arise in this fashion. Nevertheless, we will show that we can limit our attention to particular (s,t) mincuts arising in this way.

Lemma 5.5 Let $e = (s,t)$ be an edge in the Gomory-Hu tree H such that $w_e = p - 1$. At any point of UNCROSSABLE in phase p, there exists a violated set separating t from s if and only if there exists a supervertex A of G_e with $f(A) \geq p$.

Proof: Only if part. Any violated set S must be the union of supervertices A_i. By the maximality property of proper functions, at least one of these supervertices must satisfy $f(A) \geq p$.

If part. Choose the first supervertex A in the ordering such that $f(A) \geq p$. Consider the union C of all the predecessors of A in G_e. This set C induces an (s,t) mincut. Moreover, $C - A$ consists of the union of supervertices A_i's with $f(A_i) < p$, so that by maximality, $f(C - A) < p$. By symmetry, $f(V - A) = f(A) \geq p$. By maximality, $f(V - A) \leq \max(f(C - A), f(V - C))$, which implies that $f(V - C) \geq p$. Thus $f(C) \geq p$; C is a violated set at the beginning of the call to UNCROSSABLE since $f(C) \geq p$ and C is an (s,t) mincut, implying $|\delta_{F_{p-1} \cup F}(C)| = p - 1$. ■

Theorem 5.6 Let e be an edge in the Gomory-Hu tree H such that $w_e = p-1$. Let A be the first vertex in the topological ordering of G_e such that $f(A) \geq p$, and let C be A together with its predecessors in G_e. If there exists an active set separating t from s at any point of UNCROSSABLE in phase p, it must be C.

Proof: Suppose there is an active set C' containing t but not s. By Corollary 2.2, an active set cannot cross a violated set and, hence, $C' \subseteq C$. Since C' is a violated set, it must be the union of supervertices of G_e. Moreover, it must contain A since $f(C') \geq p$ and A is the only supervertex within C which has a value at least p. Furthermore, since C' corresponds to an (S,T)-dicut of G_e, it contains all predecessors of A in G_e. Thus $C' = C$. ∎

This lemma motivates the following procedure in order to find a potentially active set C containing t but not s. Scan the supervertices A of G_e in the topological order, calling the oracle for each supervertex A and stopping as soon as $f(A) \geq p$. Set C to be A together with its predecessors in G_e.

In a similar manner, we can also find a potentially active set containing s but not t. These two sets can be constructed in $O(m' + n\rho)$ time per graph G_e. By doing this for all edges $e \in H$ with $w_e = p-1$, one thus constructs in $O(nm' + n^2\rho)$ time a family of $O(n)$ violated sets guaranteed to contain all active sets; this follows from Lemma 5.4. The active sets can be obtained from this family in $O(n^2)$ time by finding the minimal sets in this family. This can be done by keeping track, for each vertex, of the set (if unique) of smallest cardinality containing it.

In summary, the modified cut tree for F_{p-1} and the initial active sets of UNCROSSABLE in phase p can be obtained in $O(f_{\max} nm' + n^2\rho)$ time: $O(f_{\max} nm')$ time to construct the modified Gomory-Hu tree, $O(nm')$ time to construct all the possible G_e, and $O(nm' + n^2\rho)$ to extract the active sets from the G_e.

When the addition step adds edge e' to F the active sets must be updated. However, the sets that are active once e' is added are sets that were violated at the beginning of a call to UNCROSSABLE in phase p. We can therefore use exactly the same procedure as above to recompute the active sets. More precisely, we update the $O(n)$ G_e by adding the (bidirected) edge e' to them and recomputing their strongly connected components in $O(m')$ time per G_e. Then we create a family of $O(n)$ potentially active sets and extract from this family the minimal sets. In this case though, we can just make one oracle call (instead of $O(n)$) per G_e since, by adding an edge to a digraph, only one new strongly connected component can be created. Recomputing the active sets for each edge added therefore takes $O(nm' + n\rho)$ time. Using our assumption that $f_{\max} \leq n$, the time to find the active sets in all iterations of a phase is thus $O(n^2m' + n^2\rho)$. Over all phases, this becomes $O(f_{\max} n^2 m' + f_{\max} n^2 \rho)$.

5.3 Active sets for SNDP

This section refines the implementation of the edge addition step given in Section 5.2 for SNDP. It achieves total time $O(nm')$ for the addition steps of one phase, and thus $O(f_{\max} nm')$ over all phases.

For SNDP, we don't need to construct the Gomory-Hu cut tree to test feasibility. We use a maximum spanning tree T of the graph having cost r_{ij} on edge (i,j). Any set satisfying the connectivity requirements of the edges of T satisfies all given requirements r_{ij} (Gomory and Hu [9]). It is easy to see that the violated sets of phase p must correspond to mincuts of value $p-1$ associated with an edge $e = (s,t)$ of T having $f(S_e) = r_{st} \geq p$. However, in this case, we have a 1-to-1 mapping: any such mincut must correspond to a violated set. As a result, any active set must be a *minimal* (s,t) mincut or a *minimal* (t,s) mincut for some $(s,t) \in T$.

We maintain these minimal mincuts over the entire algorithm. To do this we maintain, for each edge (s,t) in T, an $s-t$ flow with value no larger than r_{st}. At the beginning of phase p, we start from the flows that were computed in phase $p-1$ and find augmenting paths for each edge e in T up to value p, if possible. Thus in phase p we can detect any edge (s,t) of T for which the cut value is $p-1$ while $r_{st} \geq p$. Over the course of the algorithm we must find at most $f_{\max}$ augmenting paths for the flow for each edge of T, leading to a total time bound of $O(f_{\max} m'n)$ for maintaining these flows.

Suppose for an edge $e=(s,t)$ in T, the cut value is $p-1$ while $r_{st} \geq p$. Initially, the minimal (s,t) mincut consists of all vertices reachable from s in the residual graph of a maximum flow from s to t. The minimal (t,s) mincut is similar. As before, we can extract the active sets from these mincuts in $O(n^2)$ time. Whenever an iteration adds an edge $e=(u,v)$ to the edge set F, each minimal mincut is updated. For example for the minimal (s,t) mincut, if u is reachable from s then so is v, as is any vertex reachable from v by residual edges. If t becomes reachable then we disregard edge (s,t) in the tree T for the rest of the phase. The total time in phase p to update the minimal (s,t) mincut amounts to a search of the residual graph, and thus uses time $O(m')$. Thus the total time in a phase for updating all edges (s,t) of T is $O(nm')$. To find the new active set (if any) resulting from the addition of edge (u,v), we search through the $O(n)$ candidate sets for the smallest violated set containing u. The smallest such set will be a new active set if it contains no other currently active sets. The search for the set takes $O(n)$ time. Therefore, the total time for the edge addition step in a phase p is $O(nm')$ for SNDP, leading to a total time bound of $O(f_{\max} nm')$ over the course of the whole algorithm.

5.4 Edge selection step

This section shows how to implement the edge selection step of UNCROSSABLE in time $O(n(n+\sqrt{m \log\log n}))$. It uses a lemma allowing irrelevant edges to be ignored, plus the data structure idea of packets due to Gabow et al. [4].

Say that "time" is zero at the beginning of UNCROSSABLE, and time increases by the amount ϵ determined in each iteration (see Figure 2). As in [8] our implementation keeps track of the *addition time* at which an edge would be selected *if* the set of active sets $\mathcal{C}$ were not to change. The addition time of an edge (u,v) in the coboundary of b active sets at time t is formally defined as $t(u,v) = t + [c_{uv} - d(u) - d(v)]/b$. Here $d(u)$ and $d(v)$ denote their values at time t, $b = 0, 1$ or 2; the addition time is ∞ if $b=0$. The addition time does not change unless the active set containing u or v changes. It is easy to see that at time t the edge selection step adds an edge with the smallest addition time (at time t) to F.

We maintain a partition of V into *a-sets*. At any time in phase p the a-sets are defined as follows: Contract each currently active or previously active set to supervertices. We now have as vertices these supervertices plus the vertices which have never been in any active set. An a-set corresponds to a tree in the forest induced by edges F on these vertices. Note, then, that each vertex which has never been in any active set and is not incident to an edge of F is a singleton a-set (it corresponds to an empty tree). Note also that at any time any currently active set is an a-set. An a-set that is not a singleton or currently active set is a maximal collection of maximal previously active sets joined by edges of F, plus possibly a singleton set. We give an example of the a-sets from frame 4 of Figure 5 in Figure 9.

Notice that among edges joining the same two a-sets, we can discard all but the one with smallest addition time; the other edges will always have addition time no smaller than this smallest edge. Let $I \subseteq E$ denote the subgraph of G consisting of these edges. Assume that in choosing the next edge to add, the selection step breaks ties for smallest addition time according to some fixed numbering of the edges. Thus any set of edges is totally ordered by addition time; in particular its kth smallest edge is unique. Throughout this section, we compare edges using addition time, not cost, e.g., "kth smallest edge" refers

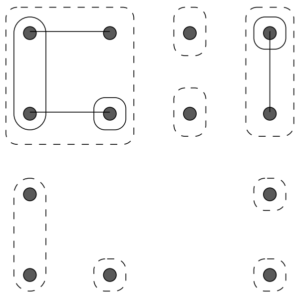

Figure 9: Example of a-sets from frame 4 of Figure 5. A-sets are contained in dashed lines. Solid lines show previously active sets.

to addition time.

Let $A(u)$ denote the a-set containing u, and let a denote the current number of a-sets. Our algorithm is based on the following principle.

Lemma 5.7 Fix an iteration in the while loop of UNCROSSABLE. Consider an edge $(u, v) \in \delta_I(A(u))$ that is not among the $2k$ smallest edges of $\delta_I(A(u))$. Then (u, v) is not added to F until $A(u)$ has changed or $A(v)$ has changed or a has decreased by k.

Proof: Suppose (u, v) is added to F in an iteration when neither $A(u)$ nor $A(v)$ has changed. Consider an edge (u', v'), one of the $2k$ smallest edges of $\delta_I(A(u))$. When (u, v) is added to F, (u', v') has not been added (since $A(u)$ has not changed). Thus the addition time of (u', v') has increased, implying that $A(v')$ has changed. Thus the $2k$ distinct sets $A(v')$ have changed. These $2k$ changes must be the result of merging various a-sets which include the $2k$ sets $A(v')$. Thus a must have decreased by at least k. ∎

The lemma above will allow us to ignore particular sets of edges during some portions of the while loop of UNCROSSABLE. In order to take advantage of the lemma, we partition the main loop into *subphases*. Let r be a parameter to be chosen later. Each time a has decreased by r or more since the start of the current subphase, a new subphase will begin. We will designate certain edges to be *awake* in such a way that Lemma 5.7 will imply that an edge (u, v) that is not awake in this subphase need not be considered unless $A(u)$ or $A(v)$ changes. At the beginning of a subphase, we choose the $2r$ smallest edges in the coboundary of every a-set. Any edge chosen by the a-sets of both its endpoints will be initially designated an awake edge.

In order to keep track of the awake edges, we use a number of priority queues. A priority queue entry is an edge that is awake and joins two distinct a-sets. Its priority is equal to its current addition time. The awake edges incident to each a-set are partitioned into *packets* of no more than $\log n$ edges each. One distinguished packet is called the *growing packet*; the other packets are *ordinary packets*. Each packet is a priority queue. Every awake edge joining two distinct a-sets is in precisely two packets, corresponding to its two ends. In addition to the packets, we also maintain a priority queue D. An edge is a *double minimum* if it is awake and is the minimum of both its packets, and these packets are ordinary. The queue D contains all such double minima. Our description implies that the edge with the smallest addition time is the minimum edge of D or the minimum edge of a growing packet.

Given these data structures, the edge addition step will work as follows. To start a subphase, each a-set chooses the $2r$ smallest edges in its coboundary. An edge that is chosen by both its ends is awake.

The awake edges are organized into packets. Any edge that is a double minimum is placed in D. To select the next edge e for F, choose the smallest edge among the minima of D and all growing packets. Let A be the new a-set created by adding e to F. Delete all edges incident to vertices of A from their packets and from D. If this causes a new packet minimum to be a double minimum, add it to D. Note that now all packets corresponding to A are empty, so initialize a new growing packet. We now need to choose the awake edges incident to A. To do this, examine all edges incident to A, awake or not. Discard any parallel edges between A and other a-sets, always keeping the smallest. The $2r$ smallest undiscarded edges incident to A will be designated awake edges. Add each such edge (u, v) to the two growing packets of $A(u)$ and $A(v)$ (one of these is A). Whenever a growing packet gets $\log n$ edges, make it an ordinary packet and start a new growing packet; also possibly add an entry to D for a new double minimum. The algorithm is correct because it maintains the defining properties of the packets and D. Lemma 5.7 justifies ignoring the edges that are not awake.

Before we estimate the running time, we observe that in any subphase, for any a-set A, at most $3r$ edges of $\delta(A)$ become awake. To see this, notice at most $2r$ such edges are awake when any a-set A is initialized. After initialization, each iteration can make at most one more edge of $\delta(A)$ awake. Thus at most r more edges are made awake before the subphase ends. As a result, any a-set A has at most $3r/\log n$ packets at any point in a subphase.

First we bound the time for deletions and insertions from the priority queues. By the observation above, an iteration that decreases the number a of a-sets by j deletes at most $O(jr)$ edges from packets and at most $O(jr/\log n)$ edges from D. Thus all addition steps delete a total of $O(nr)$ edges from packets and $O(nr/\log n)$ edges from D, for a total time of $O(nr \log\log n + nr)$ for all deletions (since packets have at most $\log n$ edges and D has at most n edges). To bound the time on the insertions, note that at most $O(nr)$ edges ever enter packets and at most $O(nr/\log n)$ edges ever enter D. Since these bounds are no larger than the total number of edge deletions, the total time for edge insertions must also be $O(nr \log\log n)$.

Putting everything together, note that there are at most n/r subphases. Using linear-time selection, we can construct packets and D in $O(m)$ additional time at the start of each subphase. We need $O(n)$ amortized time whenever an edge is selected for discarding parallel edges, examining the edges incident to A, and updating $A(u)$ values. Thus the total time is $O(n^2 + nm/r + nr \log\log n)$. Choosing $r = \sqrt{m/\log\log n}$ gives total time $O(n(n + \sqrt{m \log\log n}))$ for the edge selection step.

5.5 Linear time clean-up step

This section shows how to implement the clean-up step in time $O(n)$. The implementation is based on a tree $\bar{\mathcal{T}}$ and auxiliary arborescences τ_C which we now define.

The tree $\bar{\mathcal{T}}$ is a modification of the tree $\mathcal{T}$ used to represent the active sets over the course of the algorithm, as defined in Section 2. Recall that $\mathcal{T}$ is constructed by creating a node v_C for each $C \in \cup\mathcal{C}$, where $\cup\mathcal{C}$ is the collection of all active sets over all iterations, plus the set V. The node v_D is a parent of v_C in $\mathcal{T}$ if D is the smallest set in $\cup\mathcal{C}$ that properly contains C. To form $\bar{\mathcal{T}}$, we create one additional child node for each internal node v_C of the tree to represent the set of vertices in C that are not in the other children of v_C.

For each node v_C in $\bar{\mathcal{T}}$, we construct an arborescence τ_C. Recall that a branching is a directed forest in which every node has in-degree at most 1, and an arborescence is a connected branching. Using the edges of F_C, we form a branching β_C on the nodes corresponding to the children of C in $\bar{\mathcal{T}}$. An edge (v_A, v_B)

is created for each edge $e \in F_C$ when edge e is in the coboundary of the sets A and B. Each 2-edge is directed arbitrarily, while a 1-edge is directed towards the active set that defines it. The edges of F_C form a branching since two edges of F cannot be directed towards the same active set. It can be converted into an arborescence τ_C by adding a node connected to all the roots of the branching. Both the tree $\bar{\mathcal{T}}$ and the arborescences τ_C can be easily constructed during the edge addition stage of UNCROSSABLE and the running time for their constructions can be charged to this stage. In Figure 10, we give an example of τ_C for $C = C(e_1)$ from Figure 5.

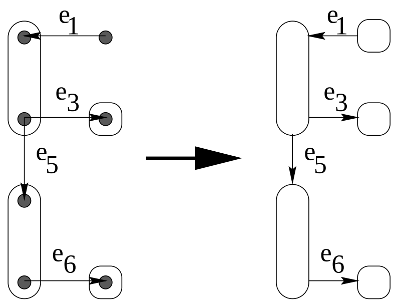

Figure 10: Example of arborescence τ_C for $C = C(e_1)$ from Figure 5. The left-hand side shows edges on underlying vertices, while the right-hand side shows arborescence.

Before we explain the clean-up procedure, we make the following observation.

Observation 5.8 A 1-edge e in F_C (for $C = C(e)$) is special if and only if all the edges of F_C added to τ_C after e form an arborescence with the head of e as its root.

Proof: This follows since a 1-edge e is special if and only if all the edges of F_C added after e form a spanning tree on the active sets corresponding to children of v_C. Such a spanning tree will necessarily cause τ_C to be an arborescence directed away from the head of e since whenever a 1-edge is directed into a node corresponding to an active set, all edges added after it must be directed away from the node. ■

We find all special edges in $O(n)$ time as follows. We consider each active set $C \in \cup\mathcal{C}$ in turn. Suppose F_C consists of $e_1, \ldots, e_t$, where e_i was added before e_j for $i < j$. Let l_i denote the least common ancestor of the heads of $e_i, \ldots, e_t$ in τ_C. By the reasoning above, a 1-edge e_i is special if and only if l_i is the head of edge e_i in τ_C. The l_i's can be found in linear time by processing the edges in reverse order, by marking the nodes along the path to the ancestor and by stopping at the first previously marked node.

We would like to implement the edge clean-up step by performing a top-down traversal of $\bar{\mathcal{T}}$. Let F'' be the set of edges remaining after a clean-up step that works as follows: we perform a top-down traversal of $\bar{\mathcal{T}}$. At each node v_C of $\mathcal{T}'$, we process the edges of F_C in the reverse order in which they appear in F, detecting and possibly removing a special edge e of F_C under the same condition as before (i.e., no other edge of F_C has been removed and all remaining edges of F in $\delta(C(e))$ are also in $\delta(A(e))$). We begin by proving the following lemma.

Lemma 5.9 $F'' = F'$.

Proof: Observe that the removal of an edge e of F_C depends only on the edges in F_C and $\delta_F(C)$, and affects only the removal of edges in F_C and in F_D, for nodes D with $e \in \delta(D)$. Such a set D must be a descendant of node v_C in the tree $\bar{\mathcal{T}}$, and any node v_A such that edges of $\delta_F(C)$ are in F_A must be an ancestor of C. By Observation 2.4, all edges of F_C occur in F before any edge in $\delta(C)$ and after any edge in F_D, for v_D a descendant of v_C. Thus the set of edges F'' formed by removing edges in a top-down traversal of $\bar{\mathcal{T}}$ will be the same as that in F'. ■

When visiting a node v_C in $\bar{\mathcal{T}}$, we need to decide which special edge of F_C to remove, if any. Call a child v_B of v_C *hit* if there exists an edge already processed but not removed that is simultaneously in the coboundaries of B and C. We show below that we can use information about hit children of v_C to determine which special edge of F_C to remove, if any.

Lemma 5.10 A special edge e of F_C is removed in the top-down traversal of $\mathcal{T}$ if and only if it is the deepest special edge e in τ_C whose head is an ancestor in τ_C of all the hit children of v_C.

Proof: Recall that while considering the edges of F_C in the reverse of the order added, we remove special edge e of F_C if and only if no other special edge has been deleted and all remaining edges in $\delta(C)$ are also in $\delta(A(e))$, where $A(e)$ is the union of sets in $\mathcal{A}(e)$. By discussion above, for any special edge e in F_C, the sets in $\mathcal{A}(e)$ correspond exactly to the descendants of the head of edge e in τ_C. Thus all remaining edges in $\delta(C)$ are also in $\delta(A(e))$ for special edge e if and only if the head of e is an ancestor in τ_C of all the hit children of v_C. Finally, by the discussion above, because any edges in F_C added after a special edge e must be edges in τ_C between descendants of the head of e, the depth of the special edges in τ_C is strictly increasing with respect to the order in which the special edges were selected. Hence the deepest special edge e in τ_C whose head is an ancestor in τ_C of all the hit children of v_C (if any such edge exists) is exactly the edge which will be removed. ■

Our procedure will be as follows. For the moment, we assume that when we visit a node v_C, the hit children of v_C have been correctly marked. Under this assumption, the lemma above tells us that we can detect the appropriate special edge to remove while determining the special edges of F_C. Once we have decided which special edge of F_C to remove (if any) and which edges to keep, for each edge $e = (u, v) \in F_C$ that we keep, we mark the nodes of the paths in $\bar{\mathcal{T}}$ from the leaves u and v up to (but excluding) their common ancestor v_C as hit nodes. Notice that this may incorrectly mark children of v_C as hit nodes, but this no longer matters since we have already processed the edges of F_C. In order for this marking procedure to run in linear time, we stop marking a path before reaching v_C if we encounter an already hit node. The validity of this stopping rule follows from the fact that we perform a top-down traversal of $\bar{\mathcal{T}}$, since if node v_D on a path to v_C is marked, so are all ancestors of v_D up to v_C. The top-down traversal also ensures that when we reach a node v_D, all of its hit children are correctly marked. Therefore, the overall running time of the clean-up step is $O(n)$ time.

6 Concluding Remarks

Since the appearance of a preliminary version of this paper [5], Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson [6] have shown how the performance guarantee of the algorithm can be improved from $2f_{\max}$ to $2\mathcal{H}(f_{\max})$, where $\mathcal{H}(k) = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$. Their algorithm uses an algorithm for uncrossable functions as a black box, so our version of UNCROSSABLE can still be used. Furthermore, they show that their algorithm can be implemented by making small modifications to our implementations, yielding a $2\mathcal{H}(f_{\max})$-approximation algorithm with the same running times as our algorithm above. This entire line of research has been summarized in the thesis of Williamson [19].

Acknowledgments

We are grateful to an anonymous referee for many useful comments. The third author would like to thank David Johnson and AT&T for inviting him to spend the summer of 1992 at Bell Labs.

References

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24:440–456, 1995.

[2] P. Berman and V. Ramaiyer. Improved approximations for the Steiner tree problem. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 325–334, 1992.

[3] G. N. Frederickson and J. Ja'Ja'. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10:270–283, 1981.

[4] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[5] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[6] M. Goemans, A. Goldberg, S. Plotkin, D. Shmoys, E. Tardos, and D. Williamson. Improved approximation algorithms for network design problems. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 1994.

[7] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:145–166, 1993.

[8] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24:296–317, 1995.

[9] R. Gomory and T. Hu. Multi-terminal network flows. *SIAM Journal of Applied Mathematics*, 9:551–570, 1961.

[10] M. Grötschel, C. L. Monma, and M. Stoer. Design of survivable networks. In *Handbook in Operations Research and Management Science*. 1993. To appear.

[11] D. Gusfield and D. Naor. Extracting maximal information about sets of minimum cuts. *Algorithmica*, 10:64–89, 1993.

[12] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. *Journal of the ACM*, 41:214–235, 1994.

[13] P. Klein and R. Ravi. When cycles collapse: A general approximation technique for constrained two-connectivity problems. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 39–55, 1993. Also appears as Brown University Technical Report CS-92-30. To appear in *Algorithmica*.

[14] P. N. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Information Processing Letters*, 52:303–307, 1994.

[15] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[16] M. W. Padberg and M. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7:67–80, 1982.

[17] J. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Study*, 13:8–16, 1980.

[18] R. Ravi and P. Klein. Approximation through uncrossing. Unpublished manuscript, 1992.

[19] D. P. Williamson. *On the design of approximation algorithms for a class of graph problems.* PhD thesis, MIT, Cambridge, MA, September 1993. Also appears as Tech Report MIT/LCS/TR-584.

[20] D. P. Williamson, M. X. Goemans, M. Mihail, and V. V. Vazirani. A primal-dual approximation algorithm for generalized Steiner network problems. *Combinatorica*, 15:435–454, 1995.

[21] A. Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9:463–470, 1993.