Mathematical Programming Series B manuscript No.
(will be inserted by the editor)

A $\frac{3}{2}$-Approximation Algorithm for Some Minimum-Cost Graph Problems

Basile Couëtoux · James M. Davis · David P. Williamson

Received: date / Accepted: date

Abstract We consider a class of graph problems introduced in a paper of Goemans and Williamson that involve finding forests of minimum edge cost. This class includes a number of location/routing problems; it also includes a problem in which we are given as input a parameter k, and want to find a forest such that each component has at least k vertices. Goemans and Williamson gave a 2-approximation algorithm for this class of problems. We give an improved $\frac{3}{2}$-approximation algorithm.

1 Introduction

Consider the following graph problem: given an undirected graph $G = (V,E)$ with nonnegative edge costs $c(e) \geq 0$ for all $e \in E$, and a positive integer k, find a minimum-cost set F of edges such that each connected component of (V,F) has at least k vertices in it. If $k = 2$, the problem is the minimum-weight edge cover problem, and

This paper is the merger of two extended abstracts, the first due to the first author (which appeared in ESA 2011 [4]) and the second due to the second and third authors (which appeared in ESA 2012 [5]). The second author is supported by the National Science Foundation under Grant No. DGE-0707428. The third author is supported in part by NSF grant CCF-1115256.

B. Couëtoux
Laboratoire d'Informatique Fondamentale de Marseille,
Université de la Mediterranée, Marseille, F-13288, Marseille Cedex 9, France.
E-mail: Basile.Couetoux@lif.univ-mrs.fr

J.M. Davis
School of Operations Research and Information Engineering,
Cornell University, Ithaca, NY 14853, USA.
E-mail: jmd388@cornell.edu

D.P. Williamson
School of Operations Research and Information Engineering,
Cornell University, Ithaca, NY 14853, USA.
E-mail: dpw@cs.cornell.edu

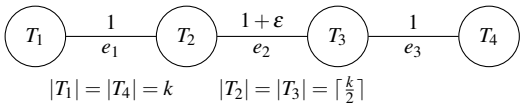

Fig. 1: The T_i are cliques of cost zero edges, with $|T_1| = |T_4| = k$, and $|T_2| = |T_3| = \lceil \frac{k}{2} \rceil$. $\{e_1, e_3, T_1, T_2, T_3, T_4\}$ is a solution of cost 2 returned by the Imielińska et al. algorithm, while $\{e_2, T_1, T_2, T_3, T_4\}$ is an optimal solution for this instance of cost $1+\varepsilon$.

if $k = |V|$, the problem is the minimum spanning tree problem; both problems are known to be polynomial-time solvable. However, for any other constant value of k, the problem is known to be NP-hard (see Imielińska, Kalantari, and Khachiyan [9] for $k \geq 4$ and Bazgan, Couëtoux, and Tuza [2] for $k = 3$).

For this reason, researchers have considered approximation algorithms for the problem. An α-approximation algorithm is a polynomial-time algorithm that produces a solution of cost at most α times the cost of an optimal solution. The parameter α is sometimes called the *performance guarantee* of the algorithm. Bazgan et al. have also shown that the problem is APX-hard for $k \geq 3$, and there is can be no approximation algorithm with performance guarantee better than $\frac{95(8k-7)+1}{95(8k-7)}$ for $k \geq 3$ unless P = NP.

Nevertheless, Imielińska, Kalantari, and Khachiyan [9] give a very simple 2-approximation algorithm for this problem, which they call the *constrained forest problem*. The algorithm is a variant of Kruskal's algorithm [10] for the minimum spanning tree problem: it considers edges in order of increasing cost, and adds an edge to the solution as long as it connects two different components (as in Kruskal's algorithm) and one of the two components has fewer than k vertices. Other 2-approximation algorithms based on edge deletion (instead of, or in addition to, edge insertion) are given by Laszlo and Mukherjee [11,12], who also perform some experimental comparisons. The performance guarantee of the algorithm is tight, as can be seen in an example in Figure 1.

Goemans and Williamson [6] show that the Imielińska et al. algorithm can be generalized to provide 2-approximation algorithms for a large class of graph problems. The graph problems are specified by a function $h : 2^V \to \{0,1\}$ (actually, [6] considers a more general case of functions $h : 2^V \to \mathbb{N}$, but we will consider only the 0-1 case here). Given a graph G and a subset of vertices S, let $\delta(S)$ be the set of all edges with exactly one endpoint in S. Then a set of edges F is a feasible solution to the problem given by h if $|F \cap \delta(S)| \geq h(S)$ for all nontrivial subsets S of vertices. Thus, for instance, the problem of finding components of size at least k is given by the function $h(S) = 1$ iff $|S| < k$. Goemans and Williamson [6] consider functions h that are *downwards monotone*; that is, if $h(S) = 1$ for some subset S, then $h(T) = 1$ for all $T \subseteq S$, $T \neq \emptyset$. They then show that the natural generalization of the Imielińska et al. algorithm is a 2-approximation algorithm for any downwards monotone function h; in particular, the algorithm considers edges in order of increasing cost, and adds an edge to the solution if it joins two different connected components C and C'

and either $h(C) = 1$ or $h(C') = 1$ (or both). Laszlo and Mukherjee [13] have shown that their edge-deletion algorithms also provide 2-approximation algorithms for these problems. Goemans and Williamson show that a number of graph problems can be modeled with downwards monotone functions, including some location-design and location-routing problems; for example, they consider a problem in which every component not only must have at least k vertices but also must have an open depot from a subset $D \subseteq V$, where there is a cost $c(d)$ for opening the depot $d \in D$ to serve the component.

In this paper we give a $\frac{3}{2}$-approximation algorithm for the class of problems introduced by Goemans and Williamson. It is easiest to explain the basic idea of the algorithm in the context of the original problem of Imielińska et al. In any partial solution constructed by the algorithm, let us call a connected component *small* if it has fewer than k vertices in it, and *big* otherwise. We call an edge *good* if it joins two small components into a big component, and *bad* if it is not good and it joins two components, at least one small. The algorithm behaves similarly to the Imielińska et al. algorithm in that it considers adding edges to the solution in order of increasing cost. However, when it considers adding an edge e to the solution (at cost $c(e)$), it also considers all good edges e' of cost $c(e') \leq 2c(e)$. If such a good edge exists, it adds the cheapest one; otherwise, it adds the edge e. The intuition for the algorithm is that we want to try to decrease the number of small components of a partial solution until there is no small component left. A good edge decreases the number of small components by two; a bad edge reduces the number of small components by one, since it will either join two small components into a small component, or a small and a big component into a big component. In order to choose the next edge to add to the partial solution, we choose the edge which will decrease the number of small components with minimal cost. Thus, we should be willing to pay up to twice as much for a good edge as a bad edge.

The extension to the more general class of downwards monotone functions is straightforward: a component C is small if $h(C) = 1$ and big otherwise.

In our analysis, we note that the algorithm can be viewed as a dual-growing algorithm similar to many primal-dual approximation algorithms for network design problems (see Goemans and Williamson [8] for a survey). However, in this case, the dual is not a feasible solution to the dual of a linear programming relaxation of the problem; in fact, it gives an overestimate on the cost of the tree generated by the algorithm. But we show that a dual-fitting style analysis works; namely, if we scale the dual solution down by a factor of 2/3, it gives a lower bound on the cost of any feasible solution. This leads directly to the performance guarantee of $\frac{3}{2}$. Our analysis is tight, as seen in Figure 2. In Section 3.3, we show that the scaled-down dual is feasible for a nonstandard linear programming relaxation of the problem.

Our result is interesting for another reason: we know of very few classes of network design problems such as this one that have a performance guarantee with constant strictly smaller than 2. For individual problems such as Steiner tree and prize-collecting Steiner tree there are approximation algorithms known with performance guarantees smaller than 2 (see, for instance, Byrka et al. [3] for Steiner tree and Archer et al. [1] for prize-collecting Steiner tree), but these results are isolated, and do not extend to well-defined classes of problems. It would be very interesting,

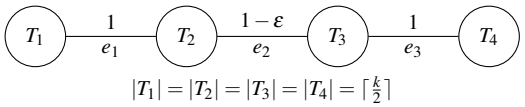

Fig. 2: The T_i are cliques of cost zero edges, with $|T_1| = |T_2| = |T_3| = |T_4| = \lceil \frac{k}{2} \rceil$. $\{e_1, e_2, e_3, T_1, T_2, T_3, T_4\}$ is a solution of cost $3 - \varepsilon$ returned by our algorithm, while $\{e_1, e_3, T_1, T_2, T_3, T_4\}$ is an optimal solution for this instance of cost 2.

for instance, to give an approximation algorithm for a class of functions known as proper functions with a constant performance guarantee smaller than 2. This class was introduced by Goemans and Williamson [7]; a function $f : 2^V \to \{0,1\}$ is proper if $f(S) = f(V - S)$ for all $S \subseteq V$ and $f(A \cup B) \leq \max(f(A), f(B))$ for all disjoint $A, B \subset V$. This class of functions includes several interesting problems, including the Steiner tree and generalized Steiner tree problems.

The paper is structured as follows. In Section 2, we give the algorithm in more detail. In Section 3, we turn to the analysis of the algorithm. We conclude with some open questions in Section 4.

2 The algorithm

In this section we give the algorithm in slightly more detail; it is summarized in Algorithm 1. As stated in the introduction, we start with an infeasible solution $F = \emptyset$. In each iteration, we first look for the cheapest *good* edge e with respect to F: a good edge with respect to F has endpoints in two different components C_1 and C_2 of (V, F) such that $h(C_1) = h(C_2) = 1$ and $h(C_1 \cup C_2) = 0$. Any edge that is not good is *bad*. We then look for the cheapest bad edge e' with respect to F that has endpoints in two different components C_1 and C_2 such that $\max(h(C_1), h(C_2)) = 1$. If the cost of the good edge e is at most twice the cost of the bad edge e', we add e to F, otherwise we add e' to F. We continue until we have a feasible solution. Note that if the step of considering good edges is removed, then we have the previous 2-approximation algorithm of Goemans and Williamson. A sample run of the algorithm is given in the appendix.

3 Analysis of the algorithm

At a high level, our analysis will work as follows. We first show that we can have the algorithm implicitly construct a set of variables $y(S)$ for $S \subseteq V$, and we prove (in Lemma 4) that the cost of the solution F found by the algorithm is at most the sum of the variables $\sum_{S \subseteq V} y(S)$. Then we will show (in Lemma 5) that scaling down this sum by 2/3 gives a lower bound on the cost of any feasible solution for the problem. These two results together imply that the cost of the solution found by the algorithm is at most $3/2$ the cost of an optimal solution for the problem. At the end of the section, we provide an alternate perspective on our analysis.

$F \leftarrow \emptyset$
while F is not a feasible solution **do**
  Let e be the cheapest (good) edge joining two components C_1, C_2 of F with
    $h(C_1) = h(C_2) = 1$ and $h(C_1 \cup C_2) = 0$ (if such an edge exists)
  Let e' be the cheapest edge joining two components C_1, C_2 of F such that
    $\max(h(C_1), h(C_2)) = 1$
  if e exists and $c(e) \leq 2c(e')$ **then**
    $F \leftarrow F \cup \{e\}$
  else
    $F \leftarrow F \cup \{e'\}$
Return F

Algorithm 1: The $\frac{3}{2}$-approximation algorithm.

$F \leftarrow \emptyset$
$y \leftarrow 0$
while F is not a feasible solution **do**
  Increase $y(C)$ uniformly for all components C with $h(C) = 1$ until either:
  (1) $\sum_{S:e \in \delta(S)} y(S) \geq c(e)$ for some good edge e; OR
  (2) $\sum_{S:a' \in \delta^+(S)} y(S) = c(a')$ for some bad arc $a' \equiv$ bad edge e';
  if (1) happens **then**
    $F \leftarrow F \cup \{e\}$
  else
    $F \leftarrow F \cup \{e'\}$
Return F

Algorithm 2: A dual-based version of the algorithm.

3.1 Some preliminaries

As stated above, we implicitly have the algorithm construct a set of variables y as it runs; we refer to y as a dual solution. While we call y a dual solution, this is a bit misleading; unlike typical primal-dual style analyses, we are not constructing a feasible dual solution to a linear programming relaxation of the problem. However, we will guarantee that it is feasible for a particular set of constraints, which we now describe. Suppose we take the original undirected graph $G = (V, E)$ and create a mixed graph $G_m = (V, E \cup A)$ by *bidirecting* every edge; that is, for each edge $e = (u, v) \in E$ of cost $c(e)$ we create two arcs $a = (u, v)$ and $a' = (v, u)$ of cost $c(a) = c(a') = c(e)$ and add them to the original set of undirected edges. Let $\delta^+(S)$ be the set of arcs of G_m whose tails are in S and whose heads are not in S; note that no undirected edge is in $\delta^+(S)$. Then the dual solution we construct will be feasible for the constraints

$$\sum_{S:a \in \delta^+(S)} y(S) \leq c(a) \qquad \text{for all } a \in A. \tag{1}$$

Note that the constraints are only over the directed arcs of the mixed graph and not the undirected edges. We say that a constraint is *tight* for some arc a if $\sum_{S:a \in \delta^+(S)} y(S) = c(a)$. We will sometimes simply say that the arc a is tight.

We give a dual-based version of the algorithm in Algorithm 2. Initially we set $y(S)$ to 0 for all $S \subseteq V$, and we maintain a selected set of edges F, initially empty.

For a connected component C of F at some iteration of the algorithm, we say C is *active* if $h(C) = 1$ and inactive otherwise. We increase $y(C)$ uniformly for all active components C until either $\sum_{S:e\in\delta(S)} y(S) \geq c(e)$ for some good edge e, or $\sum_{S:a'\in\delta^+(S)} y(S) = c(a')$ for some bad arc a' that is an orientation of bad edge e'. In the former case, we add good edge e to F, otherwise we add bad edge e'.

For the purposes of analysis, it is also useful to maintain a quantity t, called the *time* of the algorithm. It is initially zero, and in each iteration is increased by the same amount as each $y(C)$ is increased for active C.

We would like to prove that the two algorithms are equivalent. First we need the following lemma.

Lemma 1 *Consider any component C at the start of an iteration of the main loop of Algorithm 2. For any arc (u,v) with $u \in C$, $v \notin C$, and C active, $\sum_{S:(u,v)\in\delta^+(S)} y(S) = t$ at the start of the iteration.*

Proof We prove this by induction on Algorithm 2. The statement is true at the start of the algorithm since $t = 0$ and all $y(S) = 0$. Suppose the statement is true at the start of the kth iteration of the algorithm; we prove it is true at the start of the $(k+1)$st iteration. Consider an arc (u,v) and an active component C from the $(k+1)$st iteration, where $u \in C$ and $v \notin C$. Then note that since the algorithm only merges components, whatever component $C' \subseteq C$ that contained u at the previous iteration did not contain v, and since h is downwards monotone, C' was also active. Thus at the time t at the start of the kth iteration, $\sum_{S:(u,v)\in\delta^+(S)} y(S) = t$. Since C' is a component at the start of the kth iteration and it is active, both sides of the equality increase by the same amount in the kth iteration, and the equation continues to hold. □

Now we can prove the equivalence.

Lemma 2 *Algorithm 2 is equivalent to Algorithm 1.*

Proof We show that in each iteration, the algorithms add the same edge. By Lemma 1, we have that for any arc (u,v) with $u \in C$, C active, $v \notin C$, $\sum_{S:(u,v)\in\delta^+(S)} y(S) = t$ at the start of the iteration. Thus for all such arcs $a' = (u,v)$, we can increase $y(C)$ by at most $c(a') - t$ before case (2) applies in Algorithm 2. Thus case (2) will apply to the arc $a' = (u,v)$ of minimum cost such that $u \in C$ for active C and $v \notin C$. The edge e' corresponding to a' is then the cheapest bad edge joining two components C and C' in the current solution, at least one of which is active. Case (1) in Algorithm 2 applies to good edges $e = (u,v)$. For such a good edge $e = (u,v)$, it must be that $u \in C$, $v \in C'$, and both C and C' are active and distinct; then at the start of the iteration, $\sum_{S:e\in\delta(S)} y(S) = \sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) = 2t$ by Lemma 1. Thus for such edges, we can increase $y(C)$ and $y(C')$ each by at most $\max(\frac{1}{2}c(e) - t, 0)$ before case (1) applies, since after such an increase, $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) \geq 2t + c(e) - 2t = c(e)$. Thus Algorithm 2 adds the cheapest good edge e instead of the cheapest bad edge e' exactly when $\max(\frac{1}{2}c(e) - t, 0) \leq c(e') - t$, or when $c(e) \leq 2c(e')$, which implies that Algorithm 2 adds exactly the same edge in the iteration as Algorithm 1. □

Corollary 1 *For any bad edge $e' = (u,v)$ added by the algorithm, if $u \in C$ such that $h(C) = 1$, then the arc (u,v) is tight. For any good edge e added by the algorithm, $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) \geq c(e)$.*

Lemma 3 *At the end of the algorithm, the dual solution y is feasible for the constraints (1).*

Proof The statement follows by construction of Algorithm 2; the algorithm maintains the feasibility of the constraints. □

We now show that the cost of the algorithm's solution F is at most the sum of the dual variables. We will prove this on a component-by-component basis. To do this it is useful to think about directing most of the edges of the component. We now explain how we direct edges.

Consider component C of F; overloading notation, let C stand for both the set of vertices and the set of edges of the component. At the start of the algorithm, each vertex of C is in its own component, and these components are repeatedly merged until C is a component in the algorithm's solution F; at any iteration of the algorithm call the connected components whose vertices are subsets of C the *subcomponents* of C. We say that a subcomponent has *h-value* of 0 (1, respectively) if for the set of vertices S of the subcomponent $h(S) = 0$ ($h(S) = 1$ respectively). We claim that at any iteration there can be at most one subcomponent of h-value 0. Note that the algorithm never merges components both of h-value 0, and any merging of two components, one of which has h-value 1 and the other 0, must result in a component of h-value 0 by the properties of h. So if there were in any iteration two subcomponents of C both with h-value 0, the algorithm would not in any future iteration add an edge connecting the vertices in the two subcomponents, which contradicts the connectivity of C. It also follows from this reasoning that at some iteration (perhaps initially) the first subcomponent appears with h-value 0, and there is a single such subcomponent from then on. If there is a vertex $v \in C$ with $h(\{v\}) = 0$, then there is such a subcomponent initially; otherwise, such a subcomponent is formed by adding a good edge e^* to merge two subcomponents of h-value 1. In the first case, we consider directing all the edges in the component towards v and we call v the root of C. In the second case, we think about directing all the edges to the two endpoints of e^* and we call these vertices the roots of C; the edge e^* remains undirected. In either case, we say that directing the edges of the component in this way makes the component *properly birooted*; let the corresponding set of arcs (plus perhaps the one undirected edge) be denoted by $\overrightarrow{C}$.

We can now prove the needed lemma.

Lemma 4 *At the end of the algorithm $\sum_{e\in F} c(e) \leq \sum_S y(S)$.*

Proof As above, we concentrate on a single connected component C of F. Then we show that $\sum_{e\in C} c(e) \leq \sum_{S\subseteq C} y(S)$. Summing this inequality over all components gives the lemma statement.

By Corollary 1, we observe that whenever the algorithm adds a bad edge $e' = (u,v)$ joining two components C_1 and C_2 with $h(C_1) = h(C_2) = 1$, then the constraints

for both arcs (u,v) and (v,u) are tight. If $h(C_1) = 1$ and $h(C_2) = 0$ with $u \in C_1$, then the constraint for arc (u,v) is tight. Also if a good edge $e = (u,v)$ is added, then $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) \geq c(e)$.

It follows from these observations that all the arcs of $\overrightarrow{C}$ are tight; whenever we add an edge (u,v) merging two subcomponents C_1 and C_2 of C with $h(C_1) = 1$ and $h(C_2) = 0$, $u \in C_1$ and $v \in C_2$, the root(s) of C must be in C_2, and the arc (u,v) is tight. Additionally since all edges are directed towards the root(s), there is at most one arc directed out of any subcomponent of C in any iteration of the algorithm, so for any dual variable $y(S) > 0$ with $S \subseteq C$, $|\delta^+(S) \cap \overrightarrow{C}| \leq 1$; $|\delta^+(S) \cap \overrightarrow{C}| = 0$ if and only if S contains the one root, or at least one of the two roots, of C. In the case of a single root r, it then follows that

$$\begin{aligned}\sum_{e\in C} c(e) = \sum_{a\in\overrightarrow{C}} c(a) &= \sum_{a=(u,v)\in\overrightarrow{C}} \sum_{S:(u,v)\in\delta^+(S)} y(S) \\ &= \sum_{S\subseteq C} y(S)|\delta^+(S)\cap\overrightarrow{C}| \\ &\leq \sum_{S\subseteq C} y(S).\end{aligned}$$

In the case a good edge $e^* = (u,v)$ was added to C, and u,v are the roots of C, then

$$\begin{aligned}\sum_{e\in C} c(e) &= c(e^*) + \sum_{a\in\overrightarrow{C}} c(a) \\ &= c(e^*) + \sum_{a=(x,y)\in\overrightarrow{C}} \sum_{S:(x,y)\in\delta^+(S)} y(S) \\ &= c(e^*) + \sum_{S\subseteq C:u,v\notin S} y(S)|\delta^+(S)\cap\overrightarrow{C}| \\ &\leq \sum_{S\subseteq C:(u,v)\in\delta^+(S)} y(S) + \sum_{S\subseteq C:(v,u)\in\delta^+(S)} y(S) + \sum_{S\subseteq C:u,v\notin S} y(S) \\ &= \sum_{S\subseteq C} y(S).\end{aligned}$$

□

3.2 Proof of the performance guarantee

We use a dual fitting argument to show the performance guarantee. For a feasible solution F^*, let C^* be some connected component. We say $C^* \in \delta(S)$ for a subset S of vertices if there is some edge e in C^* such that $e \in \delta(S)$. We will show the following lemma.

Lemma 5 *Let F^* be a feasible solution to the problem, and let C^* be any component of F^*. Let $y(S)$ be the dual variables returned by the algorithm. Then*

$$\sum_{S:C^*\in\delta(S)} y(S) \leq \frac{3}{2} \sum_{e\in C^*} c(e).$$

From the lemma, we can easily derive the performance guarantee.

Theorem 1 *Algorithm 1 is a $\frac{3}{2}$-approximation algorithm.*

Proof Let F^* be an optimal solution to the problem, and let $\mathscr{C}^*$ be its connected components. Then

$$\frac{3}{2}\sum_{e\in F^*} c(e) = \frac{3}{2}\sum_{C^*\in\mathscr{C}^*}\sum_{e\in C^*} c(e) \geq \sum_{C^*\in\mathscr{C}^*}\sum_{S:C^*\in\delta(S)} y(S),$$

where the last inequality follows by Lemma 5. Let F be the solution returned by the algorithm. By Lemma 4, we have that

$$\sum_{e\in F} c(e) \leq \sum_{S} y(S).$$

Since we only increased variables $y(S)$ for subsets S with $h(S) = 1$, it is clear that if $y(S) > 0$, then there must exist some $C^* \in \mathscr{C}^*$ with $C^* \in \delta(S)$ in order for the solution to be feasible. Thus

$$\sum_{e\in F} c(e) \leq \sum_{S} y(S) \leq \sum_{C^*\in\mathscr{C}^*}\sum_{S:C^*\in\delta(S)} y(S) \leq \frac{3}{2}\sum_{e\in F^*} c(e).$$

□

We now turn to proving Lemma 5. The essence of the analysis is that the feasibility of the dual solution shows that the sum of most of the duals is no greater than the cost of all but one edge e in the component (the edge e that gives the birooted property). We then need to account for the duals that intersect this edge e or that contain both of its endpoints. If the sum of these duals is sufficiently small, then Lemma 5 follows. If the sum of these duals is not small, then we can show that there must be another edge e' in the component that has a large cost, and we can charge these duals to the cost of e'.

As a warmup to our techniques for proving Lemma 5, we prove a simple special case of the lemma by orienting the arcs of the component and using the dual feasibility (1).

Lemma 6 *Given any connected component C^* of a feasible solution F^*, if there is a vertex $v \in C^*$ such that $h(\{v\}) = 0$, then $\sum_{S:C^*\in\delta(S)} y(S) \leq \sum_{e\in C^*} c(e)$.*

Proof If there is a vertex $v \in C^*$ such that $h(\{v\}) = 0$, we consider a directed version of C^* which we call $\overrightarrow{C^*}$ in which all the edges are directed towards v. Because h is downwards monotone and $h(\{v\}) = 0$, any set S containing v has $h(S) = 0$ and therefore $y(S) = 0$ since we only increase y on sets S' with $h(S') = 1$. We say that $\overrightarrow{C^*} \in \delta^+(S)$ if there is some arc of $\overrightarrow{C^*}$ with a tail in S and head not in S. Then by the

previous discussion $\sum_{S:C^* \in \delta(S)} y(S) = \sum_{S:\overrightarrow{C^*} \in \delta^+(S)} y(S)$. By (1),

$$\begin{aligned}\sum_{e \in C^*} c(e) = \sum_{a \in \overrightarrow{C^*}} c(a) &\geq \sum_{a \in \overrightarrow{C^*}} \sum_{S:a \in \delta^+(S)} y(S) \\ &= \sum_{S:\overrightarrow{C^*} \in \delta^+(S)} \sum_{a \in \overrightarrow{C^*}:a \in \delta^+(S)} y(S) \\ &\geq \sum_{S:\overrightarrow{C^*} \in \delta^+(S)} y(S) = \sum_{S:C^* \in \delta(S)} y(S).\end{aligned}$$

□

We would like to prove something similar in the general case; however, if we do not have a vertex v with $h(\{v\}) = 0$, then there might not be any orientation of the arcs such that if $C^* \in \delta(S)$ and $y(S) > 0$ then $\overrightarrow{C^*} \in \delta^+(S)$, as there is in the previous case. Instead, we will use a birooted component as we did previously, and argue that we can make the lemma hold for this case. To orient the arcs of C^*, we order the edges of C^* by increasing cost, and consider adding them one-by-one, repeatedly merging subcomponents of C^*. The first edge that merges two subcomponents of h-value 1 to a subcomponent of h-value 0 we call the undirected edge of C^*, and we designate it $e^* = (u^*, v^*)$. We now let $\overrightarrow{C^*}$ be the orientation of all the edges of C^* except e^* towards the two roots u^*, v^*; e^* remains undirected. Let

$$\begin{aligned}t(u^*) &= \sum_{S:u^* \in S, v^* \notin S} y(S), \\ t(v^*) &= \sum_{S:u^* \notin S, v^* \in S} y(S), \text{ and} \\ t(e^*) &= \sum_{S:u^*, v^* \in S} y(S).\end{aligned}$$

We begin with an observation and a few lemmas. Throughout what follows we will use the quantity t as an index for the algorithm. When we say "at time t", we mean the last iteration of the algorithm at which the time is t (since the algorithm may have several iterations where t is unchanged). For a set of edges X, let $\max(X) = \max_{e \in X} c(e)$.

Observation 1 *At time t, the algorithm has considered adding all edges of cost at most t to the solution and any active component must consist of edges of cost at most t. At time t, if component C is active, then any edge $e \in \delta(C)$ must have cost greater than t.*

Lemma 7 $\max\{t(u^*) + t(e^*), t(v^*) + t(e^*)\} \leq c(e^*)$.

Proof Since $\overrightarrow{C^*}$ is birooted, we claim that there is some connected set of edges $X \subseteq C^*$ that contains u^* and v^* with $\max(X) \leq c(e^*)$ and $h(X) = 0$ (again treating X both as a connected set of edges and the associated nodes). To see the claim, simply repeat the process described above of ordering the edges of C^* by increasing cost, adding them until we merge two subcomponents of h-value 1 into a subcomponent of h-value 0; let

X be this subcomponent. At time $c(e^*)$, in the partial solution created thus far by the algorithm, u^* and v^* must be in inactive components (possibly the same component); by Observation 1, if either u^* or v^* was not in an inactive component then, since $\max(X) \leq c(e^*)$, the algorithm would have connected some superset of the vertices of X which, because $h(X) = 0$, is inactive. Thus at time $c(e^*)$ the algorithm is no longer increasing duals $y(S)$ for S containing either u^* or v^*. Then as u^* is at each point in time in at most one active set S, one has $t(u^*) + t(e^*) \leq c(e^*)$, and similarly for v^*. The inequality of the lemma follows. □

We will make extensive use of the following lemma, which tells us that the algorithm finds components of small maximum edge cost. Let C_u^t be the connected component constructed by the algorithm at time t that contains u (similarly, C_v^t).

Lemma 8 *Consider two vertices u and v and suppose the components C_u^t, C_v^t are active for all times $t < t'$ (they need not be disjoint). Then for any connected set of edges $C(u)$ in the original graph containing u (similarly $C(v)$) if $h(C(u) \cup C(v)) = 0$ while $h(C_u^t \cup C_v^t) = 1$ for all times $t < t'$, then $\max(C(u) \cup C(v)) \geq t'$.*

Proof Consider the algorithm at any time $t < t'$. Since C_u^t and C_v^t are active, by Observation 1, any edge in $\delta(C_u^t)$ or $\delta(C_v^t)$ must have cost greater than t; also, any edge in the components must have cost at most t. If $\max(C(u) \cup C(v)) \leq t$, it must be that the vertices of $C(u)$ are a subset of those of C_u^t, and similarly the vertices of $C(v)$ are a subset of those of C_v^t. This implies $C(u) \cup C(v) \subseteq C_u^t \cup C_v^t$, and since $h(\cdot)$ is downward monotone, $h(C(u) \cup C(v)) \geq h(C_u^t \cup C_v^t)$, which is a contradiction. So it must be the case that $\max(C(u) \cup C(v)) > t$. Since this is true for any time $t < t'$, it follows that $\max(C(u) \cup C(v)) \geq t'$. □

We now split the proof into two cases, depending on whether $\min(t(u^*) + t(e^*), t(v^*) + t(e^*)) > \frac{1}{2}c(e^*)$ or not. We first suppose that it is true.

Lemma 9 *If $\min(t(u^*) + t(e^*), t(v^*) + t(e^*)) > \frac{1}{2}c(e^*)$, then C^* must have some edge e' other than e^* of cost $c(e') \geq \min(t(u^*) + t(e^*), t(v^*) + t(e^*))$.*

Proof First assume that $t(e^*) > 0$; this implies that $t(u^*) = t(v^*)$ since u^* and v^* must be in active components until the point in time (at time $t(u^*) = t(v^*)$) when they are merged into a single component, which is then active until time $t(u^*) + t(e^*)$. Then for all times $t < \min(t(u^*) + t(e^*), t(v^*) + t(e^*))$, u^* and v^* are in active components $C_{u^*}^t$ and $C_{v^*}^t$, and for times t with $t(u^*) \leq t < t(u^*) + t(e^*)$, in which u^* and v^* are in the same component, $h(C_{u^*}^t \cup C_{v^*}^t) = 1$. Let $C(u^*)$ be the component containing u^* in $C^* - e^*$ and $C(v^*)$ be the component containing v^* in $C^* - e^*$. Since $h(C(u^*) \cup C(v^*)) = h(C^*) = 0$, we can apply Lemma 8, and the lemma follows.

Now assume that $t(e^*) = 0$. In this case, u^* and v^* are never in an active component together for any positive length of time (since $y(S) = 0$ for any S containing both u^* and v^*). Assume $t(u^*) = \min(t(u^*) + t(e^*), t(v^*) + t(e^*)) = \min(t(u^*), t(v^*)) > \frac{1}{2}c(e^*)$. For all $t < t(u^*)$, $C_{u^*}^t$ and $C_{v^*}^t$ are active components. Furthermore, since the algorithm did not add edge e^* during time $\frac{1}{2}c(e^*) \leq t < t(u^*)$, it must have been the case that e^* was not a good edge during that period of time: otherwise e^* would have been cheaper than whatever edge(s) the algorithm was adding in that period of time.

Since both $C^t_{u^*}$ and $C^t_{v^*}$ are active components for $t < t(u^*)$, it must have been the case that $h(C^t_{u^*} \cup C^t_{v^*}) = 1$ since otherwise e^* would have been a good edge. Thus we can apply Lemma 8: again, let $C(u^*)$ be the component containing u^* in $C^* - e^*$ and $C(v^*)$ be the component containing v^* in $C^* - e^*$. We know that $h(C(u^*) \cup C(v^*)) = 0$ since $h(C^*) = 0$. Thus there must be an edge e' in $C(u^*) \cup C(v^*)$ of cost at least $t(u^*)$. □

Finally, we can prove Lemma 5.

Proof of Lemma 5: If there is $v \in C^*$ with $h(\{v\}) = 0$, then the statement follows from Lemma 6. If not, then we can get a directed, birooted version $\overrightarrow{C^*}$ of C^* with undirected edge $e^* = (u^*, v^*)$. For any set of vertices S with $C^* \in \delta(S)$, either there is an arc of $\overrightarrow{C^*}$ directed out of S or S contains one (or both) of the two roots u^* and v^*. Thus

$$\begin{aligned}\sum_{S:C^* \in \delta(S)} y(S) &\leq \sum_{S:\overrightarrow{C^*} \in \delta^+(S)} y(S) + \sum_{S:u^* \in S, v^* \notin S} y(S) + \sum_{S:u^* \notin S, v^* \in S} y(S) + \sum_{S:u^*, v^* \in S} y(S) \\ &\leq \sum_{a \in \overrightarrow{C^*}} \sum_{S:a \in \delta^+(S)} y(S) + t(u^*) + t(v^*) + t(e^*) \\ &\leq \sum_{a \in \overrightarrow{C^*}} c(a) + t(u^*) + t(e^*) + t(v^*) + t(e^*). \end{aligned} \tag{2}$$

Without loss of generality, suppose that $t(u^*) + t(e^*) = \min(t(u^*) + t(e^*), t(v^*) + t(e^*))$. If $t(u^*) + t(e^*) \leq \frac{1}{2} c(e^*)$, then

$$\begin{aligned}\sum_{S:C^* \in \delta(S)} y(S) &\leq \sum_{a \in \overrightarrow{C^*}} c(a) + \frac{1}{2} c(e^*) + c(e^*), \\ &\leq \frac{3}{2} \sum_{e \in C^*} c(e).\end{aligned}$$

where the first inequality follows by our assumption and by Lemma 7.

If $t(u^*) + t(e^*) > \frac{1}{2} c(e^*)$, then by Lemma 9, there is an edge $e' \in C^*$, $e' \neq e^*$, of cost at least $t(u^*) + t(e^*)$. Let a' be the directed version of e' in $\overrightarrow{C^*}$. Since $c(a') \geq t(u^*) + t(e^*)$ and by Lemma 7 $c(e^*) \geq t(v^*) + t(e^*) \geq t(u^*) + t(e^*)$, then $t(u^*) + t(e^*) \leq \frac{1}{2}(c(a') + c(e^*))$. Then picking up from inequality (2) above, we have

$$\begin{aligned}\sum_{S:C^* \in \delta(S)} y(S) &\leq \sum_{a \in \overrightarrow{C^*}} c(a) + t(u^*) + t(e^*) + t(v^*) + t(e^*) \\ &\leq \sum_{a \in \overrightarrow{C^*}, a \neq a'} c(a) + c(a') + \frac{1}{2}(c(a') + c(e^*)) + c(e^*) \\ &= \sum_{a \in \overrightarrow{C^*}, a \neq a'} c(a) + \frac{3}{2} c(a') + \frac{3}{2} c(e^*) \\ &\leq \frac{3}{2} \sum_{e \in C^*} c(e),\end{aligned}$$

and we are done. □

3.3 An alternate perspective

We now give an alternate perspective on the analysis above: our algorithm creates an infeasible dual solution for the dual of a linear programming relaxation of a particular integer programming formulation of the problem. The integer programming formulation includes a primal variable $x(\overrightarrow{C})$ for each possible birooted component $\overrightarrow{C}$. Define $\gamma(S)$ to contain all birooted components $\overrightarrow{C}$ such that $\overrightarrow{C}$ has an arc leaving S, or S contains one (or both) of the roots of $\overrightarrow{C}$. Then the integer programming formulation is attempting to find a minimum-cost collection of birooted components such that each set S with $h(S) = 1$ contains some $\overrightarrow{C} \in \gamma(S)$ with $x(\overrightarrow{C}) = 1$. Let $c(\overrightarrow{C})$ be the cost of the arcs and edge in $\overrightarrow{C}$. Then the formulation is

$$\text{Min} \sum_{\overrightarrow{C}} c(\overrightarrow{C}) x(\overrightarrow{C})$$

subject to:

$$\sum_{\overrightarrow{C}:\overrightarrow{C}\in\gamma(S)} x(\overrightarrow{C}) \geq h(S), \qquad \forall S \subset V, S \neq \emptyset,$$

$$x(\overrightarrow{C}) \in \{0,1\}, \qquad \forall \overrightarrow{C}.$$

We can relax this to a linear program by replacing the integrality constraints with $x(\overrightarrow{C}) \geq 0$. The dual of the LP relaxation is then

$$\text{Max} \sum_{S} h(S) z(S)$$

subject to:

$$\sum_{S:\overrightarrow{C}\in\gamma(S)} z(S) \leq c(\overrightarrow{C}), \qquad \forall \overrightarrow{C},$$

$$z(S) \geq 0, \qquad \forall S \subset V, S \neq \emptyset.$$

Note that given variables y that obey the constraints (1), the solution $z = y$ is almost feasible for the dual above, since the sum of the duals $z(S)$ containing the arcs of a birooted component $\overrightarrow{C}$ are at most the cost of the arcs; it is only the duals $z(S)$ containing the root (or roots) of $\overrightarrow{C}$ that might be more than the cost of the undirected edge of $\overrightarrow{C}$. The proof of the previous subsection shows that scaling z down by 2/3 gives a feasible solution for the above dual of the linear programming relaxation. Our proof also shows that the integrality gap of the integer programming formulation above is 3/2.

4 Conclusion

Goemans and Williamson [6] give a 2-approximation algorithm for functions $h : 2^V \to \mathbb{N}$, if whenever $S \subseteq T$, $h(S) \geq h(T)$. In this case, we wish to select a multiset of edges F (that is, we may choose multiple copies of the same edge) such that for each $S \subset V$, the number of copies of edges in F with exactly one endpoint in S is

at least $h(S)$. It would be interesting to know whether the ideas of this paper can be extended to give a better approximation algorithm for that case.

Also, as we have already mentioned, it would be very interesting to extend this algorithm from the class of downwards monotone functions to the class of proper functions defined by Goemans and Williamson [7]. Recall that a function $f : 2^V \to \{0,1\}$ is *proper* if $f(S) = f(V-S)$ for all $S \subseteq V$ and $f(A \cup B) \leq \max(f(A), f(B))$ for all disjoint $A, B \subset V$. This class includes problems such as the Steiner tree problem and the generalized Steiner tree problem; currently no ρ-approximation algorithm for constant $\rho < 2$ is known for the latter problem. However, our algorithm makes extensive use of the property that we grow duals around active components until they are inactive, then no further dual is grown around that component; the algorithm of [7] may start growing duals around components that were previously inactive. The first step in extending the algorithm of this paper to proper functions might well be to consider the Steiner tree problem, since the dual-growing algorithm of [7] in this case does have the property that once a component becomes inactive, no further dual is grown around it.

Acknowledgements

We thank anonymous reviewers of this paper for useful comments.

References

1. Archer, A., Bateni, M., Hajiaghayi, M., Karloff, H.: Improved approximation algorithms for prize-collecting Steiner tree and TSP. SIAM Journal on Computing **40**, 309–332 (2011)
2. Bazgan, C., Couëtoux, B., Tuza, Z.: Complexity and approximation of the Constrained Forest problem. Theoretical Computer Science **412**, 4081–4091 (2011)
3. Byrka, J., Grandoni, F., Rothvoß, T., Sanità, L.: An improved LP-based approximation for Steiner tree. In: Proceedings of the 42nd Annual ACM Symposium on Theory of Computing, pp. 583–592 (2010)
4. Couëtoux, B.: A $\frac{3}{2}$ approximation for a constrained forest problem. In: C. Demetrescu, M.M. Halldórsson (eds.) Algorithms – ESA 2011, 19th Annual European Symposium, no. 6942 in Lecture Notes in Computer Science, pp. 652–663. Springer (2011)
5. Davis, J.M., Williamson, D.P.: A dual-fitting $\frac{3}{2}$-approximation algorithm for some minimum-cost graph problems. In: L. Epstein, P. Ferragina (eds.) Algorithms – ESA 2012, 20th Annual European Symposium, no. 7501 in Lecture Notes in Computer Science, pp. 373–382. Springer (2012)
6. Goemans, M.X., Williamson, D.P.: Approximating minimum-cost graph problems with spanning tree edges. Operations Research Letters **16**, 183–189 (1994)
7. Goemans, M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. SIAM Journal on Computing **24**, 296–317 (1995)
8. Goemans, M.X., Williamson, D.P.: The primal-dual method for approximation algorithms and its application to network design problems. In: D.S. Hochbaum (ed.) Approximation Algorithms for NP-hard Problems, chap. 4. PWS Publishing, Boston, MA, USA (1996)
9. Imielińska, C., Kalantari, B., Khachiyan, L.: A greedy heuristic for a minimum-weight forest problem. Operations Research Letters **14**, 65–71 (1993)
10. Kruskal, J.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7**, 48–50 (1956)
11. Laszlo, M., Mukherjee, S.: Another greedy heuristic for the constrained forest problem. Operations Research Letters **33**, 629–633 (2005)
12. Laszlo, M., Mukherjee, S.: A class of heuristics for the constrained forest problem. Discrete Applied Mathematics **154**, 6–14 (2006)

13. Laszlo, M., Mukherjee, S.: An approximation algorithm for network design problems with downwards-monotone demand functions. Optimization Letters **2**, 171–175 (2008)

Appendix

We illustrate a run of Algorithm 1 on the problem in which we want to find a minimum-cost set of edges such that each component has at least four vertices. We give an instance of the problem in which the graph contains edges of cost 0, 1, 2, or $2+\varepsilon$ to keep things simple.

The instance is described in the Figure 3; we give an optimal solution and a solution produced by the algorithm. We draw good edges with a double line.

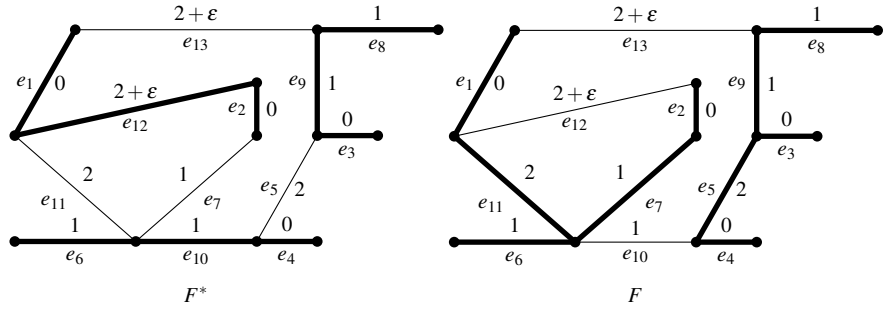

Fig. 3: The graph on the left gives an optimal solution of cost $6+\varepsilon$ and the graph on the right gives a solution produced by the algorithm of cost 8.

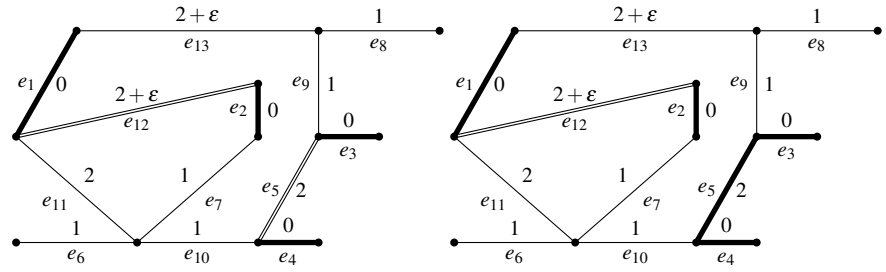

Fig. 4: The first four iterations of the algorithm pick the four edges of cost 0 (shown on the left). On the right, in the next iteration the algorithm picks e_5 as a good edge.

In Figure 4, in the first four iterations, the algorithm picks all four edges of cost 0. In the next iteration, the cheapest good edge (e_5) has cost 2, and the remaining bad edges all have cost 1, so the algorithm chooses the good edge e_5.

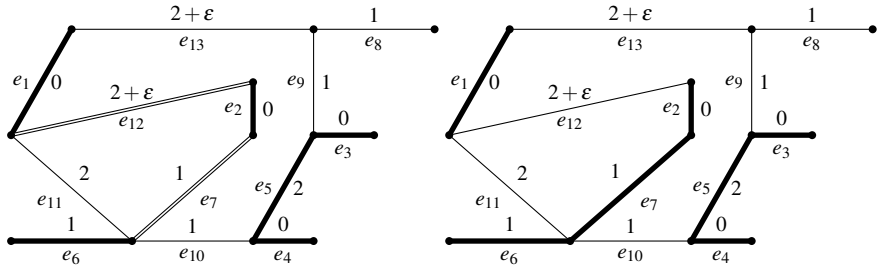

Fig. 5: The algorithm adds e_6 and e_7 to the partial solution.

In Figure 5, the algorithm now picks a bad edge of cost 1, e_6, since this is less than half the cost of the cheapest good edge (e_{12}, of cost $2+\varepsilon$). Adding this edge to the solution makes the edge e_7 a good edge. Since e_7 is of cost 1, the algorithm adds it as a good edge. After this, e_{12} is no longer a good edge.

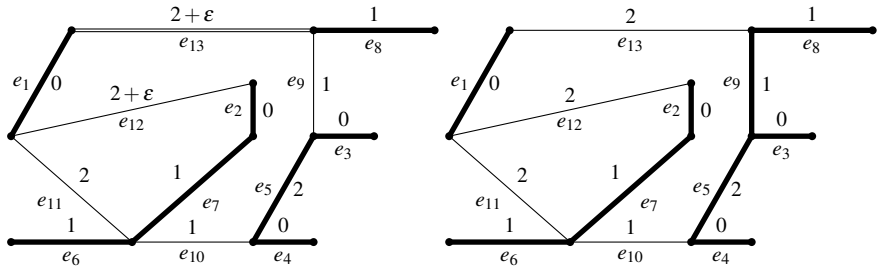

Fig. 6: The algorithm adds e_8 and e_9 to the partial solution.

In Figure 6, there are only bad edges; the algorithm picks e_8, which is of cost 1. The addition of e_8 makes e_{13} a good edge. Then the algorithm picks e_9, after which e_{13} is no longer a good edge.

In Figure 7, edge e_{10} is the last bad edge of cost 1 and, when it is examined, its addition would link two big trees, therefore it isn't added to the partial solution. Edge e_{11} is one of the bad edges of cost 2 remaining; the algorithm adds it to the solution. Now edge e_{12} would form a cycle with the partial solution and edge e_{13} would link two big trees, so neither of them are added to the solution. The algorithm returns the indicated set of edges, and each component has at least 4 vertices in it.

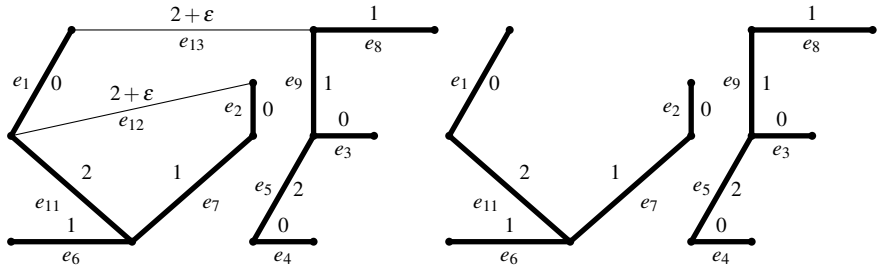

Fig. 7: On the left, e_{10} and e_{11} has been examined. On the right the two last edges e_{12} and e_{13} has been examined and removed.