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

James M. Davis[⋆] and David P. Williamson[⋆⋆]

Address: School of Operations Research and Information Engineering,
Cornell University, Ithaca, NY 14853, USA.
jmd388@cornell.edu, dpw@cs.cornell.edu

Abstract. In an ESA 2011 paper, Couëtoux [2] gives a beautiful $\frac{3}{2}$-approximation algorithm for the problem of finding a minimum-cost set of edges such that each connected component has at least k vertices in it. The algorithm improved on previous 2-approximation algorithms for the problem. In this paper, we reanalyze Couëtoux's algorithm using dual-fitting and show how to generalize the algorithm to a broader class of graph problems previously considered in the literature.

1 Introduction

Consider the following graph problem: given an undirected graph $G = (V, E)$ with nonnegative edge weights $c(e) \geq 0$ for all $e \in E$, and a positive integer k, find a 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 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 [6] for $k \geq 4$ and Bazgan, Couëtoux, and Tuza [1] 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. Imielińska, Kalantari, and Khachiyan [6] 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 [7] for the minimum spanning tree problem that 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 [8, 9], who also perform some experimental comparisons.

[⋆] This material is based upon work supported by the National Science Foundation under Grant No. DGE-0707428.

[⋆⋆] This work is supported in part by NSF grant CCF-1115256.

In an ESA 2011 paper, Couëtoux [2] introduces a beautiful little twist to the Imielińska et al. algorithm and shows that the result is a $\frac{3}{2}$-approximation algorithm. 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. As the algorithm considers whether to add edge e to the solution (at cost $c(e)$), it also considers all edges e' of cost $c(e') \leq 2c(e)$, and checks whether adding such an edge e' would join two small components into a big component. If any such edge e' exists, the algorithm adds the edge to the solution (if there is more than one, it adds the cheapest such edge), otherwise it returns to considering the addition of edge e.

In another thread of work, Goemans and Williamson [3] show that the Imielińska et al. algorithm can be generalized to provide 2-approximation algorithms to a large class of graph problems. The graph problems are specified by a function $h : 2^V \rightarrow \{0, 1\}$ (actually, [3] considers a more general case of functions $h : 2^V \rightarrow \mathbb{N}$, 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 of vertices S. 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 [3] 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 [10] 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 and 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 show that the natural generalization of Couëtoux's algorithm to downwards monotone functions gives $\frac{3}{2}$-approximation algorithms for this class of problems. In the process, we give a different and (we believe) somewhat simpler analysis than Couëtoux. 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 [5] 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 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, but these results are isolated, and do not extend to well-defined classes of problems. It would be very interesting, for instance, to give an approximation algorithm for the class of proper functions (defined in Goemans and Williamson [4]) with a constant performance guarantee smaller than 2.

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 $h(C_1 \cup C_2) = 0$; we then look for the cheapest 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 less than twice the cost of the 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.

3 Analysis of the algorithm

3.1 Some preliminaries

To give the analysis of the algorithm, we implicitly have the algorithm construct a dual solution y as it runs. 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

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

```
$F \leftarrow \emptyset$
// Set dual solution $y(S) \leftarrow 0$ for all sets $S$, $t \leftarrow 0$
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
        // Implicitly increase $y(C)$ by $h(C)\max(0, \frac{1}{2}c(e) - t)$ for all
           connected components $C$, $t$ increases by $\max(0, \frac{1}{2}c(e) - t)$
        $F \leftarrow F \cup \{e\}$
    else
        // Implicitly increase $y(C)$ by $h(C)(c(e') - t)$ for all connected
           components $C$, $t$ increases by $c(e') - t$
        $F \leftarrow F \cup \{e'\}$
Return $F$
```

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

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.

As given in the comments of Algorithm 1, initially $y(S) \leftarrow 0$ for all $S \subseteq V$. We keep track of a quantity t, initially zero; we call t the *time*. If we add a good edge e joining two components, we increase $y(C)$ by $h(C)(\max(0, \frac{1}{2}c(e) - t))$ for all current connected components of F (before adding edge e) and update t to $\max(t, \frac{1}{2}c(e))$. If we add a nongood edge e' joining two components, we increase $y(C)$ by $h(C)(c(e')-t)$ for all connected components C of F (before adding edge e') and update t to $c(e')$.

For a component C at some iteration of the algorithm, we say C is *active* if $h(C) = 1$ and inactive otherwise. We observe that because the algorithm repeatedly merges components, and because the function h is downwards monotone, any vertex u in an inactive component C is never again part of an active component; this follows since at later iterations $u \in C'$ implies $C \subseteq C'$, and since $h(C) = 0$ it must be the case that $h(C') = 0$ also.

Lemma 1. *Consider any component C at the start of an iteration of the main loop of the algorithm. 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 the algorithm. 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'

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 inequality continues to hold. $\square$

Lemma 2. *At the end of the algorithm, the dual solution y is feasible for the constraints (1).*

Proof. To prove the lemma, we prove two statements by induction on the iterations of the algorithm: first, that t is nondecreasing over the algorithm, and second that the dual solution y is feasible for the constraints (1) at the start of each iteration. Initially, since all $c(a) \geq 0$ and all $y(S) = 0$, the constraints (1) are obeyed. Now consider an arbitrary iteration of the algorithm. If we add a good edge in this iteration, then t cannot decrease. If we add a nongood edge $e' = (u,v)$ in this iteration, then one of the two components it joins (call it C) must have $h(C) = 1$ and thus was active at the start of this iteration. Suppose $u \in C$. Then since $v \notin C$, by Lemma 1, $\sum_{S:(u,v)\in\delta^+(S)} y(S) = t$. Since by induction, the constraints (1) were obeyed for arc $a = (u,v)$, $\sum_{S:(u,v)\in\delta^+(S)} y(S) \leq c(a) = c(e')$. Thus $c(e') \geq t$ or $c(e') - t \geq 0$. Hence t is nondecreasing in this iteration.

Now consider a given arc (u,v) corresponding to an edge $\hat{e}$. At the beginning of the iteration, the corresponding constraint is feasible. If both u and v are in the same connected component, then $\sum_{(u,v)\in\delta^+(S)} y(S)$ does not increase in the iteration, and the constraint remains feasible. The same is true if u is in a component C and v is in a different component but $h(C) = 0$. So we can suppose that $u \in C$, $h(C) = 1$ and v is in a different component; by Lemma 1, $\sum_{S:(u,v)\in\delta^+(S)} y(S) = t$, and potentially edge $\hat{e} = (u,v)$ can be added in this iteration. The algorithm adds either the cheapest good edge e (if one exists) or the cheapest nongood edge e'. If the nongood edge e' is added, then $\sum_{S:(u,v)\in\delta^+(S)} y(S)$ increases by $c(e')-t$ to $c(e') \leq c(\hat{e})$, where the inequality follows since $c(e')$ is the minimum-cost nongood edge. Thus the constraint continues to hold at the end of the iteration, and is tight if $e' = \hat{e}$ (that is, $\hat{e}$ is the selected edge). Now suppose that the good edge e is added. Then $c(e) \leq 2c(e') \leq 2c(\hat{e})$, and $\sum_{S:(u,v)\in\delta^+(S)} y(S)$ increases from t by $\max(0, \frac{1}{2}c(e) - t) \leq \frac{1}{2}c(e) - t \leq c(\hat{e}) - t$, so that the constraint holds at the end of the iteration. $\square$

Corollary 1. *For any nongood 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, $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) \geq c(e)$.*

Proof. The first statement follows directly from the proof above. Now consider good edge $e = (u,v)$ added by the algorithm. Suppose $u \in C_1$ and $v \in C_2$ with $h(C_1) = h(C_2) = 1$, so that at the start of the iteration $\sum_{S:(u,v)\in\delta^+(S)} y(S) = \sum_{S:(v,u)\in\delta^+(S)} y(S) = t$ by Lemma 1. If $t \leq \frac{1}{2}c(e)$, both sums increase by at least $\frac{1}{2}c(e) - t$, and $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) = c(e)$. However, if $t > \frac{1}{2}c(e)$, neither sum increases and $\sum_{S:(u,v)\in\delta^+(S)} y(S) + \sum_{S:(v,u)\in\delta^+(S)} y(S) > c(e)$. $\square$

We now need to 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. 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 $\vec{C}$.

We can now prove the needed lemma.

Lemma 3. *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 nongood 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 $\vec{C}$ 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 $S \subseteq C$, $|\delta^+(S) \cap \vec{C}| \leq 1$; $|\delta^+(S) \cap \vec{C}| = 0$ 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

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

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

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

□

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 4. *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 $\mathcal{C}^*$ be its connected components. Then

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

where the last inequality follows by Lemma 4. Let F be the solution returned by the algorithm. By Lemma 3, 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 $C^* \in \mathcal{C}^*$ with $C^* \in \delta(S)$ in order for the solution to be feasible. Thus

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

□

We now turn to proving Lemma 4. 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 4 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 4, we prove a simple special case of the lemma by orienting the arcs of the component and using the dual feasibility (1).

Lemma 5. *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 $\vec{C^*}$ 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 $\vec{C^*} \in \delta^+(S)$ if there is some arc of $\vec{C^*}$ with a tail in S and head not in S. Then by the previous discussion $\sum_{S:C^*\in\delta(S)} y(S) = \sum_{S:\vec{C^*}\in\delta^+(S)} y(S)$.

Then by (1),

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

□

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 $\vec{C^*} \in \delta^+(S)$, 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 $\vec{C^*}$ be the orientation of all the edges of C^* except e^* towards the two roots u^*, v^*; e^* remains undirected. Let

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

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

Proof. Since $\vec{C^*}$ is birooted, there is some component $X \subseteq C^*$ that contains u^* and v^* with $\max(X) \leq c(e^*)$ and $h(X) = 0$. At time $c(e^*)$, u^* and v^* must be in inactive components (possibly the same component); 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^*, and thus the inequality of the lemma must be true. $\square$

We will make extensive use of the following lemma, which tells us that the dual growing procedure 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 7. *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 2, 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 $\max(C(u) \cup C(v)) \geq t'$. $\square$

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 8. *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 7, 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 $C_{u^*}^t$ and $C_{v^*}^t$ are active components for $t < t(u^*)$, if e^* was not good, it must have been the case that $h(C^t_{u^*} \cup C^t_{v^*}) = 1$. Thus we can apply Lemma 7: 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 $t(u^*)$. $\square$

Finally, we can prove Lemma 4.

Proof of Lemma 4: If there is $v \in C^*$ with $h(\{v\}) = 0$, then the statement follows from Lemma 5. If not, then we can get a directed, birooted version $\vec{C^*}$ of C^* with undirected edge $e^* = (u^*, v^*)$. 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

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

where the penultimate inequality follows by our assumption and by Lemma 6. If $t(u^*) + t(e^*) > \frac{1}{2}c(e^*)$, then by Lemma 8, 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 $\vec{C^*}$. Since $c(a') \geq t(u^*) + t(e^*)$ and by Lemma 6 $c(e^*) \geq t(v^*) + t(e^*) \geq t(u^*) + t(e^*)$, then $t(u^*) + t(e^*) \leq \frac{1}{2}(c(a') + c(e^*))$. Then following the inequalities above, we have

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

and we are done. $\square$

4 Conclusion

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 [4]. A function $f : 2^V \rightarrow \{0, 1\}$ 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 Steiner tree and generalized Steiner tree; 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; whereas the algorithm of [4] 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 [4] in this case does have the property that once a component becomes inactive, no further dual is grown around it.

Acknowledgements

We thank the anonymous reviewers for useful comments.

References

1. Bazgan, C., Couëtoux, B., Tuza, Z.: Complexity and approximation of the Constrained Forest problem. Theoretical Computer Science 412, 4081–4091 (2011)
2. Couëtoux, B.: A $\frac{3}{2}$ approximation for a constrained forest problem. In: Demetrescu, C., Halldórsson, M.M. (eds.) Algorithms – ESA 2011, 19th Annual European Symposium, pp. 652–663. No. 6942 in Lecture Notes in Computer Science, Springer (2011)
3. Goemans, M.X., Williamson, D.P.: Approximating minimum-cost graph problems with spanning tree edges. Operations Research Letters 16, 183–189 (1994)
4. Goemans, M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. SIAM Journal on Computing 24, 296–317 (1995)
5. Goemans, M.X., Williamson, D.P.: The primal-dual method for approximation algorithms and its application to network design problems. In: Hochbaum, D.S. (ed.) Approximation Algorithms for NP-hard Problems, chap. 4. PWS Publishing, Boston, MA, USA (1996)
6. Imielińska, C., Kalantari, B., Khachiyan, L.: A greedy heuristic for a minimum-weight forest problem. Operations Research Letters 14, 65–71 (1993)
7. 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)
8. Laszlo, M., Mukherjee, S.: Another greedy heuristic for the constrained forest problem. Operations Research Letters 33, 629–633 (2005)
9. Laszlo, M., Mukherjee, S.: A class of heuristics for the constrained forest problem. Discrete Applied Mathematics 154, 6–14 (2006)
10. Laszlo, M., Mukherjee, S.: An approximation algorithm for network design problems with downwards-monotone demand functions. Optimization Letters 2, 171–175 (2008)