ELSEVIER

Theoretical Computer Science 221 (1999) 139–155

Theoretical Computer Science

www.elsevier.com/locate/tcs

Improving spanning trees by upgrading nodes

Sven O. Krumke[a,*], Hartmut Noltemeier[b], Hans-C. Wirth[b], Madhav V. Marathe[c], R. Ravi[d], S.S. Ravi[e,1], R. Sundaram[f]

[a] *Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Takustr. 7, 14195 Berlin-Dahlem, Germany*

[b] *Department of Computer Science, University of Würzburg, Am Hubland, 97074 Würzburg, Germany*

[c] *Los Alamos National Laboratory, P.O. Box 1663, MS B265, Los Alamos, NM 87545, USA*

[d] *GSIA, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

[e] *Department of Computer Science, University at Albany – SUNY, Albany, NY 12222, USA*

[f] *Delta Trading Co. Work done while at MIT, Cambridge MA 02139, USA*

Abstract

We study *bottleneck constrained network upgrading problems*. We are given an edge weighted graph $G=(V,E)$ where node $v \in V$ can be upgraded at a cost of $c(v)$. This upgrade reduces the delay of each link emanating from v. The goal is to find a minimum cost set of nodes to be upgraded so that the resulting network has good performance. The performance is measured by the bottleneck weight of a minimum spanning tree.

We give a polynomial time approximation algorithm with logarithmic performance guarantee, which is tight within a small constant factor as shown by our hardness results. © 1999 Published by Elsevier Science B.V. All rights reserved.

Keywords: NP-hardness; Approximation algorithms; Network design; Spanning tree

1. Introduction

Several problems arising in areas such as communication networks can be expressed in the following general form: Enhance the performance of an underlying network by carrying out upgrades at certain nodes or edges of the network [2, 13, 14, 9].

In communication networks, *upgrading a node* corresponds to installing faster communication equipment at that node. Such an upgrade reduces the communication

* Corresponding author.

E-mail addresses: krumke@informatik.uni-wuerzburg.d400.de (S.O. Krumke), noltemei@informatik.uni-wuerzburg.de. (H. Noltemeier), wirth@informatik.uni-wuerzburg.de. (H.C. Wirth), madhav@c3.lanl.gov. (M.V. Marathe), ravi+@cmu.edu. (R. Ravi), ravi@cs.albany.edu (S.S. Ravi), koods@theory.lcs.mit.edu. (R. Sundaram)

[1] Supported by NSF Grant CCR-9734936.

PII: S0304-3975(99)00030-4

delay along each edge emanating from the node. Similarly, *upgrading an edge* can be achieved by replacing the old line with a new optical cable. In general, there is a cost for improving each node or edge in the existing network by a unit amount. The goal is to design a strategy to upgrade the network such that it has a good performance while the upgrading cost is minimized.

2. Preliminaries and problem definition

The *node based upgrading model* discussed in this paper can be formally described as follows. Let $G=(V,E)$ be a connected undirected graph with $n:=|V|$ vertices and $m:=|E|$ edges. For each edge $e\in E$, we are given three integers $d_0(e)\geqslant d_1(e)\geqslant d_2(e)\geqslant 0$. The value $d_i(e)$ represents the *length* or *delay* of the edge e if exactly i of its endpoints are upgraded. Thus, the upgrade of a node v reduces the delay of each edge incident with v.

For each node $v\in V$ the value $c(v)$ specifies how expensive it is to upgrade the node. For a subset W of V, the cost of upgrading all the nodes in W, denoted by $c(W)$, is equal to $\sum_{v\in W} c(v)$. The edge weight function resulting from an upgrade of the node set W is denoted by d_W, that is

$$d_W(e)=\begin{cases} d_0(e) & \text{if none of the endpoints of } e \text{ belongs to } W,\\ d_1(e) & \text{if exactly one endpoint of } e \text{ belongs to } W,\\ d_2(e) & \text{if both endpoints of } e \text{ belong to } W.\end{cases}$$

We call the maximum delay of an edge in a subgraph the *bottleneck delay* of that subgraph. The *bottleneck graph* Bottleneck(G,d_W,D) contains all edges $e\in E$ with $d_W(e)\leqslant D$.

Given a bound D on the bottleneck delay of a subgraph, we partition the set of edges into four sets according to how many of the endpoints must be upgraded in order to decrease the delay of an edge below the threshold D. An edge of delay $d_0(e)\leqslant D$ is called an *uncritical* edge. An edge e is said to be *1-critical*, if $d_0(e)>D\geqslant d_1(e)$, and *2-critical*, if $d_1(e)>D\geqslant d_2(e)$. Finally, if $d_2(e)>D$, the edge e is called *useless*. Without loss of generality we can assume that the graph does not contain any useless edges.

We are now ready to formulate the problems studied in this paper.

Definition 1 (*Bottleneck tree upgrading problem*). Let $G=(V,E)$ be an edge and node weighted graph as above. Given a bound D, the *bottleneck spanning tree upgrading problem* (Node Upgrading Cost, Bottleneck, Spanning Tree), is to upgrade a set $W\subseteq V$ of nodes such that the resulting graph has a spanning tree of bottleneck delay at most D and $c(W)$ is minimized.

The long problem name (Node Upgrading Cost, Bottleneck, Spanning Tree) might read confusing. However, the above problem is an example for a *bicriteria problem*.

The notation and the framework developed for this class of problems are described in the following two subsections.

2.1. Bicriteria problems and approximations

A general bicriteria network upgrade problem $(f_1, f_2, \mathcal{S})$ is defined by two minimization objectives f_1 and f_2 and a class $\mathcal{S}$ of subgraphs. The problem specifies a budget value D on the objective f_2. A solution x is *valid*, if it belongs to the graph class $\mathcal{S}$ and satisfies the constraint $f_2(x) \leqslant D$ on the objective f_2. The goal is to find a f_1-minimal solution amongst all valid solutions.

Since the problems which arise are NP-hard in general, it is meaningful to search for approximate solutions which can be computed in polynomial time.

Definition 2 (*Performance of approximation*). Let $P = (f_1, f_2, \mathcal{S})$ be a bicriteria problem. A polynomial time algorithm has *performance* (α, β) for P, if for all instances the algorithm produces a solution $x \in \mathcal{S}$ such that $f_2(x) \leqslant \beta \cdot D$ and $f_1(x) \leqslant \alpha \cdot f_1(x^*)$, where x^* denotes an optimal (valid) solution and D is the given bound on objective f_2 in the instance.

2.2. Dual problems

The problem in Definition 1 is formulated by specifying a bound on the bottleneck delay after the upgrade, while the upgrading cost is to be minimized. It is also meaningful to consider the corresponding dual problem (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE), where we are given a bound on the upgrading cost and want to obtain the best-possible bottleneck delay while staying within our budget restrictions.

The following lemma shows that if we have a good approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE), we can convert it into a good approximation algorithm for the dual problem (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) with only a minor additional computational effort.

We will use this result and formulate our approximation algorithms only for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE), which will be more convenient.

Lemma 3. *Suppose that* A *is a bicriteria approximation algorithm for* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *with a performance of* (α, β). *Then, one can construct an approximation algorithm for* (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) *with performance of* (β, α) *by using* $\mathcal{O}(\log m) \subseteq \mathcal{O}(\log n)$ *calls to* A, *plus an overhead of* $\mathcal{O}(m \log m)$ *elementary operations.*

Proof. Let A be an (α, β)-approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE). We will show how to use A to construct a (β, α)-approximation algorithm for the dual problem.

An instance of (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) is specified by a graph $G = (V, E)$, the node cost function c, the weight functions d_0, d_1, and d_2

on the edges and the bound B on the node upgrading cost. We denote by OPT the optimum bottleneck weight of an MST after upgrading a node set of cost at most B. Observe that OPT is an integer such that $D_2 \leqslant \mathrm{OPT} \leqslant D_0$ where $D_2 := \min_{e\in E} d_2(e)$ and $D_0 := \max_{e\in E} d_0(e)$. Moreover, the set

$$M := \{d_0(e), d_1(e), d_2(e) \colon e \in E\}$$

of possible values for OPT has size $\mathcal{O}(m)$.

We sort M in time $\mathcal{O}(m \log m)$. Then we use a binary search to find the minimum integer $D \in M$ with the following property: Algorithm A outputs an upgrading set of cost at most αB, if it is applied to the instance of (Node Upgrading Cost, Bottleneck, Spanning Tree) given by the weighted graph G as above and the bound D on the bottleneck weight of an MST after the upgrade. It is easy to see that this binary search indeed works, uses $\mathcal{O}(\log |M|) \subset \mathcal{O}(\log m)$ calls to algorithm A, and terminates with a value $D \leqslant \mathrm{OPT}$. The corresponding upgrading set W leads to an MST in (G, d_W) with bottleneck weight at most $\beta \cdot D \leqslant \beta \cdot \mathrm{OPT}$ and upgrading cost $c(W) \leqslant \alpha \cdot B$. □

By similar techniques, an approximation algorithm for (Bottleneck, Node Upgrading Cost, Spanning Tree) can be converted into an approximation algorithm for (Node Upgrading Cost, Bottleneck, Spanning Tree). In this case we use the bicriteria algorithm to search for the optimal upgrading cost. This is stated in the following lemma.

Lemma 4. *Suppose that* A *is a bicriteria approximation algorithm for* (Bottleneck, Node Upgrading Cost, Spanning Tree) *with a performance of* (α, β). *Then, there is a* (β, α)*-approximation algorithm for* (Node Upgrading Cost, Bottleneck, Spanning Tree).

It should be noted that the conversion of an algorithm for (Node Upgrading Cost, Bottleneck, Spanning Tree) has the nice property that the running time increases only by a factor of $\mathcal{O}(\log n)$, while the other way round we get a factor of $\mathcal{O}(\log C)$, where $C = \sum_{v\in V} c(v)$.

3. Related work

Some node upgrading problems have been investigated under a simpler model by Paik and Sahni [14]. In their model, the delay of an edge is decreased by constant factors of δ or δ^2, when one or two of its endpoints are upgraded, respectively. Clearly, this model is a special case of the model treated in our paper.

Under their model, Paik and Sahni studied the upgrading problem for several performance measures including the maximum delay on an edge and the diameter of the network. They presented NP-hardness results for several problems. Their focus was on the development of polynomial time algorithms for special classes of networks (e.g. trees, series–parallel graphs) rather than on the development of approximation

algorithms. Our constructions can be modified to show that all the problems considered here remain NP-hard even under the Paik–Sahni model.

A special case of the problems studied in this paper is the case of all nodes having the same upgrading costs. For spanning trees, these problems, namely (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) and its dual version (BOTTLENECK, UPGRADING SIZE, SPANNING TREE), are investigated in [8]. The authors give a $(5+4\ln\Delta, 1)$-approximation algorithm for (UPGRADING SIZE, BOTTLENECK, SPANNING TREE), where Δ is the maximum degree of the graph. The algorithm can be implemented to run in time $\mathcal{O}(n+m)$. The analysis showed a better performance guarantee of $(2+2\ln\Delta, 1)$ for the case that the input does not contain any 2-critical edges.

A related problem is (UPGRADING COST, BOTTLENECK, GRAPH) which has been introduced called LINKDELAY in [14]. Paik and Sahni showed that this problem is NP-hard. A $(2,1)$-approximation algorithm for LINKDELAY has been provided in [8].

Edge-based network upgrading problems have also been considered in the literature [13, 2, 10, 7]. There, each edge has a current weight and a minimum weight below which the edge weight cannot be decreased. Upgrading an edge corresponds to decreasing the weight of that particular edge and there is a cost associated with such an upgrade. The goal is to obtain an upgraded network with the best performance.

4. An algorithm for bottleneck upgrading

In this section, we present our approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE). This algorithm provides a performance guarantee of $(2\ln n, 1)$ on a graph $G=(V,E)$ with $n:=|V|$ nodes. In Section 6 we will counterbalance this approximation result with a hardness result which shows that, unless $\mathsf{NP} \subseteq \mathsf{DTIME}(N^{\mathcal{O}(\log\log N)})$, this performance is essentially the best possible.

4.1. Overview

We first give a brief overview of our algorithm. The algorithm maintains a set W of nodes, a set F of edges and a set $\mathscr{C}$ of clusters which partition the node set V of the given graph G. The set $\mathscr{C}$ of clusters is initialized to be the set of connected components of the bottleneck graph Bottleneck(G, d_W, D),

Algorithm 1. Approximation algorithm for spanning trees

Input: A graph $G=(V,E)$, three edge weight functions d_0, d_1, d_2, a node weight function c, and a number D

1 $W \leftarrow \emptyset$
2 $G' \leftarrow$ Bottleneck(G, d_W, D)
3 $C_1, \ldots, C_q \leftarrow$ connected components of G'
4 $F \leftarrow$ set of edges of G'
5 **while** $G'=(V,F)$ has more than one connected component **do**
6 {Assume that $\mathscr{C}=\{C_1 \ldots, C_p\}$ is the set of components}

```
7   Find a node v ∈ V in the graph with minimum quotient cost as defined
    in Definition 5.
8   Let C_1,...,C_r be the components in 𝒞 chosen in Step 7 above, where
    w.l.o.g. v ∈ C_1.
9   Let e_2,...,e_r be a set of edges in G connecting v to C_2,...,C_r, respectively
10  F ← F ∪ {e_2,...,e_r} {Merge C_1,C_2,...,C_r into one component}
11  W ← W ∪ {v} {Add center (see Definition 7) to upgrading set}
12  for i ← 2,...,r do
13    if e_i = (v,v_i) is 2-critical then
14      W ← W ∪ {v_i} {Add finger (see Definition 7) to upgrading set}
15    end if
16  end for
      {Note that the total cost of the nodes added to the solution W is
    exactly c(v) + Σ_{j=1}^{r} c(v,C_j).}
17  C_1,...,C_{p'} ← connected components of G' = (V,F)
18 end while
19 return W
```

containing only those edges e which have a delay $d_0(e)$ of at most D. The set W contains the upgraded nodes and is initially empty.

The algorithm iteratively merges clusters until only one cluster remains. To this end, in each iteration it determines a node v of minimum *quotient cost*. The quotient cost of a node v is the ratio whose numerator is the cost of v plus the costs of some nodes adjacent to v in different clusters via 2-critical edges, and whose denominator is the number of clusters which have nodes adjacent to v. A precise definition of the quotient cost appears in Eq. (1) below. This quotient cost measures the "average upgrading cost" of v and the vertices that are adjacent to v through 2-critical edges. The algorithm then adds v and the nodes mentioned above to the solution set W and merges the corresponding clusters.

The algorithm is shown in Algorithm 1. It is easy to see that the set W output by the algorithm is indeed a valid upgrading set, since all the edges added to F in Step 10 will be of delay at most D after upgrading the nodes in W.

Definition 5 (*Quotient cost*). Let $\mathscr{C} = \{C_1, \ldots, C_p\}$ be the connected components of (V,F) at some iteration of the algorithm.

If $v \in C_j$ or v is adjacent to a node in C_j via a 1-critical edge, then we set $c(v, C_j) := 0$. If all the edges from v to C_j are 2-critical, then we set $c(v, C_j)$ to be the minimum cost of a node in C_j adjacent to v. If there is no edge between v and any node in C_j, then $c(v, C_j) := +\infty$.

We now define the *quotient cost* $q(v)$ of v as follows:

$$q(v) := \min_{2 \leqslant r \leqslant p} \min_{\{C_1,\ldots,C_r\} \subseteq \mathscr{C}} \frac{c(v) + \sum_{j=1}^{r} c(v, C_j)}{r}. \tag{1}$$

Notice that the quotient cost of a node can be computed in polynomial time: We can order the components in $\mathscr{C}$ as $C_1, C_2, C_3, \ldots$ in nondecreasing order of $c(v, C_j)$ (where, without loss of generality, $v \in C_1$). In computing the quotient cost of v, it is sufficient to consider the p subsets of $\mathscr{C}$ of the form $\{C_1, C_2, \ldots, C_r\}$, where $2 \leqslant r \leqslant p$.

In the sequel, we use W^* to denote an optimal upgrading set, i.e., an upgrading set of minimal cost $\mathrm{OPT} := c(W^*)$. We now proceed to prove the following theorem which indicates the performance guarantee provided by the algorithm.

Theorem 6. *Algorithm* 1 *as applied to* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *has a performance of* $(2 \ln n, 1)$, *where n denotes the number of nodes in the graph.*

Our proof of Theorem 6 relies mainly on an averaging lemma which is proved by using the notion of a *claw decomposition* introduced below.

4.2. Claw decompositions

Definition 7 (*Claw, claw decomposition*). A graph $G = (V, E)$ is called a *claw*, if there is a node $c \in V$ such that the edge set E is of the form $E = \{(c, v) : v \in V \setminus \{c\}\}$. The node c is called a *center* of the claw, the remaining nodes are called *fingers*. The center is uniquely determined if there are at least 3 nodes in the claw. A claw consisting of one single node is called a *trivial* claw.

Let G be a graph with node set V. A *claw decomposition* of V in G is a collection of node-disjoint nontrivial claws, which are all subgraphs of G and whose vertices form a partition of V.

The following theorem can be proven by an easy induction on $n := |V|$:

Theorem 8. *Let G be a connected graph with node set V, where* $|V| \geqslant 2$. *Then there is a claw decomposition of V in G.*

4.3. An averaging lemma

Lemma 9. *Let v be a node chosen in Step* 7 *of Algorithm* 1 *and let C denote the total cost of the nodes added to the solution set W in this iteration. Let there be p clusters before v is chosen and assume that in this iteration r clusters are merged. Then*

$$\frac{C}{r} \leqslant \frac{\mathrm{OPT}}{p}.$$

Proof. Let T^* be an optimal tree with the nodes W^* be the upgraded nodes. Let $\mathrm{OPT} := c(W^*)$ be the cost of the optimal solution. Let $\mathscr{C} = C_1, \ldots, C_p$ be the clusters when the node v was chosen and let $T^*(v)$ be the graph obtained from T^* by contracting each C_j to a supernode. $T^*(v)$ is connected and contains all supernodes. We then remove edges (if necessary) from $T^*(v)$ so as to make it a spanning tree. Note that all the edges in this tree are critical.

Let $A \subseteq W^*$ be the set of nodes in the optimal solution that are adjacent to another cluster in $T^*(v)$. Clearly, the cost of these nodes is no more than OPT. Take a claw decomposition of $T^*(v)$. We now obtain a set of claws in the graph G itself in the following way: Initialize E' to be the empty set. For each claw in the decomposition with center C_1' and fingers $C_2', \ldots, C_l'$ we do the following: For each edge (C_1', C_j') the optimal tree T^* must have contained an edge (u, w) with $u \in C_1'$ and $w \in C_i'$. Notice that since this edge was critical, at least one of the vertices u and w must belong to $A \subseteq W^*$. We add (u, w) to E'.

It is easy to see that the subgraph of G induced by the edges in E' consists of disjoint nontrivial claws. Also, all edges in the claws were critical and the total number of nodes in the claws is at least p. We need one more useful observation: If a claw center is not contained in A, then *all* the fingers of the claw must be contained in A, since the edges in the claw were critical.

Let A_c be the set of nodes from A acting as centers in the just generated claws. Let A_1 denote the fingers of the claws contained in A which are connected to their claw center via a 1-critical edge, whereas A_2 stands for the set of fingers adjacent to the center via a 2-critical edge and also contained in A. For each claw with exactly two nodes we designate an arbitrary one of the nodes to be the center. Then by construction, A_c, A_1, and A_2 are disjoint. Therefore,

$$\mathrm{OPT} \geqslant \sum_{u \in A_c \cup A_2} c(u) + \sum_{u \in A_1} c(u). \tag{2}$$

For a node $u \in A_c$, let N_u denote the number of vertices in the claw centered at u. We have seen that if a center is not in A, then *all* the fingers belong to the optimal solution. Clearly, this can only happen, if the claw centered at u does not contain a 2-critical edge. Thus, we can estimate the total number of nodes in the claws from above by summing up the cardinalities of the claws with centers in A and for all other claws adding twice the number of fingers. Hence,

$$\sum_{u \in A_c} N_u + 2|A_1| \geqslant |\{w\colon w \text{ belongs to some claw}\}| \geqslant p, \tag{3}$$

since the total number of nodes in the claws is at least p.

We now estimate the first sum in (2). If $u \in A_c$, then the quotient cost of u is at most the cost of u plus the cost of the fingers in the claw that are in A_2 divided by the total number of nodes in the claw. This in turn is at least C/r by the choice of the algorithm in Step 7. By summing up over all those centers, this leads to

$$\sum_{u \in A_c \cup A_2} c(u) \geqslant \frac{C}{r} \sum_{u \in A_c} N_u. \tag{4}$$

Now, for a node u in A_1, its quotient cost is at most $c(u)/2$, which again is at least C/r. Thus,

$$\sum_{u \in A_1} c(u) \geqslant \sum_{u \in A_1} 2\frac{C}{r} = 2|A_1|\frac{C}{r}. \tag{5}$$

Using (4) and (5) in (2) yields

$$\begin{aligned}\mathrm{OPT} &\geqslant \sum_{u\in A_c\cup A_2} c(u) + \sum_{u\in A_1} c(u)\\ &\geqslant \frac{C}{r}\left(\sum_{u\in A_c} N_u + 2|A_1|\right)\\ &\overset{(3)}{\geqslant} \frac{C}{r}p.\end{aligned}$$

This proves the claim. □

4.4. A potential function argument

We are now ready to complete the proof of the performance stated in Theorem 6. Assume that the algorithm uses f iterations of the loop and denote by $v_1,\ldots,v_f$ the vertices chosen in Step 7 of the algorithm.

Let ϕ_j denote the number of clusters *after* choosing node v_j in this iteration. Thus, for instance, $\phi_0 = t$, the number of components at the beginning of the whole algorithm and $\phi_f = 1$, since we end up with one cluster. Let the number of clusters merged using node v_j be r_j and the total cost of the vertices added in that iteration be c_j. Then we have

$$\phi_j = \phi_{j-1} - (r_j - 1). \tag{6}$$

Notice that, since $r_j \geqslant 2$, we have $r_j - 1 \geqslant \frac{1}{2}r_j$. Using this inequality in (6) we obtain

$$\phi_j \leqslant \phi_{j-1} - \tfrac{1}{2}r_j. \tag{7}$$

Observe that $\phi_j \geqslant 2$ for $j = 0,\ldots,f-1$, since the algorithm does not stop before the fth iteration. Notice also that $\phi_f = 1$. Then by Lemma 9, we have

$$r_j \geqslant \frac{c_j\phi_{j-1}}{\mathrm{OPT}} \tag{8}$$

for all $0 \leqslant j \leqslant f$. We now use an analysis technique due to Leighton and Rao [12] to complete the proof. Substituting Eq. (8) into (7) yields

$$\phi_j \leqslant \phi_{j-1} - \frac{1}{2}\frac{c_j\phi_{j-1}}{\mathrm{OPT}} = \phi_{j-1}\left(1 - \frac{c_j}{2\,\mathrm{OPT}}\right). \tag{9}$$

Using recurrence (9), we obtain

$$\phi_f \leqslant \phi_0 \prod_{j=1}^{f}\left(1 - \frac{c_j}{2\,\mathrm{OPT}}\right). \tag{10}$$

Taking natural logarithms on both sides and simplifying using the estimate $\ln(1-\tau) \leqslant -\tau$, we obtain

$$2\,\mathrm{OPT}\ln\left(\frac{\phi_0}{\phi_f}\right) \geqslant \sum_{j=1}^{f} c_j. \tag{11}$$

Notice that by Lemma 9 we have

$$c_j \leqslant \mathrm{OPT}\frac{r_j}{\phi_{j-1}} \leqslant \mathrm{OPT} < 2\,\mathrm{OPT},$$

and so the logarithms of all the terms in the product of (10) are well defined.

Note also that $\phi_0 \leqslant n := |V|$ and $\phi_f = 1$ and hence from (11) we get

$$\sum_{j=1}^{f} c_j \leqslant 2\,\mathrm{OPT}\ln n. \tag{12}$$

Notice that the total cost of the nodes chosen by the algorithm is exactly the sum $\sum_{j=1}^{f} c_j$. This completes the proof of Theorem 6. □

4.5. Running time

We now sketch an efficient implementation of Algorithm 1. The results are summarized in the following theorem:

Theorem 10. *Algorithm* 1 *can be implemented to run in time* $\mathcal{O}(nm\,\alpha(m,n))$, *where* n *denotes the number of nodes,* m *the number of edges in the graph, and* α *is the inverse of Ackerman's function.*

Proof. The main effort lies in the computation of the minimum quotient cost in Step 7. Suppose we have for each node $v \in V$ a sorted list $L(v) = (C_1, C_2, \ldots, C_p)$ of clusters such that $c(v, C_1) \leqslant c(v, C_2) \leqslant \cdots \leqslant c(v, C_p)$. Then, the cost of the set $\{C_1, \ldots, C_r\}$ is minimal amongst all r-element collections of clusters, so we do not have to test all possible r-element sets of clusters.

Since, for fixed v, the number p of clusters is bounded by the number of adjacent nodes, Step 7 can be implemented to run in time $\mathcal{O}(m)$.

To maintain the sorted lists we use the help of a fast *disjoint-set data structure* [4]. We initialize the data structure with the clusters formed by uncritical edges. The costs of the clusters are computed as stated in Step 7. After each step, we assure that the data structure again represents the clusters which are formed by edges whose weight does not exceed the threshold. This is done by merging those clusters which are connected by edges involved in the current upgrading, i.e. those edges which are incident with nodes upgraded in the current step. Such merging of clusters is efficiently supported by the data structure. The time needed in one iteration of the while loop is $\mathcal{O}(m\alpha(m,n))$. For details we refer to [11].

Since in each iteration the number of clusters is decreased by at least 1, there are at most n iterations. This results in a total running time of $\mathcal{O}(nm\,\alpha(m,n))$. □

Using Lemma 3 we obtain the following approximation result for the dual problem.

Theorem 11. *There exists an approximation algorithm for* (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) *with performance* $(1, 2\ln n)$. *It can be implemented to run in time* $\mathcal{O}(nm\,\alpha(m,n)\log n)$.

5. Treewidth-bounded graphs

In this section we will show that (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) can be solved in polynomial time if restricted to the class of treewidth-bounded graphs. For the sake of a better presentation we will first show how to solve the problem in polynomial time on series–parallel graphs. Then, we will describe how the ideas carry over to treewidth-bounded graphs.

Treewidth-bounded graphs were introduced by Robertson and Seymour [15]. Independently, Bern et al. [3] introduced the notion of *decomposable graphs*. Later, it was shown [1] that the class of decomposable graphs and the class of treewidth-bounded graphs coincide. A class of *decomposable graphs* Γ is given by a set of recursive rules that satisfy the following conditions [3]:

(1) The rules define a finite number of primitive graphs.
(2) Each graph in Γ has an ordered (possibly empty) set of special nodes called *terminals*. The number of terminals in each graph is bounded by a global constant.
(3) There is a finite collection of binary composition rules that operate only at terminals, either by identifying two terminals or adding an edge (called *attachment edge*) between terminals. A composition rule also determines the terminals of the resulting graph, which must be a subset of the terminals of the two graphs being composed.

Series–parallel graphs are an example of decomposable graphs and can be defined by the following rules [3].

(1) The set of primitive graphs consists of the single graph P with node set $\{s,t\}$ and the single edge (s,t). The node s is the "start-terminal" of P and the node t is the "end-terminal" of P.
(2) Let $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ be series–parallel graphs with terminals s_1, t_1 and s_2, t_2, respectively. Then
   (a) The graph obtained by identifying t_1 and s_2 is a series–parallel graph, with s_1 and t_2 as its terminals. This graph is the *series composition* of G_1 and G_2.
   (b) The graph obtained by identifying s_1 and s_2 and also t_1 and t_2 is a series–parallel graph, the *parallel composition* of G_1 and G_2. This graph has $s_1(=s_2)$ and $t_1(=t_2)$ as its terminals.

Let Γ be any class of decomposable graphs. Following [3], we assume that a given graph $G\in\Gamma$ is accompanied by a parse tree specifying how G is constructed using the rules. The size of the parse tree is linear in the size of G. Moreover, we may assume without loss of generality that the parse tree is a binary tree.

5.1. Restriction to series–parallel graphs

Let G be a series–parallel graph with the two terminals s and t. We call an edge subgraph G' of G consisting of two disjoint spanning trees containing s and t, respectively, a *terminal forest*.

For a set $M \subseteq \{s,t\}$, define $C(M)$ to be the least cost of an upgrading set W in G with $W \cap \{s,t\} = M$ such that after upgrading this set G contains a bottleneck spanning tree of delay at most D. If there is no upgrading set W such that the bottleneck delay can be reduced to be at most D and $W \cap \{s,t\} = M$, then $C(M) := +\infty$. In the same way as we defined C, we define C' for the minimum upgrading cost to obtain a terminal forest of bottleneck delay at most D.

Clearly, if we know the four values $C(M)$, we can tell the optimum objective function value. We will now show that for a series–parallel graph G we can compute C and C' by using the information of the *decomposition tree* of G in a total of $\mathcal{O}(n+m)$ time. The basic idea is to keep track of which terminals belong to an optimal upgrading set. In the sequel we write $M \backslash v$ and $M \cup v$ instead of $M \backslash \{v\}$ and $M \cup \{v\}$, respectively, for the sake of brevity.

First, we will take care of the case that G is the series composition of G_1 and G_2. Assume that we have already computed the values C and C' for G_1 and G_2. Denote them by C_1, C_1' and C_2, C_2', respectively.

It is easy to see that the restriction of any tree T to G_1 and G_2, respectively, is again a tree. Thus, we can compute C with the help of C_1 and C_2 in the following way.

$$C(M) = \min\{C_1(M \backslash t) + C_2(M \backslash s), C_1(M \cup t) + C_2(M \cup s) - c(t_1)\}.$$

The first term above considers the case when the terminal $t_1 (= s_2)$ is not upgraded. The second term takes care of t_1 being upgraded.

Similarly, we now compute C' for G. A terminal forest in G must either be a terminal forest in G_1 and a tree in G_2 or vice versa. No other possibilities exist. It now follows that C can be computed by

$$\begin{aligned} C'(M) = \min\{&C_1'(M \backslash t) + C_2(M \backslash s), C_1'(M \cup t) + C_2(M \cup s) - c(t_1), \\ &C_1(M \backslash t\}) + C_2'(M \backslash s), C_1(M \cup t) + C_2'(M \cup s) - c(t_1)\}. \end{aligned}$$

We now consider the case that G is parallely composed from G_1 and G_2. Again, we assume that the two arrays C and C' are already available for G_1 and G_2.

We start with the computation of C. A tree T in G must be a tree in exactly one of the graphs G_1 and G_2 and a terminal forest in the second one. We just need to distinguish between the cases covering the upgrade of the terminals of G_1 and G_2. We must make sure that s_1 is upgraded if and only if s_2 is. We thus obtain C by the following formula:

$$\begin{aligned} C(\{s,t\}) = \min\{&C_1'(\{s,t\}) + C_2(\{s,t\}) - c(s_1) - c(t_1), \\ &C_1(\{s,t\}) + C_2'(\{s \cup t\}) - c(s_1) - c(t_1)\}, \end{aligned}$$

$$C(\{t\}) = \min\{C_1'(\{t\}) + C_2(\{t\}) - c(t_1), C_1(\{t\}) + C_2'(\{t\})] - c(t_1)\},$$

$$C(\{s\}) = \min\{C_1'(\{s\}) + C_2(\{s\}) - c(s_1), C_1(\{s\}) + C_2'(\{s\}) - c(s_1)\},$$

$$C(\emptyset) = \min\{C_1'(\emptyset) + C_2(\emptyset), C_1(\emptyset) + C_2'(\emptyset)\}.$$

We proceed with C'. If G' is a terminal forest of G, it is straightforward to see that the restriction to *both* graphs G_1 and G_2 is a terminal forest of that particular graph. Thus, C' can be computed by using the information from C_1' and C_2' by the following formula:

$$C(\{s,t\}) = C_1'(\{s,t\}) + C_2'\{s,t\} - c(s_1) - c(t_1),$$

$$C(\{t\}) = C_1'(\{t\}) + C_2'(\{t\}) - c(t_1),$$

$$C(\{s\}) = C_1'(\{s\}) + C_2'(\{s\}) - c(s_1),$$

$$C(\emptyset) = C_1'(\emptyset) + C_2'(\emptyset).$$

Finally, observe that for a series–parallel graph consisting of the two terminals s and t and the edge (s,t) we can trivially compute the arrays C and C'.

Using the above recurrences, the array C can be computed in linear time for a series–parallel graph G, provided a decomposition tree for G is given. Since such a decomposition tree with $\mathcal{O}(n+m)$ nodes can be computed in $\mathcal{O}(n+m)$ time [16], we can conclude that the dynamic programming algorithm presented above runs in total time $\mathcal{O}(n+m)$. It should be noted that by also keeping track of the respective upgrading sets we cannot only find the optimal function value but also the optimal upgrading set.

Theorem 12. *If restricted to the class of series–parallel graphs, the problem* (Node Upgrading Cost, Bottleneck, Spanning Tree) *can be solved optimally in* $\mathcal{O}(n+m)$*-time.*

5.2. Extension to treewidth-bounded graphs

Theorem 13. *If restricted to any class of treewidth bounded graphs with no more than k terminals, where k is fixed, the problem* (Node Upgrading Cost, Bottleneck, Spanning Tree) *can be solved optimally in time* $\mathcal{O}((2k)^{2k}(n+m))$.

Proof. Let $t_1,\ldots,t_k$ be the terminals of G and let π be a partition of these terminals. Define a π-terminal forest F to be a spanning forest of G with the following properties:

(1) For each block of π the forest F contains a tree spanning all the vertices in that block.
(2) No pair of trees is connected.

The notion of a π-terminal forest generalizes the concept of spanning trees and terminal forests introduced above. In the case of series–parallel graphs, the set of terminals is $\{s,t\}$. The possible partitions of $\{s,t\}$ are $\pi_1 = (\{s,t\},\emptyset)$ and $\pi_2 = (\{s\},\{t\})$. Partition π_1 corresponds to a spanning tree of G, while π_2 gives us a terminal tree.

We keep the following information along with each partition π of terminals of G and each subset M of the terminals $\{t_1,\ldots,t_k\}$.

$$C^{\pi}(M) := \text{Minimum cost of a subset } W \subseteq V \text{ with } W \cap \{t_1,\ldots,t_k\} = M \text{ such that after upgrading the vertices in } W \text{ the graph } G \text{ contains a } \pi\text{-terminal forest of bottleneck cost at most } D.$$

For the above-defined cost, if there is no subset $W \subseteq V$ satisfying the required conditions the value of $C^{\pi}(M)$ is defined to be $+\infty$. Note that the number of cost values associated with any graph in Γ is $\mathcal{O}((2k)^k)$. We now show how the cost values can be computed in a bottom-up manner given the parse tree for G. Since the method is very similar to the case of series–parallel graphs treated above we only sketch the main ideas.

To begin with, since Γ is fixed, the number of primitive graphs is finite. For a primitive graph, each cost value can be computed in constant time, since the number of forests to be examined is fixed. Now consider computing the cost values for a graph G constructed from subgraphs G_1 and G_2, where the cost values for G_1 and G_2 have already been computed.

Let a partition π and a subset M of the terminals $\{t_1,\ldots,t_p\}$ of G be given. Any upgrading set W in G with $W \cap \{t_1,\ldots,t_p\} = M$ resulting in a π-terminal tree of bottleneck delay at most D induces two upgrading sets, one in G_1 and one in G_2. Since we have maintained the best cost values for all possibilities for G_1 and G_2, we can reconstruct for the partition π and the set M the cost value $C^{\pi}(M)$. We can do this in time independent of the sizes of G_1 and G_2 because they interact only at the terminals to form G, and we have maintained all relevant information.

Hence, we can generate all possible cost values for G by considering combinations of all relevant pairs of cost values for G_1 and G_2. This takes time $\mathcal{O}(1)$ per combination for a total time of $\mathcal{O}(2^{2k} \cdot k^{2k})$. As in [3], we assume that the size of the given parse tree for G is $\mathcal{O}(n+m)$. Thus the dynamic programming algorithm takes time $\mathcal{O}((2k)^{2k}(n+m))$. This completes the proof. □

The algorithm presented in the proof of the last theorem, although being linear for fixed k, is only practical for small values of k, since the constant factor $(2k)^{2k}$ in front of the $n+m$ grows extremely fast with k. Thus, the above results might be considered to be more of theoretical interest than application oriented.

6. Hardness results

In this section we establish our hardness results for the node upgrading problems under study. We show that (Node Upgrading Cost, Bottleneck, Spanning Tree) is hard to approximate within a logarithmic factor.

We first recall the results from [5] about the hardness of approximating MINIMUM DOMINATING SET and MIN SET COVER.

Theorem 14. *Unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(N^{\mathcal{O}(\log\log N)})$, *the* MINIMUM DOMINATING SET *problem on a graph with n vertices cannot be (polynomial time) approximated within a factor of* $\alpha < \ln n$.

Moreover, the MIN SET COVER *problem, with a ground set M, can not be approximated within a factor of* $\alpha < \ln|M|$.

Theorem 15. *For an instance of* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *denote by n the number of nodes in the input graph. Let* $\alpha < \frac{1}{2}\ln n$, *and f be any polynomial time computable function. Then, unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(N^{\mathcal{O}(\log\log N)})$, *there is no polynomial time approximation algorithm for* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *with performance* $(\alpha, f(n))$.

Proof. We give a reduction from MINIMUM DOMINATING SET [6, Problem GT2]. An instance of MINIMUM DOMINATING SET consists of a graph $G=(V,E)$. A *dominating set* is a subset $V' \subseteq V$ of nodes, such that each node $w \notin V'$ is adjacent to a node of V. A Dominating Set of an instance I is a solution for MINIMUM DOMINATING SET, if its cardinality is minimal amongst all Dominating Sets of I.

Given an instance $G=(V,E)$ of MINIMUM DOMINATING SET with $n := |V|$ nodes, we construct an instance $G'=(V',E')$ of (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) as follows. First, insert all nodes and edges from G into G'. Then, add a new node r (the root) and connect it to all nodes of V. The number of nodes in G' equals $n' = n+1$. Notice that $\ln n' = \ln(n+1) \leqslant \ln(n^2) = 2\ln n$.

The upgrading cost of the root is set to $c(r) := L := \lceil n \ln n \rceil + 1$, the upgrading costs for the remaining nodes are set to 1. For each edge $e' \in E'$, we set $d_0(e') := f(n') + 1$ and $d_1(e') := d_2(e') := 1$. The bound on the bottleneck weight of the resulting MST is set to 1.

If U is a Dominating Set in G, then there is a set of nodes to upgrade in G' such that the cost for upgrading is no more than $|U|$ and that the resulting MST has bottleneck weight no more than 1. To see this, upgrade all nodes in U. Since each node from U has upgrading cost 1 (the only node with different upgrading cost is the root which is not contained in G), the total upgrade cost is exactly $|U|$. The resulting MST is a tree of height 2: its root is the node r, at first level there are all upgraded nodes (i.e. those in U), and at second level all remaining nodes (i.e. those in $V - U$). Since all edges of this tree are incident with a node of level 1, the weight of all edges is 1.

Let there be an $(\alpha, f(n'))$-approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE). Denote by T' the resulting MST of G'. The bottleneck weight of T' is no more than $f(n')$. Therefore all of its edges have weight 1 and the upgraded nodes must form a Dominating Set on G'.

Let $\mathrm{OPT} \leqslant n$ be the cost of an optimal upgrade node set. Then, the upgrading cost of T' is at most $\mathrm{OPT}\alpha \leqslant n\frac{1}{2}\ln n' \leqslant n \ln n < L$. Consequently, the root cannot be upgraded

in the produced solution. Hence the set of upgraded nodes forms a Dominating Set on G.

We conclude that the algorithm can be used as an α-approximation algorithm for MINIMUM DOMINATING SET which is a contradiction to the result of Feige [5]. □

A similar construction shows the hardness even in the case that all vertices have upgrading cost 1.

Theorem 16. (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *is* NP*-hard even if all vertices have upgrading cost* 1. *Also, unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(N^{\mathcal{O}(\log\log N)})$, *even in this unit cost case for any* $\alpha < \frac{1}{3}\ln n$, *and polynomial time computable function* f *there is no polynomial time approximation algorithm with performance* $(\alpha, f(n))$.

Proof. We use a similar reduction as in the proof of the preceding theorem. The instance $G' = (V', E')$ of (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) is constructed as follows. First, insert all nodes and edges from G into G'. Then add a new node r (the root) and connect it to all nodes of V. Third, for each claw in G with center v and fingers $N(v)$, set up a collection $L(v)$ of new nodes. Connect each of these nodes to all nodes of $\{v\} \cup N(v)$. Choose the number L of the nodes in $L(v)$ as $L = \lceil n \ln n \rceil + 1$.

Let $K := n + Ln$, then the number of nodes in G' equals $n' = K + 1$. Notice that $\ln n' = \ln((L+1)n + 1) = \ln(n\lceil n \ln n \rceil + 2n + 1) \leqslant_{\text{ae}} \ln(n^3) = 3 \ln n$.

The upgrading cost of each node equals 1 per definition. For each edge $e' \in E'$, we set $d_0(e') := f(n') + 1$ and $d_1(e') := d_2(e') := 1$. The bottleneck weight bound on (UPGRADING SIZE, TOTAL WEIGHT, SPANNING TREE) is set to 1.

As before, upgrading all nodes in U results in an MST of bottleneck weight 1: its root is r, at first level are the nodes of U, at second level the nodes of $V - U$. We now have to deal with the remaining nodes in the collections $L(v)$ for each $v \in V$. Since U is a dominating set in G, each star $\{v\} \cup N(v)$ around v must contain at least one node v' which is contained in U. So, we can connect all nodes of $L(v)$ through edges of weight 1 via v' to the MST. Therefore, all the edges in the resulting MST have weight 1.

Let there be an $(\alpha, f(n'))$-approximation algorithm for (UPGRADING SIZE, TOTAL WEIGHT, SPANNING TREE). Denote by T' the resulting MST of G'. All edges of T' have weight 1.

Let $\mathrm{OPT} \leqslant n$ be the cost of an optimal chosen upgrade set. Then, the upgrading cost of T' is at most $\mathrm{OPT} \cdot \alpha \leqslant n \frac{1}{3} \ln n' \leqslant n \ln n < L$. Consider the star around an arbitrary node v. Each of the nodes in $L(v)$ is connected via a light edge to the tree. If none of the nodes in the star would be upgraded, then each of the L nodes in $L(v)$ must be upgraded which would exceed the available budget. Therefore, at least one node of each star of G is upgraded and the set of upgraded nodes, restricted to the node set V, forms a Dominating Set of G. We conclude that the algorithm can be used as an α-approximation algorithm for MINIMUM DOMINATING SET which is a contradiction as before. □

References

[1] S. Arnborg, B. Courcelle, A. Proskurowski, D. Seese, An algebraic theory of graph reductions, J. ACM 40 (1993) (5) 1134–1164.

[2] O. Berman, Improving the location of minisum facilities through network modification, Ann. Oper. Res. 40 (1992) 1–16.

[3] M.W. Bern, E.L. Lawler, A.L. Wong, Linear-time computation of optimal subgraphs of decomposable graphs, J. Algorithms 8 (1987) 216–235.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[5] U. Feige, A threshold of $\ln n$ for approximating set cover, Proc. 28th Annual ACM Symp. on the Theory of Computing (STOC'96), 1996, pp. 314–318.

[6] M.R. Garey, D.S. Johnson, Computers and Intractability (A Guide to the Theory of NP-Completeness), W.H. Freeman and Company, New York, 1979.

[7] S.E. Hambrush, H.-Y. Tu, Edge weight reduction problems in directed acyclic graphs, J. Algorithms 24 (1997) 66–93.

[8] S.O. Krumke, M.V. Marathe, H. Noltemeier, R. Ravi, S.S. Ravi, Network improvement problems, in: P.M. Pardalos, D. Du (Eds.), Network Design: Connectivity and Facilities Location, AMS-DIMACS Volume Series in Discrete Mathematics and Theoretical Computer Science, vol. 40, American Mathematical Society, Providence, RI, 1998, pp. 247–268.

[9] S.O. Krumke, H. Noltemeier, S.S. Ravi, M.V. Marathe, K.U. Drangmeister, Modifying networks to obtain low cost trees, Proc. 22nd Int. Workshop on Graph-Theoretic Concepts in Computer Science, Cadenabbia, Italy, Lecture Notes in Computer Science, vol. 1197, June 1996, pp. 293–307.

[10] S.O. Krumke, H. Noltemeier, S.S. Ravi, M.V. Marathe, K.U. Drangmeister, Modifying networks to obtain low cost subgraphs, Theoret. Comput. Sci. 203 (1998) 91–121.

[11] S.O. Krumke, On the approximability of location and network design problems, Ph.D. Thesis, Lehrstuhl für Informatik I, Universität Würzburg, December 1996.

[12] F.T. Leighton, S. Rao, An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms, Proc. 29th Annual IEEE Symp. on the Foundations of Computer Science (FOCS'88), 1988, pp. 422–431.

[13] C. Phillips, The network inhibition problem, Proc. 25th Annual ACM Symp. on the Theory of Computing (STOC'93), May 1993, pp. 288–293.

[14] D. Paik, S. Sahni, Network upgrading problems, Networks 26 (1995) 45–58.

[15] N. Robertson, P. Seymour, Graph minors IV, treewidth and well-quasi-ordering, J. Combin. Theory Ser. B 48 (1990) 227–254.

[16] J. Valdes, R.E. Tarjan, E.L. Lawler, The recognition of series–parallel digraphs, SIAM J. Comput. 11 (1982) (2) 298–313.