EFFICIENT ALGORITHMS FOR POWER ASSIGNMENT PROBLEMS

BY

KAN QIAO

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
May 2015

ACKNOWLEDGMENT

I am full of appreciation at the end my Ph.D journey.

First of all, I would like to convey my gratitude to Professor Gruia Calinescu, my academic adviser. Without his encouragement, guidance and patience, this thesis would never lift off the ground. I learn not only research methodologies from Dr. Calinescu, but also the attitude of being professional. There is no doubt that such skills and personalities will benefit my future career.

I would like to sincerely thank Professor Sanjiv Kapoor and Professor Pengjun Wan for their valuable suggestions on improving my work. Their encouragement and support are important to the completion of the thesis. I am grateful to Professor Michael Pelsmajer for his interest in my research and taking his valuable time to serve on my thesis committee. I would like to thank all my other research collaborators, Jungwan Shin, Benjamin Grimmer. They either contributed generous support at high-level, or helped me with technical details. I will always miss the days and nights we worked together to exploit the exciting research topics.

Last but not least, I would like to dedicate this thesis to my family, for their years of support, encouragement and understanding.

K. Q.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Power assignment problems take as input a directed simple graph $G = (V, E)$ and a cost function $c : E \to R_+$. A solution to this problem assigns every vertex a nonnegative power, $p(v)$. We use $H = (V, B(p))$ to denote the spanning subgraph of G created by this power assignment. Let $B(p)$ denote the set of all the links established between pairs of nodes in V under the power assignment p. The minimization problem then is to find the minimum power assignment, $\sum p(v)$, subject to H satisfying a specific property. 4 variants of this problem are discussed in this paper (a) **Min-Power Strong Connectivity:** $H = (V, B(p))$ is strongly connected. (b) **Min-Power Broadcast:** $H = (V, B(p))$ has a path from the fixed source z to every other vertex. (c) **Min-Power Connectivity with 2-level power (Symmetric):** $c : E \to \{0, 1\}$ and $H = (V, B(p))$ is connected. (d) **Min-Power Strong Connectivity with 2-level power (Asymmetric):** $c : E \to \{0, 1\}$ and $H = (V, B(p))$ is strongly connected.

We give the exact solution using an improved integer linear program for problem (a) and (b) (We do not have a section for the integer linear program of Min-Power Broadcast problem since it is very similar to Min-Power Strong connectivity). Then we try to speedup current best approximation algorithms while preserving their approximation ratio. For problem (a), we give a fast variant of 1.85-approximation algorithm with running time $O(n^2 \log^2 n)$. For problem (b), we give a fast variant of $2(1 + \ln n)$-approximation algorithm for the most general cost model with running time $O(n^3)$ and a fast variant of 4.2-approximation algorithm for 2-dimensional cost model with running time $O(nm)$, where $n = |V|$ and $m = |E|$. For both problem (c) and (d), We give $\frac{5}{3}$-approximation algorithms that run in $O(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

CHAPTER 1

INTRODUCTION

The problem of assigning power to vertices of a graph to achieve a desired property has important uses in ad hoc wireless networks. This class of problems take as input a simple graph $G = (V, E)$ and a cost function $c : E \rightarrow R_+$. A solution to this problem assigns every vertex a nonnegative power, $p(v)$. We use H to denote the spanning subgraph of G created by this power assignment (Described in each chapter). The minimization problem then is to find the minimum power assignment, $\sum p(v)$, subject to H satisfying a specific property.

In this paper, we will discuss 4 problems: (a): Min-Power Strong Connectivity problem, (b): Min-Power Broadcast problem, (c): Min-Power Connectivity with 2-level Power (Symmetric) problem and (d): Min-Power Strong Connectivity with 2-level Power (Asymmetric). All of them are NP-HARD [24, 26, 50, 23]. Thus for large instance we must employ heuristics and approximation algorithms.

There are many approximation algorithms and heuristics have been proposed for these problems. However, they only focus on improving the approximation ratio but overlook the running time. In this paper, we propose variants of the 5 current best approximation algorithms and improve the running time by using advanced data structure and/or simple amortized analysis while preserving or improving their approximation ratio. All of the chapters are based on the journal submissions (Chapter 3 is based on the journal submission to Ad Hoc Networks). The Min-Power Strong connectivity problem (preliminary result in [14]) is discussed in chapter 2. The Min-Power Broadcast problem (preliminary result in [15]) is discussed in chapter 3. The Min-Power Connectivity with 2-level power (Symmetric) problem (preliminary result in [38]) is discussed in section 4.3. The Min-Power Strong Connectivity with 2-level power (Asymmetric) problem (preliminary result in [38]) is discussed in section 4.4.

CHAPTER 2

MIN-POWER STRONGLY CONNECTIVITY PROBLEM

2.1 Introduction

MIN-POWER STRONG CONNECTIVITY has as input a simple undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$. A *power assignment* is a function $p : V \to \mathbb{R}^+$. A *unidirectional link* from node u to node v is established under the power assignment p if $p(u) \geq c_{uv}$. Let $B(p)$ denote the set of all unidirectional links established between pairs of nodes in V under the power assignment p. MIN-POWER STRONG CONNECTIVITY asks for minimizing $\sum_{v \in V} p(v)$ subject to p being *valid*, that is, satisfying the constraint that the directed graph $(V, B(p))$ is strongly connected.

This is the original power assignment problem introduced by Chen and Huang [24] in 1989, and it is NP-Hard. In most wireless applications, as explained above, the vertices of G are embedded in the Euclidean space with two or three dimensions, and $c_{uv} = \|uv\|^\kappa$, where $\|uv\|$ denotes the Euclidean distance between u and v. However we are not aware of any theoretical research that used that the c function has this particular definition (except when the nodes lie on one line), and we also use the graph version (with arbitrary c) in this chapter as well. Kirousis, Kranakis, Krizanc, and Pelc [44]) proved that MIN-POWER STRONG CONNECTIVITY is NP-Hard in the three dimensional Euclidean space, with $c_{uv} = \|uv\|^\kappa$ (their proof also shows that no polynomial-time approximation scheme exists). Clementi, Penna, and Silvestri [27] give an elaborate reduction proving that MIN-POWER STRONG CONNECTIVITY is NP-Hard in two dimensions.

Chen and Huang [24] propose an efficient approximation algorithm, based on computing the minimum spanning tree (MST - seen as a set of edges). After the MST is computed, assign $p(v) = \max_{u:uv \in MST} c_{uv}$. Chen and Huang (and later [44]) prove that this minimum spanning tree based algorithm has approximation ratio 2 for MIN-POWER

STRONG CONNECTIVITY, and this is known to be tight (see, for example [3]). Only recently Calinescu [9] has improved the approximation ratio for a polynomial-time algorithm, to only 1.98, but based on a natural (greedy improvement) idea. Later the approximation ratio of the exact same algorithm is improved to only 1.85[11]. We call **Greedy** the algorithm of [9] as it is a variant of Chvatal's [25] algorithm for SET COVER. [9] does not give an explicit running time, but $O(mn)$ is achievable by carefully using standard methods (similar to the algorithm for topological sort given in Exercise 22.4-5 on page 552 of [28], using arrays, linked lists, and marking. We could not find a reference to the running time of [25] other than a class project). Here and later, $n = |V|$ and $m = |E|$.

The MST algorithm has as output a digraph that is symmetric in the sense that it contains the antiparallel arc of any arc. If one such an output is required (to acknowledge packets), we have the MIN-POWER SYMMETRIC CONNECTIVITY problem for which [3] gives the best known approximation ratio: $5/3 + \epsilon$.

In this chapter, we give an involved variant of **Greedy** (same approximation ratio), with running time $O(n^2 \log^2 n)$ for arbitrary input graphs (we do not assume the input nodes are embedded in the plane or in the three dimensional space). For this purpose, we make use of a novel combination of binary search trees and convex hulls in two dimensions. This allows us to apply the greedy algorithm to bigger instances, up to 2000 nodes in 15 seconds on an ordinary desktop. Experiments on random instances show an average of up to 15% improvement over the faster (only $O(n \log n)$ running time in two dimensions) MINIMUM SPANNING TREE-based algorithm.

We also tackle obtaining exact solutions, using integer programs, as described next. It turns out that algorithms for Power Assignment use many ideas from the classical STEINER TREE optimization problem. Recently, Byrka et. al [6] obtained a breakthrough in approximating STEINER TREE using a bidirected integer linear program first proposed by [57]. This integer linear program translates naturally to MIN-POWER STRONG CON-

NECTIVITY and appears later as **IP2**. We propose another integer linear program, **IP1**, similar to the one given in section 3 of [3] for MIN-POWER SYMMETRIC CONNECTIVITY, which has the advantage of having size (non-zero entries in the matrix) of $O(mn)$, as opposed to size $O(mn^2)$ for IP2. We use both integer programs with the CPLEX solver, and the biggest size of random instances that CPLEX can solve in one minute on the same ordinary desktop turns out to be 30 nodes, by IP2. We speculate that IP2 is solved faster, even though it is bigger, because it has a proven integrality gap of 1.85 [11]. This means that, for any instance, IP2 always has a solution of power at most 1.85 times the optimum LP2 value. We are not aware of other good integrality gap bounds, and indeed one can try to use IP2 for improving the approximation ratio.

Further experimental study reveals that on instances where we can compute the optimum (random 30 points in two dimensions), the MST-based algorithm is on average 28% bigger than optimum, and the Greedy algorithm is on average 16% bigger than optimum.

The chapter outline is as follows. Next section presents the integer program, after introducing some notation. Section 2.3 presents the fast approximation algorithm. In section 2.4 we present the simulation results.

2.2 Integer Linear Programs

First, some notation. Denote by $\bar{E}$ the bidirected version of E, obtained as follows. Use the same set of vertices, and for every edge $uv \in E$, put in $\bar{E}$ both arcs uv and vu, both with cost c_{uv}. For set of vertices X, denote by $\delta^-(X)$ the set of arcs of $\bar{E}$ with head in X and tail outside X, We write $\delta^-(u)$ instead of $\delta^-(\{u\})$. Let $\delta^+(u)$ be the set of arcs of $\bar{E}$ with tail u. Let $d(v)$ be the degree of vertex v in G, the input graph. For arc $e = uv \in \bar{E}$, we use $c(e) := c_{uv}$, and for $F \subseteq \bar{E}$, $c(F) := \sum_{e \in F} c(e)$.

We have two integer linear programs for the problem, described later. We will show later that IP1' has $O(nm)$ non-zero entries in its matrix and IP2' has $O(n^2 m)$ non-zero

entries in its matrix.

Below is **IP1**, with variables y_{vu} and x_{vu} for every arc vu of $\bar{E}$. Intuitively, x_{vu} "represents" that arc vu is established, and y_{vu} "represents" that u is the farthest node from v with arc vu established; exact explanation follows the integer program.

$$\text{minimize } \sum_{vu \in \bar{E}} c_{vu} y_{vu} \text{ subject to}$$

$$x_{vu} \leq \sum_{vz \in \bar{E}: c_{vz} \geq c_{vu}} y_{vz} \qquad \forall\, vu \in \bar{E} \tag{2.1}$$

$$\sum_{vu \in \delta^-(X)} x_{vu} \geq 1 \quad \forall\, X \subset V, \emptyset \neq X \neq V \tag{2.2}$$

$$x_{vu} \geq 0 \qquad \forall\, vu \in \bar{E} \tag{2.3}$$

$$y_{vu} \geq 0 \qquad \forall\, vu \in \bar{E} \tag{2.4}$$

$$y_{vu} \in \mathbb{Z} \qquad \forall\, vu \in \bar{E} \tag{2.5}$$

Definition 2.2.1. *We call two minimization problems* equivalent *if one can efficiently construct from the instance of one problem an instance of the other problem, one can efficiently construct from an optimum solution of one problem an optimum solution of the other problem (and vice versa), and the two optimal have the same value.*

Our proofs that problems are equivalent are based on the following standard method. Take an arbitrary instance, and an arbitrary feasible solution for one of the problems. Construct from it a feasible solution for the other problem, without increasing the objective. Do this again in the reverse direction, starting with a feasible solution of the other problem.

Lemma 2.2.1. *IP1 and MIN-POWER STRONG CONNECTIVITY are equivalent.*

Proof. We first argue the fact that any minimal valid power assignment yields a feasible

IP1 solution of the same objective. A power assignment is *minimal* if no $p(v)$ can be reduced while $B(p)$ still ensures strong connectivity. Indeed, if $p(v)$ is from a minimal power assignment, then there exists a node $u \in V$ such that $p(v) = c_{vu}$. For only one such u, set $y_{vu} = 1$ and, for all $vu' \in \bar{E}$, set $x_{vu'} = \sum_{vz \in \bar{E}: c_{vz} \geq c_{vu}} y_{vz}$. Then indeed, as $B(p)$ ensures strong connectivity, for any $X \subset V$ with $\emptyset \neq X \neq V$, there is an arc e in $B(p)$ (and thus with $x_e = 1$) that has the tail outside X and the head in X. Thus Constraints 2.2 are satisfied. One can immediately check that the objective function does not increase and that the other constraints are satisfied.

Conversely, given a feasible IP1 solution, set $p(u) = \max_{v: y_{uv}=1} c_{uv}$, and we obtain a valid power assignment (as proven next) that clearly has as objective at most the objective of the IP1 solution. Indeed, $B(p)$ is strongly connected: assume for a contradiction this is not the case, and so there are $v, u \in V$ with no path from v to u in $(V, B(p))$. Let X be the set of vertices that can reach u in $(V, B(p))$; then $v \notin X$ and therefore $X \neq V$, and also $X \neq \emptyset$ since $u \in X$. Constraints (2.2) show that there exist $v' \notin X$ and $u' \in X$ with $x_{v'u'} > 0$. Then, from Constraints (2.1), there exist $z \in V$ with $c_{v'z} \geq c_{v'u'}$ and $y_{v'z} > 0$; this means $y_{v'z} = 1$ and therefore $p(v') \geq c_{v'u'}$. Therefore $v'u' \in B(p)$ and therefore v' can also reach u as u' does (since $u' \in X$), contradicting $v' \notin X$. □

LP1 is the linear relaxation of IP1; that is, the linear program given by exactly the same constraints except the last one.

While conceptually useful, IP1 has exponentially many *cut* constraints (2.2), which we replace by *flow* constraints below. That is, we give **IP1'**, with variables y_{vu} and x_{vu} for every arc uv of $\bar{E}$, and f^z_{uv} and $\bar{f}^z_{uv}$, for every arc uv of $\bar{E}$ and z in $V \setminus \{r\}$, where $r \in V$ is a vertex we choose arbitrarily. The idea is to have f^z_{uv} represent flow on arc uv, of a commodity that ships one unit of flow from r to z, and $\bar{f}^z_{uv}$ represent flow on arc uv, of a commodity that ships one unit of flow from z to r.

$$\text{minimize} \sum_{vu \in \bar{E}} c_{vu} y_{vu} \text{ subject to}$$

$$x_{vu} \leq \sum_{vz \in \bar{E}: c_{vz} \geq c_{vu}} y_{vz} \tag{2.6}$$

$$0 \leq f^z_{uv} \leq x_{uv} \tag{2.7}$$

$$0 \leq \bar{f}^z_{uv} \leq x_{uv} \tag{2.8}$$

$$\sum_{vz \in \delta^-(z)} f^z_{vz} - \sum_{zv \in \delta^+(z)} f^z_{zv} = 1 \tag{2.9}$$

$$\sum_{vz \in \delta^-(z)} \bar{f}^z_{vz} - \sum_{zv \in \delta^+(z)} \bar{f}^z_{zv} = -1 \tag{2.10}$$

$$\sum_{vu \in \delta^-(u)} f^z_{vu} = \sum_{uv \in \delta^+(u)} f^z_{uv} \tag{2.11}$$

$$\sum_{vu \in \delta^-(u)} \bar{f}^z_{vu} = \sum_{uv \in \delta^+(u)} \bar{f}^z_{uv} \tag{2.12}$$

$$x_{vu} \geq 0 \tag{2.13}$$

$$y_{vu} \geq 0 \tag{2.14}$$

$$y_{vu} \in \mathbb{Z} \tag{2.15}$$

Constraints (2.6) hold $\forall\ vu \in \bar{E}$. Constraints (2.8) and (2.7) hold $\forall\ uv \in \bar{E}\ \wedge\ \forall\ z \in (V \setminus \{r\})$. Constraints (2.10) and (2.9) hold $\forall\ z \in (V \setminus r)$. Constraints (2.11) and (2.12) hold $\forall\ z \in (V \setminus r) \wedge \forall u \in V \setminus \{r, z\}$. Constraints (2.13), (2.14), and (2.15) hold $\forall\ vu \in \bar{E}$.

IP1' has $O(nm)$ variables, and one can check also that it has $O(mn)$ nonzero entries in the constraints matrix, as each of the variables f^z_{uv} or $\bar{f}^z_{uv}$ only appears four times in the program matrix, while each of the variables x_{vu} or y_{vu} appears at most $2n$ times in the matrix. **LP1'** is the linear relaxation of IP1'; that is, the linear program given by exactly

the same constraints except the last one.

Lemma 2.2.2. *IP1 and IP1' are equivalent. (Thus* MIN-POWER STRONG CONNECTIVITY *and IP1' are equivalent).*

Proof. We first prove that for a given feasible solution in IP1, we can construct a feasible solution in IP1' of the same objective. We use the values of y_{vu} of IP1 in IP1', and as a result we do get the same objective as IP1 and satisfy constraints (2.14) and (2.15). We set x_{vu} to make constraints (2.6) satisfied with equality, and notice that constraints (2.13) are also satisfied. We construct $B(p)$ as in the proof of the previous lemma using the values x_{vu}, and $(V, B(p))$ is a strongly connected graph. Then $\forall\ z \in (V \setminus \{r\})$, we use depth first search to find a path from r to z in $(V, B(p))$. For the fixed r and given z, set $f^z_{vu} = 1$ if a arc vu is included in this path, and $f^z_{vu} = 0$ otherwise. This assignment satisfies (2.9), (2.7) and (2.11). We assign similarly values $\bar{f}^z_{vu}$ using a path from z to r in $(V, B(p))$. This assignment satisfies (2.10), (2.8) and (2.12). So it is a feasible solution for IP1'.

Conversely, given a feasible solution of IP1', we construct a feasible solution for IP1. Using the same value of x_{vu} and y_{vu} in IP1' as in IP1 immediately satisfies every constraints except (2.2). It also gives same objective as that of IP1'. Let $X \subset V$ be such that $\emptyset \neq X \neq V$. Assume first that $r \in V \setminus X$, and let z be a vertex in X. Use x_{vu} as capacity of edge vu of $\bar{E}$. Then the variables f^z_{vu} give a feasible flow that ships one unit of flow from r to z this network, and thus any $r - z$ cut has capacity at least 1. Constraint (2.2) follows. If $r \in X$, we find a vertex z in $V \setminus X$ and apply the same argument using flow values $\bar{f}^z_{vu}$. □

Below is **IP2**, with variables x_{vu} for every arc vu of $\bar{E}$. The idea is that x_{vu} being 1 "represents" that u is the farthest node from v with arc vu established.

Also to define the integer program, for every arc $uv \in \bar{E}$, let D_{uv} be the directed hyperedge with tail u and heads the vertices w such that $uw \in \bar{E}$ and $c_{uw} \leq c_{uv}$. We say

that directed hyperedge $D_{uv} \in \delta^-(X)$, for $X \subset V, \emptyset \neq X \neq V$ iff $u \notin X$ and $D_{uv} \cap X \neq \emptyset$ (in words, the tail of D_{uv} is not in X, while at least one head is in X).

$$\text{minimize} \sum_{vu \in \bar{E}} c_{vu} x_{vu} \text{ subject to}$$

$$\sum_{D_{vu} \in \delta^-(X)} x_{vu} \geq 1 \quad \forall\, X \subset V, \emptyset \neq X \neq V \tag{2.16}$$

$$x_{vu} \geq 0 \qquad \forall\, vu \in \bar{E} \tag{2.17}$$

$$x_{vu} \in \mathbb{Z} \qquad \forall\, vu \in \bar{E} \tag{2.18}$$

LP2 is the linear relaxation of IP2; that is, the linear program given by exactly the same constraints except the last one.

IP2/LP2 is potentially stronger than IP1, as we don't know how to obtain from an LP1 solution an LP2 feasible solution, without increasing the objective, while we can construct from any LP2 feasible solution an LP1 feasible solution, without increasing the objective, as follows. If $(x'_{vu})_{uv \in \bar{E}}$ is a valid LP2 solution, set in LP1 $y_{vu} = x'_{vu}$ and $x_{vu} = \sum_{vz \in \bar{E}: c_{vz} \geq c_{vu}} y_{vz}$ for all $vu \in \bar{E}$. Then, indeed, one can check that Constraints (2.2) are satisfied using the fact that if $vu \in \delta^-(X)$, then for all z satisfying $c_{vz} \geq c_{vu}$, we have $D_{vz} \in \delta^-(X)$.

Lemma 2.2.3. MIN-POWER STRONG CONNECTIVITY *and IP2 are equivalent.*

Proof. We first prove that, given a feasible IP2 solution, we obtain a valid power assignment that has as objective at most the objective of the IP2 solution. To do so, set $p(u) = \max_{v: x_{uv}=1} c_{uv}$, and clearly the objective is at most the objective of the IP2 solution. Next we prove that p is a valid power assignment. Indeed, $B(p)$ is strongly connected: assume for a contradiction this is not the case, and so there are $v, u \in V$ with no path from

v to u in $(V, B(p))$. Let X be the set of vertices that can reach u in $(V, B(p))$; then $v \notin X$ and therefore $X \neq V$, and also $X \neq \emptyset$ since $u \in X$. Constraints (2.16) show that there exist $v' \notin X$, $u' \in V$, and $u'' \in X$ with $c_{v'u'} \geq c_{v'u''}$ and $x_{v'u'} > 0$. This means $x_{v'u'} = 1$ and therefore $p(v') \geq c_{v'u''}$, and therefore $v'u'' \in B(p)$ and therefore v' can also reach u as u'' does (since $u'' \in X$), contradicting $v' \notin X$.

We also need to argue the fact that any minimal valid power assignment p yields a feasible IP2 solution of the same objective. We can let x_{vu} to be 1 if vu is the longest edge with tail v in $B(p)$. Otherwise we let x_{vu} to be 0. Then we get an objective in IP2 at most the objective of the solution of the MIN-POWER STRONG CONNECTIVITY problem. All the constraints except (2.16) are satisfied. If (2.16) is not always satisfied, we have $\exists X \subset V \ \sum_{D_{vu} \in \delta^-(X)} x_{vu} < 1$. So there are no $v \notin X$, $u \in V$, and $u' \in X$ such that $c_{vu} \geq c_{vu'}$ and $x_{vu} = 1$ (and thus with $c_{vu'} \leq p(v)$). Then we can not find any edge of $B(p)$ coming in X. This contradicts the fact that $B(p)$ is strongly connected. So (2.16) is also satisfied. □

While IP2 appears to be "tighter" (in the sense that it has extra constraints, thus making LP2 approximate IP2 better), this comes at the price of a larger integer programming instance. As IP1, IP2 has exponentially many "cut" constraints, but when we replace them by flows, (as we did for IP1), we have $O(mn^2)$ non-zero entries in the matrix, as one can verify soon.

Fix arbitrary $r \in V$. Commodity rz ships one unit of flow from r to node z. Variables $f^z_{uv}(w)$ represent the total flow of commodity rz from node u to node w "through" directed hyperedge D_{uv}, for any $uv, uw \in \bar{E}$ with $c_{uw} \leq c_{uv}$, and $z \in V \setminus \{r\}$. f^z_{uv} represents the flow of commodity rz leaving u through hyperedge D_{uv}. $\bar{f}^z_{uv}(w)$ and $\bar{f}^z_{uv}$ have the symmetric role of shipping one unit of flow from z to r. Thus we have $O(n \sum_{v \in V}(d(v))^2)$ variables, each appearing four times in the constraints; and since $d(u) \leq n - 1$ for all u,

$\sum_{v \in V} (d(v))^2 \leq n \sum_{v \in V} d(v) = 2nm.$

Define D'_{vu} to be $V(D_{vu}) \setminus \{v\}$. Call **IP2'** the integer program below.

$$\text{minimize} \sum_{vu \in \bar{E}} c_{vu} x_{vu} \text{ subject to}$$

$$0 \leq f^z_{uv} \leq x_{uv} \tag{2.19}$$

$$0 \leq \bar{f}^z_{uv} \leq x_{uv} \tag{2.20}$$

$$1 + \sum_{zv \in \delta^+(z)} f^z_{zv} = \sum_{vz \in \delta^-(z)} \left(\sum_{u : z \in D'_{vu}} f^z_{vu}(z) \right) \tag{2.21}$$

$$\sum_{zv \in \delta^+(z)} \bar{f}^z_{zv} = 1 + \sum_{vz \in \delta^-(z)} \left(\sum_{u : z \in D'_{vu}} \bar{f}^z_{vu}(z) \right) \tag{2.22}$$

$$\sum_{yv \in \delta^+(y)} f^z_{yv} = \sum_{vy \in \delta^-(y)} \left(\sum_{u : y \in D'_{vu}} f^z_{vu}(y) \right) \tag{2.23}$$

$$\sum_{yv \in \delta^+(y)} \bar{f}^z_{yv} = \sum_{vy \in \delta^-(y)} \left(\sum_{u : y \in D'_{vu}} \bar{f}^z_{vu}(y) \right) \tag{2.24}$$

$$f^z_{yv} = \sum_{u \in D'_{yv}} f^z_{yv}(u) \tag{2.25}$$

$$\bar{f}^z_{yv} = \sum_{u \in D'_{yv}} \bar{f}^z_{yv}(u) \tag{2.26}$$

$$f^z_{uv}(w) \geq 0 \tag{2.27}$$

$$\bar{f}^z_{uv}(w) \geq 0 \tag{2.28}$$

$$x_{vu} \geq 0 \tag{2.29}$$

$$x_{vu} \in \mathbb{Z} \tag{2.30}$$

Constraints (2.19) and (2.20) hold $\forall\ uv \in \bar{E} \wedge \forall\ z \in (V \setminus \{r\})$. Constraints (2.21) and (2.22) hold $\forall\ z \in (V \setminus r)$. Constraints (2.23) and (2.24) hold $\forall\ z \in (V \setminus r) \wedge \forall y \in V \setminus \{r, z\}$. Constraints (2.25) and (2.26) hold $\forall\ z \in (V \setminus r) \wedge \forall yv \in \bar{E}$. Constraints (2.27)

and (2.28) hold $\forall uv \in \bar{E} \wedge \forall z \in (V \setminus \{r\}) \wedge \forall w \in D'_{uv} \setminus \{u\}$. Constraints (2.29) and (2.30) hold $\forall\ vu \in \bar{E}$.

We call **LP2'** the linear relaxation of IP2'; that is, the linear program given by exactly the same constraints except the last one.

Lemma 2.2.4. *LP2 and LP2' are equivalent.*

Proof. We first prove that we can construct a feasible solution of LP2' from any feasible solution of LP2. Given the solution x_{vu} of LP2, we use the same value of x_{vu} in LP2'. Then (2.29) is satisfied. We also have the same objective value. Construct a new graph (V', E') as following. Start with $V'' = \emptyset$ and $E' = \emptyset$. For all $vu \in \bar{E}$, create a vertex v_u and add it to a set V'', and create an arc vv_u with capacity equal x_{vu} and add it to E'. Set V' to be $V \cup V''$. Also create (and add it to E') arcs $v_u w$ with capacity ∞ for any $v, u, w \in V$ if $vu \in \bar{E}$ and $vw \in E$ and $c_{vw} \leq c_{vu}$.

For fixed $z \in V$ and recall the fixed $r \in V$, we will obtain the value of f^z_{vu} and $f^z_{vu}(w)$ as described below. For any cut of V' which separates vertices as (S, T) where $r \in S$ and $z \in T$, we will show that the capacity of this cut is at least 1. If the min cut contains an infinity edge, the capacity of the cut is obviously greater than 1. Let T' be $T \setminus V''$. T' is not empty because $r \in T$. Since (2.16) is satisfied, $\sum_{D_{vu} \in \delta^-(T')} x_{vu} \geq 1$. For any v, u such that $D_{vu} \in \delta^-(T')$, v belongs to S and there exists arc $v_u w$ such that w belongs to T'.

So $v_u \in T$. Then vv_u with size x_{vu} is an arc cross the cut (S, T). So the cut (S, T) of V' has capacity at least 1. So we can always find a flow with value 1 from the fixed node r to any node z. We set $f^z_{vw}(u)$ to be the flow on the edge $v_u w$, and set $f^z_{vu} = \sum_{y \in D'_{vu}} f^z_{vu}(y)$ so (2.21), (2.23), (2.27) and (2.25) are satisfied. Similarly, we obtain the value of $\bar{f}^z_{vu}$ and $\bar{f}^z_{vu}(w)$, and (2.22), (2.24), (2.28) and (2.26) are satisfied.

Conversely, given the feasible solution of LP2', we construct a feasible solution for LP2. We use the same values of x_{vu} from LP2' for LP2, and they immediately satisfy every constraints except (2.16). This also gives the same objective in LP2 as in LP2'. Let $X \subset V$ be such that $\emptyset \neq X \neq V$. Assume first that $r \in V \setminus X$, and let z be a vertex in X. We use the network (V', E') constructed earlier. Let $X'' = \{u_v \,|\, u \notin X \wedge \exists w \in X \, c_{uw} \leq c_{uv}\}$, and let $X' = X \cup X'$. Consider the $r - z$ cut of (V', E'): $(V' \setminus X', X')$. One can check that the capacity of this cut is exactly $\sum_{D_{vu} \in \delta^-(X)} x_{vu}$, and since the values $f^z_{vw}(u)$ give a feasible flow of value 1 from r to z in this network, we obtain that Constraint (2.16) is satisfied. If $r \in X$, we find a vertex z in $V \setminus X$ and apply the same argument using flow values $\bar{f}^z_{vw}(u)$. □

Lemma 2.2.5. *IP2' and* MIN-POWER STRONG CONNECTIVITY *are equivalent. (So IP2' and IP2 are also equivalent.)*

Proof. We first prove that we can construct a feasible solution of IP2' from an arbitrary feasible solution of MIN-POWER STRONG CONNECTIVITY. We can (without increasing the objective function) start with a minimal feasible solution of MIN-POWER STRONG CONNECTIVITY, and therefore there exists a node $u \in V$ such that $p(v) = c_{vu}$. For only one such u, set $x_{vu} = 1$; thus we have the same objective in IP2'. Also constraints (2.29) and (2.30) immediately satisfied. Since the $(V, B(p))$ is strongly connected, we can find a path between any pair of nodes. Let z be an arbitrary node, and recall that r is a fixed node. By depth first search, we find the directed path from r to z in $(V, B(p))$. Let arc vw be on this path, then there exist an edge $vu \in \bar{E}$ such that $c_{vw} \leq c_{vu}$ and $c_{vu} = p(v)$ and $x_{vu} = 1$. Set both $f^z_{vu}(w) = 1$ and $f^z_{vu} = 1$. Do this for all the arcs of the path, and set all the other f^z-values to 0. This finds a flow if value 1, so (2.21), (2.23), and (2.27) are satisfied. As for constraints (2.25), note that $f^z_{vu} = 1$ only if $x_{vu} = 1$. One similarly finds a path from z to r in $(V, B(p))$ and use it to assign values $\bar{f}^z$ that satisfy all the constraints.

Conversely, we prove that we can construct a feasible solution of MIN-POWER

STRONG CONNECTIVITY problem from an arbitrary solution of IP2'. Since we already know how to construct a solution of MIN-POWER STRONG CONNECTIVITY problem from a solution of IP2, we only need to show how to construct a solution of IP2 from our solution of IP2'. This is done exactly as in the second part of the previous lemma; one only needs to notice that Constraints 2.18 are also satisfied. □

2.3 Fast 1.85-approximation

From previous work, we present only the algorithm. We refer to [9] and its submitted journal version for details. The algorithm directly outputs a set of arcs $A \subseteq \bar{E}$; then we assign $p(u) = \max_{uv \in A} c_{uv}$.

2.3.1 The algorithm as previously presented in [9]. Let T be the undirected minimum spanning tree of G. Let $\bar{T}$ be the bidirected version of T.

For $u \in V$ and $r \in \{c_{uv} : uv \in E\}$, let $S(u,r)$ be the directed star with center u containing all the arcs uv with $c_{uv} \leq r$; note that r is the power of S. For a directed star S, let $E(S)$ be its set of arcs and $V(S)$ be its set of vertices.

For given $S(u,r)$, let $Q(u,r)$ be the set of edges e of T such that there exist $x, y \in V(S(u,r))$ with e on the path from x to y in T. Let $\bar{Q}(u,r)$ be the set of arcs of $\bar{T}$ on a directed path from u to some $x \in V(S(u,r))$; it is easy to verify that the undirected version of $\bar{Q}(u,r)$ is $Q(u,r)$.

For a collection $\mathcal{A}$ of directed stars $S(u_i,r_i)$, define $Q(\mathcal{A}) = \cup_{S(u_i,r_i)\in\mathcal{A}} Q(u_i,r_i)$ and $f(\mathcal{A}) = \sum_{e\in Q(\mathcal{A})} c(e)$. For $S = S(u,r)$, define $f_{\mathcal{A}}(S) = f(\mathcal{A} \cup \{S\}) - f(\mathcal{A}) = \sum_{e\in Q(u,r)\setminus Q(\mathcal{A})} c(e) = \sum_{e\in I_{\mathcal{A}}(S)} c(e)$, where $I_{\mathcal{A}}(S)$ is defined to be those arcs of $\bar{Q}(u,r)$ for which the undirected version is not in $Q(\mathcal{A})$.

The algorithm starts with $M = \bar{T}$ as the set of arcs, and adds directed stars to collection $\mathcal{A}$ (initially empty) replacing arcs from M to greedily reduce the sum of costs of

the arcs in M plus the sum of the powers of the stars in $\mathcal{A}$. For intuition, we mention that this sum is an upper bound on the power of the algorithm's output. To simplify later proofs, the algorithm makes changes (adding directed stars and removing arcs from M) even if our sum stays the same. To be precise:

Algorithm **Greedy:**

$\mathcal{A} \leftarrow \emptyset$, $M \leftarrow \bar{T}$

While ($f(\mathcal{A}) < c(T)$) do

$(u, r) \leftarrow \text{argmax}_{(u',r')} f_{\mathcal{A}}(S(u', r'))/r'$

$M \leftarrow M \setminus I_{\mathcal{A}}(S(u, r))$

$\mathcal{A} \leftarrow \mathcal{A} \cup \{S(u, r)\}$

Output $\cup_{S \in \mathcal{A}} E(S) \cup M$

[11] improves the approximation ratio of this algorithm from [9], spending four and a half pages on its correctness and approximation ratio analysis, with no explicit running time given.

2.3.2 Faster implementation. For intuition, Figure 2.1 shows two iterations of the algorithm.

For every u, let $v_1, \ldots, v_{d(u)}$ be the neighbors of u in G, sorted in non-decreasing order by c_{uv_i}, and make $v_0 = u$ for convenience, with $c_{uu} = 0$ below. Strictly speaking, $v_i = v_i(u)$, but u is always clear from the context. For simplicity, we duplicate those values r that appears twice in the list of c_{uv_i}'s. Recall that $Q(u, r)$ is the set of edges of T on a path in T between some $x, y \in V(S(u, r))$. We construct, as linked lists, and explicitly keep the sets $\hat{Q}(u, 1) = Q(u, c_{uv_1})$ and for $2 \leq j \leq d(u)$: $\hat{Q}(u, j) = Q(u, c_{uv_j}) \setminus Q(u, c_{uv_{j-1}})$. Constructing all these link lists can be done in total time $O(n)$ for each u, as explained next.

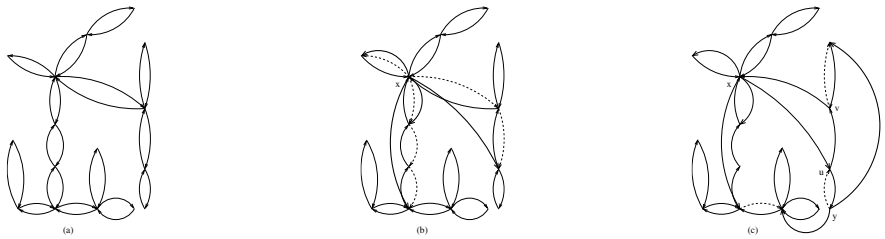

Figure 2.1. (a) Initial M. (b) Solid arcs give $\cup_{S\in\mathcal{A}}E(S)\cup M$ after adding to $\mathcal{A}$ the star S_1 (with thick solid arcs) centered at x. The arcs removed from M are dashed. (c) Solid arcs give $\cup_{S\in\mathcal{A}}E(S)\cup M$ after adding to $\mathcal{A}$ the star S_2 (with thick solid arcs) centered at y. The arcs removed from M are dashed. Note that the algorithm does not remove arc uv, as for the purpose of making the approximation proofs hold, only one arc from two anti-parallel arcs of $\bar{T}$ can be removed.

Root T at an arbitrary vertex and prepare it for *Least Common Ancestor* queries [41, 59]; thus $lca(x, y)$ gives in $O(1)$ time the node in T furthest from the root that is an ancestor of both nodes x and y. One can avoid the complicated Least Common Ancestor data structures of [41, 59] by using the method of [60] when computing alternating paths, or use simpler Least Common Ancestor data structures with $O(\log n)$ query time. Initially, every vertex of T is unmarked. For vertex x, let $parent(x)$ be its parent in T, and to avoid confusion we may write $x - y$ to represent the edge of $E(T)$ with endpoints x, y. Then execute:

Algorithm **Compute** $\hat{Q}(u, i)$ **for all** i

$z_0 \leftarrow u$; $mark(u)$

for $i \leftarrow 1$ to $d(u)$ do

$\hat{Q}(u, i) \leftarrow \emptyset$

$z_i \leftarrow lca(v_i, z_{i-1})$

$x \leftarrow z_{i-1}$

while ($x \neq z_i$) do

$\hat{Q}(u, i) \leftarrow \hat{Q}(u, i) \cup \{x - parent(x)\}$

$mark(parent(x))$; $x \leftarrow parent(x)$

endwhile

$x \leftarrow v_i$

while ($unmarked(x)$) do

$\hat{Q}(u,i) \leftarrow \hat{Q}(u,i) \cup \{parent(x) - x\}$

$mark(x)$; $x \leftarrow parent(x)$

endwhile

endfor

Note that with z_i being an ancestor of z_{i-1}, the first **while** loop does not make $x = nil$. Also, z_i is marked by the end of the first **while** loop, either during the loop, or, if $z_i = z_{i-1}$, during a previous iteration of the **for** loop, or, if $z_i = u$ at initialization. Thus also the second **while** loop does not make $x = nil$.

Each time we execute the code inside a **while** loop, we mark a new vertex, and thus the total running time is $O(n + d(u)) = O(n)$.

By traversing the computed $\hat{Q}(u,i)$ (recall, it is a linked list), store beforehand (in a $n \times (n-1)$ table), for every edge $e \in T$ and for every $u \in V$, the smallest index i, if any, with $e \in \hat{Q}(u,i)$. In the new version of **Greedy**, below $\tilde{Q}$ is used to represent $Q(\mathcal{A})$.

Algorithm **Greedy1:**

1. $\mathcal{A} \leftarrow \emptyset$; $M \leftarrow \bar{T}$; $\tilde{Q} \leftarrow \emptyset$
2. while ($c(\tilde{Q}) < c(T)$) do
3. $\quad maxv \leftarrow 0$
4. $\quad$ for ($u \in V$) do
5. $\quad\quad i_u \leftarrow \text{argmax}_{i \in \{1,\ldots,d(u)\}} \frac{c\left(Q(u,c_{uv_i}) \setminus \tilde{Q}\right)}{c_{uv_i}}$
6. $\quad\quad$ if ($\frac{c\left(Q(u,c_{uv_{i_u}}) \setminus \tilde{Q}\right)}{c_{uv_{i_u}}} > maxv$) then

7. $\hat{u} \leftarrow u$
8. $maxv \leftarrow \frac{c\left(Q(u,c_{uv_{i_u}})\setminus\tilde{Q}\right)}{c_{uv_{i_u}}}$
9. endif
10. endfor
11. $M \leftarrow M \setminus I_{\mathcal{A}}(S(\hat{u}, c_{\hat{u}v_{i_{\hat{u}}}}))$
12. $\mathcal{A} \leftarrow \mathcal{A} \cup \{S(\hat{u}, c_{\hat{u}v_{i_{\hat{u}}}})\}$
13. for ($e \in \cup_{i=1}^{i_{\hat{u}}} \hat{Q}(\hat{u}, c_{\hat{u}v_{i_{\hat{u}}}})$) do
14. $\tilde{Q} \leftarrow \tilde{Q} \cup \{e\}$.
15. endfor
16. endwhile
17. Output $\cup_{S\in\mathcal{A}} E(S) \cup M$

Other than Line 5, whose implementation and analysis we delay, everything can be made to run in time $O(n^2)$. Indeed, the **while** loop of Line 2 is run at most $n-1$ times, and Line 13 runs in time $O(n)$, and it is run at most $n-1$ times. For Line 11, we repeat Algorithm **Compute** $\hat{Q}$ for $u = \hat{u}$, which actually does "point" the edges of T away from u. We would need to check if an edge is in $\tilde{Q}$, which we keep as a bit vector.

The crux of the fast approximation is doing Line 5, each in $O(\log^2 n)$. The data structure is separate for each $u \in V$. Plot in two dimensions the points P_i with coordinates c_{uv_i} and $c\left(Q(u, c_{uv_i}) \setminus \tilde{Q}\right)$; also have P_0 with coordinates $(0, 0)$. Please refer to the left side of Figure 2.2 for intuition.

Consider the upper convex hull of these points, also drawn in the figure. (Given set of two-dimensional points S, a point $x \in S$ is an upper hull of a set of S iff for any two points $y, z \in S$, if the first coordinate of x (x_1 here) in the closed interval $[y_1, z_1]$, we have that x is on or above the segment yz.) Then (as in Graham's scan [28]), the neighbor of P_0 is the answer to the argmax of Line 5.

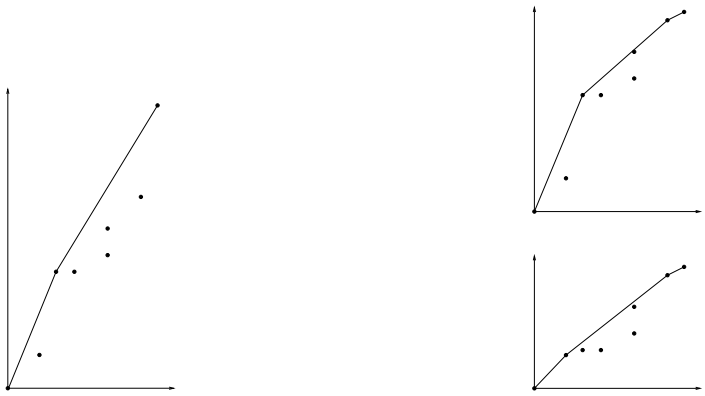

Figure 2.2. On the left, an example of points P_i, with the upper convex hull drawn. Recall that P_0 has coordinates $(0, 0)$. On the right, top, the points P_i after an edge e in $\hat{Q}(u, 7)$ is added to $\tilde{Q}$, causing $c\left(Q(u, c_{uv_7}) \setminus \tilde{Q}\right)$ to decrease. The upper convex hull is also drawn. On the right, bottom, the points P_i after an edge e in $\hat{Q}(u, 2)$ is added to $\tilde{Q}$, causing the second coordinate to drop for points $P_2 \dots P_7$. The upper convex hull is also drawn.

We need to quickly retrieve this neighbor of P_0 while $\tilde{Q}$ changes in Line 14, at most n times. Putting an edge in $\tilde{Q}$ decreases $c\left(Q(u, c_{uv_i}) \setminus \tilde{Q}\right)$ for all i with $e \in Q(u, c_{uv_i})$, by exactly $c(e)$. Thus the effect is felt by $i \geq i'$, where i' is such that $e \in \hat{Q}(u, i')$, and i' can be found in $O(1)$ time from e, as this information is stored beforehand. An example of this change appears on the upper right side of Figure 2.2, followed by another such change on the lower right side of Figure 2.2.

Assume for simplicity that $d(u) + 1$ is a power of 2. Keep a binary tree, defined recursively as follows. The root represents points $P_0 \dots P_{d(u)}$, its left child represent the first half of these points, and the right child the second half. A leaf of the tree represents exactly two points.

Each node x of this binary tree keeps, besides links to the left and right child, three values Δ, Δ_l, and Δ_r, and four indices i_l, i_r, j_l, and j_r. The indices i_l and i_r are such that x represents the points $P_{i_l} \dots P_{i_r}$, and do not change after initialization. The indices j_l and j_r are such that $j_l < (i_l + i_r)/2$, $j_r > (i_l + i_r)/2$, and $P_{j_l}P_{j_r}$ is a segment on the upper convex envelope of points $P_{i_l} \dots P_{i_r}$; they do change during the algorithm.

To explain the meaning of Δ, Δ_l, and Δ_r, we define the following quantity, which is not stored in the node x of the tree:

$$\Gamma(x) = \sum_{y\,:\,y \text{ ancestor of } x} \Delta(y), \tag{2.31}$$

where a node is consider to be an ancestor of itself. Then $\Delta_l(x)$ is such that the second coordinate of $P_{j_l}(x)$ is $\Gamma(x) + \Delta_l(x)$; similarly $\Delta_r(x)$ is such that the second coordinate of $P_{j_r}(x)$ is $\Gamma(x) + \Delta_r(x)$. Of-course, one needs to verify that the algorithm maintains that each record field represents what is explained above.

Now the computation of the i_u in Line 5 is easily done by a binary search: start at the root x, and if $j_l(x) = 0$, then $i_u = j_r(x)$ since the segment $P_0P_{j_r(x)}$ is on the upper convex hull. If $j_l(x) \neq 0$, then replace x by the left child of x, and resume.

The procedure for initializing the tree in time $O(n \log n)$ is not trivial, and discussed later as it uses the same methods as updating the tree (the more difficult part, below).

Updating the binary tree of vertex u after an edge e is added to $\tilde{Q}$ in Line 14 (we do this only for edges *added* to $\tilde{Q}$ - those edges of T that are already in $\tilde{Q}$ do not modify the data structure - and exactly $n-1$ edges are added to $\tilde{Q}$, in total) is considerably harder. Updating the $\Delta(x)$ values is fairly standard as in interval trees [29] (we still present pseudocode), but updating $j_l, \Delta_l, j_r, \Delta_r$ is harder and shown later below. i_l and i_r do not need updating. The procedure **Update**(i', x, c) is first called for x being the root of the binary tree of u, i' is smallest i such that $e \in \hat{Q}(u, i)$, and $c = c(e)$. Recall that such $i' = i'(u, e)$ is stored beforehand. An invariant is that the procedure is only called for x satisfying $i_l(x) \leq i' \leq i_r(x)$.

Algorithm **Update**(i', x, c)

1. if ($i' = i_l(x)$) then
2. $\Delta(x) \leftarrow \Delta(x) - c$
3. else
4. if (x is a leaf) then
5. $\Delta_r(x) \leftarrow \Delta_r(x) - c$
6. else
7. if ($i' < (i_l(x) + i_r(x))/2$) then
8. $\Delta(right\_child(x)) \leftarrow \Delta(right\_child(x)) - c$
9. Update($i', left\_child(x), c$)
10. else
11. Update($i', right\_child(x), c$)
12. endif
13. Readjust(x)
14. endif
15. endif

To somehow simplify the pseudocode, we moved **Readjust(x)** as a separate procedure. It is only called for non-leaf nodes x, and it assumes that all the fields in the subtree rooted at x are correct (the values correspond to what they represent), except for $j_l(x)$, $j_r(x)$, $\Delta_l(x)$, and $\Delta_r(x)$, which are what **Readjust(x)** must compute. We further separate two cases from **Readjust(x)**, making it finish after either **Readjust_left(x)** or **Readjust_right(x)** is called; these two functions take each care of the simpler cases when $j_r(x)$ and $\Delta_r(x)$ are already computed, or respectively $j_l(x)$ and $\Delta_l(x)$ are already computed.

Recall that every non-leaf node in the binary tree has exactly two children. The variable β is computed such that all points in the subtree of y are to the left of the vertical line at β, and all points in the subtree of z are to the right of this vertical line. Computations below can be done without knowing $\Gamma(x)$. In Line 5, the second coordinate of $P_{j_l(y)}$ is

$\Gamma(x) + \Delta_l(x) + \Delta_l(y)$. In Line 6, the second coordinate of $P_{j_r(y)}$ is $\Gamma(x) + \Delta_l(x) + \Delta_r(y)$. In Line 7, the second coordinate of $P_{j_r(z)}$ is $\Gamma(x) + \Delta_r(x) + \Delta_r(z)$. In Line 8, the second coordinate of $P_{j_l(z)}$ is $\Gamma(x) + \Delta_r(x) + \Delta_l(z)$. The same formulas are used in lines 2,3 of **Readjust_left(x)** and **Readjust_right(x)**.

Algorithm **Readjust(x)**

1. $y \leftarrow left\_child(x)$; $z \leftarrow right\_child(x)$
2. $\Delta_l(x) \leftarrow \Delta(y)$; $\Delta_r(x) \leftarrow \Delta(z)$
3. $\beta \leftarrow$ average of first coordinates of $P_{i_r(y)}$ and $P_{i_l(z)}$
4. do
5. compute the second coordinate of $P_{j_l(y)}$;
6. compute the second coordinate of $P_{j_r(y)}$;
7. compute the second coordinate of $P_{j_l(z)}$;
8. compute the second coordinate of $P_{j_r(z)}$;
9. compute P, the intersection of lines
   $P_{j_l(y)}P_{j_r(y)}$ and $P_{j_l(z)}P_{j_r(z)}$
10. if (P is to the left of vertical line β) then
11. if ($slope(P_{j_l(y)}P_{j_r(y)}) < slope(P_{j_l(z)}P_{j_r(z)})$) then
12. if (y is a leaf) then
13. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta_l(y)$; $j_l(x) \leftarrow j_l(y)$
14. Readjust_right(x); exit
16. else
17. $y \leftarrow left\_child(y)$
18. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta(y)$
19. endif
20. else

21. if (y is a leaf) then
22. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta_r(y)$; $j_l(x) \leftarrow j_r(y)$
23. Readjust_right(x); exit
24. else
25. $y \leftarrow right\_child(y)$
26. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta(y)$
27. endif
28. endif
29. else
30. if ($slope(P_{j_l(y)}P_{j_r(y)}) < slope(P_{j_l(z)}P_{j_r(z)})$) then
31. if (z is a leaf) then
32. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta_r(z)$; $j_r(x) \leftarrow j_r(z)$
33. Readjust_left(x); exit
34. else
35. $z \leftarrow right\_child(z)$
36. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta(z)$
37. endif
38. else
39. if (z is a leaf) then
40. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta_l(z)$; $j_r(x) \leftarrow j_l(z)$
41. Readjust_left(x); exit
42. else
43. $z \leftarrow left\_child(z)$
44. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta(z)$
45. endif
46. endif
47. endif

48. while ($true$)

Algorithm **Readjust_right(x)**

(called when $j_l(x)$ and $\Delta_l(x)$ are final)

1. do
2. compute the second coordinate of $P_{j_l(z)}$;
3. compute the second coordinate of $P_{j_r(z)}$;
4. if ($P_{j_l(z)}$ is above segment $P_{j_l(x)}P_{j_r(z)}$)
5. if (z is a leaf)
6. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta_l(z)$
7. $j_r(x) \leftarrow j_l(z)$; exit
8. else
9. $z \leftarrow left\_child(z)$
10. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta(z)$
11. endif
12. else
13. if (z is a leaf)
14. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta_r(z)$
15. $j_r(x) \leftarrow j_r(z)$; exit
16. else
17. $z \leftarrow right\_child(z)$
18. $\Delta_r(x) \leftarrow \Delta_r(x) + \Delta(z)$
19. endif
20. endif
21. while ($true$)

Algorithm **Readjust_left(x)**

(called when $j_r(x)$ and $\Delta_r(x)$ are final)

1. do
2. compute the second coordinate of $P_{j_l(y)}$;
3. compute the second coordinate of $P_{j_r(y)}$;
4. if ($P_{j_r(y)}$ is above segment $P_{j_l(y)}P_{j_r(x)}$)
5. if (y is a leaf)
6. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta_r(y)$
7. $j_l(x) \leftarrow j_r(y)$; exit
8. else
9. $y \leftarrow right\_child(y)$
10. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta(y)$
11. endif
12. else
13. if (y is a leaf)
14. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta_l(y)$
15. $j_l(x) \leftarrow j_l(y)$; exit
16. else
17. $y \leftarrow left\_child(y)$
18. $\Delta_l(x) \leftarrow \Delta_l(x) + \Delta(y)$
19. endif
20. endif
21. while ($true$)

Figure 2.3 illustrates several possible iterations of the algorithm. When doing **Readjust(x')**, we look at the upper convex hull of points P_0, P_1, P_2, P_3, which has as edges

the segments P_0P_2 and P_2P_3. We see that Line 9 is executed with y in the pseudocode being y' from the figure, and then Line 18, and Line 20 makes $j_l(x') \leftarrow j_l(y') = 0$. We also see that Line 26 is executed with z in the pseudocode being y" in the figure, and then Line 27, and Line 29 makes $j_r(x') \leftarrow j_l(y'') = 2$. The segment from $P_{j_l(x')}$ to $P_{j_r(x')}$ is drawn dashed. When doing **Readjust(x")**, we look at the upper convex hull of points P_4, P_5, P_6, P_7, which has as edges the segments P_4P_5, P_5P_6, and P_6P_7. We see that Line 9 is executed with y in the pseudocode being z' in the figure, and then Line 10, and Line 12 makes $j_l(x") \leftarrow j_r(z') = 5$. We also see that Line 26 is executed with z in the pseudocode being y" in the figure, and then Line 27, and Line 29 makes $j_r(x") \leftarrow j_l(z") = 6$. The segment from $P_{j_l(x")}$ to $P_{j_r(x")}$ is drawn dashed.

When doing **Readjust(x)**, we look at the upper convex hull of points P_0, P_2, P_5, P_6, which has as edges the dashed segment P_0P_2 and the dotted segment P_2P_6. We see that Line 9 is executed with y in the pseudocode being x' from the figure, and then Line 10, and Line 14 makes $y \leftarrow right\_child(x') = y"$. We also see that Line 26 is executed with z in the pseudocode being x" from the figure, and then Line 35, and Line 39 makes $z \leftarrow right\_child(x") = z"$. Line 43 is reached, and we now look at the upper convex hull of points P_2, P_3, P_6, P_7, which has as edges the segments P_2P_3 and P_3P_7, drawn in a dot-dashed pattern with arrows at both ends. Again Line 9 is executed, this time with y in the pseudocode being y" from the figure, and then Line 10, and Line 12 makes $j_l(x) \leftarrow j_r(y") = 3$. We also see that Line 26 is executed with z in the pseudocode being y" from the figure, z in the pseudocode being z" from the figure, and then Line 35, and Line 37 makes $j_r(x) \leftarrow j_r(z") = 7$. At this moment, the procedure terminates.

Lemma 2.3.1. Readjust(x) *correctly computes* $j_l(x)$, $j_r(x)$, $\Delta_l(x)$, *and* $\Delta_r(x)$.

Proof. The lemma can be proved by the similar idea in [56] and [30]. □

It immediate that **Readjust_left(x)** runs in time $O(\log n)$ for a given x: in every

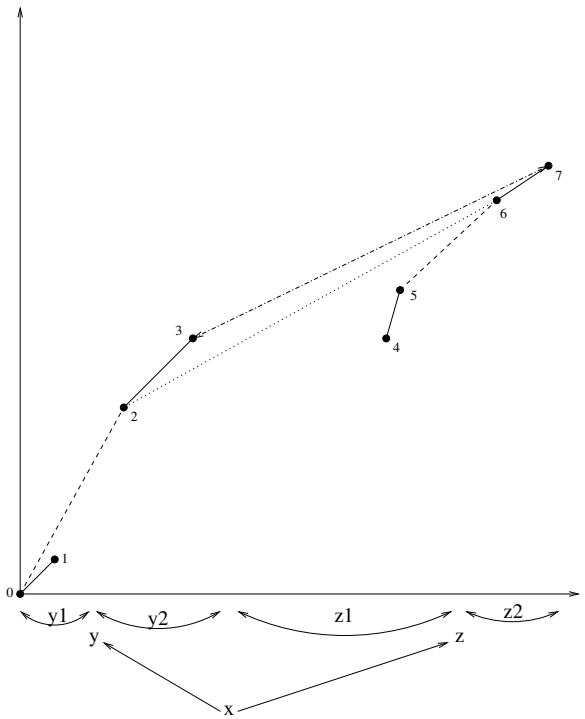

Figure 2.3. We illustrate the situation for a node u with $d(u) = 7$. The eight points P_i are given, with first coordinate c_{uv_i}, and second coordinate $c\left(Q(u, c_{uv_i}) \setminus \tilde{Q}\right)$. From the binary tree, we illustrate x as the root, and x' as its left child and x" as its right child. y' is the left child of x', and y" is the right child of x'. Also, z' is the left child of x", and z" is the right child of x". We have $j_l(y') = 0$, $j_r(y') = 1$, $j_l(y") = 2$, $j_r(y") = 3$, $j_l(z') = 4$, $j_r(z') = 5$, $j_l(z") = 6$, and $j_r(z") = 7$. For $v = y', y", z', z"$, the segment from $P_{j_l(v)}$ to $P_{j_r(v)}$ is drawn solid.

iteration of the **while** loop, y is replaced by one of its children, and the perfect balanced binary tree has depth $\log n$. A symmetric argument applies to **Readjust_right(x)**. Also, **Readjust(x)** runs in time $O(\log n)$ for a given x: in every iteration of the **while** loop (Lines 4-48), either y or z are replaced by a child, and when one becomes a leaf, exactly one call to either **Readjust_left(x)** or **Readjust_right(x)** is made. This makes constructing the initial tree possible in time $O(n \log n)$, in bottom-up fashion. It also gives a $O((\log n)^2)$ running time for **Update(**i', x, c**)**, as needed for a total running time of $O(n^2 \log^2 n)$.

2.4 Simulation results

We implemented the following four algorithms: the MST-based algorithms, **Greedy1** (and the slower **Greedy** as well), the IP1'-based algorithm and the IP2'-based algorithm, both using the free CPLEX Optimization Studio Academic Research Edition 12.2. All the code, other than CPLEX, was written in C++ and compiled with `gpp`, and run on a Intel(R) Core(TM) i7-2600M CPU @ 2.10GHz desktop with 16G memory. We checked using a separate program that our experimental outputs are strongly connected. We used the multi-thread capability using multi-core processors to speed up both **Greedy1** (where the **for** loop in Line 4 can be done in parallel) and the IP-based algorithms (CPLEX did use threads).

We included in our comparison faster versions of IP2' and Greedy which speeds up the computation by working on the Delaunay graph (see, e.g., [29]) defined by the nodes instead of the complete graph.

We use $\kappa = 2$ in our experiments in the two dimensional plane. Table 2.1 gives the percent improvement over MST and the runtimes for the compared algorithms; solution quality is also presented in graphical form in Figure 2.4. For each instance size, we generated 50 uniformly random instances, and 50 instances using the propellant setting of the GenSeN topology generator [16]. In the table, the size of an instance is followed by r

if uniformly random, and p if propellant. CPU is used to denote the CPU-time measured in seconds.

The results show that IP2' has a practical running time up to 40 nodes, and produces an average improvement over MST of 9-35%. The Delaunay version of IP2' (IP2-D in the table) has practical runtime up to 100 nodes, but gives slightly worse solutions. Note that this is not an optimum algorithm, as it ignores the non-Delaunay possible links, and in fact gives worse solutions than Greedy on some instances.

Greedy/Greedy1 comes within 4-16% of optimum on those instances where we can compute optimum. Greedy1 is practical up to 2000 nodes, where the straightforward Greedy requires more memory than the desktop provides. On bigger instances (where we cannot compare with optimum solutions), Greedy1 gives a bigger improvement in the quality of the solution compared to MST, and this improvement of Greedy1 seems to converge to circa 15%. Greedy-D produces slightly worse solutions and we cannot figure out a version of time complexity better than $O(n^2)$. For its simplicity, Greedy-D can be a practical choice, even though it has good running time only in two dimensions, and no proof of approximation ratio (by contrast, Greedy1's running time and approximation ratio are claimed for arbitrary input graphs). We also remark that for MIN-POWER SYMMETRIC CONNECTIVITY [3] suggests that the improvement over MST, on random instances, does not exceed 6.25%.

Table 2.1. Average percent improvement over the MST (%) and runtime in seconds (CPU) for the compared algorithms.

n	OPT-IP1		OPT-IP2		IP2-D		Greedy-D		Greedy1	
	%	CPU	%	CPU	%	CPU	%	CPU	%	CPU
20r	10.5	364	10.5	1.74	10.5	0.68	5.66	0.00	5.66	0.00
20p	12.1	331	12.1	3.43	12.1	0.85	5.2	0.00	5.20	0.00
25r	—	—	9.87	15.87	9.87	1.35	6.32	0.00	6.58	0.00
25p	—	—	9.32	16.36	9.32	1.47	7.31	0.00	7.51	0.00
30r	—	—	22.1	67.31	20.1	1.97	8.15	0.00	9.02	0.00
30p	—	—	19.4	69.56	17.4	2.23	7.32	0.00	8.77	0.00
35r	—	—	26.2	149.31	13.6	3.35	7.43	0.00	8.74	0.00
35p	—	—	27.11-	156.88	16.2	4.19	6.54	0.00	7.72	0.00
40r	—	—	35.32	441.22	11.36	4.58	7.89	0.00	9.11	0.00
40p	—	—	34.18	456.87	10.89	4.72	8.34	0.00	9.13	0.00
50r	—	—	—	—	17.3	6.43	8.67	0.00	11.42	0.00
50p	—	—	—	—	15.78	6.89	7.23	0.00	9.37	0.00
60r	—	—	—	—	14.7	10.43	10.95	0.00	11.80	0.00
60p	—	—	—	—	13.9	11.35	10.05	0.00	10.24	0.00
70r	—	—	—	—	12.2	34.19	11.45	0.00	12.64	0.00
80r	—	—	—	—	11.67	88.31	10.97	0.00	11.46	0.00
90r	—	—	—	—	13.31	135.31	11.89	0.00	12.24	0.00
100r	—	—	—	—	14.58	198.31	12.71	0.00	12.86	0.00
200r	—	—	—	—	—	—	13.43	0.00	13.67	0.00
400r	—	—	—	—	—	—	13.72	0.00	13.85	0.37
1000r	—	—	—	—	—	—	14.05	0.42	14.16	1.95
2000r	—	—	—	—	—	—	14.30	0.77	14.46	14.75

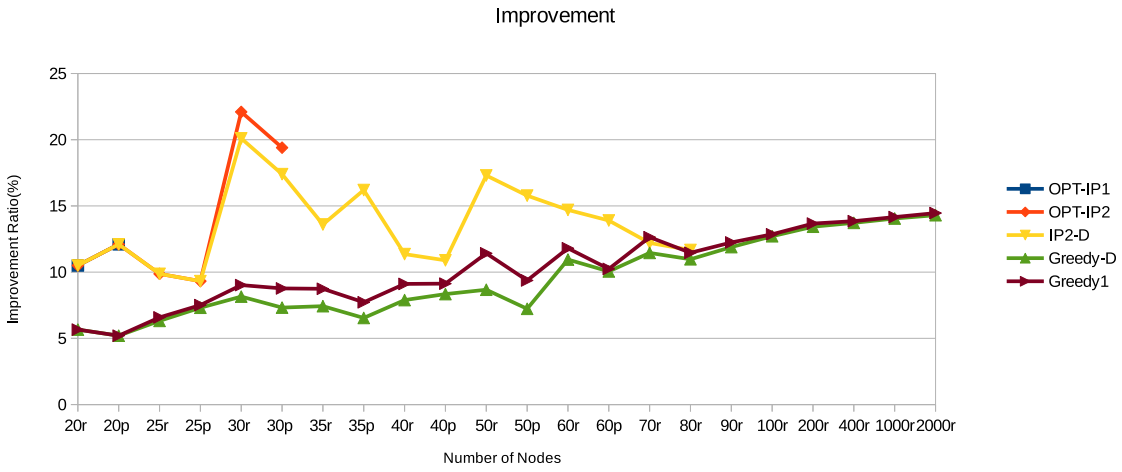

Figure 2.4. Improvement over MST

CHAPTER 3

MIN-POWER BROADCAST PROBLEM

3.1 Introduction

In this chapter, we make an experimental study of the problem of assigning transmission power to the nodes of ad hoc wireless networks to minimize total power consumption during a broadcast session (unidirectional links allowed). Precisely, MIN-POWER BROADCAST has as input a simple directed graph $G = (V, E)$, a vertex $z \in V$ (the source), and a cost function (sometimes called "power requirement") $c : E \to \mathbb{R}^+$. A *power assignment* is a function $p : V \to \mathbb{R}^+$. A *unidirectional link* from node u to node v is established under the power assignment p if $uv \in E$ and $p(u) \geq c_{uv}$. Let $B(p)$ denote the set of all unidirectional links established between pairs of nodes in V under the power assignment p. MIN-POWER BROADCAST asks for minimizing $\sum_{v \in V} p(v)$ subject to p being *valid*, that is, satisfying the constraint that the directed graph $(V, B(p))$ has a path from the source z to every other vertex.

The same problem was also studied in the bidirected input model (sometimes called "undirected" or "symmetric" in the literature), where the edge set of the input E is bidirected, (that is, $uv \in E$ if and only if $vu \in E$, and if weighted, the two edges have the same cost). In some wireless settings, it is reasonable to assume an *Euclidean* input model, where c_{uv} is proportional to the Euclidean distance from the position of u to the position of v, raised to a power κ, where κ is fixed constant between 2 and 5.

A survey of Power Assignment problems is given by Santi [58]; as there we only consider centralized algorithms (there is a vast literature on distributed algorithms). The general (directed) input model is appropriate in certain scenarios (i.e, it can take into account the residual battery of the nodes [10]), while the bidirected input model is more general than the Euclidean input model and is also appropriate in some scenarios (i.e, when

obstacles make communication between two nodes more power-consuming, or even impossible). From an algorithmic standpoint, it is easiest to tackle the two-dimensional Euclidean input model, then the three-dimensional input model, followed by the bidirected input model, and the general input model is the hardest. Even in the two-dimensional Euclidean input model, Min-Power Broadcast was proven NP-Hard [26]. Only in the one-dimensional Euclidean input model, Min-Power Broadcast has polynomial-time algorithms.

A fair number of approximation algorithms and heuristics have been proposed and will be discussed in the next subsections. The simplest (and fastest) is the minimum spanning tree (MST) algorithm, and the most sophisticated are the "greedy spider" [10] algorithm[1] and the "relative greedy" [19] algorithm. These two approximation algorithms are however also the slowest, with neither [10] nor [19] making an explicit running time analysis. We have improved their running time, while keeping the approximation ratio.

Precisely, in Section 3.3, we present a variant of the Relative-Greedy algorithm with running time $O(nm)$ and the same approximation ratio of 4.2 in the two-dimensional Euclidean input model, where n is the number of nodes and m is the number of edges in the input graph (m could be as high as $n(n-1)/2$, in which case we have an $O(n^3)$-time algorithm). The space complexity of this algorithm is $O(m+n)$. In Section 3.4 we present a variant of the Greedy-Spider algorithm with running time $O(n^3)$ and the same approximation ratio of $2(1+\ln n)$ in the most general model. The space complexity of this algorithm is $O(mn)$, compared to $O(n^2)$ for the naive variants ($O(m+n)$-space also seems possible with $O(nm(m+n\log n))$ running time).

[1] Actually, more complicated (and slow) algorithms may very well exists. Precisely, [10] suggests the existence of a $1.35\ln n$ approximation algorithm based on the ideas of [39]; this algorithm needs in the worst case $\Omega(n)$ calls to Minimum Weight Perfect Matching. Such running time would be impractical for large instances. A $1.5\ln(n)$ approximation algorithm was claimed by [37]; however this paper has errors. These may be fixable at the expense of making $\Omega(n)$ calls to Minimum Weight Perfect Matching or an $O(m)$ increase in the running time.

The new variants make use of existing advanced data structures and/or simple amortized analysis, improving naive variants by a factor of $\Theta(n)$. This improvement allows us to apply these algorithms to large instances (2000 nodes, compared to 210 in [17]). We then conducted extensive experiments on random input data in the two-dimensional Euclidean input model, using these two algorithms and many of the other existing ones (we only excluded ideas that have never been tried on large instances and for which the running time appears to be high).

We also use a post-processing technique as suggested by [7], once after each algorithm. It turns out that the algorithms are varied enough that one should try all the practical ones and pick the best output. Huge instances can be solved by the MST algorithm, and we measure an algorithm, or combination of algorithms, by the percentage improvement over the objective function of the MST algorithm. The improved running times allows adding Greedy-Spider and Relative-Greedy to the combination of algorithms for large instances (500 to 2000 nodes), which in turn increases the improvement over the MST algorithm from an average of circa 9% (on large random instances) to an average of circa 14% (a circa 50% increase in this improvement).

We also tackle obtaining exact solutions, using natural integer programs similar to those used by Byrka et. al [6] in approximating STEINER TREE (a bidirected integer linear program first proposed by [57]). Using an academic version of CPLEX, we can solve exactly instances with up to 30 nodes in the Euclidean input model within circa 72 seconds. Further experimental study reveals that on instances where we can compute the optimum (random 40 points in two dimensions), the MST-based algorithm is on average circa 66% bigger than optimum, and the combination of approximation algorithms and fast heuristics is on average circa 37% bigger than optimum.

3.1.1 Approximation Algorithms and Heuristics. Min-Power Broadcast was first studied by Wieselthier et al. [66]. They proposed three heuristics but do not prove approxima-

tion ratios. The first heuristic, SPT, uses a shortest-path tree from the source z. The second heuristic, MST, uses a minimum spanning tree. In both cases the resulting undirected graph is oriented away from the source, and power is assigned accordingly. The third heuristic, called BIP (broadcasting incremental power), is a Prim-like heuristic which starts with an outgoing branching consisting of the source, and iteratively adds an arc connecting the set of nodes currently reached from the source to an unreached node, with minimum increase in the total power given by the selected arcs.

For the two-dimensional Euclidean input model, Wan et al. [63] study the approximation ratios of the above three heuristics. An instance was constructed in [63] to show that the approximation ratio of SPT is as large as $\frac{n}{2} - o(1)$. On the other hand, MST has a constant approximation ratio [26, 64, 63, 65]. The approximation ratio of BIP is at most the ratio of MST [63]. A sequence of further papers [8, 34, 32, 53, 33, 4] improved the analysis of the MST algorithm to obtain a ratio of 6; this was known to be best possible since [64]. In three dimensions, the MST algorithm also has constant approximation ratio [54]. The running time of the MST algorithm is $O(m + n)$ in bidirected input graphs (where it has approximation ratio $\Theta(n)$), and $O(n \ln n)$ in the two-dimensional Euclidean input model. Thus it can be used on huge (million-node) instances; it was experimentally analyzed in [35].

It is straightforward to give a variant of BIP running in $O(n^2)$; its approximation ratio is between 4.598 [5] and 6 (as explained above). Thus BIP is a practical algorithm. The best approximation ratio in the two-dimensional Euclidean input model is 4.2 and is achieved by the Relative-Greedy algorithm of [19].

In the bidirected input model, a standard reduction from Set Cover shows that no approximation ratio better than $O(\ln n)$ is possible unless $P = NP$ (using [31]). This reduction was known in 2000 and appears in several papers [26, 49, 20, 7, 47, 10]. The first $O(\ln n)$ approximation algorithm was given by Caragiannis et al. [20] (journal version:

[21]). Similar algorithms were presented in [48, 62], and the simplest and best variant (ratio of $2(1+\ln n)$) of this algorithm was presented by [52] and achieves a $O(mn)$ running time (their analysis claims $O(mn\alpha(m,n))$ running time, but one observation can get rid of the inverse Ackermann function α in the analysis). This algorithm and the crux of its proof of approximation ratio can be traced back to [68] (see also [36] for a simpler analysis), which is why we call it Hypergraph-Greedy. Later, [22] obtains another $O(\ln n)$ approximation algorithm, with a complicated algorithms based on [39], that needs in the worst case $\Omega(n)$ calls to Minimum Weight Perfect Matching. Relative-Greedy [19] also has approximation ratio of circa $2(1+\ln n)$ in this model.

In the general (directed) model, [10] were the first to publish a $O(\ln n)$-approximation: the Greedy-Spider algorithm (adapted from [46]). In a simultaneous submission [18] (journal version: [22]) also obtains a $O(\ln n)$-approximation, with a worse constant in front of $\ln n$, and with what appears to be a large running time.

Two other heuristics that do not have a good approximation ratio are fast enough and make sense. In [7, 62, 45], a localized reduction in power is proposed. We adopt this as a postprocessing for all of our algorithms (to do so after each iteration of various algorithms is too time consuming). Algorithm 2 of [52], which we call Greedy-Broom, is somehow similar to BIP, but chooses a directed star (defined in the next section) instead of an arc, minimizing the ratio of the power of the star by the number of newly reached nodes. It has a $O(mn)$ running time.

We do not consider the algorithms of [40, 42, 67, 69, 2, 51] as they seem unsuitable for large instances.

3.2 Preliminaries

When discussing approximation ratios, natural logarithm ($\ln$) was used. For set X and element v, we may use $X+v$ to denote $X\cup\{v\}$ and $X-v$ to denote $X\setminus\{v\}$. For a

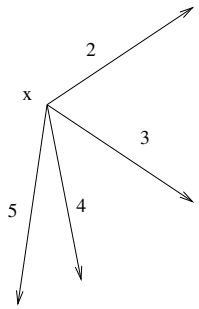

Figure 3.1. A star with center x and four arcs, of power $\max\{2, 3, 4, 5\} = 5$

set of arcs/edges X, we write $c(X) = \sum_{uv \in X} c_{uv}$. For a tree T and $u, v \in V(T)$, we use $P_T(u, v)$ to denote the simple path from u to v in T. Given a subset of edges $B \subseteq E(T)$, we say that an edge $e \in E(T)$ *disconnects* u from v in $(V, E(T) \setminus B)$ if u, v are connected in $(V, E(T) \setminus B)$ and $e \in E(P_T(u, v))$.

Given a directed spanning subgraph H of G, the *power* of a vertex u in H is given by $p_H(u) = \max_{uv \in E(H)} c_{uv}$. The *power* of H is given by $p(H) = \sum_{u \in V} p_H(u)$. It is easy to check that $p(H)$ is the minimum total power that results in establishing all the arcs of H (and possibly more arcs).

For $u \in V$ and $r \in \{c_{uv} \mid uv \in E\}$, let $S(u, r)$ be the *directed star* consisting of all the arcs uv with $c_{uv} \leq r$. We call u the center of S and note that r is the power of $S(u, r)$. For a directed star S, let $p(S)$ denote its power, let $E(S)$ be its set of arcs and define $V(S)$, its set of vertices, to be its center plus the heads of its arcs. See Figure 3.1 for an example. Many algorithms described above inspect all the possible stars, and there are $O(m)$ stars (for each vertex u, the number of stars with center u is u's degree/out-degree in the input graph).

The Hypergraph-Greedy algorithm [52] keeps a set of stars (initially empty), giving a set of arcs H. It then selects the next star such that to maximize the decrease in the number of weakly connected components in (V, H) divided by the power of the star. At the

end, it re-orients if needed the edges of H to lead away from the source z and thus obtain an out-going arborescence from the source (this idea, first applied in [20], only works if the input graph is bidirected, and typically loses a factor of 2 in the approximation ratio). See [52] for the $O(mn)$ variant. Recently, Calinescu [12] has obtained an $O(m \ln^2 n)$ variant of this algorithm using geometric data structures.

3.3 The Relative-Greedy algorithm

The Relative-Greedy algorithm of [19] (using ideas from [70]) keeps an undirected spanning tree T (initially, the minimum spanning tree in G according to cost c), and then does greedy local improvements as follows: For a set of vertices X, define the *swap-set* of X in T, $W_T(X)$, to be the maximum-cost set of edges of T such that, removing $W_T(X)$ from T leaves $|X|$ components, each containing exactly one vertex of X. Recall that $V(S)$ is the set of vertices of a star S. Choose a star $S = S(u, r)$ to maximize the ratio $c(W_T(V(S)))/p(S)$ and add it to the set of stars (which starts empty), provided that this ratio exceeds 2. Remove $W_T(X)$ from T, and add "fake" edges: the undirected version of all the arcs of S, with the new cost 0, to obtain the tree used for the next iteration. Repeat choosing another star as long as such a star exists. Once no star has ratio exceeding 2, take the graph consisting of the "real" edges of T and the arcs of all the selected stars, and just as for the Hypergraph-Greedy algorithm above, re-orient if needed its edges to obtain an out-going arborescence from the source z.

One needs to find the next star at most n times, and the challenge is to compute $W_T(V(S))$ and $c(W_T(V(S)))$ for every S. One naive method (as in Problem 23-1 of [28]) takes $O(n^2)$ per star. This can be improved to $O(n)$ per star by using the following known fact: For any spanning tree T, set $X \subseteq V$, and $v \in V \setminus X$, the swap-set $W_T(X+v)$ consists of $W_T(X)$ plus one edge. The stars $S(u, r)$ with the same center u do have the property that each has one more vertex compared to the previous one, if they are sorted by r.

We believe that the running time can be further improved to $O(\ln n)$ amortized time per star, using the linking-and-cutting trees data structure of Sleator and Tarjan (based on Splay Trees; see [60]). Here we do even better (details to follow), with the main operation being: once the tree T is changed into T' by the addition of a fake edge and the removal of a "true" edge, our algorithm finds the edge in $W_{T'}(X+v) \setminus W_{T'}(X)$ (for each of the relevant X and v) in constant time using some precomputation of the tree T', and the knowledge accumulated and saved for T.

Let $x_0, x_1, \ldots, x_q$ be an ordered list of $q+1$ vertices, and for $j = 1, 2, \ldots, q$, let $X_j = \{x_0, x_1, \ldots, x_j\}$. Let $B_j := W_T(X_j)$. We use the following claim that is implicit in several earlier papers (such as [70]). Its proof, included for completeness, is in the Appendix.

Claim 3.3.1 (Folklore)**.** *Assume all the tree costs of the minimum spanning tree T are nonnegative.*

1. *For any $j = 1, 2, \ldots, q$, we have $|B_j| = j$.*
2. *B_j can be computed by the following* **first procedure***. Start with $B = \emptyset$, and repeatedly find (if not possible, stop, and output $B_j = B$) any two vertices of X_j that are in the same connected component of $(V, E(T) \setminus B)$, then find the largest-cost edge on the unique simple path of T between these two vertices, and add it to B.*
3. *For any $j = 1, 2, \ldots, q$, B_j can be obtained from B_{j-1} by the following* **second procedure***: the graph $(V, E(T) \setminus B_{j-1})$ has j connected components, each containing one vertex of X_{j-1}. Find the component with v_j, and then find e, the largest-cost edge of T on its unique simple path from v_j to the unique vertex of X_{j-1} in this component. Then $B_j = B_{j-1} + e$.*

Proof. Assume from now on that no two edges have the same cost (if edges have the same

cost, break ties arbitrarily but consistently). Note that the removal of i edges from T leaves exactly $i + 1$ connected components.

This paragraph proves the first point. If $|B_j| > j$, then $(V, E(T) \setminus B_j)$ has more than $j + 1$ connected components and thus one without any vertex of X_j; in this case one edge of B_j, incident to such a component, can be removed from B_j and still have every vertex of X_j in a distinct component of $(V, E(T) \setminus B_j)$. If $|B_j| < j$, then $(V, E(T) \setminus B_j)$ has at most j components and thus one contains two vertices of X_j, which makes B_j not a valid swap-set for X_j. Thus we conclude that $|B_j| = j$.

We continue with the second point. Let B be the current set of the first procedure. Every edge selected in B is on the unique simple path of two vertices of X_j in T and thus after selecting i edges, we have $i + 1$ components of $(V, E(T) \setminus B)$, each with at least one vertex of X_j. Assume (in order to obtain a contradiction) that there is an $i > 0$ such that the i^{th} edge selected by the procedure to enter B is not in B_j, and choose in the following the smallest such i. Let B' be B right before this selection, and e' be the edge selected that is not in B_j. Note that $B' \subset B_j$. Let y and y' be the two vertices of X_j that are used by the procedure; thus y and y' are in the same component (which we call Q) of $(V, E(T) \setminus B')$. Let x and x' be the endpoints of e'; since $e' \notin B_j$, x and x' are in the same connected component (which we call Q') of $(V, E(T) \setminus B_j)$. Let v' be the unique vertex of $V(Q') \cap X_j$ (v' could be y' or y or x or x'). Switch the notation of x and x' so that x' is on the unique simple path $P_T(v', x)$ of Q'.

In a first case (another case being symmetric is omitted) x is on the unique simple path $P_T(x', y')$ of Q (the other case has y instead of y'). Note that in this case $v' \neq y'$. Let f be the first edge of B_j on the simple path $P_T(x, y')$ of Q; such an edge must exist since otherwise v' and y' would be connected in $(V, E(T) \setminus B_j)$. Note also that f is on the unique simple path $P_T(y, y')$ of Q and therefore the cost of f is smaller than the cost of $e' = xx'$. Add e' to B_j; this splits Q' into two components, one with v' and another with x; note that

no vertex of X_j is left in the connected component of x in $(V, E(T) \setminus (B_j + e'))$. Then $(B_j + e') - f$ is another candidate for the swap-set of X_j in T, with a bigger cost than B_j, a contradiction.

The second procedure (the third point of the claim) is the same as the first procedure (from the second point of the claim) when we only use as one of the two vertices of the procedure the vertex v_j (the only vertex in $X_j \setminus X_{j-1}$) only for the last edge (the j^{th}) to be picked in B. □

The algorithm keeps the following information, for current tree T. Fix in the discussion vertex $u \in V$. For this u, let $v_1, \ldots, v_{d'(u)}$ be the neighbors of u in G, sorted in non-decreasing order by c_{uv_i}. Here $d'(u)$ stands for the degree of u in the graph G. For convenience, set $v_0 = u$ and $d' = d'(u)$. Let $S_1, \ldots, S_{d'}$ be the stars with center u, where $V(S_j) = v_0, v_1, \ldots, v_j$ and the arcs connecting u to each v_i are included. Note that these stars with center u are sorted such that their power is non-decreasing and $p(S_i) = c_{uv_i}$. For convenience, define S_0 to be the one-vertex graph with only v_0. Let $A_j := W_T(V(S_j))$.

Based on Claim 3.3.1, with the list $x_0, x_1, \ldots, x_q$ being $v_0, v_1, \ldots, v_{d'}$ and, therefore, $A_j = B_j$, the algorithm "stores" the sets A_j by keeping a pointer to the edge $e_j := A_j \setminus A_{j-1}$, using an array of size d'. This is done for every vertex u (as the center of the stars).

With this information, in time $O(m)$, one can go through all the stars S (for each u, use the order $S_1, \ldots, S_{d'(u)}$) to compute $c(W_T(V(S)))$ and identify the star S that maximizes the ratio $c(W_T(V(S)))/p(S)$. Next, the algorithm removes $W_T(V(S))$ from T, and adds "fake" edges: all the undirected version of the arcs of S, with the new cost 0, in order to obtain the tree used for the next iteration. Call u the center of S, and let j be such that $S = S_j$ (from the list $S_1, \ldots, S_{d'(u)}$ mentioned above). Recall that $A_j = W_T(V(S_j))$. If, instead of removing A_j from T and adding the fake edges from S_j (all at once), we execute

sequentially, for $i = 1$ to j, the step that adds a fake edge of cost 0 and endpoints u and v_i, and removes e_i from T (recall that $e_i = A_i \setminus A_{i-1}$), then one can check that we have exactly the same effect (that we keep a tree after each single-edge replacement follows from the next lemma). We call one such replacement a *mini-iteration*. We have another property:

Lemma 3.3.1. *For all $i = 1, \ldots, j$, e_i becomes the largest-cost edge on the unique simple path in the current tree from u to v_i, before it is removed and replaced by uv_i.*

Proof. Let $i \in \{1, \ldots, j\}$ be arbitrary. Let T be the tree before the iterations starts, and T' be the current tree, before the i^{th} mini-iteration. From the second procedure of Claim 3.3.1, e_i is the largest-cost edge on $P_T(v_i, v_l)$, for some $l \in \{0, \ldots, i-1\}$, and $P_T(v_i, v_l) \cap A_{i-1} = \emptyset$. Then $P_T(v_i, v_l) = P_{T'}(v_i, v_l)$, and thus, as T does not contain the edge uv_l, $P_{T'}(v_i, u)$ is either $P_T(v_i, v_l)$, if $l = 0$, or $P_T(v_i, v_l)$ followed by the edge uv_l. As the edge uv_l has cost 0, the lemma follows. □

3.3.1 Handling the fake edges. To obtain a better bound on the running time, we modify the algorithm such that it does not change T in the case that $e_i \in A_j$ is a fake edge (we are unable to rule out e_i being a fake edge). This modification requires arguments that the approximation ratio is not changed; for this we prove (below) by induction on the number of iterations (stars selected and used) that both variants of the algorithm pick the same stars, and keep a tree of the same total cost. We do assume that ties regarding edge costs are broken the same way.

Definition 3.3.1. *A path is called a* fake *path if every edge in this path is a fake edge. For any tree $\hat{T}$, $P^r_{\hat{T}}(u, v)$ is the ordered list of non-fake* directed *edges, in the order they appear on $P_{\hat{T}}(u, v)$. Here we obtain a directed edge xy from an undirected edge $\{x, y\}$ by using the direction of $P_{\hat{T}}(u, v)$ (in other words, u is closer to x than to y on $P_{\hat{T}}(u, v)$).*

Let T be the tree kept by the original variant of the algorithm and $\bar{T}$ be the tree kept by the modified variant. We also prove the stronger **first condition** that for any set $X \subseteq V$,

(and at any moment), $W_T(X) \setminus W_{\bar{T}}(X)$ and $W_{\bar{T}}(X) \setminus W_T(X)$ are either sets consisting only of fake edges, or empty. We also prove the **second condition** that, for any two vertices u, v, $P^r_T(u,v) = P^r_{\bar{T}}(u,v)$. The two variants of the algorithm start with the same tree and an empty set of stars, so both of the conditions above hold initially.

From now on, we prove the induction step of our conditions. So we reach the situation that both the original and the modified algorithms have selected the same set of stars, that for any set $X \subseteq V$, $W_T(X) \setminus W_{\bar{T}}(X)$ and $W_{\bar{T}}(X) \setminus W_T(X)$ are either sets consisting only of fake edges, or empty, and that, for any two vertices u, v, $P^r_T(u,v) = P^r_{\bar{T}}(u,v)$.

Consider any star S. The condition that $W_T(X) \setminus W_{\bar{T}}(X)$ and $W_{\bar{T}}(X) \setminus W_T(X)$ implies that $c(W_T(X)) = c(W_{\bar{T}}(X))$ for any $X \subseteq V$, and therefore $c(W_T(V(S))) = c(W_{\bar{T}}(V(S)))$. Therefore the modified algorithm selects as the next star exactly the same star as the one of the original variant, and terminates at the same time. Note also that $c(T) = c(W_T(V)) = c(W_{\bar{T}}(V)) = c(\bar{T})$.

Consider now one iteration, in which both variants select the star S_j with center $\tilde{u}$ and arcs (sorted by costs) $\tilde{u}\tilde{v}_i$, for $1 \leq i \leq j$. Recall that $A_j = W_T(V(S_j))$. Let $\bar{A}_j$ be $W_{\bar{T}}(V(S_j))$. As argued before, instead of removing A_j from T and adding the fake edges from S_j (all at once), we execute sequentially, for $i = 1$ to j, the step that adds a fake edge of cost 0 and endpoints $\tilde{u}$ and $\tilde{v}_i$, and removes e_i from T (recall that $e_i = A_i \setminus A_{i-1}$). This step was called one mini-iteration. The modified algorithm proceeds the same way, with the same star S_j except that, with $\bar{e}_i = \bar{A}_i \setminus \bar{A}_{i-1}$, it keeps $\bar{e}_i$ if it is fake, and does not add in this case the fake edge of cost 0 and endpoints $\tilde{u}$ and $\tilde{v}_i$. Below, $T(i)$ denotes the tree in the original algorithm after the i^{th} mini-iteration, and $\bar{T}(i)$ denotes the tree in the modified algorithm after the i^{th} mini-iteration. $T(0)$ and $\bar{T}(0)$ are the trees of the original algorithm and the modified algorithm respectively before the first mini-iteration.

Lemma 3.3.2. *The following holds for every i. For any two vertices u, v, $P^r_{T(i)}(u,v) =$*

$P^r_{\bar{T}(i)}(u,v)$. *If in the* i^{th} *mini-iteration, the original algorithm replaces one edge* e *of* $T(i-1)$ *by an edge* g *(from a star), and* e *is not a fake edge, then the modified algorithm also replaces* e *in* $\bar{T}(i-1)$ *by the edge* g*. And if* e *is fake, then* $\bar{T}(i) = \bar{T}(i-1)$.

Proof. We will prove this lemma by induction, with the base case holding since, for $T(0)$ and $\bar{T}(0)$, this lemma is the induction hypothesis of the second condition.

During mini-iteration i, the original algorithm replaces edge $e = e_i$ of $T(i-1)$ by star edge $g = \{\tilde{u}, \tilde{v}_i\}$ (recall that here $\tilde{u}$ is the center of the star). Also recall that $e_i = A_i \setminus A_{i-1}$ and $\bar{e}_i = \bar{A}_i \setminus \bar{A}_{i-1}$.

We claim that e_i is a fake edge if and only if $\bar{e}_i$ is a fake edge. Indeed, if e_i is not fake, then by the first condition (w.r.t. $T(0)$ and $\bar{T}(0)$, and used with $X = V(S_i)$), using that $A_i = W_{T(0)}(V(S_i))$ and $\bar{A}_i = W_{\bar{T}(0)}(V(S_i))$, we obtain that $e_i \in \bar{A}_i$. Moreover, $e_i \notin \bar{A}_{i-1}$, since otherwise, from the first condition, used for $A_{i-1} = W_{T(0)}(V(S_{i-1}))$ and $\bar{A}_{i-1} = W_{\bar{T}(0)}(V(S_{i-1}))$, we get that $e_i \in A_{i-1}$, contradicting the fact that $e_i = A_i \setminus A_{i-1}$. A symmetric argument works for $\bar{e}_i$ being a fake edge. Moreover, the argument above gives that if at least one of e_i and $\bar{e}_i$ is not a fake edge, then $e_i = \bar{e}_i$.

Thus indeed, if the original algorithm replaces one edge e of $T(i-1)$ by an edge g (from a star) and e is not a fake edge, then $g = \{\tilde{u}, \tilde{v}_i\}$ and the modified algorithm algorithm also replaces e (since $e_i = \bar{e}_i$) with the same edge g. And if the original algorithm replaces one edge e of $T(i-1)$ by an edge g (from a star) and e is a fake edge, then $\bar{e}_i$ is also a fake edge, and as required $\bar{T}(i) = \bar{T}(i-1)$.

Let now u and v be arbitrary vertices. We proceed to prove that $P^r_{T(i)}(u,v) = P^r_{\bar{T}(i)}(u,v)$.

In the first case, e $(= e_i)$ is not a fake edge. In a first subcase, e does not belong to $P_{T(i-1)}(u,v)$. Then $P_{T(i)}(u,v) = P_{T(i-1)}(u,v)$ and therefore $P^r_{T(i)}(u,v) = P^r_{T(i-1)}(u,v)$.

By the induction hypothesis, $P^r_{T(i-1)}(u,v) = P^r_{\bar{T}(i-1)}(u,v)$, and therefore e, being non-fake, does not belong to $P_{\bar{T}(i-1)}(u,v)$. Thus $P_{\bar{T}(i)}(u,v) = P_{\bar{T}(i-1)}(u,v)$ and therefore $P^r_{\bar{T}(i)}(u,v) = P^r_{\bar{T}(i-1)}(u,v)$. We conclude that $P^r_{\bar{T}(i)}(u,v) = P^r_{T(i)}(u,v)$.

In a second subcase, e belongs to $P_{T(i-1)}(u,v)$. In this case $P_{T(i)}(u,v)$ uses g. We rename the endpoints of g to be called $\hat{u}$ and $\hat{v}$ such that $P_{T(i-1)}(\hat{u},\hat{v})$ uses e in the same direction as $P_{T(i-1)}(u,v)$. Then $P_{T(i)}(u,v)$ consists of $P_{T(i-1)}(u,\hat{u})$, concatenated with the edge g, followed by $P_{T(i-1)}(\hat{v},v)$. Also, with g being a fake edge, $P^r_{T(i)}(u,v)$ consists of $P^r_{T(i-1)}(u,\hat{u})$ concatenated with $P^r_{T(i-1)}(\hat{v},v)$. By the induction hypothesis, $P^r_{T(i-1)}(u,v) = P^r_{\bar{T}(i-1)}(u,v)$, and therefore, e being non-fake, belongs to $P_{\bar{T}(i-1)}(u,v)$ and appears in the same direction as on $P_{T(i)}(u,v)$. Also by the induction hypothesis, $P^r_{T(i-1)}(\hat{u},\hat{v}) = P^r_{\bar{T}(i-1)}(\hat{u},\hat{v})$, and therefore e, being non-fake, belongs to $P_{\bar{T}(i-1)}(\hat{u},\hat{v})$ and appears in the same direction as on $P_{T(i-1)}(\hat{u},\hat{v})$. Thus e appears in the same direction on $P_{\bar{T}(i-1)}(u,v)$ and $P_{\bar{T}(i-1)}(\hat{u},\hat{v})$. This means that $P_{\bar{T}(i)}(u,v)$ consists of $P_{\bar{T}(i-1)}(u,\hat{u})$, concatenated with the edge g, followed by $P_{\bar{T}(i-1)}(\hat{v},v)$. Then, $P^r_{\bar{T}(i)}(u,v)$ consists of $P^r_{\bar{T}(i-1)}(u,\hat{u})$ concatenated with $P^r_{\bar{T}(i-1)}(\hat{v},v)$, and thus it equals $P^r_{T(i)}(u,v)$ (here we used the induction hypothesis: $P^r_{\bar{T}(i-1)}(u,\hat{u}) = P^r_{T(i-1)}(u,\hat{u})$ and $P^r_{\bar{T}(i-1)}(\hat{v},v) = P^r_{T(i-1)}(\hat{v},v)$).

In the second case, e is a fake edge. In a first subcase, e does not belong to $P_{T(i-1)}(u,v)$. Then $P_{T(i)}(u,v) = P_{T(i-1)}(u,v)$ and therefore $P^r_{T(i)}(u,v) = P^r_{T(i-1)}(u,v)$. Moreover, as $\bar{T}(i) = \bar{T}(i-1)$, we have that $P^r_{\bar{T}(i)}(u,v) = P^r_{\bar{T}(i-1)}(u,v)$, and since by the induction hypothesis, $P^r_{T(i-1)}(u,v) = P^r_{\bar{T}(i-1)}(u,v)$, we conclude that $P^r_{T(i)}(u,v) = P^r_{\bar{T}(i)}(u,v)$.

In a second subcase, e belongs to $P_{T(i-1)}(u,v)$. As e is fake, Lemma 3.3.1 implies that $P^r_{T(i-1)}(\tilde{u},\tilde{v}_i)$ is an empty list (recall that $\tilde{u}$ and $\tilde{v}_i$ are the endpoints of g). In this subcase $P_{T(i)}(u,v)$ uses g, and in fact all the edges in $P_{T(i)}(u,v) \setminus \{g\}$ but not in $P_{T(i-1)}(u,v)$ belong to the $P_{T(i-1)}(\tilde{u},\tilde{v}_i)$, a fake path. Thus in this subcase, we also have that $P^r_{T(i)}(u,v) = P^r_{T(i-1)}(u,v)$. The arguments from the previous paragraph, using

$\bar{T}(i) = \bar{T}(i-1)$, apply and allow us to conclude that $P^r_{T(i)}(u,v) = P^r_{\bar{T}(i)}(u,v)$.

This completes the induction step of Lemma 3.3.2. □

Note that the above lemma immediately implies that, for any fake edge $\{x,y\} \in E(T(i))$, $P_{\bar{T}(i)}(x,y)$ is a fake path. Similarly, for any fake edge $\{x',y'\} \in E(\bar{T}(i))$, $P_{T(i)}(x',y')$ is a fake path. Also, Lemma 3.3.2 used with $i = j$ gives that the second condition is maintained after the iteration in which star S_j is selected. That the first condition is maintained after this iteration is proven below. We use T and $\bar{T}$ in this lemma to denote the trees after this iteration of the original and the modified algorithm respectively.

Lemma 3.3.3. *For any $X \subseteq V$, $W_T(X) \setminus W_{\bar{T}}(X)$ and $W_{\bar{T}}(X) \setminus W_T(X)$ are either sets consisting only of fake edges, or empty.*

Proof. Let $j = |X| - 1$, thus chosen such that $|W_T(X)| = |W_{\bar{T}}(X)| = j$. Let B_j be the swap set of X in $T(i)$, and $\bar{B}_j$ be the swap set of X in $\bar{T}(i)$. Let us build B and $\bar{B}$ by the first procedure of Claim 3.3.1, using B and $\bar{B}$ respectively as the variable during the procedure. Start with $l = 1$, and, as long as possible, pick two vertices $u_l, v_l \in X$ such that, besides being in the same connected component of $(V, E(T) \setminus B)$, we also have that $P^r_T(u_l, v_l)$ is non-empty. Add the largest-cost edge of $P^r_T(u_l, v_l)$ to B and increment l. Use the same u_l, v_l for constructing $\bar{B}$, which is always possible using the following argument.

We claim the property that $B = \bar{B}$ and that u_l, v_l are in the same connected component of $(V, E(\bar{T}) \setminus \bar{B})$.

This claim is proven by induction on l. Initially, $B = \bar{B} = \emptyset$, and u_1, v_1 are connected in the tree $\bar{T}$. Assume now that the property holds for l. With u_l, v_l in the same same connected component of $(V, E(\bar{T}) \setminus \bar{B})$, and $P^r_T(u_l, v_l) = P^r_{\bar{T}}(u_l, v_l)$, we obtain that $B = \bar{B}$ after the addition of the same edge. Moreover, if u_{l+1}, v_{l+1} are such that u_{l+1}, v_{l+1} are in the same connected component of $(V, E(T) \setminus B)$ and $P^r_T(u_{l+1}, v_{l+1})$ is non-

empty, then $P^r_{\bar{T}}(u_{l+1}, v_{l+1})$ is also non-empty and disjoint from $\bar{B}$ (since $P^r_T(u_{l+1}, v_{l+1}) = P^r_{\bar{T}}(u_{l+1}, v_{l+1})$ and $B = \bar{B}$). As all the edges of $P_{\bar{T}}(u_{l+1}, v_{l+1})$ are either fake or in $P^r_{\bar{T}}(u_{l+1}, v_{l+1})$, and no fake edge is in B, we do obtain that u_{l+1}, v_{l+1} are in the same connected component of $(V, E(\bar{T}) \setminus \bar{B})$.

When such $u_l, v_l \in X$ do not exist, one can also check, by an argument symmetric to the one above, that also there are no $u', v' \in X$ such that they are the same connected component of $(V, E(\bar{T}) \setminus \bar{B})$ and with $P^r_{\bar{T}}(u', v')$ non-empty.

At this moment, the procedure has put in B and respectively in $\bar{B}$ all the non-fake edges that will ever be put in B and $\bar{B}$. The lemma follows, as $B = \bar{B}$ before the addition of fake edges. □

We next obtain the last lemma as part of our arguments that not replacing fake edges by fake edges keeps the same approximation ratio. Please refer to Lemma 3.3.1 and the discussion proceeding it, and the paragraph before Lemma 3.3.2.

Lemma 3.3.4. *For all $i = 1, \ldots, j$, if $\bar{e}_i$ is not a fake edge, then it becomes the largest-cost edge on the unique simple path $P_{\bar{T}(i-1)}(\tilde{u}, \tilde{v}_i)$, before it is removed and replaced by fake edge $\tilde{u}\tilde{v}_i$.*

Proof. From the proof of Lemma 3.3.2, with $\bar{e}_i$ not fake, we have $\bar{e}_i = e_i$. From Lemma 3.3.1, e_i is the largest-cost edge on $P_{T(i-1)}(\tilde{u}, \tilde{v}_i)$. As e_i is not fake, e_i belongs to $P^r_{T(i-1)}(\tilde{u}, \tilde{v}_i)$, and by Lemma 3.3.2, $P^r_{T(i-1)}(\tilde{u}, \tilde{v}_i) = P^r_{\bar{T}(i-1)}(\tilde{u}, \tilde{v}_i)$. This makes e_i the largest-cost edge on the unique simple path $P_{\bar{T}(i-1)}(\tilde{u}, \tilde{v}_i)$ and, again by Lemma 3.3.2, e_i is removed from $\bar{T}(i-1)$ and replaced by $\tilde{u}\tilde{v}_i$. □

Based on Lemma 3.3.3 and the discussion proceeding Lemma 3.3.2, the modified variant chooses the same stars as the original algorithm [19]. Also in their (original) approximation-ratio analysis, the power of the output is bounded by twice the power of the

selected stars plus the cost of the remaining "true" edges of the tree. We have exactly the same quantity for this bound, since the modified variant selects the same stars and the final tree of the modified variant differs from the final tree of the original variant only in fake edges (since $W_T(V) = W_{\bar{T}}(V)$). Thus our modified variant has the same approximation ratio of 4.2 in the two-dimensional Euclidean input model.

3.3.2 The data structures used. Since we do not remove any fake edges, there are in total at most $n-1$ times when we change the tree. There are also at most $n-1$ stars that are added by the algorithm (each must introduce at least one fake edge), and thus in total at most $(n-1)^2$ times when some fake edge is considered (even if not added). One also has to construct the data structures (the arrays, one for each u, with e_j in position j) for the initial tree T. This can be easily accomplished in time $O(mn)$ using ideas developed later for maintaining the data structures. We omit the details; note that we allocate much more time to this initialization than the $O(m)$ allocated to an update - and indeed a running time of $O(m \lg n)$ in total seems possible using the linking-and-cutting trees data structure of Sleator and Tarjan (based on Splay Trees; see [60]).

Based on this discussion, once we establish how to maintain our data structures in time $O(m)$ whenever an edge of T is replaced by a fake edge, we obtain a $O(mn)$ running time for this new variant of the Relative-Greedy algorithm. Thus from now we concentrate on the following "main" operation: given tree T, edge xx' of T and fake edge yy' of cost 0, recompute the data structures for the tree T' obtained from T by deleting xx' and adding yy'. One can check that, for one tree, these data occupies $O(m+n)$ space. We also note that once the data structures for T' are computed, we have no need to keep the data of T. Thus the overall space requirement of the modified variant is $O(m+n)$.

For fixed center u, as above, we have A_j as the swap-set of $V(S_j)$ before T is transformed in T'. We are computing A'_j, the swap set of S_j in T'. Note that x, x', y, y' and T are not related to u and j. We know from Claim 3.3.1 and the discussion following it

that A'_{j+1} has exactly one more edge than A'_j , and we aim to identify this edge, which we called e'_{j+1}. Recall that v_j is the only vertex in $V(S_j) \setminus V(S_{j-1})$. Let Q be the connected component of $(V, E(T) - xx')$ that contains x and let Q' be the connected component of $(V, E(T) - xx')$ that contains x' . Rename y, y', the endpoints of the fake edge, such that $V(Q)$ contains y (and $V(Q')$ contains y'). Note that Q is also the connected component of $(V, E(T') - yy')$ that contains x.

Claim 3.3.2. $|A'_j \setminus A_j| \leq 1$.

Proof. Use the first procedure (second point) of Claim 3.3.1 to construct A_j (as B_j in the claim's statement) using only pairs of vertices of $V(S_j)$ that are both in $V(Q)$, or both in $V(Q')$. We select this way $|V(S_j) \cap V(Q)| - 1$ edges for pairs of vertices of $V(S_j) \cap V(Q)$, unless $V(S_j) \cap V(Q) = \emptyset$, in which case no such edge is selected. Similarly, we select this way $|V(S_j) \cap V(Q')| - 1$ edges for pairs of vertices of $V(S_j) \cap V(Q')$, unless $V(S_j) \cap V(Q') = \emptyset$, in which case no such edge is selected. In total, we select either $|V(S_j)| - 2$ or $|V(S_j)| - 1$ edges.

We select exactly the same edges constructing A'_j using the same procedure. Thus $|A_j \cap A'_j| \geq |V(S_j)| - 2$, and with $|A_j| = |A'_j| = |V(S_j)| - 1$, the claim follows. □

Both A'_j and A_{j+1} are used when computing e'_{j+1}. In fact, we do not access the full A'_j and A_{j+1}, but only the (at most three) elements of $A_{j+1} \setminus A'_j$ and $A'_j \setminus A_{j+1}$. If A'_j would be removed from T', there would be exactly one node $v \in V(S_j)$ that is connected to y and y' in T'; this vertex is useful and the algorithm calls it v_c. However, if the edge yy' is selected in A'_j, v_c is not used by the algorithm any more (since v_c is only used in cases I through V, and if yy' is selected in A'_j, then f' will always be yy' and we cannot have $f' = f$ or $f' = e_{j+1}$ anymore). and the fact that it is not correctly defined does not matter. Moreover, the tree T' is pre-processed (as described later, in time $O(n)$) such that, for any v, v', with $v \in V(Q)$ and $v' \in V(Q')$, the maximum cost of an edge on the unique

simple path $P_{T'}(v, v')$ can be found in constant time. Then one can obtain e'_{j+1} in constant time, as shown in Algorithm 1 (with correctness in Proposition 3.3.1). In the algorithm, the input consists of v_c, and tree edges f and f' whose meaning is as follows: if $A'_j = A_j$, then $f = f'$ is some arbitrary edge in A_j; otherwise $f = A_j \setminus A'_j$ and $f' = A'_j \setminus f$. The algorithm also makes use of the following queries, which are all answered in constant time (with a precomputation that takes time $O(n)$, and which we also describe below).

1. $max\_to\_y\_root(v)$ returns, for a vertex v, the largest-cost edge of T' on the unique simple path $P_{T'}(v, y)$. If $v = y$, it returns NULL, which has, for convenience, negative cost. This quantity is stored in an array of size n which is computed in a preorder traversal of T' when rooted at y (if we process node v, a child of v', then $max\_to\_y\_root(v)$ is either $max\_to\_y\_root(v')$ or the edge vv'). A look-up in the table takes constant time.

2. $max\_T'(v, v')$ returns, for a vertices v and v' such that one of them is in $V(Q)$ and the other in $V(Q')$, the largest-cost edge of T' on the unique simple path $P_{T'}(v, v')$. It is done in constant time by calling $max\_to\_y\_root(v)$ and $max\_to\_y\_root(v')$ and comparing the results.

3. $max\_to\_x\_root(v)$ returns, for a vertex v, the largest-cost edge of T on the unique simple path $P_T(v, x)$. If $v = x$, it returns NULL, which has, for convenience, negative cost.

4. $v\_descendant\_of\_e$, for input vertex v and edge e, tests if e disconnects v from y in T'. For this, T' is preprocessed, in time $O(n)$, to answer in constant time Least Common Ancestor queries (see [41, 59]).

Proposition 3.3.1. *The procedure from Algorithm 1 correctly computes A'_j for every j.*

Computes e'_{j+1} and updates f, f', v_c.
if $f = f'$ **then**
  if $v_c, v_{j+1} \in V(Q)$, *or* $v_c, v_{j+1} \in V(Q')$ **then**
    $e'_{j+1} \leftarrow e_{j+1}$ // Case I
  end
  else
    if $xx' = e_{j+1}$ **then**
      $g \leftarrow max\_T'(v_c, v_{j+1})$
      $e'_{j+1} \leftarrow g$ // Case II
    end
    else
      $e'_{j+1} \leftarrow e_{j+1}$ // Case III
    end
    $f' \leftarrow e'_{j+1}$
    $f \leftarrow e_{j+1}$
  end
end
else if $f' = e_{j+1}$ **then**
  if $v_c, v_{j+1} \in V(Q)$, *or* $v_c, v_{j+1} \in V(Q')$ **then**
    $e'_{j+1} \leftarrow f$
    $f' = f$ // Case IV
  end
  else
    $g \leftarrow max\_T'(v_c, v_{j+1})$
    $e'_{j+1} \leftarrow g$
    $f' \leftarrow g$ // Case V
  end
end
else if f *is not on the same side as* v_{j+1} **then**
  $e'_{j+1} \leftarrow e_{j+1}$ // Case VI
end
else if e_{j+1} *is not on the same side as* v_{j+1} **then**
  $e'_{j+1} \leftarrow f$
  $f \leftarrow e_{j+1}$ // Case VII
end
else
  if e_{j+1} *is the largest-cost edge from* v_{j+1} *to* x **then**
    $e'_{j+1} \leftarrow \max(e_{j+1}, f)$
    $f \leftarrow \min(e_{j+1}, f)$ // Case VIII
  end
  else
    $e'_{j+1} = e_{j+1}$ // Case IX
  end
end
if $v_c\_descendant\_of\_e'_{j+1}$ **then**
  $v_c = v_{j+1}$
end

Algorithm 1: Filling up one entry in the data structure for T'

Proof. We proceed case by case and discuss the correctness of computing f, f', e'_{j+1}. The update of v_c that is done at the end of the algorithm is addressed last in this proof.

In Cases I, II, and III, $f = f'$ (signifying $A_j = A'_j$). Consider trying to find A'_{j+1} from A'_j using the second procedure from Claim 3.3.1 ($A'_j = B_j$ in the claim's statement, also T' is used instead of T). Then v_{j+1} is in the same component with some $v_{i'}$, for $0 \leq i' \leq j$, and e_{j+1} is the largest-cost edge on $P_T(v_{i'}, v_{j+1})$. The algorithm is not aware of which i' is the correct one! (we cannot figure out a method that finds this i' in constant time). If one of $v_{i'}, v_{j+1}$ is in $V(Q)$ and the other is in $V(Q')$, then the unique simple path $P_{T'}(v_{i'}, v_{j+1})$ goes through the edge yy'; as v_c is the only vertex among $\{v_0, \ldots, v_j\}$ that is connected to y in $(V, E(T') \setminus A'_j)$, we must have $v_{i'} = v_c$, and thus we are not in Case I. Thus in Case I we must have that $v_{i'}$ and v_{j+1} are both in $V(Q)$, or in a symmetric situation, both in $V(Q')$ (the argument is however the same regardless if $v_{i'} = v_c$ or not, see subcases 1.1 and 1.2 in Figure 3.2). Then $P_{T'}(v_{i'}, v_{j+1})$ is the same as $P_T(v_{i'}, v_{j+1})$, and we have indeed $e'_{j+1} = e_{j+1}$ (as the largest-cost edge on this path), and moreover $A'_{j+1} = A_{j+1}$, and thus it is correct that we do not update f, f'. The algorithm is correct in Case I.

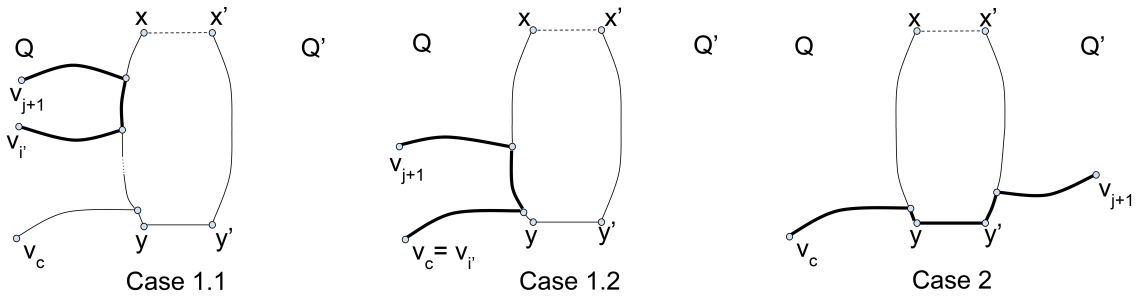

Figure 3.2. In Case I, $f = f'$, $v_{i'}$ and v_{j+1} are both in $V(Q)$ or both in $V(Q')$. Represented here is only when both are in $V(Q)$. Subcase 1.1: $v_c \neq v_{i'}$ (then $v_{i'}$ is not connected to v_c in $(V, E(T') \setminus A'_j)$). Subcase 1.2: $v_c = v_{i'}$. In all our figures, we use thick curves to represent paths in T' where e_{j+1} must belong. Dotted curves have edges that belong to A'_j. In both subcases, e'_{j+1} is on the path from $v_{i'}$ to v_{j+1}, and thus $e'_{j+1} = e_{j+1}$. Case 2: $f = f'$, $v_c \in V(Q)$, $v_{j+1} \in V(Q')$ and $xx' = e_{j+1}$. e'_{j+1} ($e'_{j+1} = g$) is on the path from v_c to v_{j+1}.

Further consider Case II. Assume by symmetry that $v_c \in V(Q)$, and thus $v_{j+1} \in$

$V(Q')$ (See Case 2 in Figure 3.2). Consider trying to find A_{j+1} from A_j using the second procedure from Claim 3.3.1 ($A_j = B_j$ in the claim's statement). That this procedure finds xx' means that, in $(V, E(T) \setminus A_j)$, the component of v_{j+1} has, as vertex in $V(S_j)$, some vertex not in $V(Q')$ (otherwise, the unique simple path from v_{j+1} to this vertex of $V(S_j)$ will stay inside Q'). With v_{j+1} already separated by A_j from all the vertices of $V(Q') \cap V(S_j)$, it must be the case that, in $(V, E(T') \setminus A_j)$, v_{j+1} must be in the same connected component with y and thus with v_c. Note that also that $A'_j = A_j$ in this case, and thus the second procedure from Claim 3.3.1, when trying to find e'_{j+1}, will indeed find g (from pseudocode). Thus the algorithm correctly computes $e'_{j+1} = g$. Setting $f' = e'_{j+1}$ and $f = e_{j+1}$ is correct in Cases II and III whether $e'_{j+1} = e_{j+1}$ or not.

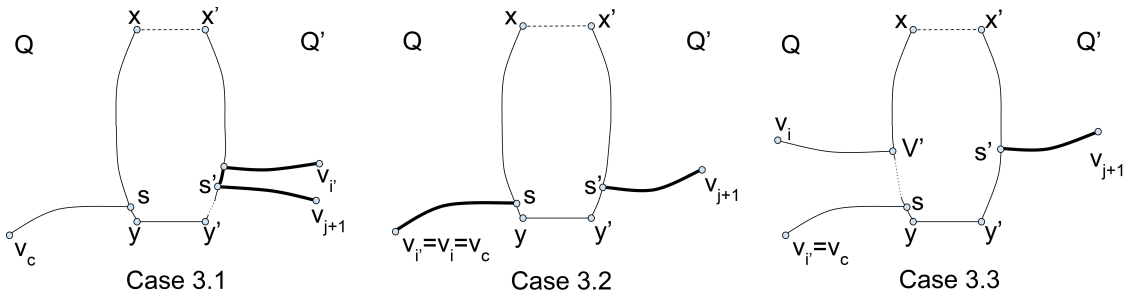

Figure 3.3. In Case III, $f = f'$, $v_c \in V(Q)$, $v_{j+1} \in V(Q')$ and $e_{j+1} \neq xx'$. In this case, the algorithm sets e'_{j+1} to e_{j+1}. Subcase 3.1: $v_{i'} \in V(Q')$ (it follows that v_{j+1} and y' are not connected in $(V, E(T') \setminus A'_j)$, as $v_{i'} \neq v_c$ and v_c is connected to y' in $(V, E(T') \setminus A'_j)$). e'_{j+1} is on the path from $v_{i'}$ to v_{j+1}. Subcase 3.2: $v_c = v_i = v_{i'}$. e'_{j+1} is on the path from v_c to s or v_{j+1} to s'. Subcase 3.3: $v_i \neq v_c$ (it follows that v_c is not connected to v_i in $(V, E(T') \setminus A'_j)$). e'_{j+1} is on the path from s' to v_{j+1}.

Case III is the hardest so far; recall that we have that $A'_j = A_j$ and $xx' \neq e_{j+1}$, and assume by symmetry that $v_c \in V(Q)$ and $v_{j+1} \in V(Q')$ (As in Figure 3.3). The algorithm "magically" sets $e'_{j+1} = e_{j+1}$, and we must verify that this is correct. As above, consider trying to find A'_{j+1} from A'_j using the second procedure from Claim 3.3.1. Then v_{j+1} is in the same component of $(V, E(T') \setminus A'_j)$ with some $v_{i'}$, for $0 \leq i' \leq j$. The unique simple path $P_{T'}(v_{j+1}, v_{i'})$ contains no edge of $A'_j = A_j$. If $v_{i'} \in V(Q')$ (As Subcase 3.1 in Figure 3.3), then the unique simple path $P_{T'}(v_{j+1}, v_{i'})$ is also the unique simple path $P_T(v_{j+1}, v_{i'})$,

and, since this path has no edge of A_j, we must have that e_{j+1} is the largest-cost edge on this path. Using Claim 3.3.1, setting $e'_{j+1} = e_{j+1}$ is correct.

Assume from now on that we are in Case III and $v_{i'} \notin V(Q')$. Thus, as one has to use edge yy' to cross from Q' to Q, we have that $v_{i'} = v_c$. The largest-cost edge on this path $P' = P_{T'}(v_c, v_{j+1})$ is g, as computed in Case II. Since we are however in Case III, and the algorithm sets $e'_{j+1} = e_{j+1}$, we need to prove that $g = e_{j+1}$ (for which we use that $e_{j+1} \neq xx'$). Consider trying to find A_{j+1} from A_j using the second procedure from Claim 3.3.1. Then v_{j+1} is in the same component of $(V, E(T) \setminus A_j)$ with some v_i, for $0 \leq i \leq j$. Let P be the unique simple path $P_T(v_i, v_{j+1})$. If $v_i \in V(Q')$, then v_i and v_c are in the same component in $(V, E(T) \setminus A_j)$, as both v_i and v_c are connected to v_{j+1} in $(V, E(T) \setminus A_j)$. This contradicts the definition of A_j. Therefore $v_i \in V(Q)$ and thus P uses the edge xx', and e_{j+1} is the largest-cost edge on P. Define $s \in V$ to be the first vertex as we travel on P' from v_c to v_{j+1} with s on the unique simple path $P_T(x, y)$. Define $s' \in V$ to be the first vertex as we travel on P' from v_{j+1} to v_c with s' on the unique simple path $P_T(x', y')$.

If $v_i = v_c$ (See Subcase 3.2 in Figure 3.3), we argue as follows. If e_{j+1} belongs to the subpath of P' from v_c to s, then e_{j+1} has cost larger than the edges on the subpath of P' from v_{j+1} to s', and e_{j+1} has cost larger than xx'. By Lemma 3.3.4, every edge on P' from s to s' has cost smaller than xx', and thus smaller than e_{j+1}. We conclude that if e_{j+1} belongs to the subpath of P' from v_c to s, then e_{j+1} has the largest-cost among the edges of P', and thus setting $e'_{j+1} = e_{j+1}$ is correct. If e_{j+1} belongs to the subpath of P' from s to s', then e_{j+1} must be xx' (by Lemma 3.3.4), which cannot happen in Case III. If e_{j+1} belongs to the subpath of P' from v_{j+1} to s', then e_{j+1} has cost larger than the edges on the subpath of P' from v_c to s, and e_{j+1} has cost larger than xx'. By Lemma 3.3.4, every edge on P' from s to s' has cost smaller than xx', and thus smaller than e_{j+1}. We conclude that if e_{j+1} belongs to the subpath of P' from v_{j+1} to s', then e_{j+1} has the largest-cost among the edges of P', and thus setting $e'_{j+1} = e_{j+1}$ is correct.

If $v_i \neq v_c$ (See Subcase 3.3 in Figure 3.3), we argue as follows. Define $v' \in V$ to be the first vertex as we travel on $P = P_T(v_i, v_{j+1})$ with v' on the unique simple path $P_T(x, y)$. If e_{j+1} belongs to the subpath of P from v' to s', then e_{j+1} must be xx' (by Lemma 3.3.4), which cannot happen in Case III. If e_{j+1} belongs to the subpath of P' from v_{j+1} to s', then e_{j+1} has cost larger than xx', and by Lemma 3.3.4, every edge on P' from s to s' has cost smaller than xx', and thus smaller than e_{j+1}. Let h be the edge of $A_j = A'_j$ which is first on the simple path $P_T(s, x)$. If $x = s$, then v_i and v_c are in the same component in $(V, E(T) \setminus A_j)$. This can not happen and $x \neq s$. So h must exist, by the same argument (in Subcase 3.3 in Figure 3.3, h belongs to the dotted path). Let h' be the largest-cost edge on the subpath of P' from v_c to s. We must have that h' has cost less than h, or else removing h from A_j allows a simple path P'' from v_c to some other vertex of $Q \cap V(S_j)$, and h' could be used instead of h in A_j. Moreover, by Lemma 3.3.4, the cost of xx' exceeds the cost of h. Combined with e_{j+1} having cost larger than xx', we obtain that e_{j+1} has cost larger than all the edges of the subpath of P' from v_c to s. We conclude that e_{j+1} has the largest-cost among the edges of P', and thus setting $e'_{j+1} = e_{j+1}$ is correct.

In a last subcase of Case III (we already reached $v_i \neq v_c$), we show that e_{j+1} cannot belong to the subpath of P from v_i to v' (and thus, in Subcase 3.3 in Figure 3.3, the thick curve correctly represents the only path where e_{j+1} can be). Let h be the edge of $A_j = A'_j$ which is first on the simple path $P_T(v', y)$; h must exist since $v_i \neq v_c$. We must have that e_{j+1} has cost less than h, or else removing h from A_j allows a simple path P'' from v_i to some other vertex of $Q \cap V(S_j)$, and e_{j+1} could be used instead of h in A_j. Moreover, by Lemma 3.3.4, the cost of xx' exceeds the cost of h. Thus e_{j+1} has cost less than xx', contradicting that we are in Case III.

We proceed to Cases IV and V, where $f' = e_{j+1}$ (and therefore $A'_j \neq A_j$). Construct A'_j as in Claim 3.3.2: use the first procedure (second point) of Claim 3.3.1 using only pairs of vertices of $V(S_j)$ that are both in $V(Q)$, or both in $V(Q')$. All the edges selected are in

both A_j and A'_j. Doing this should not completely construct A'_j since $A'_j \neq A_j$. One last simple path P' with starting vertex in $V(Q) \cap V(S_j)$ and ending at a vertex in $V(Q') \cap V(S_j)$ is found, and f' is the largest-cost edge of T' on this path; as $f' = e_{j+1}$, we have $f' \neq yy'$.

Assume by symmetry that $v_c \in V(Q)$; then P' started at v_c and ended at some other vertex $v_i \in V(Q') \cap V(S_j)$; moreover $f' \in E(Q')$ (since v_c remains connected to y). Let v' be the endpoint of f' closest to v_i (some possibilities are illustrated in the subcases of figures 3.4 and 3.5); we argue below that f cannot be on the subpath of P' from v' to v_i. Define $s' \in V$ to be the first vertex as we travel on P' from v_i to v_c with s' on the unique simple path $P_T(x', y')$. Like f', f must disconnect a vertex in $V(Q) \cap V(S_j)$ from a vertex in $V(Q') \cap V(S_j)$ in $(V, E(T) \setminus (A_j - f))$. If f were to be on the subpath of P' from v' to v_i and an edge $\bar{g}$ of A_j can disconnect the endpoints of f from v_i in $(V, E(T) \setminus (A_j - \bar{g}))$, then $\bar{g}$ will also exist in A'_j and disconnect v_c from v_i in $(V, E(T') \setminus (A'_j - \bar{g}))$. That is a contradiction. So the vertex in $V(Q') \cap V(S_j)$ that is disconnected from a vertex of $V(Q) \cap V(S_j)$ by f in $(V, E(T) \setminus (A_j - f))$ must be v_i. Thus f must be the largest-cost edge from x to v_i (any path in T from a vertex in $V(Q)$ to v_i must pass through x), and in particular the cost of f exceeds the cost of xx'. If f' is on the subpath of P' from s' to v_i, then we should have used f instead of f', or vice versa. If f' is not on this subpath, then f' is on the simple path $P_T(x', y')$, and therefore, by Lemma 3.3.4, we have that f' has cost less than xx'. Thus f has cost larger than f', contradicting the choice of f'.

Thus f is not on the simple path $P_T(v', v_i)$ of Q'. Moreover, no edge in A_j is on the simple path $P_T(v', v_i)$ of Q'. If $e_{j+1}(= f')$ disconnects v_{j+1} from v_k in $(V, E(T) \setminus A_j)$ where $v_k \neq v_i$, then either v_k or v_{j+1} is connected to v_i in $(V, E(T) \setminus A_{j+1})$. Therefore $f'(= e_{j+1})$ disconnects v_{j+1} from v_i in $(V, E(T) \setminus A_j)$. In Case V (depicted in Figure 3.4), we also must have $v_{j+1} \in V(Q')$. We have that v_{j+1} is still connected to y' in $(V, E(T') \setminus A'_j)$, since as argued above, in $(V, E(T) \setminus A_j)$, v_{j+1} is connected to the endpoint of f' other than v', and this other endpoint of f' is connected to y' in $(V, E(T') \setminus A'_j)$ by the choice of v_i being

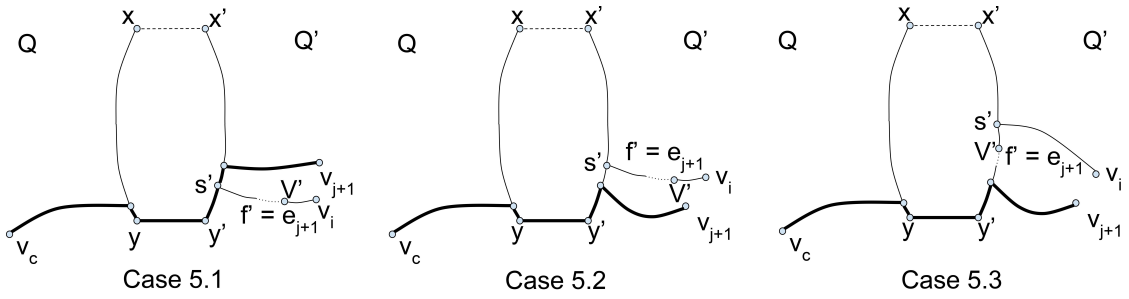

Figure 3.4. In Case V, $f \neq f'$, $f' = e_{j+1}$, $v_c \in V(Q)$, $v_{j+1} \in V(Q')$. There are three subcases depicted, based on where e_{j+1} is on the path from v_{j+1} to v_i. In both Subcase 5.1 and Subcase 5.2, f' is on the path from s' to v_i. In Subcase 5.3, f' is on P' but not on the path from v_c to v_{j+1} or s' to v_i. In all subcases, the same argument in the text gives that e'_{j+1} $(= g)$ is on the path from v_c to v_{j+1}.

such that f' is the only edge of A'_j on the path $P' = P_{T'}(v_c, v_i)$. Therefore v_{j+1} is in the same component of $(V, E(T') \setminus A'_j)$ as v_c; g as computed by algorithm is the largest-cost edge on the simple path $P_{T'}(v_c, v_{j+1})$, and therefore e'_{j+1} is correctly set to be g by the algorithm.

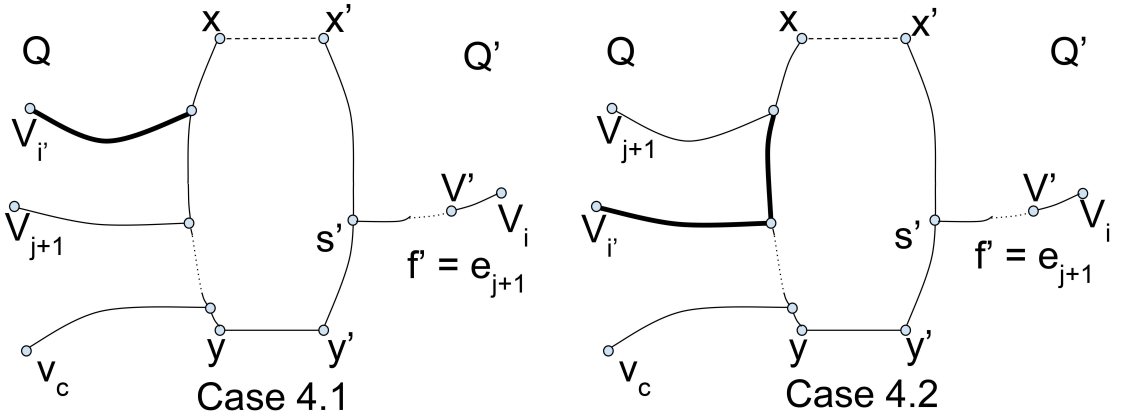

Figure 3.5. In Case IV, $f \neq f'$, $f' = e_{j+1}$. $v_c, v_{j+1} \in V(Q)$ or $v_c, v_{j+1} \in V(Q')$. Only the former case is depicted. e'_{j+1} is set by the algorithm to f (as proven below, f disconnects $v_{i'}$ from v_{j+1} in $(V, E(T) \setminus A'_j)$).

In Case IV (depicted in Figure 3.5), $v_{j+1} \in V(Q)$; note that $e_{j+1} = f'$ disconnects v_{j+1} from v_i in $(V, E(T) \setminus A_j)$, since e_{j+1} must disconnect v_{j+1} from some vertex of $V(S_j)$ in $(V, E(T) \setminus A_j)$, and v', the endpoint of f' closest to v_i, is connected to v_i in $(V, E(T) \setminus A_j)$ (as argued earlier). This means that v_i and x are connected in $(V, E(T) \setminus A_j)$. However,

when constructing A_j as in Claim 3.3.2, with the first procedure (second point) of Claim 3.3.1 using only pairs of vertices of $V(S_j)$ that are both in $V(Q)$, or both in $V(Q')$, we first obtain all the edges of $A'_j \cap A_j$, and the last edge chosen is f. Thus f disconnects v_i from some $v_{i'}$ (with $0 \leq i' \leq j$) in $(V, E(T) \setminus (A_j - f))$. Also, $v_{i'} \in V(Q)$ since $v_i \in V(Q')$, and v_i is still connected to x after removing A_j from $E(T)$. We have that both $v_{i'}$ and v_{j+1} are connected to x in $(V, E(T) \setminus (A_j - f))$. With e_{j+1} having both endpoints in Q', we obtain that f is the only edge of A_{j+1} on $P_T(v_{i'}, v_{j+1})$; by Claim 3.3.1, f is also the largest-cost edge on this path. Also, with $f' = e_{j+1}$ having both endpoints in Q', we obtain that no edge of A'_j is on $P_{T'}(v_{i'}, v_{j+1}) = P_T(v_{i'}, v_{j+1})$. Since A'_{j+1} must contain an edge on $P_{T'}(v_{i'}, v_{j+1})$, using Claim 3.3.1, we obtain that this edge must be f and thus $e'_{j+1} = f$.

After excluding Cases I-V, we have that $A_{j+1} \setminus A'_j = \{f, e_{j+1}\}$, and we know from Claim 3.3.2 that e'_{j+1} is either f or e_{j+1}. Note that in both cases, after the algorithm, f should be the other of f, e_{j+1}. If we run the first procedure from Claim 3.3.1 to obtain A_j, first separating pairs of vertices both in $V(Q) \cap V(S_j)$ or both in $V(Q') \cap V(S_j)$, we get all the edges of $A'_j - f' = A_j - f$. Then f, which is not in A'_j, must be taken by being the only edge of A_j on the path from $P_T(v_i, v_{i'})$ with $v_i \in V(Q)$ and $v_{i'} \in V(Q')$.

Consider now Case VI (depicted in Figure 3.6). Assume by symmetry that $v_{j+1} \in V(Q)$, and therefore f does not have both endpoints in $V(Q)$. Then x is in the same component of $(V(Q), E(Q) \setminus A_j)$ as v_i. If e_{j+1} disconnects v_{j+1} from some vertex of $V(Q')$ in $(V, E(T) \setminus A_j)$, then v_{j+1} is connected to v_i in $(V, E(T) \setminus A_j)$ since both v_i and v_{j+1} are connected to x in $(V, E(T) \setminus A_j)$. In this case, e_{j+1} also disconnects v_{j+1} from v_i in $(V, E(T) \setminus A_j)$; this contradicts the fact (obtained from from the second procedure of Claim 3.3.1) that e_{j+1} disconnects v_{j+1} from exactly one vertex of $V(S_j)$ in $(V, E(T) \setminus A_j)$. Therefore, e_{j+1} disconnects v_{j+1} from a vertex $v \in V(Q) \cap V(S_j)$ in $(V, E(T) \setminus A_j)$. From Claim 3.3.1, e_{j+1} is the largest-cost edge on $P_T(v_{j+1}, v)$. With $v \in V(Q) \cap V(S_j)$, $P_T(v_{j+1}, v) = P_{T'}(v_{j+1}, v)$, and moreover, since we are in Case VI, we have $e_{j+1} \neq f'$.

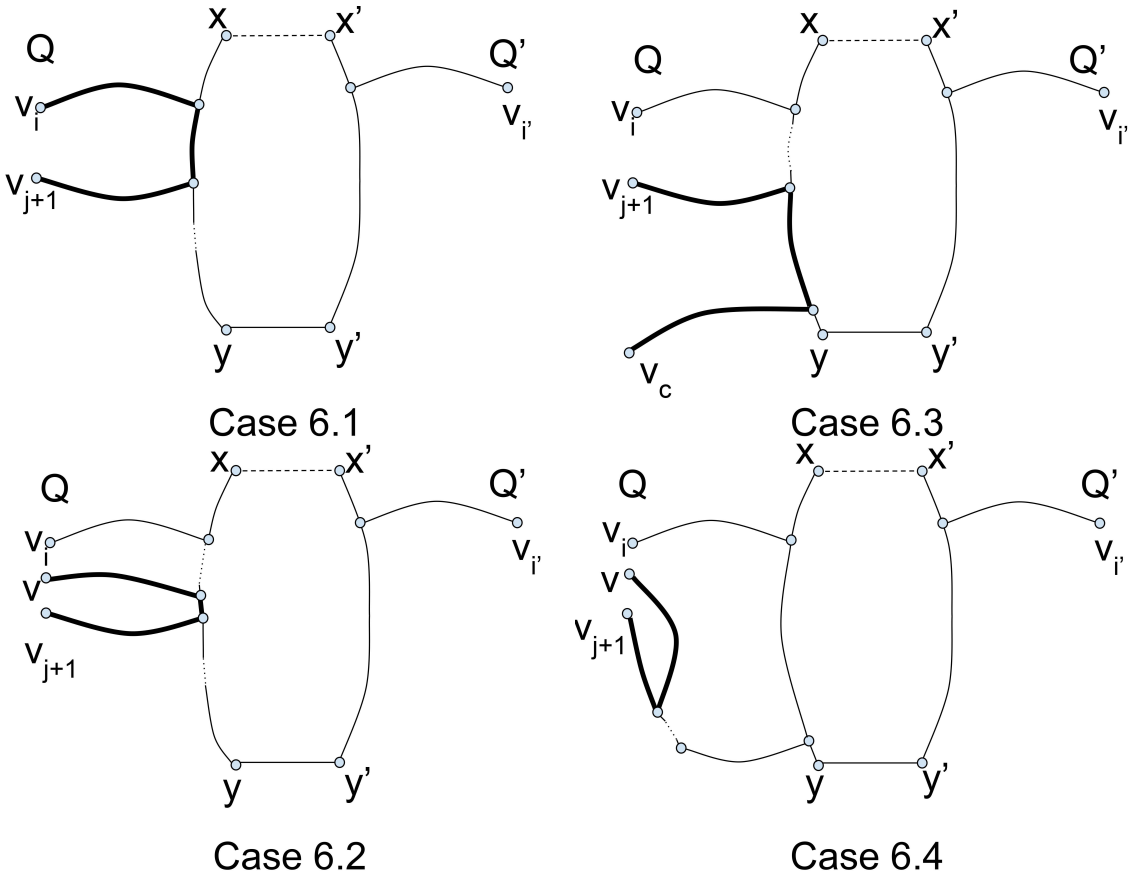

Figure 3.6. In Case VI, $f \neq f'$, $f' \neq e_{j+1}$ and f is not on the same side as v_{j+1}. In this case, the algorithms sets e'_{j+1} to be e_{j+1}. Subcase 6.1: e'_{j+1} is on the path from v_i to v_{j+1}. Subcase 6.3: e'_{j+1} is on the path from v_{j+1} to v_c. For both Subcase 6.2 and Subcase 6.4, e'_{j+1} is on the path from v_{j+1} to some vertex $v \in V(Q) \cap V(S_j)$. Recall that in all our figures, we use thick curves to represent paths in T' where e_{j+1} must belong. Dotted curves have edges that belong to A'_j.

Thus we can get that $e_{j+1} \in A'_{j+1}$ by running the first procedure from Claim 3.3.1 to obtain A'_{j+1} by first disconnecting v from v_{j+1} in T', and thus selecting e_{j+1}. This means that $e'_{j+1} = e_{j+1}$.

Consider now Case VII (depicted in the Figure 3.7). Assume by symmetry that $v_{j+1} \in V(Q)$, but e_{j+1} does not have both endpoints in $V(Q)$. Thus the second procedure from Claim 3.3.1 has considered a simple path P from v_{j+1} to some vertex in $V(Q')$. That vertex must be $v_{i'}$, since this is the only vertex of $V(Q') \cap V(S_j)$ that can be connected to x' in $(V, E(T) \setminus A_j)$. Previously, we just found f on the simple path $P' = P_T(v_i, v_{i'})$ where

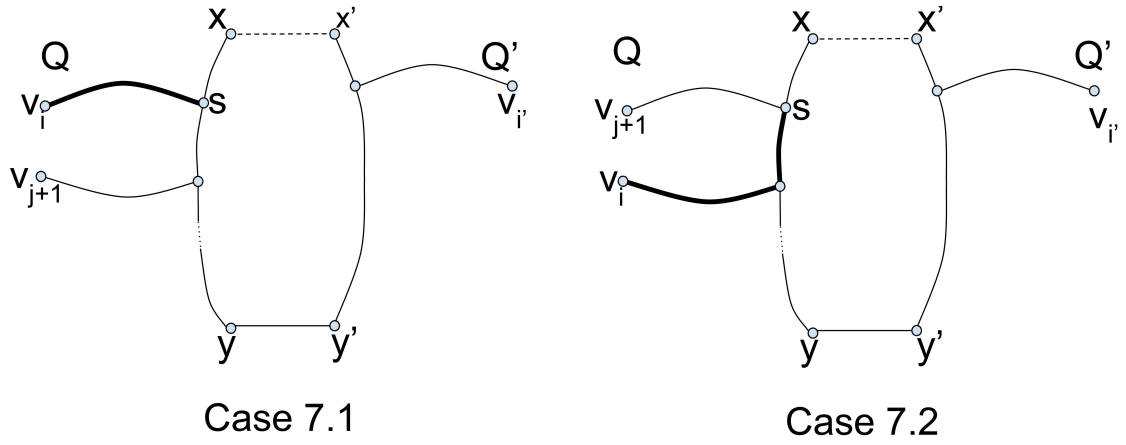

Figure 3.7. In Case VII, $f \neq f'$, $f' \neq e_{j+1}$, and e_{j+1} is not on the same side as v_{j+1} (and therefore, as argued in the text, f is on the same side as v_{j+1}). In both Subcases 7.1 and 7.2, e'_{j+1} ($e'_{j+1} = f$) is on the path from v_i to s. The dotted paths in the figures may not exist (in which case $v_c = v_i$).

$v_i \in V(Q)$. Thus, in $(V, E(T) \setminus (A_j - f))$, both v_i and v_{j+1} are connected to x and therefore are connected. As e_{j+1} does not have both endpoints in $V(Q)$, it follows that f is the only edge of A_{j+1} on $P_T(v_i, v_{j+1})$; moreover, as in Claim 3.3.1, f is the largest-cost edge on $P_T(v_i, v_{j+1})$. Note that $P_T(v_{j+1}, v_i) = P_{T'}(v_{j+1}, v_i)$, and moreover, since we are in Case VII, we have $f \neq f'$. Thus we can get that $f \in A'_{j+1}$ by running the first procedure from Claim 3.3.1 to obtain A'_{j+1} by first disconnecting v_i from v_{j+1} in T', and thus selecting f. This means that $e'_{j+1} = f$, as set by the algorithm. This finishes Case VII.

Recall that we have that $A_{j+1} \setminus A'_j = \{f, e_{j+1}\}$, which means e'_{j+1} is either f or e_{j+1}, and since we excluded cases I-VII, either $v_{j+1} \in V(Q)$ and $f, e_{j+1} \in E(Q)$, or $v_{j+1} \in V(Q')$ and $f, e_{j+1} \in E(Q')$. Assume by symmetry that $f, e_{j+1} \in E(Q)$ and $v_{j+1} \in V(Q)$. Start constructing A_{j+1} as in the first procedure of Claim 3.3.1, picking pairs of vertices to be separated such that both such vertices are either both in in $V(S_j) \cap V(Q)$ or both in $V(S_j) \cap V(Q')$. Neither f nor e_{j+1} were selected yet (since all the edge selected will also be selected by the procedure constructing A'_j). Thus, the edge f is obtained to be the largest-cost on $P_T(v_i, v_{i'})$, with $v_i \in V(Q) \cap V(S_j)$ and $v_{i'} \in V(Q') \cap V(S_j)$. Since $f \in E(Q)$, we have that v_i is not connected to x in $(V, E(T) \setminus A_j)$, while $v_{i'}$ is connected

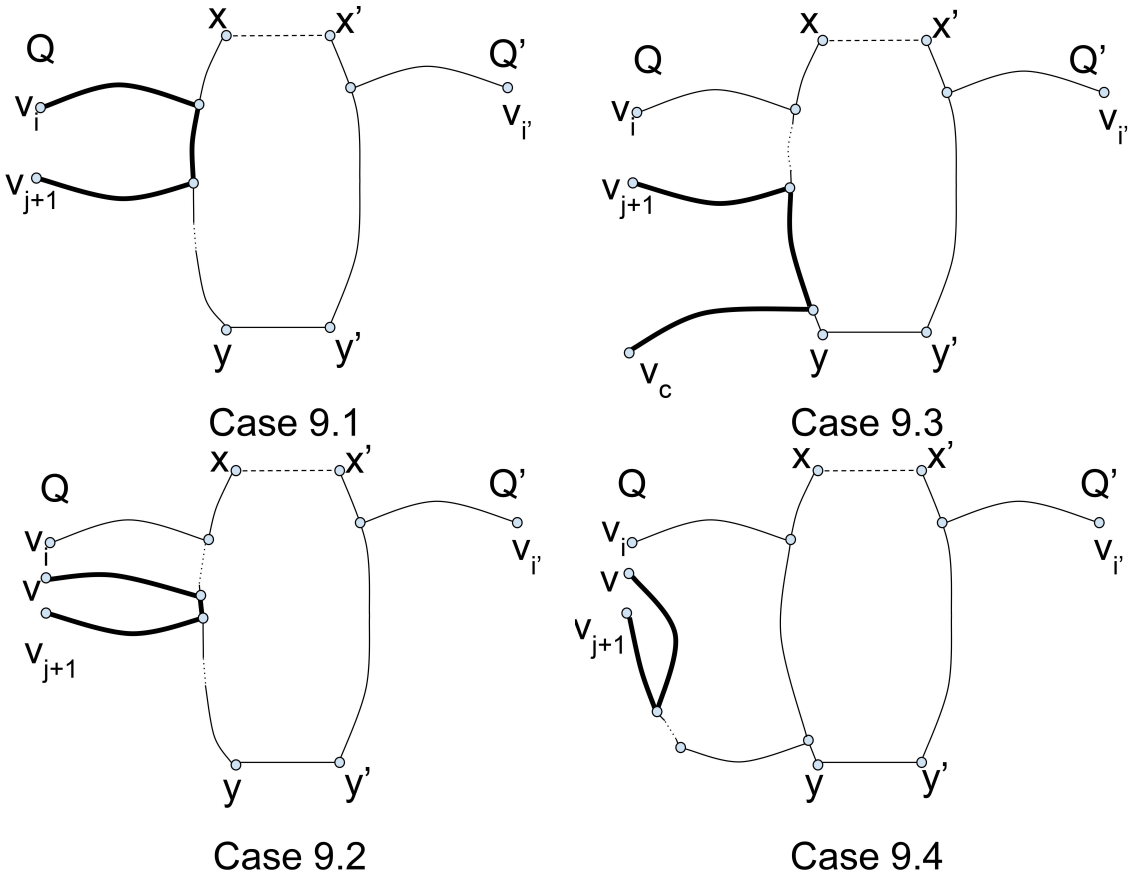

Figure 3.8. In Case IX, $f \neq f'$, $f' \neq e_{j+1}$. f, e_{j+1} are on the same side as v_{j+1} and e_{j+1} is not the largest-cost edge from v_{j+1} to x. In this case, the algorithm sets e'_{j+1} to be e_{j+1}. Subcase 9.1: e'_{j+1} is on the path from v_i to v_{j+1}. Subcase 9.3: e'_{j+1} is on the path from v_{j+1} to v_c. For both Subcase 9.2 and Subcase 9.4, e'_{j+1} is on the path from v_{j+1} to some vertex $v \in V(Q) \cap v(S_j)$.

to x in $(V, E(T) \setminus A_j)$.

Now consider Case IX (the more complicated Case VIII is left to be last), where e_{j+1} is not the largest-cost edge on the simple path $P_T(v_{j+1}, x)$. This means that, when computing A_{j+1} according to the second procedure of Claim 3.3.1, e_{j+1} (which belongs to $E(Q)$) is not used to disconnect v_{j+1} from a vertex in $V(Q')$ in $(V, E(T) \setminus A_j)$. Thus e_{j+1} is the largest-cost edge on $P_T(v_{j+1}, v)$, with $v \in V(Q) \cap V(S_j)$ and with $P_T(v_{j+1}, v) \cap A_j = \emptyset$. Then e_{j+1} should be selected in A'_{j+1} if we construct A'_{j+1} by first separating v_{j+1} from v. Since $e_{j+1} \notin A'_j$, it is correct to set $e'_{j+1} = e_{j+1}$ (As depicted in Figure 3.8).

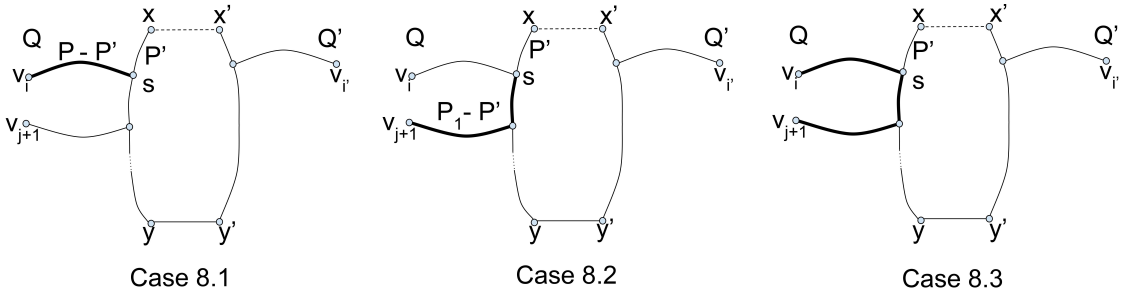

Figure 3.9. In Case VIII, $f \neq f'$, $f' \neq e_{j+1}$. f, e_{j+1} are on the same side as v_{j+1} and e_{j+1} is the largest-cost edge from v_{j+1} to x. In this case, e'_{j+1} is set to $max(e_{j+1}, f)$. Subcase 8.1: e'_{j+1} is on the path $P - P'$. Subcase 8.2: e'_{j+1} is on the path $P_1 - P'$. Subcase 8.3: e'_{j+1} is on the path from v_i to v_{j+1}.

Now consider Case VIII. Let P_1 be the simple path $P_T(v_{j+1}, x)$. The first procedure of Claim 3.3.1 to compute A_j could first pick pairs of vertices to be separated such that both such vertices are either both in in $V(S_j) \cap V(Q)$ or both in $V(S_j) \cap V(Q')$. All the edges thus selected are in A'_j as well. Then one last edge, which must be f since $f \notin A'_j$, is obtained by separating a vertex $v_i \in V(Q)$ from $v_{i'} \in V(Q')$. Let P be the simple path $P_T(v_i, x)$; since $f \in E(Q)$ in Case VIII, we have that f is the largest-cost edge on P. Let s be the first vertex of P_1, when going from v_{j+1} to x, that is also a vertex of P. Call P' the subpath of P (or P_1) from s to x. We have subcases.

We first notice we cannot have that both f and e_{j+1} are on P', since f is the largest-cost edge of P and e_{j+1} is the largest-cost edge of P_1 and P' is contained in both P and P_1.

If e_{j+1} belongs to P' but f does not (See Subcase 8.1 in Figure 3.9), then f is the largest-cost edge of A_{j+1} on the simple path $P_T(v_{j+1}, v_i)$, since f has larger cost than e_{j+1} since both are on P, and e_{j+1} is the largest-cost edge on P_1. Thus if we start constructing A'_{j+1}, using the first procedure of 3.3.1, by choosing to separate v_i from v_{j+1}, we pick f and thus $f \in A'_{j+1}$. The algorithm correctly chooses f to be e'_{j+1}.

If f belongs to P' but e_{j+1} does not (See Subcase 8.2 in Figure 3.9), then e_{j+1} is the

largest-cost edge of A_{j+1} on the simple path $P_T(v_{j+1}, v_i)$, since e_{j+1} has larger cost than f since both are on P_1, and f is the largest-cost edge on P. Thus if we start constructing A'_{j+1}, using the first procedure of 3.3.1, by choosing to separate v_i from v_{j+1}, we pick e_{j+1} and thus $e_{j+1} \in A'_{j+1}$. The algorithm correctly chooses e_{j+1} to be e'_{j+1}.

In the last subcase, neither e_{j+1} nor f are on the path P' (As Subcase 8.3 in Figure 3.9). Note that the simple path $P_T(v_{j+1}, v_i)$ goes through s, and that f is the largest-cost edge between v_i and s. Since e_{j+1} is the largest-cost edge on $P_T(v_{j+1}, x)$, it is also the largest-cost edge on $P_T(v_{j+1}, s)$. Thus if we start constructing A'_{j+1}, using the first procedure of 3.3.1, by choosing to separate v_i from v_{j+1}, we pick $\max(e_{j+1}, f)$ and thus $\max(e_{j+1}, f) \in A'_{j+1}$. That is exactly what the algorithm does, setting $e'_{j+1} = \max(e_{j+1}, f)$ and $f = \min(e_{j+1}, f)$.

For the correctness of the (possible) update of v_c after e'_{j+1} was found, Consider trying to find A'_{j+1} from A'_j using the second procedure from Claim 3.3.1. Before the procedure, v_c and y are in the same component of $(V, E(T') \setminus A'_j)$. So only e'_{j+1} is the only edge in A_{j+1} on the path $P_{T'}(v_c, y)$. This can happen exactly when, as the algorithm checks, when v_c is a descendant of e'_{j+1} in T'. Assume now that v_c is separated by e'_{j+1} from y. Since the procedure chooses as e'_{j+1} the largest-cost edge between v_{j+1} and the only vertex of $V(S_j)$ in the same component with v_{j+1} in the graph $(V, E(T') \setminus A'_j)$, it must be the case that v_{j+1} is in the same component with v_c. This component is broken into two by the removal of e'_{j+1}, and it must be the case that v_{j+1} is on the same "side" with y, and should become the new v_c. □

Preprocessing is $O(n)$, and applies to all stars S with all the possible centers. Thus in time $O(\sum_{u \in V} d'(u)) = O(m)$ we can obtain $W_{T'}(V(S))$ for all the stars S with all the centers.

Based on the discussion of this section, we obtain one of our main results:

Theorem 3.3.1. *For Min-Power Broadcast, there exists an algorithm with running time* $O(mn)$ *and approximation ratio* 4.2 *in the two-dimensional Euclidean input model and* $2(1 + \ln n)$ *in the bidirected input model.*

3.4 The Greedy-Spider Algorithm

For each algorithm described in this section, the input is a directed graph $G = (V, E)$ with costs $c : E \to R^+$. Whenever we refer to the *cost* of an arc uv in the following, we implicitly assume $uv \in E(G)$ and we mean c_{uv}, the power requirement of the arc.

Part of the description below is copied with minor corrections from [10]. The algorithm works in iterations. It starts iteration i with a directed graph H_i seen as a set of arcs with vertex set V (initially, $H_1 = \emptyset$). The strongly connected components of H_i which do not contain the source z and have no incoming arc are called *unhit components*. The algorithm stops if no unhit components exists, since in this case the source can reach every vertex in H_i. Otherwise, a weighted structure called a *spider* (details below) is computed such that it achieves the biggest reduction in the number of unhit components divided by the weight of the spider. The algorithm then adds the spider (seen as a set of arcs) to H_i to obtain H_{i+1}.

Definition 3.4.1. *A* spider *is a directed graph consisting of one vertex called* head *and a set of non-trivial directed paths (called* legs*), each of them from the head to a (vertex called)* foot *of the spider. The* weight *of the spider* S*, denoted by* $w(S)$*, is the maximum cost of the arcs leaving the head plus the sum of costs of the legs, where the cost of a leg is the sum of the costs of its arcs without the arc leaving the head.*

See Figure 3.10 for an illustration of a spider and its weight. As opposed to stars, the weight of the spider S can be higher than $p(S)$ (here we assume S is a set of arcs), as the legs of the spider can share vertices, and for those vertices the sum (as opposed to the maximum) of the costs of outgoing arcs contributes to $w(S)$. From every unhit component

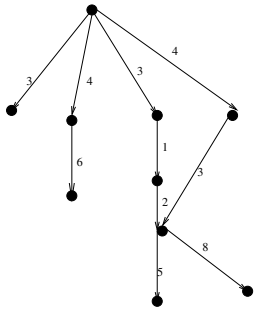

Figure 3.10. A spider with four legs, weight equal to $\max\{3,4,3,4\}+6+(1+2+5)+(3+8)=29$, and power equal to 24.

of H_i we arbitrarily pick a vertex and we call it a *representative*.

Definition 3.4.2. *The* shrink factor *of a spider S with head h with respect to H, denoted by $sf(S)$, is:*

1. *the number of representatives among its feet if h is reachable (where, by convention, a vertex is reachable from itself) from the source z or if h is not reachable from any of its feet, or*

2. *the number of representatives among its feet minus one, otherwise.*

The algorithm of [10] appears in Figure 3.11, with small differences in notation. For its approximation ratio, please refer to [10]. We describe later the detailed implementation of Step 2.1 of the algorithm.

A method for finding the spider S which minimizes $w(S)/(sf(S))$ with respect to H is described in [10] (a proof of correctness was not given in the conference paper [10]). A running time is not given there, but a naive implementation, with some careful precomputation, is still $\Omega(n^3)$. Up to n spiders may be added to H by the algorithm, and to achieve an overall running time of $O(n^3)$ requires keeping data structures and simple amortized analysis, described next.

Input: A complete directed graph $G = (V, E)$, cost function c_{uv} and a source vertex z
Output: A directed spanning graph H (seen as a set of arcs, with $V(H) = V$) such that in H there is a path from the source z to every vertex of V.

(1) Initialize $H = \emptyset$

(2) While H has at least one unhit component

  2.1 Find the spider S which minimizes $w(S)/(sf(S))$ with respect to H

  2.2 Set $H \leftarrow H \cup S$

(3) For all vertices v assign $p(v) = \max_{vu \in H} c_{vu}$

Figure 3.11. The Greedy-Spider algorithm for Min-Power Broadcast with asymmetric costs

3.4.1 The original variant - copied from [10]. Next we describe how to find the spider which minimizes its weight divided by its shrink factor. In fact, we search for *powered* spiders (the power of the head is given). This does not cause any problems as finding the best powered spider finds the best spider.

We try all possible heads h, and all possible discrete power values for the head (there are at most n such discrete power values - precisely the values c_{hu} for every $u \in G$, where $c_{hh} = 0$ by convention). Define the *children* of the head to be the vertices within its power value - where the head is also considered a child. For each representative r_i, compute the shortest path P_i from a child of h to r_i. If h is not reachable from the source, partition the representatives in two sets - R_1 which cannot reach h and R_2 which can reach h; otherwise let $R_1 = R$ and $R_2 = \emptyset$. Sort R_1 and R_2 such that the lengths of the paths P_i are in nondecreasing order. Then the best spider with head h and the given power value can be obtained by trying all integers j_1 and j_2 satisfying $0 \leq j_1 \leq |R_1|$ and $0 \leq j_2 \leq |R_2|$, and taking the paths P_i leading to the first j_1 representatives of R_1 and the first j_2 representatives of R_2.

3.4.2 The new variant and its correctness. First, we explain a situation in the original variant, in the confusing case when the head is a representative. Since in definition 3.4.1

the legs are required to be non-trivial, the head of a spider is not a foot. However, in the algorithm from the previous subsection, paths of length 0 are implicitly considered, since the head is a child of itself in the spider. Then it appears as if the number of reached representative is over counted by one compared to the Definition 3.4.2. However, the head can reach itself and is in R_2, in fact the only vertex in R_2, since distinct representatives cannot reach each other, because each is in a separate component with no arcs coming in. So while the first case of Definition 3.4.2 should apply for any spider whose head is a representative, the fact that the algorithm from the previous subsection may add representatives from R_2 and subtract 1 leads to the same result.

Instead of minimizing the ratio $w(S)/(sf(S))$, we prefer to use below maximizing the ratio $(sf(S)/w(S))$, which we call the *M-ratio* of a spider. Using Floyd-Warshall, all pairs shortest paths are precomputed and stored in $SP[i][j]$. This takes time $O(n^3)$.

We clarify in this paragraph the ideas from the original variant. For every u, let $v_1, \ldots, v_{d'(u)}$ be the out-neighbors of u in G, sorted in non-decreasing order by c_{uv_i}. As in the original variant, we try all possible heads and all possible discrete power values for the head. That is, for each node u and each i, we have a data structure of size n (a constant number of variables and arrays of length n, actually), which is designed to compute the spider with the maximum M-ratio w.r.t H among those with head u and such that u has power c_{uv_i}; that is, with powered head u, i. The total space used here is $O(mn)$, as there are $\sum_{u \in V}(1 + d'(u)) = n + m$ powered heads.

In the following u is fixed. The children of the powered head u, i (as defined in the original variant) are $u, v_1, \ldots, v_i$. Call $v_0 = u$. Also as in the original variant (with a more precise definition), for every vertex v, let $d[v]$ be the length of a shortest path from a child of the spider to v (it is 0, if v itself is a child). Then $d[v] = \min_{k=0}^{i} SP[v_k][v]$. The algorithm computes these quantities as follows. For powered head $u, 0$, we have $d[v] = SP[v_0][v]$ for all $v \in V$. In the next iteration, we try to update $d[v]$ by $SP[v_1][v]$. Now $d[v]$ is correct for

powered head $u, 1$. Repeat for $2, \ldots$, until we finish powered head $u, d'(u)$. The total time used here is $O(n^3)$.

Recall the vertex sets R_1 and R_2 as defined in the previous subsection. Fix i as well so we discuss powered head u, i. All the nodes v are sorted, as described in Subsection 3.4.3, in non-decreasing order of $d[v]$, and their indices kept in an array $I[..]$; we call this order u, i-sorted. u itself is the first in I (first position). Another array I^{-1} keeps for each $x \in V$, its index in I; that is, the j with $I[j] = x$.

The list R contains the representatives that cannot reach u, and is kept implicitly, as follows: array $Next[j]$ keeps the next index of I; that is, such that $I[Next[j]]$ is the node that follows $I[j]$ in the u, i-sorted list R. Similarly, $Prev[j]$ is the node that precedes $I[j]$ in the u, i-sorted list R. In the case that the head u is not reachable from the source, we have that $R = R_1$ (R_1 is defined in Subsection 3.4.1). We do not keep R_2 (the representatives that can reach u; u itself may be in R_2), but instead keep $\overline{R} = R_1 \cup R_2$ (which contains all representatives) implicitly, as R is kept, using arrays $\overline{Prev}[j]$, $\overline{Next}[j]$.

Also, the algorithm keeps three variables, q_{opt}, S, and j_{opt}, which are meant to describe the following mathematical indices. Let $x_1, x_2, \ldots, x_{|R|}$ be the elements of R, u, i-sorted. Define $P := c_{uv_i}$ and

$$M_q := \frac{q}{P + \sum_{l=1}^{q} d[x_l]} \tag{3.1}$$

Then q_{opt} should be $\arg\max_{q \in \{1,2,\ldots,|R|\}} M_q$, S should be $\sum_{l=1}^{q_{opt}} d[x_l]$, and j_{opt} should be that $I[j_{opt}] = x_{q_{opt}}$ (or in other words, $j_{opt} = I^{-1}[x_{q_{opt}}]$). We will prove later that these variables stored are indeed what they should be.

The algorithm also keeps another three variables, $\overline{q_{opt}}$, $\overline{S}$, and $\overline{j_{opt}}$, which are meant to describe the following mathematical indices. Let $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_{|\overline{R}|}$ be the elements of $\overline{R}$,

u, i-sorted. Define:

$$\overline{M}_q := \frac{q-1}{P + \sum_{l=1}^{q} d[\overline{x}_l]} \tag{3.2}$$

Then $\overline{q_{opt}}$ should be $\arg\max_{q \in \{1,2,\ldots,|\overline{R}|\}} \overline{M}_q$, $\overline{S}$ should be $\sum_{l=1}^{\overline{q_{opt}}} d[\overline{x}_l]$, and $\overline{j_{opt}}$ should be that $I[\overline{j_{opt}}] = \overline{x}_{\overline{q_{opt}}}$ (or in other words, $\overline{j_{opt}} = I^{-1}[\overline{x}_{\overline{q_{opt}}}]$). We will prove later that these variables stored are indeed what they should be.

The algorithm also keeps $\widehat{q_{opt}}$, $\widehat{S}$, and $\widehat{j_{opt}}$, and uses them when u is reachable from z. They are meant to describe the following mathematical indices. Define:

$$\widehat{M}_q := \frac{q}{P + \sum_{l=1}^{q} d[\overline{x}_l]} \tag{3.3}$$

Then $\widehat{q_{opt}}$ should be $\arg\max_{q \in \{1,2,\ldots,|\overline{R}|\}} \widehat{M}_q$, $\widehat{S}$ should be $\sum_{l=1}^{\widehat{q_{opt}}} d[\overline{x}_l]$, and $\widehat{j_{opt}}$ should be that $I[\widehat{j_{opt}}] = \overline{x}_{\widehat{q_{opt}}}$ (or in other words, $\widehat{j_{opt}} = I^{-1}[\overline{x}_{\widehat{q_{opt}}}]$) We will prove later that these variables stored are indeed what they should be.

Lemma 3.4.1. *For every q, there exists a spider with powered head u, i and M-ratio at least M_q and a spider with powered head u, i and M-ratio at least $\overline{M}_q$. If u is reachable from the source, there exists a spider with powered head u, i and M-ratio at least $\widehat{M}_q$.*

Proof. We construct 3 spiders S_1, S_2, S_3 with powered head u, i. Let S_1 use $\{u, v_i, \ldots, v_i\}$ as the children of u (recall that we defined earlier $v_1, \ldots, v_{d'(u)}$ to be the out-neighbors of u in G sorted in non-decreasing order by c_{uv_j}), and shortest paths as its legs to reach $x_1, x_2, \ldots, x_{q_{opt}} \in R$ from a child of u.

Let S_2 use the same children as S_1 and shortest paths as its legs to reach $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_{\overline{q_{opt}}} \in \overline{R}$ from a child of u. Also let S_3 use the same children as S_1 and S_2 and shortest paths as its legs to reach $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_{\widehat{q_{opt}}} \in \overline{R}$ from a child of u.

By its definition, $M_{q_{opt}} = \frac{q_{opt}}{P + \sum_{l=1}^{q_{opt}} d[x_l]}$. The denominator of this term is $P +$

$\sum_{l=1}^{q_{opt}} d[x_l]$. It is power of the head plus the sum of shortest paths of all the legs, which is $w(S_1)$. q_{opt} is exactly the shrinking factor of S_1, by Definition 3.4.2, so $M_{q_{opt}}$ is the M-ratio of S_1. By its definition, $\overline{M}_{\overline{q_{opt}}} = \frac{\overline{q_{opt}}-1}{P+\sum_{l=1}^{\overline{q_{opt}}} d[\overline{x}_l]}$. The denominator is $w(S_2)$. Also, $\overline{q_{opt}} - 1$ is exactly the shrinking factor of S_2, by Definition 3.4.2, so $\overline{M}_{\overline{q_{opt}}}$ is the M-ratio of S_2. By its definition, $\widehat{M}_{\widehat{q_{opt}}} = \frac{\widehat{q_{opt}}}{P+\sum_{l=1}^{\overline{q_{opt}}} d[\overline{x}_l]}$. The denominator is exactly $w(S_3)$. Since u is reachable from the source, $\widehat{q_{opt}}$ is exactly the shrinking factor of S_3, by Definition 3.4.2. So $\widehat{M}_{\widehat{q_{opt}}}$ is the M-ratio of S_3. □

Lemma 3.4.2. *If u is not reachable from z, and if the spider maximizing the M-ratio for powered head u, i does not have any feet in R_2, then $M_{q_{opt}} = \max_{q\in\{1,2,\ldots,|R|\}} M_q$ provides such a spider.*

Proof. Assume that the spider S maximizing the M-ratio does not have any feet in R_2. Since S does not have any feet in R_2, the feet of S are all in R. For any q, if S has q feet, the shrink factor is q by Definition 3.4.2. $w(S)$ is P plus the sum of the cost of the q legs. We can minimize this value by choosing $x_1, x_2, \ldots, x_q \in R$ as its feet since R is sorted. Since $M_{q_{opt}} = \max_{q\in\{1,2,\ldots,|R|\}} M_q$, we can build the spider which maximizes the M-ratio as in Lemma 3.4.1. □

Lemma 3.4.3. *If u is not reachable from z, and if the optimal spider for powered head u, i has at least one foot in R_2, then $\overline{M}_{\overline{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \overline{M}_q$ provides such a spider.*

Proof. Assume that the spider S maximizing the M-ratio has at least one foot in R_2. Since S has at least one foot in R_2, the feet of S are all in $\overline{R}$. For any q, if S has q feet, the shrink factor is $q-1$ by Definition 3.4.2. $w(S)$ is P plus the sum of the costs of the q legs. We can minimize this value by choosing $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_q \in \overline{R}$ as its feet since $\overline{R}$ is sorted. Since $M_{\overline{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \overline{M}_q$, we can build the spider which maximizes the M-ratio as in Lemma 3.4.1. □

Lemma 3.4.4. *If u is reachable from z, then $\widehat{M}_{\widehat{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \widehat{M}_q$ provides the optimal spider for powered head u, i.*

Proof. Assume that the spider S maximizes the M-ratio. Since S may use some feet in R_2, the possible feet of S are all in $\overline{R}$. For any q, if S has q feet, the shrink factor of S is q, by Definition 3.4.2. $w(S)$ is P plus the sum of the costs of the q legs. We can minimize this value by choosing $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_q \in \overline{R}$, as its feet since $\overline{R}$ is sorted. Since $\widehat{M}_{\widehat{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \widehat{M}_q$, we can build the spider which maximizes the M-ratio as in Lemma 3.4.1. □

Theorem 3.4.1. *Assume that q_{opt} is indeed such that $M_{q_{opt}} = \max_{q\in\{1,2,\ldots,|R|\}} M_q$, $\overline{q_{opt}}$ is indeed such that $\overline{M}_{\overline{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \overline{M}_q$, and $\widehat{M}_{\widehat{q_{opt}}} = \max_{q\in\{1,2,\ldots,|\overline{R}|\}} \widehat{M}_q$. The maximum M-ratio of a spider w.r.t. H equals $\max\{M_{q_{opt}}, \overline{M}_{\overline{q_{opt}}}\}$ over all possible u, i when u is not reachable from z, or $\widehat{M}_{\widehat{q_{opt}}}$ over all possible u, i when u is reachable from z.*

Proof. Assume that the spider that maximize the M-ratio is S, with head u and power c_{uv_i}. In a first case u is not reachable from z. If S does not have any feet that can reach u in H, then a spider no worse than S is found according to lemma 3.4.2, used for the powered head u, i. In S does have feet that can reach u in H, then a spider no worse than S is found according to lemma 3.4.3, used for the powered head u, i. In the second case, when u is reachable from z, then a spider no worse than S is found according to lemma 3.4.4, used for the powered head u, i. □

We use the following properties of M_q, previously known in similar settings.

Lemma 3.4.5 (folklore)**.** *Assume $2 \leq q \leq |R| - 1$. If $M_{q+1} \geq M_q$, then $M_q \geq M_{q-1}$. If $M_{q-1} \geq M_q$, then $M_q \geq M_{q+1}$.*

Proof. For the first part, from $M_{q+1} \geq M_q$, we deduce:

$$\frac{q+1}{P + d[x_{q+1}] + \sum_{l=1}^{q} d[x_l]} \geq \frac{q}{P + \sum_{l=1}^{q} d[x_l]}, \tag{3.4}$$

from which we deduce:

$$P + \sum_{l=1}^{q} d[x_l] \geq q d[x_{q+1}], \tag{3.5}$$

and therefore, using $d[x_{q+1}] \geq d[x_q]$, we have:

$$P + \sum_{l=1}^{q-1} d[x_l] \geq q d[x_{q+1}] - d[x_q] \geq (q-1) d[x_q], \tag{3.6}$$

which can be seen to imply $M_q \geq M_{q-1}$.

Similarly, for the second part, from $M_{q-1} \geq M_q$, we deduce:

$$\frac{q-1}{P + \sum_{l=1}^{q-1} d[x_l]} \geq \frac{q}{P + d[x_q] + \sum_{l=1}^{q-1} d[x_l]}, \tag{3.7}$$

from which we deduce:

$$(q-1) d[x_q] \geq P + \sum_{l=1}^{q-1} d[x_l], \tag{3.8}$$

and therefore, using $d[x_{q+1}] \geq d[x_q]$, we have:

$$q d[x_{q+1}] \geq (q-1) d[x_q] + d[x_q] \geq P + \sum_{l=1}^{q} d[x_l], \tag{3.9}$$

which can be seen to imply $M_q \geq M_{q+1}$. □

Lemma 3.4.6 (folklore)**.** *Assume* $2 \leq q \leq |\overline{R}| - 1$. *If* $\overline{M}_{q+1} \geq \overline{M}_q$, *then* $\overline{M}_q \geq \overline{M}_{q-1}$. *If* $\overline{M}_{q-1} \geq \overline{M}_q$, *then* $\overline{M}_q \geq \overline{M}_{q+1}$.

Proof. From $\overline{M}_{q+1} \geq \overline{M}_q$, we deduce:

$$\frac{q}{P + d[\overline{x}_{q+1}] + \sum_{l=1}^{q} d[\overline{x}_l]} \geq \frac{q-1}{P + \sum_{l=1}^{q} d[\overline{x}_l]}, \tag{3.10}$$

from which we deduce:

$$P + \sum_{l=1}^{q} d[\overline{x}_l] \geq (q-1) d[\overline{x}_{q+1}], \tag{3.11}$$

and therefore, using $d[\overline{x}_{q+1}] \geq d[\overline{x}_q]$, we have:

$$P + \sum_{l=1}^{q-1} d[\overline{x}_l] \geq (q-1) d[\overline{x}_{q+1}] - d[\overline{x}_q] \geq (q-2) d[\overline{x}_q], \tag{3.12}$$

which can be seen to imply $\overline{M}_q \geq \overline{M}_{q-1}$.

Similarly, from $\overline{M}_{q-1} \geq \overline{M}_q$, we deduce:

$$\frac{q-2}{P + \sum_{l=1}^{q-1} d[\overline{x}_l]} \geq \frac{q-1}{P + d[\overline{x}_q] + \sum_{i=l}^{q-1} d[\overline{x}_l]}, \tag{3.13}$$

from which we deduce:

$$(q-2) d[\overline{x}_q] \geq P + \sum_{l=1}^{q-1} d[\overline{x}_l] \tag{3.14}$$

and therefore, using $d[\overline{x}_{q+1}] \geq d[\overline{x}_q]$, we have:

$$(q-1) d[\overline{x}_{q+1}] \geq (q-2) d[\overline{x}_q] + d[\overline{x}_q] \geq P + \sum_{l=1}^{q} d[\overline{x}_l] \tag{3.15}$$

which can be seen to imply $\overline{M}_q \geq \overline{M}_{q+1}$. □

The next corollary is proven just as Lemma 3.4.5.

Corollary 3.4.1. *Assume* $2 \leq q \leq |\overline{R}| - 1$. *If* $\widehat{M}_{q+1} \geq \widehat{M}_q$, *then* $\widehat{M}_q \geq \widehat{M}_{q-1}$. *If* $\widehat{M}_{q-1} \geq \widehat{M}_q$, *then* $\widehat{M}_q \geq \widehat{M}_{q+1}$.

As the algorithm progresses, more arcs are added to H and as result we have fewer strongly connected components, and fewer unhit strongly connected components. The algorithm can easily be careful to never select new representatives, and thus each R ($R = R(u,i)$ for powered head u,i) is shrinking. $\overline{R}$ is also shrinking. Whenever a vertex y becomes a non-representative, we go to each u and each i and as we describe starting with the next paragraph, in constant amortized time, modify the data structure for the u,i-powered head. This modification includes updating the arrays $Prev[]$, $Next[]$, and the variables q_{opt}, S, and j_{opt}, as well as $\overline{Prev}[]$, $\overline{Next}[]$, $\overline{q_{opt}}$, $\overline{S}$, $\overline{j_{opt}}$, $\widehat{q_{opt}}$, $\widehat{S}$, and $\widehat{j_{opt}}$. There are $O(n^2)$ powered heads (in fact, $O(m)$), and thus modifying all the data structures when one vertex becomes a non-representative takes $O(n^2)$ time, for a total contribution of $O(n^3)$ to the running time of the algorithm.

Modifying the data structure, for a given, fixed u,i, is as follows. Using I^{-1}, find the j such that $I[j] = y$. We have three cases: $j < j_{opt}$, $j = j_{opt}$, and $j > j_{opt}$. In all cases we adjust $Prev[Next[j]] = Prev[j]$ and $Next[Prev[j]] = Next[j]$; in other words, we remove j from the doubly-linked list that implicitly stores $R(u,i)$.

For the purpose of the proof, let R' be $R(u,i)$ without the element y (same sorted order). Also, define M'_l (for $l = 1,\ldots,|R'|$) to be as in Equation (3.1) using the new list R' instead of R. We assume, as an invariant of the algorithm, that the variables q_{opt}, S, and j_{opt} match their mathematical definition (given right after Equation (3.1)) w.r.t. R.

If $j > j_{opt}$, there is nothing else left to do, as we prove below that $M'_{q_{opt}}$ is still largest among all M'_l values, for $1 \le l \le |R'|$. Indeed, we have that $M'_{q_{opt}} \ge M'_{q_{opt}-1}$ as $q_{opt} - 1$ and q_{opt} refer to the same list (a sublist of R' that did not change from the corresponding sublist of R). And we still have that $M'_{q_{opt}} \ge M'_{q_{opt}+1}$, since even if the $(q_{opt}+1)^{st}$ element of R' differs versus R, the change resulted in an increase of $P + \sum_{l=1}^{q_{opt}+1} d[x_l]$. The facts that $M'_{q_{opt}} \ge M'_{q_{opt}-1}$ and that $M'_{q_{opt}} \ge M'_{q_{opt}+1}$, combined with Lemma 3.4.5, imply that $M'_{q_{opt}} \ge M'_l$ for every $l = 1,2,\ldots,|R'|$. Thus q_{opt}, S, and j_{opt}

match their mathematical definition w.r.t. R'.

If $j < j_{opt}$, the update procedure sets $j' = j_{opt}$, and if $j = j_{opt}$, then the procedure sets $j' = Prev[j_{opt}]$. In both cases, the procedure sets $q' = q_{opt} - 1$, and $S' = S - d[I[j]]$. Note that $M'_{q'}$ is correctly computed as $q'/(P + \sum_{l=1}^{q'} d[x_l])$. The procedure then compares $M'_{q'}$ with $M'_{q'+1}$, where $M'_{q'+1}$ can be computed in constant time using $Next$ links. If $M'_{q'} \geq M'_{q'+1}$ the update procedure stops with $j_{opt} = j'$, $q_{opt} = q'$, and $S = S'$; else it sets $q' = q' + 1$, $j' = Next[j']$, and $S' = S' + d[I[j']]$, and continues by comparing $M'_{q'}$ with $M'_{q'+1}$. We need to argue (soon) that this procedure is correct; note however that it takes time $O(1)$ plus the number of times when $M'_{q'} < M'_{q'+1}$. Each such comparison increases $Next[j_{opt}]$ by at least one; note also that $Next[j_{opt}]$ can never decrease. Thus the aggregate cost of such comparisons is $O(n)$ for every powered head u, i, for a total contribution to the algorithm's running time of $O(n^3)$. This means that the overall cost for updating the data structure after $n - 1$ representatives becoming non-representatives is $O(n^3)$.

Now we argue that this update procedure is correct, in both the case $j < j_{opt}$ and the case $j = j_{opt}$. Lemma 3.4.5 ensures that as long as $M_{q'} \geq M_{q'-1}$ (or $q' = 1$), the largest M'_l-value is obtained for an index at least q'; and in fact as soon as $M'_{q'}$ becomes at least $M'_{q'+1}$, we do have indeed that $q_{opt} = q'$. What is left is to show is that, when $q' > 1$, with q' being the value first assigned (this is $q_{opt} - 1$ in both cases), $M'_{q'} \geq M'_{q'-1}$. Indeed, in the case $j = j_{opt}$, then, before $I[j_{opt}]$ was removed from R, we had $M_{q_{opt}} \geq M_{q_{opt}-1}$, and Lemma 3.4.5 ensures that we had $M_{q_{opt}-1} \geq M_{q_{opt}-2}$, which is the same as $M_{q'} \geq M_{q'-1}$. In the case $j < j_{opt}$, let $k < q_{opt}$ be such that x_k is removed from R; that is, $x_k = I[j]$. Let $\sigma = \sum_{l=1}^{q_{opt}} d[x_l]$, before x_k is remove from R. Before x_k is removed from R_1, we had

$$\frac{q_{opt} - 1}{P - d[x_{q_{opt}}] + \sigma} = M_{q_{opt}-1} \leq M_{q_{opt}} = \frac{q_{opt}}{P + \sigma} \tag{3.16}$$

We have that, after x_k (recall this is $I[j]$) is removed from R,

$$M'_{q'} = \frac{q_{opt} - 1}{P - d[x_k] + \sigma} \tag{3.17}$$

and

$$M'_{q'-1} = \frac{q_{opt} - 2}{P - d[x_k] - d[x_{q_{opt}}] + \sigma}. \tag{3.18}$$

Calculations show that (3.16) implies that

$$q_{opt} d[x_{q_{opt}}] \leq P + \sigma. \tag{3.19}$$

We have that $d[x_{q_{opt}}] \geq d[x_k]$, and therefore, by adding $-d[x_{q_{opt}}] \leq -d[x_k]$ to the equation above, we obtain:

$$(q_{opt} - 1) d[x_{q_{opt}}] \leq P - d[x_k] + \sigma, \tag{3.20}$$

which can be seen to imply that, after x_k is removed from R, $M'_{q'-1} \leq M'_{q'}$. This completes the argument that $M'_{q'} \geq M'_{q'-1}$ and thus the correctness of the update procedure for R.

For updating $\overline{R}$, the update procedure is similar, using $\overline{Prev}[]$ instead of $Prev[]$, $\overline{Next}$ instead of $Next[]$, and $\overline{j_{opt}}$, $\overline{q_{opt}}$, and $\overline{S}$ instead of j_{opt}, q_{opt}, and S respectively. The proof of correctness is also similar, except that we use Lemma 3.4.6 instead of Lemma 3.4.5.

For updating and $\widehat{j_{opt}}$, $\widehat{q_{opt}}$, and $\widehat{S}$ (defined after Equation (3.3) w.r.t. $\widehat{M}$), the update procedure is the same as the one for updating j_{opt}, q_{opt}, and S, but operating on $\overline{R}$ instead of R.

3.4.3 Book-keeping. Using Floyd-Warshall, all pairs shortest paths are precomputed and

we know the optimal $SP[i][j]$. This taken $O(n^3)$ time.

In each iteration that the algorithm adds a spider to H, at most n^2 arcs are added (and for every such arc, if it already in H, we do not do it again). We use the data structure of [43] to keep all the reachability data of H, with an overall running time of $O(n^3)$.

Now we describe how the array I is computed, for each powered head u, i. We actually populate all these $I^{u,i}[]$ arrays in one procedure, described next. Since we only have n^2 pairs of shortest paths, we can have at most n^2 values in $SP[i][j]$. We copy all the values in SP to a list L, then sort L (breaking ties arbitrarily). L has at most n^2 elements. This sorting will take $O(n^2 \lg n)$ time. Then we create another $n \times n$ matrix, $\overline{SP}$, and assign $\overline{SP}[i][j] = k$ where k is the index of $SP[i][j]$ in L. $\overline{SP}$ can be populated in $O(n^2 \lg n)$. Recall that $I^{u,i}[]$ must have the elements v sorted in non-decreasing order of $d^{u,i}[v]$. Since each $d[]$ is in L, we can create m other arrays $\bar{d}^{u,i}[]$ which are computed by similar logic as $d[]$ but use $\overline{SP}[i][j]$ instead of $SP[i][j]$. As in Hashing with Chains, we use an auxiliary array A of size $|L|$, where each element is a linked list. All elements of A are initialized to be empty lists. Then we go through all the $O(mn)$ entries of all $\bar{d}^{u,i}[v]$, and add the triplet (u, i, v) to the list $A[\bar{d}^{u,i}[v]]$. After this (as in Bucket Sort), we go through each list $A[i]$, for $i = 1, 2, \ldots, |L|$, and whenever we encounter a triplet (u, i, v), we place v in $I^{u,i}[]$, filling up this array starting from position 1 to position n. We also fill one entry in some I^{-1}. Then the total time this procedure uses for populating all the $I^{u,i}[]$ arrays is $O(n^3)$.

Based on the discussion of this section, we obtain one of our main results:

Theorem 3.4.2. *For Min-Power Broadcast, there exists an algorithm with running time $O(n^3)$ and approximation ratio $2(1 + \ln n)$ in the directed/asymmetric input model.*

3.5 Simulation results

We implemented the following algorithms: the MST-based algorithm, BIP, SPT (shortest-paths tree), RG (Relative-Greedy), HG (Hypergraph-Greedy), GS (Greedy-Spider),

and an IP-based algorithm using the free CPLEX Optimization Studio Academic Research Edition 12.2. We also wrote a post-processing heuristic that makes the solution minimal (goes through the vertices and reduces the power of each vertex as much as possible while keeping the same connectivity). We used the multi-thread capability using multi-core processors to further speed up the heuristics (these greedy algorithms allow some parallelization) and the IP-based algorithms (CPLEX did use eight threads).

We write MST-P to denote the MST algorithm, followed by this post-processing, and similarly have BIP-P, SPT-P, RG-P, HG-P, and GS-P. All the code, other than CPLEX, was written in C++ and compiled with `gpp`, and run on a Intel(R) Core(TM) i7-2310M CPU @ 2.10GHz desktop with 16G memory. We checked using a separate program that our experimental outputs are out-connected from the source.

We use $\kappa = 2$ in our experiments in the two dimensional plane. Tables 3.1 and 3.2 give the percent improvement over MST and the runtimes for the compared algorithms. For each instance size, we generated 50 uniformly random instances, and 50 instances using the propellant setting of the GenSeN topology generator [16]. In the table, the size of an instance is followed by r if uniformly random, and p if propellant. CPU is used to denote the CPU-time measured in seconds.

The results show that the exact algorithm has a practical running time up to 30 nodes, and produces an average improvement over MST of 22-39%. Post-processing is useful and fast enough. The Shortest-Paths based algorithm underperforms frequently. By itself, the Relative-Greedy algorithm does best in average, but we run into instances where its output is worse than the MST output; this can be explained since Relative-Greedy improves over the cost of the minimum spanning tree, and the power of a tree could be much less than it's cost (up to $(1/6)$-th, in the two-dimensional Euclidean input model). This is the result with multi-threading and a few other optimizations

Table 3.1. Average percent improvement over the MST (%) and runtime in seconds (CPU, multicore) for the compared algorithms.

n	MST-P		BIP		BIP-P		SPT-P		RG-P	
	%	CPU	%	CPU	%	CPU	%	CPU	%	CPU
20r	1.23	0.00	6.45	0.00	8.13	0.00	-3.13	0.00	10.34	0.00
20p	3.13	0.00	7.88	0.00	8.45	0.00	6.12	0.00	7.77	0.00
25r	5.65	0.00	8.77	0.00	8.98	0.00	5.33	0.00	9.14	0.00
25p	7.46	0.00	6.45	0.00	8.35	0.00	8.06	0.00	9.36	0.00
30r	8.17	0.00	4.31	0.00	5.33	0.00	3.66	0.00	10.66	0.00
30p	6.43	0.00	7.22	0.00	8.45	0.00	5.01	0.00	9.31	0.00
40r	5.34	0.00	10.31	0.00	11.14	0.00	4.52	0.00	10.11	0.00
40p	8.34	0.00	6.58	0.00	7.34	0.00	4.03	0.00	16.71	0.00
50r	4.66	0.00	7.71	0.00	8.23	0.00	3.03	0.00	15.34	0.00
50p	6.53	0.00	7.11	0.00	8.32	0.00	2.18	0.00	11.88	0.00
60r	3.89	0.00	10.66	0.00	11.54	0.00	6.21	0.00	13.98	0.00
60p	5.76	0.00	8.44	0.00	10.14	0.00	-4.45	0.00	10.56	0.00
70r	5.41	0.00	6.53	0.00	7.91	0.00	-6.38	0.00	15.32	0.00
80r	4.12	0.00	7.86	0.00	10.53	0.00	-5.12	0.00	12.37	0.00
90r	6.23	0.00	8.98	0.00	9.28	0.00	-4.57	0.00	14.43	0.00
100r	5.12	0.00	7.24	0.00	10.31	0.00	-3.64	0.00	12.13	0.00
200r	4.14	0.00	8.13	0.00	10.12	0.00	-0.21	0.00	12.47	0.00
500r	4.56	0.00	4.14	0.00	8.94	0.00	-3.85	0.00	12.88	0.24
1000r	5.12	0.00	6.64	0.00	7.16	0.00	-1.71	0.00	13.13	1.11
2000r	3.14	0.00	6.15	0.00	7.45	0.00	-3.41	0.00	11.45	5.78

Our data shows that the algorithms are varied enough that one should try all the practical ones and pick the best output. Table 3.2 includes MIN-E, which is obtained by running all the existing fast algorithms (including post-processing), and picking the best (the running time is pretty much the sum of the running times, and is dominated on large instances by the $O(mn)$-time HG-P, much slower than the $O(n^2)$-time algorithms MST, BIP, or SP), and MIN-A, which is obtained the same way using our two new fast variants, plus all the existing fast algorithms. For 2000 nodes, the Greedy-Spider algorithm was too slow, and only a heuristic based on it (no proven approximation ratio) was included in MIN-A. The improvement over MST of MIN-A is in the range 13-24%, and is circa 50% more than the improvement of MIN-E.

We also present in tables for 3.3 and 3.4 for $\kappa = 5$. While not identical to the results obtained for $\kappa = 2$, we can draw the same conclusions from the experiments. Many algorithms work so fast on these instances that the system time.

Table 3.2. Average percent improvement over the MST (%) and runtime in seconds (CPU, multicore) for the compared algorithms.

n	IP		HG-P		MIN-E		GS-P		MIN-A	
	%	CPU	%	CPU	%	CPU	%	CPU	%	CPU
20r	28.3	1.4	7.34	0.00	12.13	0.00	5.42	0.00	16.02	0.00
20p	27.4	2.3	8.23	0.00	9.08	0.00	6.33	0.00	13.57	0.00
25r	34.5	21.0	9.13	0.00	14.13	0.00	6.52	0.00	17.93	0.00
25p	35.6	23.1	7.57	0.00	13.11	0.00	12.65	0.00	22.06	0.00
30r	35.7	69.4	5.68	0.00	12.30	0.00	9.67	0.00	20.66	0.00
30p	39.4	71.5	8.43	0.00	14.33	0.00	7.56	0.00	19.01	0.00
40r	40.13	460.3	5.78	0.00	15.39	0.00	11.45	0.00	17.83	0.00
40p	42.34	430.4	9.13	0.00	14.26	0.00	8.85	0.00	21.31	0.00
50r	—	—	7.34	0.00	15.27	0.00	7.36	0.00	19.73	0.00
50p	—	—	12.13	0.00	13.35	0.00	9.74	0.00	17.88	0.00
60r	—	—	9.34	0.00	12.45	0.00	11.26	0.00	24.78	0.00
60p	—	—	8.56	0.00	15.15	0.00	8.36	0.00	19.45	0.00
70r	—	—	6.45	0.00	14.34	0.00	9.36	0.00	21.14	0.00
80r	—	—	7.31	0.00	13.64	0.00	9.73	0.00	23.92	0.00
90r	—	—	8.56	0.00	12.42	0.00	8.63	0.00	20.11	0.00
100r	—	—	8.57	0.00	13.62	0.00	11.85	0.00	18.01	0.00
200r	—	—	9.45	0.00	12.42	0.00	11.65	0.18	17.10	0.34
500r	—	—	8.15	0.48	9.25	0.52	10.52	2.03	13.7	2.77
1000r	—	—	7.83	1.45	9.26	1.67	11.52	2.78	14.61	5.56
2000r	—	—	7.46	15.43	9.23	16.21	12.43	26.77	15.43	48.76

Table 3.3. Average percent improvement over the MST (%) and runtime in seconds (CPU, multicore, $\kappa = 5$) for the compared algorithms.

n	MST-P		BIP		BIP-P		SPT-P		RG-P	
	%	CPU	%	CPU	%	CPU	%	CPU	%	CPU
20r	2.27	0.00	6.12	0.00	8.69	0.00	1.38	0.00	9.31	0.00
20p	4.32	0.00	8.03	0.00	8.95	0.00	6.42	0.00	9.01	0.00
25r	4.14	0.00	7.98	0.00	8.62	0.00	5.64	0.00	9.59	0.00
25p	8.05	0.00	6.06	0.00	7.59	0.00	7.97	0.00	10.76	0.00
30r	7.92	0.00	5.35	0.00	5.75	0.00	3.95	0.00	11.51	0.00
30p	5.91	0.00	6.71	0.00	7.68	0.00	4.61	0.00	10.42	0.00
40r	5.44	0.00	8.92	0.00	9.14	0.00	4.11	0.00	11.32	0.00
40p	7.67	0.00	6.44	0.00	7.92	0.00	4.03	0.00	18.88	0.00
50r	5.93	0.00	7.78	0.00	8.72	0.00	3.15	0.00	17.94	0.00
50p	6.13	0.00	10.25	0.00	8.81	0.00	2.33	0.00	13.34	0.00
60r	5.23	0.00	11.02	0.00	9.84	0.00	5.77	0.00	15.65	0.00
60p	5.58	0.00	8.01	0.00	9.84	0.00	2.36	0.00	12.35	0.00
70r	5.03	0.00	5.87	0.00	8.36	0.00	3.93	0.00	15.32	0.00
80r	4.79	0.00	8.41	0.00	9.68	0.00	-4.22	0.00	12.98	0.00
90r	6.66	0.00	8.17	0.00	9.27	0.00	-5.38	0.00	16.32	0.00
100r	4.71	0.00	6.66	0.00	9.27	0.00	-6.13	0.00	14.19	0.00
200r	4.34	0.00	7.56	0.00	8.72	0.00	-3.23	0.00	12.96	0.00
500r	4.19	0.00	7.42	0.00	8.31	0.00	-1.45	0.00	14.81	0.28
1000r	4.71	0.00	8.97	0.00	7.73	0.00	-3.34	0.00	14.18	1.32
2000r	3.26	0.00	7.21	0.00	7.22	0.00	-4.54	0.00	13.05	5.44

Table 3.4. Average percent improvement over the MST (%) and runtime in seconds (CPU, multicore, $\kappa = 5$) for the compared algorithms.

n	IP		HG-P		MIN-E		GS-P		MIN-A	
	%	CPU	%	CPU	%	CPU	%	CPU	%	CPU
20r	38.58	1.31	6.75	0.00	9.88	0.00	8.87	0.00	14.73	0.00
20p	36.03	2.13	8.47	0.00	9.26	0.00	9.51	0.00	15.57	0.00
25r	36.91	21.42	8.33	0.00	13.98	0.00	9.91	0.00	17.57	0.00
25p	36.31	22.17	7.64	0.00	13.11	0.00	9.38	0.00	21.63	0.00
30r	37.12	70.09	7.16	0.00	13.28	0.00	9.18	0.00	20.59	0.00
30p	39.79	75.79	8.93	0.00	14.91	0.00	10.01	0.00	19.62	0.00
40r	42.53	418.87	6.43	0.00	15.23	0.00	10.87	0.00	21.07	0.00
40p	43.18	447.61	8.31	0.00	13.54	0.00	9.23	0.00	22.85	0.00
50r	—	—	9.56	0.00	14.81	0.00	9.72	0.00	20.91	0.00
50p	—	—	9.13	0.00	13.08	0.00	10.03	0.00	21.62	0.00
60r	—	—	8.18	0.00	12.94	0.00	11.59	0.00	23.29	0.00
60p	—	—	8.21	0.00	14.39	0.00	10.52	0.00	20.22	0.00
70r	—	—	7.03	0.00	14.05	0.00	10.74	0.00	20.57	0.00
80r	—	—	8.65	0.00	14.86	0.00	9.63	0.00	25.83	0.00
90r	—	—	9.07	0.00	11.34	0.00	9.82	0.00	19.94	0.00
100r	—	—	8.22	0.00	13.21	0.00	12.79	0.00	18.19	0.00
200r	—	—	9.07	0.00	12.79	0.00	11.31	0.18	15.56	0.24
500r	—	—	8.15	0.51	10.88	0.55	10.88	1.97	15.46	2.83
1000r	—	—	8.45	1.69	10.81	1.72	11.63	2.75	15.93	5.82
2000r	—	—	8.23	16.31	10.52	16.37	12.33	27.03	16.66	48.84

CHAPTER 4

TWO-LEVEL POWER PROBLEM

4.1 Introduction

Two variants of this problem will be discussed in this section. One is the symmetric variant (Min-Power connectivity problem). The power assignment induces a simple undirected graph $H(p)$ on vertex set V given by $\{x, y\} \in E(H(p))$ if and only if the edge $\{x, y\} \in E$ and $p(x) \geq c(xy)$ and $p(y) \geq c(xy)$. Then the goal is to minimize the *total power* subject to $H(p)$ being connected. The other one is the asymmetric variant (Min-Power strong connectivity problem). The power assignment induces a simple directed subgraph $H(p)$ on vertex set V where $xy \in E(H(p))$ if and only if the arc $xy \in E$ and $p(x) \geq c(xy)$. The goal is to minimize the *total power* subject to $H(p)$ being strongly connected. Both of these problems have been shown to be NP-Complete in [50, 23].

We assume that $c : E \to \{A, B\}$ where $0 \leq A \leq B$. This corresponds to a situation where all wireless nodes operate on one of two power levels. In practice, network nodes cannot operate at an arbitrary power level, but rather have fixed operating powers. We consider networks that have each node able to operate at low or high power, transmitting at short or long range. This problem is equivalent to minimizing the number of maximum power nodes, which allows us to assume $c : E \to \{0, 1\}$ without loss of generality.

Previous works on the symmetric and asymmetric power assignment problems have presented 1.5 and 11/7≈1.571-approximation algorithms respectively [55, 1]. Although these algorithms are polynomial, their runtimes are high degree polynomials which prevents them from being used in many cases. Many previous papers have given faster variations of their approximation algorithms for these problems. We will use $n = |V|$ and $m = |E|$. On the symmetric problem, a 5/3-approximation algorithm has been published with claimed $O(nm\alpha(n))$ runtime [50]. The asymmetric problem has a 7/4-approximation

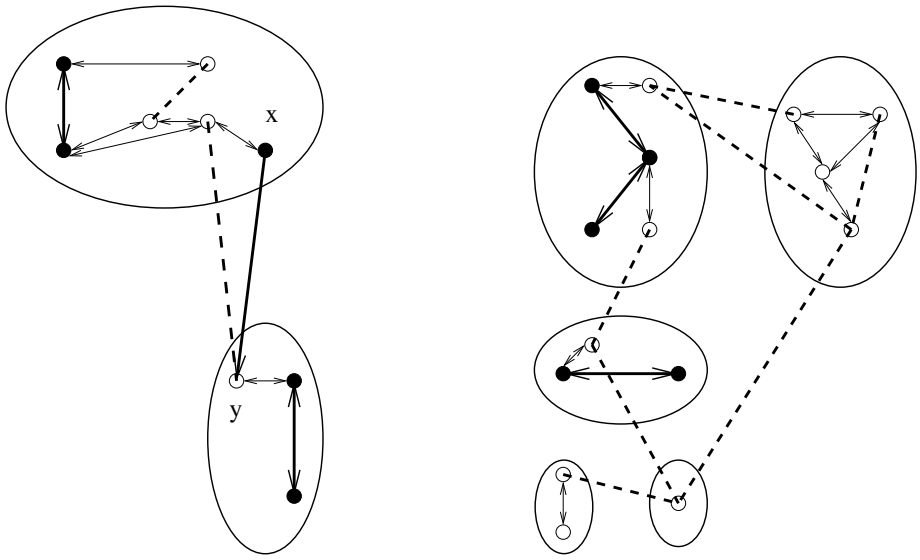

Figure 4.1. Left: a set S that is not impeccable. Black nodes have power 1 and make up S. All other nodes have power 0. Thin arrows are edges (arcs) of $H(S)$ of cost 0, thicker arrows are edges (arcs) of $H(S)$ of cost 1, and dashed segments are edges of E of cost 1 with no corresponding connection in $H(S)$. Ellipses represent connected(strongly connected) components of $H(S)$. S is not impeccable as x has power one, y has power zero, and they are in different components. The arc xy is only included in $H(S)$ for the asymmetric problem. Right: an impeccable set.

with claimed runtime of $O(n^2)$[23, 13].

We give a fast algorithm for both of these problems. In section 4.2, we present our algorithm and then prove its 5/3-approximation bound. In sections 4.3 and 4.4, we show that our algorithm is able to be implemented in $O(m\alpha(n))$ time for the symmetric and asymmetric problems respectively.

4.2 General Algorithm

Our general algorithm will iteratively select groups of vertices that if assigned power one, their separate components will merge into a single component. This approach follows from the concept of perfect sets defined in [13]. In Section 4.2.1, we formalize the definitions related to perfect sets and then in Section 4.2.2 we present our algorithm and prove its approximation bound and tightness.

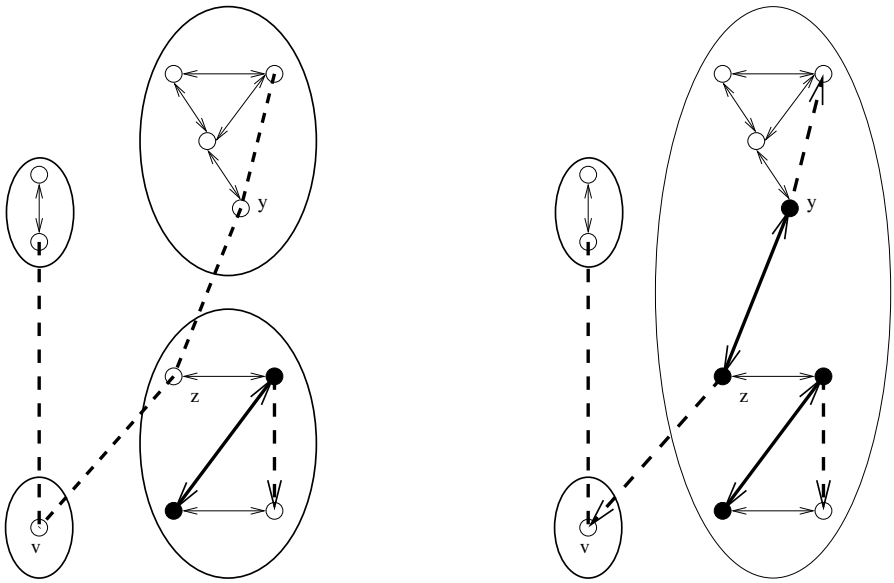

Figure 4.2. Left: The set $Q = \{y, z\}$ is quasi-perfect w.r.t. S, as explained next. Right: $H(Q \cup S)$. All the vertices of Q are in the same strongly connected component of $H(Q \cup S)$. However, Q is not perfect since z has power one and v has power zero, but they are in different connected(strongly connected) components of $H(Q \cup S)$. The dashed arc zv and the dashed arc leaving y are only included in $H(S \cup Q)$ for the asymmetric problem.

4.2.1 Definitions. Since we only consider graphs where all arcs have either cost 0 or 1, we can replace the notation of a power assignment with a set of maximum power vertices. Given a set of vertices $S \subseteq V$, we define the function $p^S : V \to \{0, 1\}$ as follows: $p^S(u) = 1$ if $u \in S$ and $p^S(u) = 0$ if $u \in V \setminus S$. We will then abuse notation and write $H(S)$ instead of $H(p^S)$. For the symmetric problem $H(S)$ is an undirected graph and for the asymmetric problem $H(S)$ is a directed graph. The distinction between these two functions will always be obvious from context.

For any $H(S)$, we will use $Comp(v)$ for $v \in V$ to be the connected (strongly connected) component of v. Initially the components of $H(\emptyset)$ are defined by the zero cost arcs. We define the component graph of some $H(S)$ to be a graph $G' = (V', E')$ describing the relationship between all current components as follows. V' is the set of connected (strongly connected) components of $H(S)$. E' has an edge $\{Comp(x), Comp(y)\}$ if and

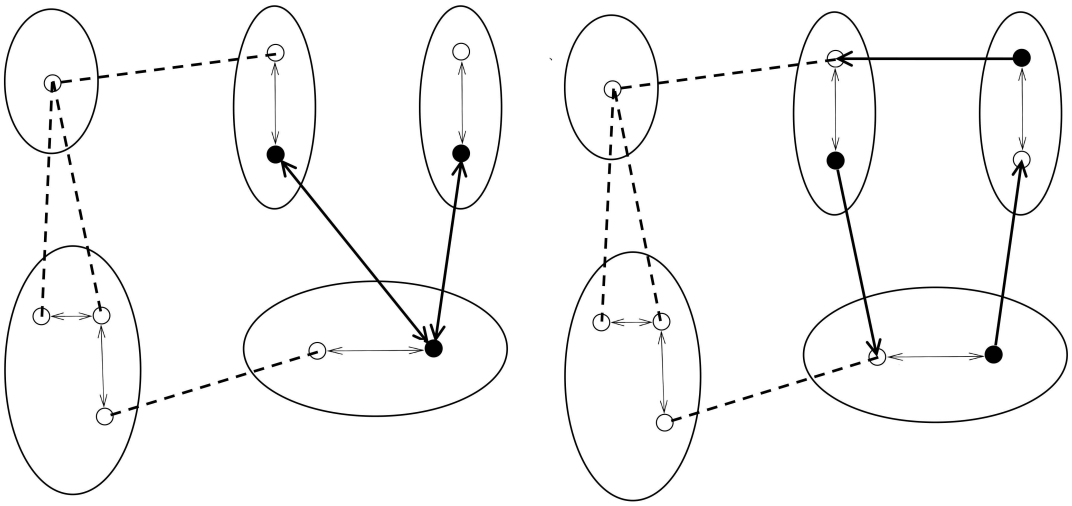

Figure 4.3. Set Q are black vertices. Left: A perfect set Q that is also symmetric. Right: A perfect set Q that is not symmetric since the graph induced by Q is not strongly connected.

only if $\{x, y\} \in E\ Comp(x) \neq Comp(y)$. For the purposes of our algorithms, we also associate all the $\{x, y\}$ endpoints in V with each edge added to the component graph. Whenever we refer to a vertex $u' \in V'$ of the component graph, we will use a prime mark to distinguish it from vertices of the original graph. Define $V(v') = \{v | v \in V \cap Comp(v) = v'\}$ for a vertex $v' \in V'$. Define $E(\{u', v'\}) = \{\{u, v\} | \{u, v\} \in E \cap Comp(u) = u' \cap Comp(v) = v'\}$ for an edge $\{u', v'\} \in E'$. Define $V(\{u', v'\})$ as the set of end points of the edges in $E(\{u', v'\})$. Define $Comp(Q)$ as $\{v' | v \in Q \cap Comp(v) = v'\}$ for $Q \subseteq V$.

For any $S \subseteq V$, S is *impeccable* if and only if no $\{u, v\} \in E$ exists with $p(u) = 1$, $p(v) = 0$ and $Comp(u) \neq Comp(v)$ (See Figure 4.1 borrowed from [13] for an example). Note that $H(\emptyset)$ is impeccable.

We say that $Q \subseteq V$ is *quasiperfect* with respect to impeccable S if the following holds: For all $u' \in V'$, we have that $|Q \cap V(u')| \leq 1$, and all vertices of Q are in the same connected (strongly connected) component of $H(Q \cup S)$. Therefore, adding a quasiperfect Q to our current impeccable solution will contract $|Q|$ components into one.

However, adding a quasiperfect set may make our current solution no longer impeccable. If Q is quasiperfect w.r.t. impeccable S and $Q \cup S$ is impeccable, then Q is called *perfect* w.r.t. S (See Figure 4.2 borrowed from [13] for an example). A set Q is *symmetric* w.r.t. S (See Figure 4.3 for an example) if Q is perfect w.r.t. S and the induced subgraph on the vertices of Q in $H(Q \cup S)$ is connected (Strongly connected).

4.2.2 Algorithm and Analysis. Our approach to both symmetric and asymmetric problems will maintain an impeccable set, S, that grows until $H(S)$ is connected (strongly connected). This is accomplished by repeatedly adding sets perfect w.r.t. S into S. Adding such a set, Q, to S will decrease the number of components in $H(S)$ by $|Q| - 1$. The greedy heuristic of adding larger perfect sets to S first follows from this idea. Calinescu's $\approx$1.61-approximation of the asymmetric problem uses this heuristic by adding perfect sets of size k or more, then size $k-1$... then size 4, and finally solving the problem exactly once only perfect sets of size 2 and 3 remain [13].

Following this heuristic, our algorithm for both problem variations, shown in Algorithm 2, first adds perfect sets where $|Q| \geq 4$. Once no sets of size four or more exist, it adds sets of size three, and then sets of size two. For the symmetric case, this approach is essentially the same as the algorithm in [50] and our proof of its approximation bound follows closely from their proofs. We begin with sets of size four or more because we are unable to maintain $O(m\alpha(n))$ runtime while searching exclusively for larger perfect sets. We denote S at after i sets are added as S_i.

Theorem 4.2.1. *Algorithm 1 has a 5/3 approximation ratio.*

Given any graph G, we denote an optimal solution as $OPT(G)$, Algorithm 2's solution as $A(G)$, the number of connected (strongly connected) components in $H(S)$ before the first loop as K, before the second loop as M, before the third loop as P. We will denote each set of size four or more added as k_i and each set of size three as m_i. It follows

1: Set $i = 0$; $S_0 = \emptyset$
2: **while** $\exists$Q perfect w.r.t. S_i and $|Q| \geq 4$ **do**
3: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
4: **end while**
5: **while** $\exists$Q perfect w.r.t. S_i and $|Q| = 3$ **do**
6: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
7: **end while**
8: **while** $\exists$Q perfect w.r.t. S_i and $|Q| = 2$ **do**
9: $\quad S_{i+1} := S_i \cup Q$; $i := i + 1$;
10: **end while**
11: **return** S_i

Algorithm 2: General Approximation Algorithm

immediately that

$$|A(G)| = \sum_i |k_i| + \sum_i |m_i| + 2(P-1) \tag{4.1}$$

$$\leq \frac{4}{3}\sum_i (|k_i| - 1) + \frac{3}{2}\sum_i (|m_i| - 1) + 2(P-1) \tag{4.2}$$

$$= \frac{4(K-M)}{3} + \frac{3(M-P)}{2} + 2(P-1) \tag{4.3}$$

$$= \frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}P - 2 \tag{4.4}$$

where (4.2)=(4.3) holds because the first loop contracts $K - M$ connected (strongly connected) components by reducing the number of components by $|k_i|-1$ for each k_i found. Similarly the second loop contracts $M - P$ connected (strongly connected) components.

Lemma 4.2.1. $|OPT(G)| \geq K$

Proof. Any solution has a vertex in each connected (strongly connected) component of $H(\emptyset)$. □

Consider the reduced problem of strongly connecting $H(S_i)$ after the first loop has finished. An optimal solution to the reduced problem will add some perfect sets of size three and some sets of size two since no perfect sets of size four or more can exist in this

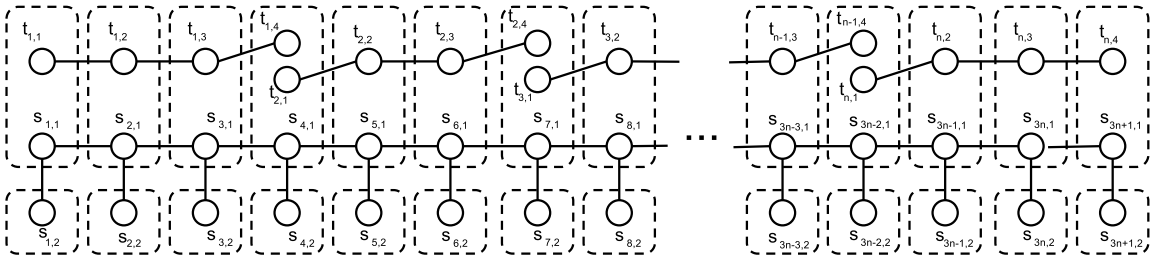

Figure 4.4. An example for tightness. Dotted boxes represent the connected (strongly connected) components of $H(\emptyset)$. All edges shown have cost 1.

process (by lemma 5 in [13]). We will denote the number of sets of size two used in the optimal solution as $L-1$. Note that $M \geq L-1$.

Lemma 4.2.2. $|OPT(G)| \geq \frac{3}{2}M + \frac{1}{2}L - 2$

Proof. Strongly connecting $H(\emptyset)$ requires at least as many vertices as strongly connecting $H(S_i)$ in the reduced problem, which requires $\frac{3}{2}(M-L)$ vertices from sets of size three and $2(L-1)$ vertices from sets of size two. □

Lemma 4.2.3. $\frac{M-P}{2} \geq \frac{M-L}{4}$, *therefore* $P \leq \frac{1}{2}M + \frac{1}{2}L$

Proof. Consider the $\frac{M-L}{2}$ perfect sets of size three added by an optimal solution to the reduced problem. We claim that when Algorithm 2 greedily adds a perfect set, Q, of size three at most two of the optimal perfect sets are no longer available. Those sets become unavailable since their size reduced to 1 or 2. For any perfect set Q' of size 3 used in optimal, if $|Comp(Q) \cap Comp(Q')| < 2$, Q' in optimal will not be affected after algorithm 2 add Q to S. If $|Comp(Q) \cap Comp(Q')| = 3$, then only Q' in the optimal will be affected. Now consider $|Comp(Q) \cap Comp(Q')| = 2$ and Q_1, Q_2 are 2 sets used in optimal. $|Comp(Q_1) \cap Comp(Q_2)| \leq 1$ because Q_1, Q_2 are in the optimal solution. Moreover, at most 2 sets in optimal need to be considered since 3 sets can not satisfy all the conditions above. Therefore we have only one possibility: $|Comp(Q_1) \cap Comp(Q_2)| = 1$, $|Comp(Q) \cap Comp(Q_1)| = 2$ and $|Comp(Q) \cap Comp(Q_2)| = 2$. This is the only case

which can happen. After Q is added by algorithm 2. Both Q_1 and Q_2 have size less than 3. By all the cases discussed above, we can conclude that when Algorithm 2 greedily adds a perfect set, Q, of size three at most two of the optimal perfect sets are no longer available. □

By taking a convex combination of Lemmas 4.2.1 and 4.2.2, we find that

$$|OPT(G)| \geq \frac{4}{5}K + \frac{1}{5}(\frac{3}{2}M + \frac{1}{2}L - 2) \tag{4.5}$$

$$= \frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{2}{5} \tag{4.6}$$

From equations (4.4), (4.6) and Lemma 4.2.3, it follows that

$$\frac{|A(G)|}{|OPT(G)|} \leq \frac{\frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}P - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{2}{5}} \tag{4.7}$$

$$\leq \frac{\frac{4}{3}K + \frac{1}{6}M + \frac{1}{2}(\frac{1}{2}M + \frac{1}{2}L) - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{2}{5}} \tag{4.8}$$

$$= \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{3}{10}M + \frac{1}{10}L - \frac{2}{5}} \tag{4.9}$$

$$= \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{5}{20}M + \frac{1}{20}M - \frac{1}{20}L + \frac{3}{20}L - \frac{2}{5}} \tag{4.10}$$

$$\leq \frac{\frac{4}{3}K + \frac{5}{12}M + \frac{1}{4}L - 2}{\frac{4}{5}K + \frac{5}{20}M + \frac{3}{20}L - \frac{2}{5}} \leq \frac{5}{3} \tag{4.11}$$

Therefore Algorithm 2 has a 5/3 approximation ratio. □

Theorem 4.2.2. *For both the symmetric and asymmetric problem, the 5/3 approximation ratio of Algorithm 2 is tight.*

Proof. We prove this by giving an class of graphs that Algorithm 2 may assign arbitrarily close to 5/3 times the number of vertices to maximum power as the optimal solution. Consider the example given in Figure 4.4. This example has $6n + 2$ connected (strongly connected) components that could be optimally connected with $6n + 2$ maximum power

vertices, namely all vertices s_{ij}. Algorithm 2 could choose the set $\{t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}\}$ for each iteration i=1,..,n of the first loop. After this, the only remaining perfect sets are the pairs $\{s_{i,1}, s_{i,2}\}$. Therefore, Algorithm 2 would add each of these pairs in its third loop. This example could have $10n + 2$ vertices assigned maximum power. Therefore, we have an approximation ratio of $\frac{10n+2}{6n+2}$, which asymptotically approaches 5/3. □

The tightness example in [50] could also be used to show our approximation is tight.

Theorem 4.2.3. *Algorithm 2 has $O(m\alpha(n))$ runtime implementations for both the symmetric and asymmetric problems, where $\alpha(n)$ is the inverse Ackermann function.*

Theorem 4.2.3 is our main result. We first show how to implement Algorithm 2 for the symmetric problem in Section 4.3, and then expand on this method to give an implementation for the asymmetric problem in Section 4.4.

4.3 Symmetric Implementation

For this version of the problem, we only need to consider symmetric sets instead of perfect sets.

Claim 4.3.1. *Every perfect set in the symmetric problem is a symmetric set.*

Proof. Consider a perfect set Q in the symmetric problem. Since S is impeccable, each edge in $H(S \cup Q)$ but not in $H(S)$ must have both endpoints in Q. Since Q is perfect w.r.t S, $H(S \cup Q)$ has the vertices of Q connected. Since each vertex of Q is in a different component of $H(S)$, we can conclude that the induced subgraph on the vertices of Q in $H(Q \cup S)$ is also connected. □

Since we only need to consider symmetric sets, we can search for connected subgraphs with each vertex in a different component of $H(S_i)$ with size four or more, then size three, and finally size two. To allow us to quickly check the component of a vertex, we

will maintain a Union Find data structure grouping each vertex with all others in the same connected (strongly connected) component of $H(S_i)$. Initializing this data structure only requires computing the connected (strongly connected) components of $H(\emptyset)$, which can be done in $O(m)$. This data structure supports both union and find operations in $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function [61].

Any connected subgraph with at most one vertex per component of $H(S_i)$ will have a quasiperfect vertex set. We claim that every quasiperfect set is a subset of some perfect set. Consider the simple construction, called $Augment(Q)$, which takes a quasiperfect Q and will return a set Q' where $Q \subseteq Q'$ and Q' is perfect w.r.t. S.

1: Union components of Q into single component
2: **while** Q is not perfect w.r.t. S_i **do**
3: $Q = Q \cup \{v\}$ where $u \in Q$, $\{u, v\} \in E$, and $Comp(u) \neq Comp(v)$ in $H(S_i \cup Q)$
4: Union component of v with component of u
5: **end while**
6: **return** Q

When a connected subgraph is found, it will be augmented to make it perfect and then added to our solution, S_i. Further, the Union-Find data structure will be updated to associate all vertices in components with a vertex in Q together. Every vertex processed by Augment will be added to S_i immediately afterwards. Therefore each vertex is inspected at most once. We can conclude that total time spent augmenting sets is bounded by $O(\sum d(v)\alpha(n)) = O(m\alpha(n))$. For all following implementations we will ignore the cost of the augment function, since this argument can be used to bound its total cost over the entire algorithm.

4.3.1 Implementation of Loop 1 (lines 2-4). For the symmetric problem, the first loop needs to find symmetric sets of size four or more. We will process this loop in two phases, first adding sets symmetric w.r.t. S_i that have a vertex with degree three or more in their

connected subgraph. Such sets can be easily found by augmenting the set $\{v\}$ for any v with three neighbors in other components (lines 1-5 below).

When no more such vertices exist, all sets symmetric w.r.t. S_i must either have a connected subgraph that is a cycle or a path, because all vertices are adjacent to at most two other components of $H(S_i)$. Note that every cycle or path on at least four vertices contains a path of length three. Therefore all remaining symmetric sets of size four or more have such a path in its connected subgraph. We can then find all of these structures in linear time by first precomputing the at most two components adjacent to each vertex, and then checking if each edge can be the middle edge of such a path (lines 6-13 below).

1: **for all** $u \in V$ **do**
2: **if** u adjacent to three or more other components in $H(S)$ **then**
3: $S_{i+1} := S_i \cup Augment(\{\mathtt{u}\})$; $i := i + 1$;
4: **end if**
5: **end for**
6: **for all** $u \in V$ **do**
7: $Adjacent[u] := \{Comp(v) | \{u, v\} \in E, Comp(v) \neq Comp(u)\}$
8: **end for**
9: **for all** $\{u, v\} \in E$ **do**
10: **if** $|Adjacent[u] \cup Adjacent[v]| = 4$ **then**
11: $S_{i+1} := S_i \cup Augment(\{u, v\})$; $i := i + 1$;
12: **end if**
13: **end for**

The first two loops will each require iterating through the adjacency list of every vertex doing a find operation on the other end of each edge. Therefore these loops can be implemented in $O(\sum d(v)\alpha(n)) = O(m\alpha(n))$. The final loop inspects each edge and checks if its endpoints are adjacent to two distinct components (four components including

the components of the endpoints), taking $O(m\alpha(n))$.

4.3.2 Implementation of Loop 2 (lines 5-7). All symmetric sets of size three have a connected subgraph that is either a path or cycle. Both of these can be found in near linear time by searching for the necessary vertices with degree two.

1: **for all** $u \in V$ **do**
2: **if** u adjacent to two other components in $H(S_i)$ **then**
3: $S_{i+1} := S_i \cup Augment(\{u\})$; $i := i + 1$;
4: **end if**
5: **end for**

This loop only needs to read the adjacency list of each vertex, checking the number of neighboring components. So we can implement it in $O(m\alpha(n))$.

4.3.3 Implementation of Loop 3 (lines 8-10). After Loop 2 of Algorithm 2 completes, all symmetric sets are either a single vertex or a pair of adjacent vertices. So we only need to find edges with both endpoints in different components of $H(S_i)$.

1: **for all** $\{u, v\} \in E$ **do**
2: **if** $Comp(u) \neq Comp(v)$ **then**
3: $S_{i+1} := S_i \cup \{u, v\}$; $i := i + 1$;
4: Union the components of u and v
5: **end if**
6: **end for**

Note: we have to union the component's of u and v directly since we do not use the Augment function. This loop's implementation has $O(m\alpha(n))$ runtime. Therefore Algorithm 2 can be implemented in $O(m\alpha(n))$ for the symmetric problem. □

4.4 Asymmetric Implementation

For the asymmetric version of this problem, we need to consider all perfect sets, not only symmetric sets. We will use the concept of a component cycle as a new structure for finding perfect sets.

We define C to be a *component cycle* if the following holds: $C = \{c_1, c_2, \ldots, c_k\}$ is quasiperfect w.r.t. S, and for $1 \leq i < k$ there is an edge $\{c_i, u\} \in E$ where $Comp(u) = Comp(c_{i+1})$ and there is edge $\{c_k, u\} \in E$ where $Comp(u) = Comp(c_1)$. We will then use the following lemma to find perfect sets.

Lemma 4.4.1. *Given Q perfect w.r.t. S, Q has a subset that is a component cycle of size at least two.*

Proof. Let Q be perfect w.r.t. some S. Let u and v be different vertices of Q. Since Q is perfect, there must be paths from u to v and vice versa in $H(S \cup Q)$. Consider the finite sequence of vertices in Q in each of these paths. If these sequences have no common vertices other than u and v, then combined they make a component cycle. Otherwise, let w be the first vertex of Q in the u,v-path other than u that is common to both paths. Then combining the sequence of vertices in Q from u to w with the vertices from w to u will create a component cycle of size at least two. □

Since any set perfect w.r.t. S_i has a component cycle, we can extend the implementation for the symmetric problem to build perfect sets based on their component cycles. We still maintain the Union-Find data structure grouping vertices by their component in $H(S_i)$. Our approach to the asymmetric problem is given in Algorithm 3. The first four loops of Algorithm 3 (lines 2-13) add perfect sets with size at least four. Then loops five and six (lines 14-19) will handle all perfect sets of size three. The final loop (lines 20-22) will finish connecting $H(S_i)$ by adding the remaining size two sets.

The asymmetric approximation algorithm has considerable overlap with the simpler algorithm for the symmetric variant. Fast implementations of loops 1, 5 and 7 of Algorithm

1: Set $i = 0$; $S_0 = \emptyset$
2: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| \geq 4$ **do**
3: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 1
4: **end while**
5: **while** $\exists Q$ perfect w.r.t. S_i and $\exists C \subseteq Q$ component cycle and $|C| \geq 4$ **do**
6: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 2
7: **end while**
8: **while** $\exists Q$ perfect w.r.t. S_i, $|Q| \geq 4$ and $\exists C \subseteq Q$ component cycle and $|C| = 3$ **do**
9: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 3
10: **end while**
11: **while** $\exists Q$ perfect w.r.t. S_i, $|Q| \geq 4$ and $\exists C \subseteq Q$ component cycle and $|C| = 2$ **do**
12: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 4
13: **end while**
14: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| = 3$ **do**
15: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 5
16: **end while**
17: **while** $\exists Q$ perfect w.r.t. S_i and $\exists C \subseteq Q$ component cycle and $|C| = 3$ **do**
18: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 6
19: **end while**
20: **while** $\exists Q$ symmetric w.r.t. S_i and $|Q| = 2$ **do**
21: $S_{i+1} := S_i \cup Q$; $i := i + 1$; //Loop 7
22: **end while**
23: **return** S_i

Algorithm 3: 5/3-Approximation of Asymmetric Problem

3 have already been shown in Section 4.3. So we only need to show that Loops 2, 3, 4 and 6 can be implemented under our runtime bound.

These four loops are all based on finding component cycles. Each vertex in a component cycle could have arcs going to a component that is not in the cycle. We will refer to any arc that goes from a quasiperfect set Q to a vertex in a different component of $H(Q \cup S)$ as a *branch*. After the first loop of Algorithm 3 finishes, we can conclude multiple constraints on the structure of all branches, shown in the following claim.

Claim 4.4.1. *After loop 1 of Algorithm 3 finishes, for any Q quasiperfect w.r.t. S_i, $q \in Q$ and branch qu: All other branches qv have $Comp(u) = Comp(v)$, and u has no adjacent components other than ones with a vertex in Q.*

Proof. If a branch qv existed leading to a different component, then augmenting $\{q\}$ will create a symmetric set with at least four vertices. If u had another adjacent component, then augmenting $\{q, u\}$ will create a symmetric set with at least four vertices. Neither of these can occur after the first loop finishes. □

4.4.1 Implementation of Loop 2 (lines 5-7). The second loop of Algorithm 3 will contract perfect sets that contain a component cycle of size four or more until none exist. To implement this, we will use a modified depth first search algorithm on the component graph. When our search finds a cycle of size four or more, the DFS will find a corresponding component cycle and build a perfect set containing that component cycle. Then it will add this set into S, contract the components of the perfect set, and continue processing the DFS.

The DFS will start at an arbitrary vertex of the component graph called $root \in V'$ that is assigned $Depth(root)$=0. When a new component u' is reached from some component v', it will have $Discovered(u')$ change from $false$ to $true$. Further, it will have $Parent(u') = v'$ and $Depth(u') = Depth(v') + 1$. This parent relationship defines a path at any time from the current component being processed to the root. The set of edges each component still has to process will be denoted by $E_{new}(u')$, which initially has value $E'(u')$, namely all edges incident to u' in the component graph.

As the DFS runs, there are three ways it could find a cycle of size four or more (shown in Figure 4.5 (a)). A Type 1 cycle is found when a back edge extends more than two components up the path. A Type 2 cycle is found when two three cycles are found in the path that share an edge. Note that the DFS may process either of these two back edges first. A Type 3 cycle is found when a single component in the path is in the middle of two three cycles.

As our DFS runs, whenever a three cycle is found, we will mark that each of its components are in a three cycle. The component with the greatest depth has $Bottom3$ set

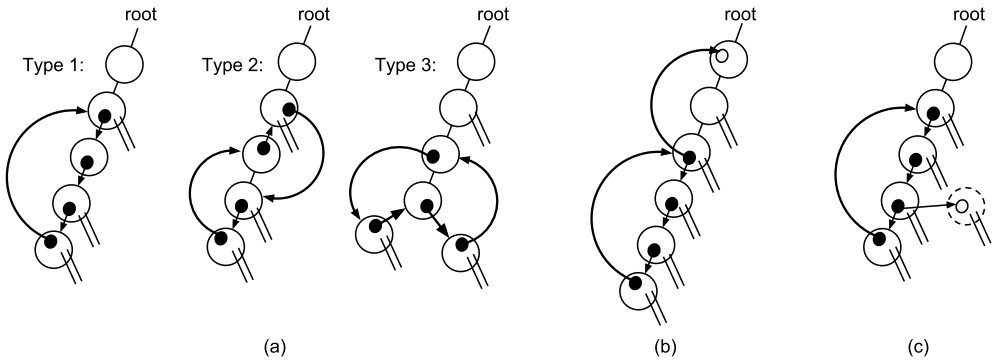

Figure 4.5. (a) The three types of cycles in the component graph contracted during the DFS. (b) A component cycle with a branch to a component in the path. (c) A component cycle with a branch to an undiscovered component (dashed).

to $true$. The middle component has $Middle3$ set to $true$. Finally the component closest to the root has $Top3$ set to $true$. These values are initially $false$ for every component. We will use $Bottom3$ and $Top3$ to find type 2 cycles and $Middle3$ to find type 3 cycles. However, for any $x' \in V'$, we only want $Top3(Parent(x'))$ to indicate that $Parent(x')$ is the top of a three cycle containing x'. To maintain this, we will reset $Top3(Parent(x'))$ to $false$ whenever we backtrack from a vertex x'.

By associating the edges that make these three cycles with each of these boolean flags, we can easily detect and construct type 2 and 3 cycles while the DFS runs. While processing a component x', if we find a three cycle and have $Bottom3(Parent(x'))$ or $Top3(Parent(x'))$ we can construct the type 2 cycle shown in Figure 4.5 (a). Similarly if we find a three cycle and have $Middle3(Parent(x'))$, we can construct the type 3 cycle shown in Figure 4.5 (a).

Whenever a cycle is found, we will expand the corresponding component cycle into a perfect set. As shown in Figure 4.5 (b) and (c), our component cycle could have two different types of branches. There could be a branch yz that leads to a component higher in the path. In which case, we can expand our component cycle into a larger quasiperfect set by adding a vertex in each component between $Comp(z)$ and the highest component with

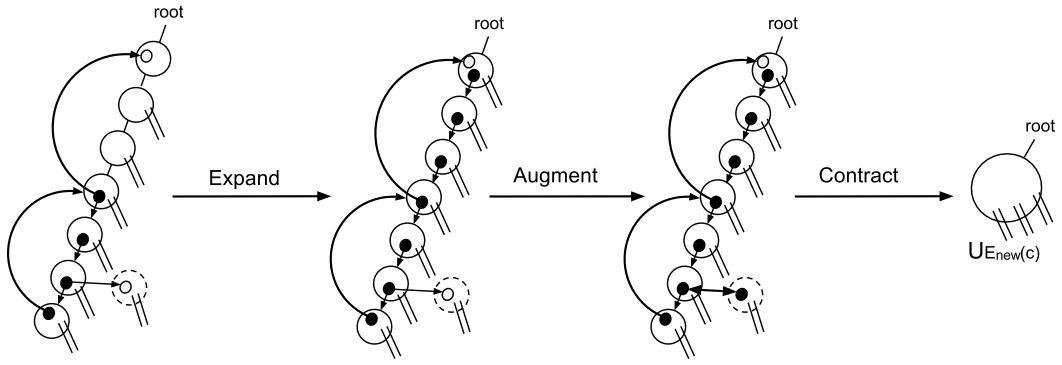

Figure 4.6. When a component cycle is found, any branch to the path can be removed as shown by the Expand operation. After Expand runs, all remaining tails to undiscovered components are handled by Augment. The final perfect set can be contracted, combining all E_{new}

.

a vertex in our current set. Iterating this expanding process must eventually terminate in a quasiperfect set with no branches into the path. This procedure will be called $Expand(Q)$, which takes a quasiperfect Q, and returns a quasiperfect Q' such that $Q \subseteq Q'$ and there are no branches from Q' to components in the path. $Expand(Q)$ is defined as follows (An example is shown in Figure 4.6):

1: $t' :=$ component with a vertex in Q of minimum depth
2: **while** Q has a branch uv and Discovered(v) **do**
3: **while** $t' \neq Comp(v)$ **do**
4: Add a vertex in $Parent(t')$ adjacent to t' into Q
5: $t' := Parent(t')$
6: **end while**
7: **end while**
8: **return** Q

The same type of runtime argument made for $Augment(Q)$ can be applied to $Expand(Q)$. Every vertex needs to be inspected for branches, and finding a branch will add more vertices that need to be inspected. However, since we will add all vertices found by expand into S_i,

we will never inspect a vertex twice. Therefore, we can amortize the cost of $Expand(Q)$ to $O(m\alpha(n))$.

The other possible type of branch goes to an undiscovered component. Once these are the only type of arcs leaving our quasiperfect set, we can augment it to get a perfect set. By Claim 4.4.1, know that augmenting a branch will only add that component (Only one component) to our quasiperfect set. Therefore we can expand and augment our component cycle to find a perfect set that only contracts consecutive vertices at the end of our DFS path and undiscovered components.

Claim 4.4.2. *Expand and Augment preserve the DFS tree.*

Proof. After Expand and Augment, t' is still a leaf and all the relation of $Parent(v)$ is maintained correctly for any $v \in V$. Then we need to show that no cycle using the edges on the DFS path is created by Expand or Augment. Assume we had a valid DFS tree, we will prove Expand and Augment will not create cycle using the edges on the DFS path. We know that Expand and Augment will only merge the consecutive components from the end of our DFS path and undiscovered components. If a new cycle is created, it must contain the contracted component t'. This cycle can not contain an edge connected to an undiscovered component since that edge must be on the DFS path. So there is another edge on the DFS path from the ancestors of t' to t' other than the edge from $Parent(t')$. If there are 2 different edges on the DFS entering the consecutive vertices from the end of DFS path, the previous DFS tree is invalid. □

When we contract a perfect set Q that merges vertices in the path from the current component x' to the highest component with a vertex of Q, t', we must modify the depth first search state to stay valid. The new component must have all of the remaining E_{new} of components of Q in the path and every edge incident to an undiscovered component of Q. This new component is the bottom of a three cycle that we have already found if and only if

t' was the bottom of a cycle before contraction. Similarly, our new component will only be the middle of a three cycle we have found if t' was the middle of a cycle that doesn't use the current child of t' in the path. The flag $Middle3(t')$ is set to $true$ the first time a three cycle is found. Thus if the cycle found when $Middle3(t')$ was marked uses the current child of t' in the path, we know no other such three cycle exists. This allows us to quickly determine if the new component should have $Middle3(t') = true$. Moreover, $Top3(Parent(t'))$ should have same value as $Middle3(t')$. Any states of $Parent(Parent(t'))$ or the vertices above on the DFS path are not affected. As with backtracking, we will set the $Top3(t')$ flag of our new component to $false$.

Our full implementation of this modified depth first search follows:

1: $Discovered(root) := true$
2: $x' := root$
3: **while** $E_{new}(x') \neq \emptyset$ **or** $x' \neq root$ **do**
4: **if** $E_{new}(x') = \emptyset$ **then**
5: $x' := Parent(x')$
6: $Top3(x') := false$
7: **else**
8: Pick $\{x', u'\} \in E_{new}(x')$
9: Remove $\{x', u'\}$ from $E_{new}(x')$
10: **if not** $Discovered(u')$ **then**
11: $Parent(u') := x'$
12: $Depth(u') := Depth(x') + 1$
13: $Discovered(u') := true$
14: $x' := u'$
15: **else**
16: **if** $Depth(x') - Depth(u') = 1$ **then**
17: Continue to Next Iteration

18: **end if**

19: **if** $Depth(x') - Depth(u') = 2$ **then**

20: **if** $Bottom3(Parent(x'))$, $Middle3(Parent(x'))$ **or** $Top3(Parent(x'))$ **then**

21: Construct a type 2 or 3 component cycle C

22: **else**

23: $Bottom3(x') := true$

24: $Middle3(Parent(x')) := true$

25: $Top3(u') := true$

26: Continue to Next Iteration

27: **end if**

28: **end if**

29: **if** $Depth(x') - Depth(u') \geq 3$ **then**

30: Construct a type 1 component cycle C

31: **end if**

32: $Q := Expand(C)$

33: $Q := Augment(Q)$

34: t' := Minimum depth component of Q

35: $E_{new}(t') := \bigcup_{q \in Q} E_{new}(Comp(q))$

36: Contract the components of Q into t'

37: $S_{i+1} = S_i \cup Q$; $i = i + 1$

38: $x' := t'$

39: $Top3(x') := false$

40: **if** $Middle3(x')$ and cycle for $Middle3(x')$ was contracted **then**

41: $Middle3(x') := false$

42: $Top3(parent(x')) := false$

43: **end if**

44: **end if**

45: **end if**

46: **end while**

Claim 4.4.3. *After the modified depth first search for loop 2 terminates, no cycle of size four or more exist in the component graph.*

Proof. Assume to the contrary that cycles of size four or more exist in the component graph after the DFS terminates. Then let C be such a cycle in the component graph of minimum size, and let x' be the last component in $V(C)$ discovered by the DFS. Then let u' and v' be the predecessor and successor of x' in C. Since there are no cross edges in a DFS of an undirected graph, we can conclude u' and v' are both above x' in the path. If u' and v' are not the $Parent(x')$ and $Parent(Parent(x'))$, then when one of the edges $\{x', u'\}$ or $\{x', v'\}$ was processed the if-statement on line 29 was entered and a type 1 cycle was found contracting multiple components of C. This contracts our assumption.

We can conclude that u' and v' must been $Parent(x')$ and $Parent(Parent(x'))$. Since they are adjacent in the path, the edge $\{u', v'\}$ exists. Since we chose C to be a minimum cycle of size four or more, C must have size exactly four because a smaller cycle is created by replacing u', x', v' in C with u', v'. We denote the fourth component of this cycle by w'. Then w' is either in the path when x' is being processed or not.

Suppose w' is in the path when x' is processed. Then w' must immediately precede u' and v' in the path, since w' being any higher would create a type 1 cycle. We consider the two three cycles v', w', u' and x', v', u' in the path. If the cycle v', w', u' is found first, we will mark $Parent(x')$ with $Bottom3 = true$. Then when processing x' we will enter the if-statement on line 20 and contract C. Alternatively, if we find the cycle x', v', u' first, we will mark $Parent(Parent(x'))$ with $Top3$. $Top3$ is only reset when we backtrack, which cannot happen before $Parent(x')$ finishes processing. Therefore, when we are process-

ing $Parent(x)$ and find the cycle v', w', u', we will enter the if-statement on line 20 and contract C. Both of these results contradict our assumption.

Finally, suppose w' was not in the path when x' was processed. Then when w' was processed, u' and v' must have been its two predecessors in the path by the same argument used for x'. The DFS must have finished processing all of w''s edges before discovering x'. So the cycle v', w', u' was found and $Parent(x')$ marked with $Middle3$ before x' was discovered. Then when the cycle x', v', u' was found, the if-statement on line 20 must have been entered and C contracted. This result also contradicts our assumption.

We can conclude that after the depth first search terminates no component cycles of size four or more exist. □

Each iteration of our DFS will consider a new edge in the graph. This bounds the number of iterations at $O(m)$ in the same way that a regular DFS takes $O(m)$ steps. Every operation in the in the loop takes $O(1)$ or $O(\alpha(n))$ except for lines 32-39. However, we have already shown that we can amortize the cost of Expand and Augment over the entire algorithm to get our $O(m\alpha(n))$ runtime. Each of lines 34-39 have an straightforward implementation with runtime $O(|Q|\alpha(n))$, which is bounded by $O(\sum |Q_i|\alpha(n)) = O(n\alpha(n))$ over the entire algorithm. It follows then that our modified DFS maintains near linear runtime, $O(m\alpha(n))$.

4.4.2 Implementation of Loop 3 (lines 8-10). The third loop of Algorithm 3 iteratively adds perfect sets of size four or more that have a component cycle of size three. Thus every cycle added by this loop must have at least one branch. We will again use a modified depth first search to find cycles, but our search is made easier since no cycles of size four or more exist after the second loop has finished.

When our new DFS finds a three cycle, we will check all of the endpoints in the original graph of edges between the three components for a branch. If a branch is found, the

DFS will expand and augment a component cycle with this branch, contract the resulting perfect set, and then resume the search. We use the same definitions for our DFS as in the second loop of Algorithm 3.

1: $Discovered(root) := true$
2: $x' := root$
3: **while** $E_{new}(x') \neq \emptyset$ **or** $x' \neq root$ **do**
4: **if** $E_{new}(x') = \emptyset$ **then**
5: $x' := Parent(x')$
6: **else**
7: Pick $\{x', u'\} \in E_{new}(x')$
8: Remove $\{x', u'\}$ from $E_{new}(x')$
9: **if not** $Discovered(u')$ **then**
10: $Parent(u') := x'$
11: $Depth(u') := Depth(x') + 1$
12: $Discovered(u') := true$
13: $x' := u'$
14: **else**
15: **if** $Depth(x') - Depth(u') = 2$ **then**
16: Set C to the three cycle $\{x', u', parent(x')\}$
17: **if** exists $v \in V(\{x', u'\}) \cup V(\{x', parent(x')\}) \cup V(\{u', parent(x')\})$ and $b \in V$ such that $\{v, b\} \in E$ and $Comp(b) \notin C$ **then**
18: Construct component cycle C' with one branch vb
19: $Q := Expand(C')$
20: $Q := Augment(Q)$
21: $t' :=$ Minimum depth component of Q
22: $E_{new}(t') := \bigcup_{q \in Q} E_{new}(Comp(q))$
23: Contract the components of Q into t'

24: $S_{i+1} = S_i \cup Q; i = i + 1$

25: $x' := t'$

26: **end if**

27: **end if**

28: **end if**

29: **end if**

30: **end while**

Claim 4.4.4. *After the modified DFS for loop 3 terminates, no component cycles of size three with at least one branch exist in* $H(S_i)$.

Proof. Assume to the contrary that there exists a component cycle $\{u, v, w\}$ with a branch ut after this DFS finishes. Let u', v', w' and t' be the components of u, v, w and t, respectively. Then u', v' and w' must follow each other immediately in the path when the last of them is processed, otherwise a cycle of size four or more exists. Since u is an endpoint in V of the edges of this cycle, the DFS will enter the if-statement on line 18 and add some perfect set contracting u', v' and w'. Therefore $\{u, v, w\}$ is not a component cycle after the DFS finishes. Contradiction. □

The runtime of this DFS follows from the same argument made for the implementation of the second loop of Algorithm 3. The only addition work is searching for a branch in the endpoints of our cycle. We can reduce this cost by precomputing the at most two components that every vertex in the original graph can be adjacent to. Then checking a single endpoint for a branch will take $O(\alpha(n))$. Thus checking for a branch in the endpoints between two components that are connected by k edges in the underlying graph will take time $O(k\alpha(n))$. We will never check for such an endpoint between the same two components again, because two distinct three cycles sharing an edge implies the existence of a cycle of size four. Thus we can bound the total time checking endpoints by

$O(\sum k_i \alpha(n)) = O(m\alpha(n))$.

4.4.3 Implementation of Loop 4 (lines 11-13). The fourth loop of Algorithm 3 will find and contract component cycles of size two that have two branches. A component cycle of size two is 2 vertices in 2 components. Our implementation will iterate through all the edges in the component graph and check if it has a branch on each end of the edge that go to different components.

As in loop 3 of Algorithm 3, we can reduce the runtime of this implementation by precomputing the at most two components that every vertex in the original graph can be adjacent to. Our implementation follows:

1: **for all** $u'v' \in E'$ **do**
2: **if** $\exists b_1, b_2 \in V$ and $u, v \in V(\{u', v'\})$ and $\{b_1, u\}, \{b_2, v\} \in E$, $Comp(u) = u', Comp(v) = v'$ **then**
3: **if** $u', v', Comp(b_1), Comp(b_2)$ are distinct **then**
4: Construct a component cycle $C = \{u, v\}$ with two branches ub_1, vb_2
5: $Q := Augment(C)$
6: $S_{i+1} := S_i \cup Q$; $i := i + 1$;
7: **end if**
8: **end if**
9: **end for**

By precomputing the adjacency of each vertex, the time spent checking the if-condition for one iteration is at most $O(k\alpha(n))$ where k is the number of edges between the two components in the underlying graph. Then the amortizing argument used for loop 3 also implies this implementation has most $O(m\alpha(n))$ runtime.

4.4.4 Implementation of Loop 6 (lines 17-19). This loop is simpler than the three previous loop, since no branches can exist after loop 5 finishes. A simplified version of the

third loop can be used to run in $O(m\alpha(n))$ time. We find that all seven loops and therefore Algorithm 3 run in $O(m\alpha(n))$.$\square$

CHAPTER 5

CONCLUSION

This study of Power Assignment problems showed that several recent improved approximation algorithms can be applied to large instances.

For Min-Power Strong Connectivity problem, we give a fast variant of 1.85-approximation algorithm with running time $O(n^2 \log^2 n)$. For Min-Power Broadcast problem, we give a fast variant of $2(1 + \ln n)$-approximation algorithm for the most general cost model with running time $O(n^3)$ and a fast variant of 4.2-approximation algorithm for 2-dimensional cost model with running time $O(nm)$, where $n = |V|$ and $m = |E|$.

For Min-Power Strong connectivity and Min-Power broadcast problem, We also made progress in exactly solving moderate-sized instances.

For Min-Power connectivity with two-power level problem, we give $\frac{5}{3}$-approximation algorithms that run in $O(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

BIBLIOGRAPHY

[1] A. Karim Abu-Affash, Paz Carmi, and Anat Parush Tzur. Dual power assignment via second Hamiltonian cycle. *CoRR*, abs/1402.5783, 2014.

[2] Sameh Al-Shihabi, Peter Merz, and Steffen Wolf. Nested partitioning for the minimum energy broadcast problem. In Vittorio Maniezzo, Roberto Battiti, and Jean-Paul Watson, editors, *Learning and Intelligent Optimization*, volume 5313 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg, 2008.

[3] E. Althaus, G. Calinescu, I. Mandoiu, S. Prasad, N.Tchervenski, and A. Zelikovsky. Power efficient range assignment for symmetric connectivity in static ad hoc wireless networks. *Wireless Networks*, 12(3):287–299, 2006.

[4] C. Ambühl. An optimal bound for the MST algorithm to compute energy efficient broadcast trees in wireless networks. In *Proceedings of 32th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1139–1150. Springer Verlag, 2005.

[5] Joanna Bauer, Dag Haugland, and Di Yuan. New results on the time complexity and approximation ratio of the broadcast incremental power algorithm. *Inf. Process. Lett.*, 109(12):615–619, May 2009.

[6] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An improved LP-based approximation for Steiner tree. In *STOC '10: Proceedings of the 42nd ACM symposium on Theory of computing*, pages 583–592, New York, NY, USA, 2010. ACM.

[7] Mario Cagalj, Jean-Pierre Hubaux, and Christian Enz. Minimum-energy broadcast in all-wireless networks: NP-completeness and distribution issues. In *ACM Mobicom*, pages 172–182, 2002.

[8] Hongxu Cai and Yingchao Zhao. On approximation ratios of minimum-energy multicast routing in wireless networks. *Journal of Combinatorial Optimization*, 9(3):243–262, 2005.

[9] G. Calinescu. Min-power strong connectivity. In M. Serna, K. Jansen, and J. Rolin, editors, *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization*, number 6302 in Lecture Notes in Computer Science, pages 67–80. Springer, 2010.

[10] G. Calinescu, S. Kapoor, A. Olshevsky, and A. Zelikovsky. Network lifetime and power assignment in ad-hoc wireless networks. In *Proc. 11th European Symphosium on Algorithms*, pages 114–126, 2003.

[11] Gruia Calinescu. Approximate Min-Power Strong Connectivity. *SIAM J. Discrete Mathematics*, to appear, 2013.

[12] Gruia Călinescu. Faster approximation for symmetric min-power broadcast. In *CCCG*. Carleton University, Ottawa, Canada, 2013.

[13] Gruia Calinescu. 1.61-approximation for min-power strong connectivity with two power levels. *Journal of Combinatorial Optimization*, pages 1–21, 2014.

[14] Gruia Calinescu and K Qiao. Asymmetric topology control: exact solutions and fast approximations. In *INFOCOM, 2012 Proceedings IEEE*, pages 783–791. IEEE, 2012.

[15] Gruia Calinescu and Kan Qiao. Minimum power broadcast: Fast variants of greedy approximations. In *Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on*, pages 479–487. IEEE, 2014.

[16] Tiago Camilo, Jorge Sá Silva, André Rodrigues, and Fernando Boavida. Gensen: A topology generator for real wireless sensor networks deployment. In *Proc. SEUS'07*, pages 436–445, 2007.

[17] I. Caragiannis, M. Flammini, and L. Moscardelli. An exponential improvement on the mst heuristic for minimum energy broadcasting in ad hoc wireless networks. *Networking, IEEE/ACM Transactions on*, 21(4):1322–1331, Aug 2013.

[18] I. Caragiannis, C. Kaklamanis, and P. Kanellopoulos. Energy-efficient wireless network design. In *ISAAC*, 2003.

[19] Ioannis Caragiannis, Michele Flammini, and Luca Moscardelli. An Exponential Improvement on the MST Heuristic for Minimum Energy Broadcasting in Ad Hoc Wireless Networks. In *ICALP*, pages 447–458, 2007.

[20] Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos. New results for energy-efficient broadcasting in wireless networks. In Prosenjit Bose and Pat Morin, editors, *ISAAC*, volume 2518 of *Lecture Notes in Computer Science*, pages 332–343. Springer, 2002.

[21] Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos. A logarithmic approximation algorithm for the minimum energy consumption broadcast subgraph problem. *Inf. Process. Lett.*, 86(3):149–154, May 2003.

[22] Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos. Energy-efficient wireless network design. *Theor. Comp. Sys.*, 39(5):593–617, September 2006.

[23] Paz Carmi and Matthew J. Katz. Power assignment in radio networks with two power levels. *Algorithmica*, 47(2):183–201, 2007.

[24] W.T. Chen and N.F. Huang. The strongly connecting problem on multihop packet radio networks. *IEEE Transactions on Communications*, 37(3):293–295, 1989.

[25] V. Chvatal. A greedy heuristic for the set covering problem. *Mathematics of Operation Research*, 4:233–235, 1979.

[26] A. Clementi, P. Crescenzi, P. Penna, G. Rossi, and P. Vocca. On the Complexity of Computing Minimum Energy Consumption Broadcast Subgraphs. In *18th Annual Symposium on Theoretical Aspects of Computer Science, LNCS 2010, 2001*, pages 121–131, 2001.

[27] A.E.F. Clementi, P. Penna, and R. Silvestri. On the power assignment problem in radio networks. *Electronic Colloquium on Computational Complexity (ECCC)*, (TR00-054), 2000.

[28] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2. edition, 2001.

[29] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry (second edition)*. Springer-Verlag, 2000.

[30] David P. Dobkin, David Eppstein, and Don P. Mitchell. Computing the discrepancy with applications to supersampling patterns. *ACM Trans. Graph.*, 15(4):354–376, October 1996.

[31] U. Feige. A treshold of $\ln n$ for approximating set cover. *JACM*, 45:634–652, 1998.

[32] Michele Flammini, Ralf Klasing, Alfredo Navarra, and Stephane Perennes. Improved approximation results for the minimum energy broadcasting problem. *Algorithmica*, 49(4):318–336, 2007.

[33] Michele Flammini, Ralf Klasing, Alfredo Navarra, and Stephane Perennes. Tightening the upper bound for the minimum energy broadcasting. *Wirel. Netw.*, 14(5):659–669, October 2008.

[34] Michele Flammini, Alfredo Navarra, Ralf Klasing, and Stéphane Pérennes. Improved approximation results for the minimum energy broadcasting problem. In *DIALM-POMC*, pages 85–91, 2004.

[35] Michele Flammini, Alfredo Navarra, and Stephane Perennes. The "real" approximation factor of the mst heuristic for the minimum energy broadcasting. In *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*, WEA'05, pages 22–31, Berlin, Heidelberg, 2005. Springer-Verlag.

[36] G. Baudis and C. Gropl and S. Hougardy and T. Nierhoff and H. J. Prömel. Approximating minimum spanning sets in hypergraphs and polymatroids. In *ICALP*, 2000.

[37] S.K. Ghosh. Energy efficient broadcast in distributed ad hoc wireless networks. In *Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on*, pages 394–401, 2008.

[38] Benjamin Grimmer and Kan Qiao. Near linear time 5/3-approximation algorithms for two-level power assignment problems. In *Proceedings of the 10th ACM international workshop on Foundations of mobile computing*, pages 29–38. ACM, 2014.

[39] Sudipto Guha and Samir Khuller. Improved Methods for Approximating Node Weighted Steiner Trees and Connected Dominating Sets. *Information and Computation*, 150:57–74, 1999.

[40] Song Guo and Oliver Yang. Minimum-energy multicast in wireless ad hoc networks with adaptive antennas: Milp formulations and heuristic algorithms. *IEEE Transactions on Mobile Computing*, 5(4):333–346, April 2006.

[41] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[42] Hugo Hernández, Christian Blum, and Guillem Francès. Ant colony optimization for energy-efficient broadcasting in ad-hoc networks. In *Proceedings of the 6th international conference on Ant Colony Optimization and Swarm Intelligence*, ANTS '08, pages 25–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] Toshihide Ibaraki and Naoki Katoh. On-line computation of transitive closures of graphs. *Inf. Process. Lett.*, 16(2):95–97, 1983.

[44] L. M. Kirousis, E. Kranakis, D. Krizanc, and A. Pelc. Power consumption in packet radio networks. *Theoretical Computer Science*, 243(1-2):289–305, 2000.

[45] Ralf Klasing, Alfredo Navarra, Aris Papadopoulos, and Stphane Prennes. Adaptive broadcast consumption (abc), a new heuristic and new bounds for the minimum energy broadcast routing problem. In NikolasM. Mitrou, Kimon Kontovasilis, GeorgeN. Rouskas, Ilias Iliadis, and Lazaros Merakos, editors, *NETWORKING 2004. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications*, volume 3042 of *Lecture Notes in Computer Science*, pages 866–877. Springer Berlin Heidelberg, 2004.

[46] P. Klein and R.Ravi. A nearly best-possible approximation algorithm for node-weighted Steiner trees. *Journal of Algorithms*, 19:104–115, 1995.

[47] S. Krumke, R. Liu, E. Lloyd, M. Marathe, R. Ramanathan, and S.S. Ravi. Topology control problems under symmetric and asymmetric power thresholds. In *Proc. Ad-Hoc Now*, pages 187–198, 2003.

[48] Deying Li, Xiaohua Jia, Senior Member, and Hai Liu. Energy efficient broadcast routing in static ad hoc wireless networks. *IEEE Trans. Mobile Comput*, 3:144–151, 2004.

[49] Weifa Liang. Constructing minimum-energy broadcast trees in wireless ad hoc networks. In *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 112–122. ACM Press, 2002.

[50] ErrolL. Lloyd, Rui Liu, and S.S. Ravi. Approximating the minimum number of maximum power users in ad hoc networks. *Mobile Networks and Applications*, 11(2):129–142, 2006.

[51] Manki Min, Austin F. O'Brien, and Sung Y. Shin. Improved psor algorithm for minimum power multicast tree problem in wireless ad hoc networks. *Int. J. Sen. Netw.*, 8(3/4):193–201, October 2010.

[52] Fredrick Mtenzi and Yingyu Wan. The minimum-energy broadcast problem in symmetric wireless ad hoc networks. In *Proceedings of the 5th WSEAS international conference on Applied computer science*, ACOS'06, pages 68–76, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[53] Alfredo Navarra. Tighter bounds for the minimum energy broadcasting problem. In *Proceedings of the Third International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, WIOPT '05, pages 313–322, Washington, DC, USA, 2005. IEEE Computer Society.

[54] Alfredo Navarra. 3-d minimum energy broadcasting. In Paola Flocchini and Leszek Gsieniec, editors, *Structural Information and Communication Complexity*, volume 4056 of *Lecture Notes in Computer Science*, pages 240–252. Springer Berlin Heidelberg, 2006.

[55] Z. Nutov and A. Yaroshevitch. Wireless network design via 3-decompositions. *Inf. Process. Lett.*, 109(19):1136–1140, 2009.

[56] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

[57] Tobias Polzin and Siavash Vahdati Daneshmand. On Steiner trees and minimum spanning trees in hypergraphs. *Oper. Res. Lett.*, 31(1):12–20, 2003.

[58] Paolo Santi. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv.*, 37(2):164–194, 2005.

[59] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[60] R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[61] A. Tucker. Coloring a family of circular arcs. *SIAM Journal of Applied Math.*, 29:493–502, 1975.

[62] Mario Čagalj, Jean-Pierre Hubaux, and Christian C. Enz. Energy-efficient broadcasting in all-wireless networks. *Wirel. Netw.*, 11(1-2):177–188, January 2005.

[63] P.-J. Wan, G. Calinescu, X.-Y. Li, and O. Frieder. Minimum Energy Broadcast Routing in Static Ad Hoc Wireless Networks. *Wireless Networks*, 8(6):607–617, 2002.

[64] Peng-Jun Wan, Gruia Calinescu, Xiang-Yang Li, and Ophir Frieder. Minimum-energy broadcast routing in static ad hoc wireless networks. In *INFOCOM*, pages 1162–1171, 2001.

[65] Peng-Jun Wan, Gruia Calinescu, Xiang-Yang Li, and Ophir Frieder. Erratum: Minimum-energy broadcast in static ad hoc wireless networks. *Wireless Networks*, 11(4):531–533, 2005.

[66] J.E. Wieselthier, G.D. Nguyen, and A. Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *Proc. IEEE INFOCOM*, pages 585–594, 2000.

[67] Steffen Wolf and Peter Merz. Evolutionary local search for the minimum energy broadcast problem. In *Proceedings of the 8th European conference on Evolutionary computation in combinatorial optimization*, EvoCOP'08, pages 61–72, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] L.A. Wolsey. Analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2:385–392, 1982.

[69] Di Yuan, Joanna Bauer, and Dag Haugland. Minimum-energy broadcast and multicast in wireless networks: An integer programming approach and improved heuristic algorithms. *Ad Hoc Netw.*, 6(5):696–717, July 2008.

[70] A. Zelikovsky. Better approximation bounds for the network and Euclidean Steiner tree problems. Technical Report CS-96-06, Department of Computer Science, University of Virginia, 1996.