APPROXIMATION ALGORITHMS FOR NEW GRAPH PARTITIONING AND FACILITY LOCATION PROBLEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Zoya Svitkina

August 2007

APPROXIMATION ALGORITHMS FOR NEW GRAPH PARTITIONING AND FACILITY LOCATION PROBLEMS

Zoya Svitkina, Ph.D.

Cornell University 2007

In applications as diverse as data placement in peer-to-peer systems, control of epidemic outbreaks, and routing in sensor networks, the fundamental questions can be abstracted as problems in combinatorial optimization. However, many of these problems are NP-hard, which makes it unlikely that exact polynomial-time algorithms for them exist. Approximation algorithms are designed to circumvent this difficulty, by finding provably near-optimal solutions in polynomial time. This thesis introduces a number of new combinatorial optimization problems that arise from various applications and proposes approximation algorithms for them. These problems fall into two general areas: graph partitioning and facility location.

The first problem that we introduce is the *unbalanced graph cut* problem. Here the goal is to find a graph cut, minimizing the size of one of the sides, while also respecting an upper bound on the number of edges cut. We develop two bicriteria approximation algorithms for this problem using the technique of Lagrangian relaxation, and a different algorithm for its maximization version. The other graph partitioning problem that we introduce and study is the *min-max multiway cut* problem. It aims to partition a graph into multiple components, minimizing the maximum number of edges coming out of any component. We present an approximation algorithm for this problem which uses unbalanced cuts as well as the greedy

technique.

In the second part of the thesis, we study two generalizations of the facility location problem, which aims to open facilities, assigning clients to them, in order to minimize the facility opening costs and the connection costs. In the *facility location with hierarchical facility costs* problem, the facility costs are more general, and depend on the set of assigned clients. Our algorithm, based on the local search technique, uses two new local improvement operations, achieving a constant-factor approximation guarantee. The second generalization is the *load-balanced facility location* problem, which specifies a lower bound for the number of clients assigned to an open facility. We give the first true constant-factor approximation algorithm, which uses a reduction to the capacitated facility location problem.

The thesis is concluded with related open problems and directions for future research.

BIOGRAPHICAL SKETCH

Zoya Svitkina was born on August 30, 1979 in Moscow, Russia, which was then part of the Soviet Union. In 1994, she moved to the United States, where she graduated from West High School in Madison, Wisconsin. She went on to receive a Bachelor of Science degree from the University of Wisconsin-Madison, with a major in Computer Sciences. After that she joined the Computer Science Department of Cornell University as a graduate student, where she has spent five years starting in 2002.

Dedicated to my grandparents

ACKNOWLEDGEMENTS

Most of all, I would like to thank my advisor, Éva Tardos, for taking me as her student and for being infinitely patient and supportive throughout the years. I would also like to thank her for her wisdom and valuable advice on many topics of life, research, and career development.

I thank David Shmoys for valuable and insightful discussions about various algorithmic problems. I also thank Éva Tardos, David Shmoys, and Joe Halpern for serving on my thesis committee and for teaching some of the best courses that I've taken. I am grateful to my committee and other Cornell faculty, including Dexter Kozen, John Hopcroft, Johannes Gehrke, David Williamson, Radu Rugina, Jon Kleinberg, Juris Hartmanis, Graeme Bailey, Bobby Kleinberg and others for teaching wonderful courses and just for being there and providing inspiration. In addition, I thank all those who have written letters of recommendation for me and apologize for some last-minute requests.

I thank my co-authors Ara Hayrapetyan, David Kempe and Martin Pál for collaboration on the unbalanced cut problem, and Éva Tardos for collaboration on the multiway cut and facility location problems.

My big thanks to my roommates, officemates, friends and of course family for providing moral support and company throughout these years. I hope not to lose touch with any of them in the future.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In many real-world applications, such as deciding how to build roads or where to place warehouses, the fundamental questions that arise can be abstracted as problems in combinatorial optimization. In these problems, the goal is to minimize or maximize an objective function over a space of feasible solutions. The total size of the solution space is usually exponential in the size of the input, so listing and evaluating each possible solution is not a practical approach. However, in some cases it is possible to find optimal solutions using a much smaller amount of time than would be taken by listing all solutions. A well-established notion that is used to define such efficient algorithms is that of polynomial running time (see, e.g. [14]). Polynomial-time algorithms are ones whose running time, as a function of the size of the input instance, is asymptotically bounded by a polynomial function.

Some combinatorial optimization problems can be solved to optimality in polynomial time. Well-known examples of such problems include the *shortest path*, the *minimum spanning tree*, and the *minimum s-t cut* problems [14]. However, for many other natural and useful problems no polynomial-time algorithms are known. Moreover, the theory of NP-completeness [13] tells us that for a large class of such problems, called NP-complete problems, if there exists a polynomial-time algorithm for one of them, then there exist polynomial-time algorithms for all the others as well. Given this result, it is widely conjectured that in fact there do not exist polynomial-time algorithms for NP-complete problems. A related notion of NP-hardness defines a class of problems that are at least as hard as the NP-complete ones.

Approximation algorithms [67] are designed to overcome the difficulties presented by the NP-hard problems. These are algorithms that run in polynomial time, but instead of always producing the optimal solution, they produce a feasible solution whose objective function is guaranteed to be close to the optimum. This guarantee usually takes the form of a bound on the multiplicative factor by which the value of the produced solution differs from the value of the optimal solution. This factor is interchangeably called the approximation ratio, approximation factor or approximation guarantee. It may or may not depend on the size of the problem instance. Good approximation algorithms have been found for many problems, such as *knapsack* [39], *vertex cover* [35], *set cover* [43], and *sparsest cut* [51, 2]. The approximation guarantees obtained for these problems are different, ranging from a polynomial time approximation scheme (PTAS) for knapsack, which can get arbitrarily close to the optimum, to a logarithmic factor for the set cover problem.

A more relaxed kind of guarantee is provided by the bicriteria approximation algorithms. Their name refers to the two criteria which are being approximated: one is the objective function, like in the regular approximation algorithms, and the second is a constraint (or a set of constraints) of the problem. The solution produced by a bicriteria algorithm may violate one or more of the problem's constraints, so strictly speaking it may not be feasible. The approximation guarantee of such an algorithm specifies two multiplicative factors, indicating by how much each of the two criteria can be violated. For example, in the *bisection* problem, the goal is to divide the nodes of a graph into two equal parts, minimizing the number of edges that cross the cut. But a well-known bicriteria approximation algorithm [51] for this problem produces two parts which are not exactly equal-sized, but are

within a constant factor (for example, they could contain one-third and two-thirds of the nodes). For the objective function, the approximation guarantee of a bicriteria algorithm compares the value of the produced solution to the optimal value of a solution that does not violate the constraints. For example, in the case of the bisection problem, the number of edges in the slightly-unbalanced cut would be compared to the minimum number of edges of any completely-balanced cut.

In this thesis, we introduce several new combinatorial optimization problems that arise from various applications, and we design approximation algorithms for them, some of which are bicriteria approximations. The first few of these problems deal with graph cuts. In these problems, the goal is to remove some of the edges of a graph in order to satisfy certain requirements. First, we consider two versions of the *unbalanced cut* problem, and then we consider a multi-terminal cut problem called *min-max multiway cut*. In the next part of the thesis, we study two optimization problems related to facility location, where the goal is to open a set of facilities and assign clients to them. The following sections of the introduction present the framework and definitions of the problems studied in this thesis.

1.1 Graph cuts

In the graph cut problems, we are given a graph with nodes and edges, and the goal is to remove some of its edges, disconnecting the nodes into multiple components, in order to satisfy certain requirements and to optimize an objective function. The most basic and well-studied version of the problem is the *minimum s-t cut* problem. In it, the graph contains two special vertices, called the source s and the sink t, and the requirement is to disconnect s from t, with the objective of removing as few edges as possible. This problem is known to be solvable in polynomial

time [26]. However, when additional constraints are imposed, the problem often becomes NP-hard. For example, an important problem in graph cuts is to find a *bisection*, or a balanced cut, in a graph [29]. In this problem, one has to remove a minimum number of edges from the graph so as to divide the vertices into two equal-sized sets, disconnected from each other. This problem is NP-hard, but it has a poly-logarithmic approximation algorithm [21].

In this thesis we introduce a new *unbalanced cut* problem, which in a way is the opposite of the balanced cut. More specifically, we are given a graph with a source and a sink nodes, as well as a number B. The goal then is to find a cut in this graph that separates the source from the sink, such that the number of edges removed is at most B, and the number of nodes which are on the same side of the cut as the source is minimized. This problem is motivated by finding communities in social networks, as well as by applications in epidemiology and disaster containment. We show that the unbalanced cut problem is NP-hard, and then consider two different directions for approximation. One is the minimization version, in which the number of nodes on the source side of the cut, which should be as small as possible, is to be approximated. The other is the maximization version, in which we approximate the number of nodes on the sink side, which has to be as large as possible. These two versions are equivalent if the problem is to be solved to optimality, but they are different for the purposes of approximation. We present a constant-factor bicriteria approximation algorithm for the minimization version, and a polylogarithmic approximation algorithm for the maximization problem.

Another class of graph cut problems that has been widely studied are the multi-terminal graph cuts. In these problems, the graph has to be partitioned into more than two components. For example, in the *multiway cut* problem [15], the

graph has a number of special vertices, called terminals, and the goal is to remove a minimum number of edges in such a way that all terminals are separated from each other. The multiway cut problem is well-studied, and a number of constant-factor approximation algorithms for it are known [15, 6]. In this thesis we introduce a different multi-terminal cut problem, called *min-max multiway cut*. Analogously to the multiway cut, the input consists of a graph with a subset of nodes labeled as terminals, and the goal is to remove some edges, separating all terminals from each other. However, the objective function is different from the multiway cut: instead of minimizing the total number of edges removed, the aim of min-max multiway cut is to minimize the *maximum* number of edges coming out of any terminal's component. The motivation for this problem comes from the area of distributed databases, and the task of partitioning data among servers in a peer-to-peer system. We present a polylogarithmic-factor approximation algorithm for the min-max multiway cut problem.

The results for problems related to graph cuts are presented in Chapters 2 and 3, and are based on joint work with Ara Hayrapetyan, David Kempe, Martin Pál [36], and Éva Tardos [65].

1.2 Facility location

Facility location problems are motivated by applications such as deciding the placement of stores or warehouses in a geographical area or of servers on a network. These problems involve a tradeoff between the costs of building the facilities and the resulting distances between the clients and the facilities. The best-known version of this problem is the *metric uncapacitated facility location* [17], in which the input consists of a set of clients, a set of potential facilities, distances between the

clients and the facilities, and a facility opening cost for each facility. The goal is to open a subset of the facilities and to connect each client to an open facility, with the objective of minimizing the sum of the opening costs for the facilities that were opened, plus the sum of connection costs of all the clients, which are equal to their distances to their assigned facilities. This problem has received a lot of attention in the recent years, and many constant-factor approximation algorithms, utilizing several different techniques, are known for it [62, 10, 11, 8, 48, 3, 42, 63, 54].

Extensions and generalizations have also been considered for the facility location problem. For example, in the *capacitated facility location* [48, 12, 56, 69], facilities may not serve more clients than allowed by their specified capacities. The *universal facility location* problem [34, 53, 30] is a generalization that allows more complex facility costs, which, instead of being fixed for each facility, are now allowed to be arbitrary non-decreasing functions of the number of clients that the facility is assigned to serve. Another recently introduced generalization of the problem is the *facility location with service installation costs* [58, 61], which introduces a new type of more complex facility costs. In particular, in this setting each client requests a particular kind of service, and when clients are assigned to a facility, their required services have to be installed at that facility, incurring an extra cost in addition to the plain facility opening cost. Constant-factor approximation algorithms have been discovered for all of these extensions of the facility location problem.

In this thesis, we study two models that generalizes the facility location problem. In the first one, the facility cost is a monotone and submodular function of the set of clients that are assigned to it. We focus on a special case of this problem, called *facility location with hierarchical facility costs*, which is analogous to the

service installation model, but instead of having two levels of facility costs (facility opening and service installation), there can be arbitrarily many levels of such costs. Besides multi-level service installation, this problem has applications for data storage and data gathering in sensor networks. We present a constant-factor approximation algorithm for this problem, based on the local search technique, for the case that facility cost functions are identical on all facilities. The second generalization that we consider is the *load-balanced facility location* problem, which in some sense is the opposite of capacitated facility location. Instead of capacities, which specify a maximum number of clients that a facility is allowed to serve, this problem has lower bounds, which specify the minimum number of clients a facility is allowed to serve if opened. We present a constant-factor approximation algorithm for this problem which is based on a reduction to capacitated facility location.

The results for the facility location problem with hierarchical facility costs are presented in Chapter 4. They were obtained in joint work with Éva Tardos and have appeared in an extended abstract [66]. The results for the load-balanced facility location problem are presented in Chapter 5, and are under preparation for publication [64].

CHAPTER 2

UNBALANCED GRAPH CUTS

2.1 Introduction

Graph cuts are among the most well-studied objects in theoretical computer science. In the most pristine form of the problem, two given vertices s and t have to be separated by the removal of an edge set of minimum total capacity. By a fundamental result of Ford and Fulkerson [26], such an edge set can be found in polynomial time. Since then, many problems have been shown to reduce to graph cut problems, sometimes quite surprisingly (see, e.g., [47]). One way to view the min-cut problem is to think of "protecting" the sink node t from the presumably harmful node s by way of removing edges: the capacity of the cut then corresponds to the cost of edge removals. This interpretation in turn suggests a very natural variant of the graph cut problem: given a node s and a bound B on the total edge removal cost, try to "protect" as many nodes from s as possible, while cutting at most a total edge capacity of B. In other words, find an s-t cut of capacity at most B, minimizing the size of the s-side of the cut. This is the *unbalanced cut* problem studied in this chapter.

Naturally, the unbalanced cut problem has direct applications in the areas of disaster, military, or crime containment. In all of these cases, a limited amount of resources can be used to monitor or block the edges by which the disaster could spread, or people could escape. At the same time, the area to which the disaster is confined should be as small as possible. For instance, in the firefighter's problem [16], a fixed small number of firefighters must confine a fire to within a small area, trying to minimize the value of the property and lives inside.

Perhaps even more importantly, the unbalanced cut problem arises naturally in the control of epidemic outbreaks. While traditional models of epidemics [5] have ignored the network structure in order to model epidemic diseases via differential equations, recent work by Eubank et al. [18, 19], using highly realistic large-scale simulations, has shown that the graph structure of the social contacts has a significant impact on the spread of the epidemic, and crucially, on the type of actions that most effectively contain the epidemic. If we assume that patient 0, the first infected member of the network, is known, then the problem of choosing which individuals to vaccinate in order to confine the epidemic to a small set of people is exactly the node cut version of the unbalanced cut problem.

Besides the obvious connections to the containment of damage or epidemics, the unbalanced cut problem can also be used for finding small dense subgraphs and communities in graphs. Discovering communities in graphs has received much attention recently, in the context of analyzing social networks and the World Wide Web [24, 49]. It involves examining the link structure of the underlying graph so as to extract a small set of nodes sharing a common property. This property is usually expressed by high internal connectivity, sometimes in combination with small expansion. We show how to reduce the community finding problem to unbalanced cut.

Our models and results We consider two versions of the unbalanced cut problem, the minimization version, which we call MinSBCC (minimum size bounded capacity cut), and the maximization version, MaxSBCC. Formally, the problems are defined as follows.

Definition 2.1.1 *Given an (undirected or directed) graph* $G = (V, E)$ *with edge*

capacities c_e, source and sink nodes s and t, as well as a total capacity bound (also called the budget*) B, the* unbalanced cut problem *is to find an s-t cut $(S, \overline{S})$, $s \in S$ of capacity no more than B. In the MinSBCC version, the objective is to minimize $|S|$, the number of nodes on the source side of the cut. In MaxSBCC, the objective is to maximize $|\overline{S}|$, the number of nodes on the sink side of the cut. We also consider a generalization in which the nodes are assigned weights w_v, and the objective is to optimize the total node weight in S or $\overline{S}$, subject to the budget constraint.*[1]

We use $\delta(S)$ to denote the capacity of the cut $(S, \overline{S})$ in G, and S^* to denote the minimum-size set of nodes, containing s but not containing t, such that $\delta(S^*) \leq B$, i.e. $(S^*, \overline{S^*})$ is the optimum unbalanced cut. For any two sets of nodes S and T, we use $c(S,T)$ for the total capacity of edges with one endpoint in S and the other in T. Let us define (α, β)-bicriteria approximation algorithms for the unbalanced cut problems.

Definition 2.1.2 *For $\alpha \geq 1$ and $0 < \beta \leq 1$, an* (α, β)-approximation algorithm for MaxSBCC *is an algorithm that, given an instance of MaxSBCC, produces in polynomial time an s-t cut $(S', \overline{S'})$, such that $\delta(S') \leq \alpha B$ and $w(\overline{S'}) \geq \beta w(\overline{S^*})$. An* (α, β)-approximation algorithm for MinSBCC, *with $\alpha \geq 1$ and $\beta \geq 1$, produces a cut $(S', \overline{S'})$, such that $\delta(S') \leq \alpha B$ and $w(S') \leq \beta w(S^*)$.*

We show in Sections 2.3 and 2.5.2 that the unbalanced cut problem is NP-hard on general graphs with uniform node weights, and on trees with non-uniform node weights. For the minimization problem, we develop two $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$-bicriteria approximation algorithms, where $0 < \lambda < 1$. The first algorithm obtains this guarantee by a simple rounding of a linear programming relaxation of MinSBCC.

[1]Some of our motivating examples and applications do not specify a sink; this can be resolved by adding an isolated sink to the graph.

The second one bypasses solving the linear program by running a single parametric maximum flow computation and is thus very efficient [27]. It also has a better guarantee: it outputs either a $(\frac{1}{\lambda}, 1)$-approximation or a $(1, \frac{1}{1-\lambda})$-approximation, thus violating at most one of the constraints by the corresponding factor, but we cannot control which one it is. The analysis of this algorithm is based on the same linear programming formulation of MinSBCC and its Lagrangian relaxation.

For the maximization version of the problem, we give $(O(\log^{3/2} n), 1)$-bicriteria approximation in Section 2.3.3, using the algorithm of Feige and Krauthgamer [21]. The same technique also yields a $(O(\log^{3/2} n), 1)$-bicriteria approximation algorithm for the MinSBCC problem.

In Section 2.4, we give a polynomial-time algorithm based on dynamic programming to optimally solve the unbalanced cut problem for trees with unit node weights. We then extend the algorithm to a PTAS for general node weights. Section 2.5 discusses the reductions from node cut and dense subgraph problems to MinSBCC, thus showing that our algorithms can be used to address the applications described above.

Related Work Minimum cuts have a long history of study and form part of the bread-and-butter of everyday work in algorithms [1]. While minimum cuts can be computed in polynomial time, additional constraints on the size of the cut or on the relationship between its capacity and size (such as its density) usually make the problem NP-hard.

Much recent attention has been given to the computation of *sparse cuts*, partly due to their application in divide-and-conquer algorithms [60]. The seminal work of Leighton and Rao [51] gave the first $O(\log n)$ approximation algorithm for sparsest

and balanced cut problems using region growing techniques. This work was later extended by Garg, Vazirani, and Yannakakis [31]. In a recent breakthrough result, the approximation factor for these problems was improved to $O(\sqrt{\log n})$ by Arora, Rao, and Vazirani [2].

A problem similar to MinSBCC is studied by Feige et al. [21, 22]: given a number k, find an s-t cut $(S, \overline{S})$ with $|S| = k$ of minimum capacity. They obtain an $O(\log^{3/2} n)$ approximation algorithm in the general case [21][1], and improve the approximation guarantees when k is small [22]. Our algorithm in Section 2.3.3 builds on these results.

2.2 NP-hardness

We first establish the NP-hardness of the unbalanced cut problems. Since the two versions of the unbalanced cut problem are equivalent when solved to optimality, showing that one of them is NP-hard is sufficient for proving that both of them are NP-hard.

Proposition 2.2.1 *The MaxSBCC problem with arbitrary edge capacities and node weights is NP-complete even when restricted to trees.*

Proof. We give a reduction from the knapsack problem [28]. Let the knapsack instance consist of items $1, \ldots, n$ with sizes $s_1, \ldots, s_n$ and values $a_1, \ldots, a_n$, and let the total knapsack size be B. We create a source s, a sink t, and a node v_i for each item i. The node weight of v_i is a_i, and it is connected to the source by an edge of capacity s_i. The sink t has weight 0, and is connected to v_1 by an edge of

[1]The result stated in [21] is an $O(\log^2 n)$-approximation, but given the recent improvement in the approximation ratio for the *sparsest cut* problem [2], it automatically improves to $O(\log^{3/2} n)$.

capacity 0. The budget for the MaxSBCC problem is B.

The capacity of any s-t cut is exactly the total size of items corresponding to nodes on the t-side, and maximizing the total node weight on the t-side is equivalent to maximizing the total value of items in the knapsack. □

2.3 Bicriteria approximation algorithms

Now, we turn attention to our main approximation results. The first two, presented in Sections 2.3.1 and 2.3.2, are the two $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$-bicriteria approximation algorithms for MinSBCC on general graphs. The third algorithm, presented in Section 2.3.3, is a $(O(\log^{3/2} n), 1)$-approximation for both MinSBCC and MaxSBCC.

The analysis of the first two algorithms is based on the following linear programming (LP) relaxation of the natural integer program for MinSBCC. We use a variable x_v for every vertex $v \in V$ to denote which side of the cut it is on, and a variable y_e for every edge e to denote whether or not the edge is cut.

$$\begin{array}{lllll} \text{Minimize} & \sum_{v \in V} x_v & & & \\ \text{subject to} & x_s & = & 1 & \\ & x_t & = & 0 & \\ & y_e & \geq & x_u - x_v & \text{for all } e = (u, v) \in E \\ & \sum_{e \in E} y_e \cdot c_e & \leq & B & \\ & x_v, y_e & \geq & 0 & \end{array} \tag{2.1}$$

2.3.1 Randomized rounding-based algorithm

Our first algorithm is based on randomized rounding of the solution to (2.1).

Algorithm 1 Randomized LP-rounding algorithm with parameter λ

1: Let (x^*, y^*) be the optimal solution to LP (2.1).

2: Choose $\ell \in [1-\lambda, 1]$ uniformly at random.

3: Let $S = \{v \mid x_v^* \geq \ell\}$, and output S.

Theorem 2.3.1 *The Randomized Rounding algorithm (Algorithm 1) outputs a set S of size at most $\frac{1}{1-\lambda}$ times the LP objective value. The expected capacity of the cut $(S, \overline{S})$ is at most $\frac{1}{\lambda}B$.*

Proof. To prove the first statement of the theorem, observe that for each $v \in S$, $x_v^* \geq \ell \geq 1-\lambda$. Therefore $\sum_{v \in V} x_v^* \geq \sum_{v \in S} x_v^* \geq (1-\lambda)|S|$.

For the second statement, observe that ℓ is selected uniformly at random from an interval of size λ. Furthermore, an edge $e = (u, v)$ will be cut only if ℓ lies between x_u^* and x_v^*. The probability of this happening is thus at most $\frac{|x_u^* - x_v^*|}{\lambda} \leq \frac{y_e^*}{\lambda}$. Summing over all edges yields that the expected total capacity of the cut is at most $\sum_e \frac{c_e y_e^*}{\lambda} \leq \frac{1}{\lambda}B$. □

This algorithm can easily be derandomized: instead of choosing ℓ randomly, we can try all possibilities and output the smallest cut we encounter. Since there are at most $|V|$ different values of x_v^*, we need to try at most $|V|$ cuts. The algorithm can also be extended to instances with non-negative node weights w_v: simply add the weights in the objective function — the analysis stays the same.

2.3.2 A parametric flow-based algorithm

Next, we show how to avoid solving the LP, and instead compute the cuts directly via a parametric max-flow computation. The somewhat more involved analysis will also show that in fact, at most one of the two criteria is approximated, while

the other is not violated.

Algorithm Description: The algorithm searches for candidate solutions among the parametrized minimum cuts in the graph G^α, which is obtained from G by adding an edge of capacity α from every vertex v to the sink t (introducing parallel edges if necessary). Here, α is a parameter ranging over non-negative values. Observe that the capacity of a cut $(S, \overline{S})$ in the graph G^α is $\alpha|S| + \delta(S)$, so the minimum s-t cut in G^α minimizes $\alpha|S| + \delta(S)$.

Initially, as $\alpha = 0$, the min-cut of G^α is the min-cut of G. As α increases, the source side of the min-cut of G^α will contain fewer and fewer nodes, until eventually it contains the single node $\{s\}$. All these cuts for the different values of α can be found efficiently using a single run of the push relabel algorithm. Moreover, the source sides of these cuts form a nested family $S_0 \supset S_1 \supset ... \supset S_k$ of sets [27]. (S_0 is the minimum s-t cut in the original graph, and $S_k = \{s\}$) . Our solution will be one of these cuts S_j.

We first observe that $\delta(S_i) < \delta(S_j)$ if $i < j$; for if it were not, then S_j would be a superior cut to S_i for all values of α. If $\delta(S_k) \leq B$, then, of course, $\{s\}$ is the optimal solution. On the other hand, if $\delta(S_0) > B$, then no solution exists. In all other cases, choose i such that $\delta(S_i) \leq B \leq \delta(S_{i+1})$. If $\delta(S_{i+1}) \leq \frac{1}{\lambda}B$, then output S_{i+1}; otherwise, output S_i.

Theorem 2.3.2 *The above algorithm produces either*
(1) a cut S^- such that $\delta(S^-) \leq B$ and $|S^-| \leq \frac{1}{1-\lambda}|S^*|$, or
(2) a cut S^+ such that $\delta(S^+) \leq \frac{1}{\lambda}B$ and $|S^+| \leq |S^*|$.

Proof. For the index i chosen by the algorithm, we let $S^- = S_i$ and $S^+ = S_{i+1}$. Hence, $\delta(S^-) \leq B \leq \delta(S^+)$.

First, observe that $|S^+| \leq |S^*|$, as otherwise, the parametric cut procedure would have returned S^* instead of S^+. If S^+ also satisfies $\delta(S^+) \leq \frac{1}{\lambda}B$, then we are done, so we focus on the case that $\delta(S^+) > \frac{1}{\lambda}B$. In that case, we will prove that $|S^-| \leq \frac{1}{1-\lambda}|S^*|$.

Because S^+ and S^- are neighbors in our sequence of parametric cuts, there is a value of α, call it α^*, for which both are minimum cuts of G^{α^*}. Applying the Lagrangian Relaxation technique, we remove the constraint $\sum_e y_e c_e \leq B$ from LP (2.1) and put it into the objective function using the constant α^*.

$$\begin{array}{lllll} \text{Minimize} & \alpha^* \cdot \sum_{v \in V} x_v + \sum_{e \in E} y_e \cdot c_e & & & \\ \text{subject to} & x_s & = & 1 & \\ & x_t & = & 0 & \\ & y_e & \geq & x_u - x_v \quad \text{for all } e = (u,v) \in E & \\ & x_v, y_e & \geq & 0 & \end{array} \tag{2.2}$$

Lemma 2.3.3 *LP (2.2) has an integer optimal solution.*

Proof. Recall that in G^{α^*} we added edges of capacity α^* from every node to the sink. Extend any solution of LP (2.2) to these edges by setting $y_e = x_v - x_t = x_v$ for the newly added edge e connecting v to t. We claim that after this extension, the objective function of LP (2.2) is equivalent to $\sum_{e \in G^{\alpha^*}} y_e c'_e$, where c'_e is the edge capacity in the graph G^{α^*}. Indeed, this claim follows from observing that the first part of the objective of LP (2.2) is identical to the contribution that the newly added edges of G^{α^*} are making towards $\sum_{e \in G^{\alpha^*}} y_e c'_e$.

Consider some fractional optimal solution $(\hat{x}, \hat{y})$ to LP (2.2) that has objective function value $L^* = \sum_{e \in G^{\alpha^*}} \hat{y}_e c'_e$. As this is an optimal solution, we can assume without loss of generality that $y_e = \max(0, x_u - x_v)$ for all edges $e = (u,v)$. So

if we define $w_x = \sum_{u,v:x_u \geq x \geq x_v} c'_{uv}$, then the value of the objective function is $L^* = \int_0^1 w_x \mathrm{d}x$.

Also, for any $x \in (0,1)$, we can obtain an integral solution to LP (2.2) whose objective function value is w_x by rounding $\hat{x}_v$ to 0 if it is no more than x, and to 1 otherwise (and setting $y_{uv} = \max(0, x_u - x_v)$). Since this process yields feasible solutions, we know that $w_x \geq L^*$ for all x. On the other hand, L^* is a weighted average (integral) of w_x's, and hence in fact $w_x = L^*$ for all x, and any of the rounded solutions is an integral optimal solution to LP (2.2). □

Notice that feasible integral solutions to LP (2.2) correspond to s-t cuts in G^{α^*}. Therefore, by Lemma 2.3.3, the optimal solutions to LP (2.2) are the minimum s-t cuts in G^{α^*}. In particular, S^+ and S^- are two such cuts. From S^+ and S^-, we naturally obtain solutions to LP (2.2), by setting $x_v^+ = 1$ for $v \in S^+$ and $x_v^+ = 0$ otherwise, with $y_e^+ = 1$ if e is cut by $(S^+, \overline{S^+})$, and 0 otherwise (similarly for S^-). By definition of α^*, both (x^+, y^+) and (x^-, y^-) are then optimal solutions to LP (2.2). Thus, their linear combination $(x^*, y^*) = \ell \cdot (x^+, y^+) + (1-\ell) \cdot (x^-, y^-)$ is also an optimal feasible solution. Choose ℓ such that

$$\ell \cdot \textstyle\sum_{e \in E} y_e^+ c_e + (1-\ell) \cdot \sum_{e \in E} y_e^- c_e \;\; = \;\; B. \tag{2.3}$$

Such an ℓ exists because our choice of S^- and S^+ ensured that $\delta(S^-) \leq B \leq \delta(S^+)$. For this choice of ℓ, the fractional solution (x^*, y^*), in addition to being optimal for the Lagrangian relaxation, also satisfies the constraint $\sum_e y_e^* c_e \leq B$ of LP (2.1) with equality, implying that it is optimal for LP (2.1) as well. Crudely bounding the second term in Equation (2.3) by 0, we obtain that $\delta(S^+) = \sum_{e \in E} y_e^+ c_e \leq \frac{B}{\ell}$.

As we assumed that $\delta(S^+) > \frac{B}{\lambda}$, we conclude that $\ell < \lambda$. Because (x^*, y^*) is an

optimal solution to LP (2.1), it provides the lower bound $\sum_v x_v^* \leq |S^*|$, and the fact that $x_v^* \geq (1-\ell)x_v^-$ now implies that $|S^-| = \sum_{v \in V} x_v^- \leq \frac{|S^*|}{1-\ell} \leq \frac{1}{1-\lambda} \cdot |S^*|$.

Hence, in this case, S^- meets the capacity constraint, and exceeds the optimal size by at most a factor of $\frac{1}{1-\lambda}$. □

As in the case of the LP rounding algorithm presented above, some simple cosmetic modifications allow for the inclusion of node weights in addition to edge capacities.

2.3.3 Dynamic programming-based algorithm

Feige and Krauthgamer [21] give an $O(\log^{3/2} n)$ approximation algorithm for the problem of finding cuts with specified number of nodes, and an improved $O(\log n)$ approximation for the case when the input graph G is assumed not to contain a fixed graph as a minor (e.g., for planar graphs). We use this algorithm to give an $(O(\log^{3/2} n), 1)$ approximation algorithm for the unbalanced cut problems in general graphs and an improved $(O(\log n), 1)$ approximation algorithm in the special case. This yields the following theorem.

Theorem 2.3.4 *There are $(O(\log^{3/2} n), 1)$ bicriteria approximation algorithms for the MinSBCC and the MaxSBCC problems on general graphs, and $(O(\log n), 1)$-approximations for graphs excluding any fixed graph as a minor (e.g., planar graphs).*

Proof. Feige and Krauthgamer [21] give an algorithm for finding cuts with specified sizes. For a graph G with n nodes and each number $d < n$, their algorithm finds a cut $(S_d, \overline{S_d})$ with $|S_d| = d$ and capacity $\delta(S_d)$ within $\alpha = O(\log^{3/2} n)$ of the

minimum capacity for such a cut. For graphs excluding any fixed graph as a minor, this guarantee is improved to $\alpha' = O(\log n)$. The algorithm also works for finding s-t cuts on graphs with node weights and edge capacities.

We claim that the cut that corresponds to the smallest value d^* such that $\delta(S_{d^*}) \leq \alpha B$ is an $(\alpha, 1)$-approximate MinSBCC as well as MaxSBCC. By definition, its capacity is at most αB. And, if the optimal unbalanced cut had size $|S^*| = d' < d^*$, then, by the guarantee of the algorithm, $\delta(S_{d'})$ would be at most αB, contradicting our choice of d^*. Thus, the size of S_{d^*} is at most that of S^*, and the size of the sink side $\overline{S_{d^*}}$ is at least that of $\overline{S^*}$. □

2.4 Algorithm for unbalanced cut on trees

As we saw in Section 2.2, the unbalanced cut problem is NP-hard even on trees with general node weights and edge capacities. However, if all nodes have non-negative integer weights bounded by a polynomial, then the problem can be solved in polynomial time for trees, via a dynamic programming algorithm.

We root the tree at the source node s and direct all edges away from s. When all edges have capacity 1, then clearly, only edges incident with s should be cut. They must include the edge on the unique s-t path, and in addition, the edges to the roots of the largest subtrees. Choosing these B edges gives the smallest possible size for the s-side of the cut.

For the case of general edge capacities, consider the tree T_v rooted at a node v, together with the edge e_v into v. We define the quantity a_v^k to be the smallest total capacity of edges in T_v that must be cut if nodes of total weight at most k from T_v are to be included in the source side of the cut. Notice that $a_v^0 = c_{e_v}$.

Also, as the sink must always be excluded, we have $a_t^k = c_{e_t}$ for all k.

For a leaf v, we have $a_v^k = c_{e_v}$ for $k < w_v$, and $a_v^k = 0$ for $k \geq w_v$. For an internal node v with children $v_1, \ldots, v_d$, we can either cut the edge e_v into v, or otherwise include v and solve the problem recursively for the children of v, hence

$$a_v^k \;=\; \min\left(c_{e_v}, \min_{k_1,\ldots,k_d \geq 0 : \sum k_i = k - w_v} \sum_i a_{v_i}^{k_i}\right) \qquad \text{for } k > 0, v \neq t.$$

Note that the optimal partition into k_i's can be found in polynomial time by a nested dynamic programming subroutine that uses optimal partitions of each k into $k_1 \ldots k_j$ in order to calculate the optimal partition into $k_1 \ldots k_{j+1}$.

Once we have computed a_s^k at the source s for all values of k, we simply pick the smallest k^* such that $a_s^{k^*} \leq B$.

2.4.1 A PTAS for node-weighted trees

The above algorithm for solving unbalanced cut on trees with polynomial node weights can be used to obtain a polynomial-time approximation scheme (PTAS) for unbalanced cut on trees with arbitrary node weights.

Suppose we want to achieve a $(1 + 2\epsilon)$ guarantee for the MinSBCC problem. Let S^* denote the optimal solution and OPT denote its value. We first guess a value W such that $OPT \leq W \leq 2\ OPT$ (one can test all powers of 2). Next, we remove all *heavy* nodes, i.e. those whose weight is more than W, in the sense that we treat all of them like sinks. We then rescale the remaining node weights w_v to $w'_v := \lceil \frac{w_v n}{\epsilon W} \rceil$. Notice that the largest node weight is now at most $\frac{n}{\epsilon}$. Hence, we can run the above dynamic programming algorithm on the rescaled graph in time polynomial in n.

We now bound the cost of the obtained solution, which we call S. The scaled weight of the solution S^* is at most $\sum_{v \in S^*} \lceil \frac{w_v n}{\epsilon W} \rceil \leq \frac{n}{\epsilon W} OPT + n$ (using the fact

that S^* contains no more than n nodes). Since S^* is a feasible solution for the rescaled problem, the solution S found by the algorithm has (rescaled) weight no more than that of S^*. Therefore, the original weight of S is at most $(OPT + \epsilon W)$. Considering that $W \leq 2\ OPT$, we obtain the desired guarantee, namely that the cost of S is at most $(1 + 2\epsilon)OPT$. An analogous algorithm can be used to obtain a PTAS for the MaxSBCC problem on trees.

2.5 Applications

2.5.1 Epidemiology and node cuts

Several of the problems given in the introduction are phrased much more naturally in terms of node cuts than edge cuts. For instance, while military or other disaster containment problems can be thought of as controlling escape or spread pathways, i.e., edges, controlling the outbreak of a disease involves the vaccination of individuals, and thus the removal of nodes from the graph. Therefore, it is natural to study the node cut version. Here, each node has a *weight* w_v, the cost of including it on the s-side of the cut, and a *capacity* c_v, the cost of removing (cutting) it from the graph. The goal is to find a set $R \subseteq V$, not containing s, of capacity $c(R)$ not exceeding a budget B, such that after removing R, the connected component S containing s has minimum total weight $w(S)$.

This problem can be reduced to (node-weighted) MinSBCC in the standard way. The same reduction works for the maximization version as well. First, if the original graph G is undirected, we bidirect each edge. Now, each vertex v is split into two vertices v_{in} and v_{out}; all edges into v now enter v_{in}, while all edges out of v now leave v_{out}. We add a directed edge from v_{in} to v_{out} of capacity c_v.

Each originally present edge, i.e., each edge into v_{in} or out of v_{out}, is given infinite capacity. Finally, v_{in} is given node weight 0, and v_{out} is given node weight w_v. Call the resulting graph G' and solve the unbalanced cut problem on it.

If it happens that for some node v, the solution places v_{out} on the source side of the cut and v_{in} on the sink side, then v_{out} can be moved to the sink side, only improving the solution. Now, it is easy to verify that (1) no edge cut in G' ever cuts any originally present edges, (2) the capacity of an edge cut in G' is equal to the node capacity of a node cut in G, and (3) the total node weight on the s-side of an edge cut in G' is exactly the total node weight in the s component of the corresponding node cut in G. Hence, we have reduced the node cut problem to unbalanced cut, and any approximation guarantees obtained for MinSBCC or MaxSBCC equally carry over.

2.5.2 Graph communities

Identifying "communities" has been an important and much studied problem for social or biological networks, and more recently, the web graph [24, 25]. Different mathematical formalizations for the notion of a community have been proposed, but they usually share the property that a community is a node set with high edge density within the set, and comparatively small expansion.

It is well known [50] that the *densest subgraph*, i.e., the set S maximizing $\frac{c(S)}{|S|} := \frac{c(S,S)}{|S|}$ can be found in polynomial time via a reduction to *minimum cut*; Charikar [7] showed that a simple greedy algorithm gives a 2-approximation for this problem. On the other hand, if the size of the set S is prescribed to be at most k, then the problem is the well-studied *densest k-subgraph problem* [4, 20, 23], which is known to be NP-complete (via a simple reduction from *clique*), and was

recently shown to be hard to approximate to within $(1+\epsilon)$ for some $\epsilon > 0$ [46]. The best known approximation ratio is $O(n^{1/3-\epsilon})$ for some $\epsilon > 0$ [20]. What we consider below is the converse of the densest k-subgraph problem, in which the density of the subgraph is given, and the size has to be minimized.

The definition of a graph community as the densest subgraph has the disadvantage that it lacks specificity. For example, adding a high-degree node tends to increase the density of a subgraph, but intuitively such a node should not belong to the community. The notion of a community that we consider avoids this difficulty by requiring that a certain fraction of a community's edges lie inside of it. Formally, let an *α-community* be a set of nodes S with $\frac{c(S)}{d(S)} \geq \alpha$, where $c(S)$ is the number of edges within S, and $d(S)$ is the sum of degrees of nodes in S. This definition is a relaxation of one introduced by Flake et al. [24] and is used in [57]. We are interested in finding such communities of smallest size.

The problem of finding the smallest α-community and the problem of finding the smallest subgraph of a given density have a common generalization, which is obtained by defining a node weight w_v which is equal to node degree for the former problem and to 1 for the latter. We show how to reduce this general size minimization problem to MinSBCC in an approximation-preserving way. In particular, by applying this reduction to the densest k-subgraph problem, we show that unbalanced cut is NP-hard even for the case of unit node weights.

Given a graph $G = (V, E)$ with edge capacities c_e, node weights w_v, and a specified node $s \in V$, we consider the problem of finding the smallest (in terms of the number of nodes) set S containing s with $\frac{c(S)}{w(S)} \geq \alpha$. (The version where s is not specified can be reduced to this one by trying all nodes s.) We modify G to obtain a graph G' as follows. Add a sink t, connect each vertex v to the source s

with an edge of capacity $d(v) := \sum_u c_{(v,u)}$, and to the sink with an edge of capacity $2\alpha w_v$. The capacity for all edges $e \in E$ stays unchanged.

Theorem 2.5.1 *A set $S \subseteq V$ with $s \in S$ has $\frac{c(S)}{w(S)} \geq \alpha$ if and only if $(S, \overline{S} \cup \{t\})$ is an s-t cut of capacity at most $2c(V) = 2\sum_{e \in E} c_e$ in G'.*

Notice that this implies that any approximation guarantees on the size of S carry over from the MinSBCC problem to the problem of finding communities. Also notice that by making all node weights and edge capacities 1, and setting $\alpha = \frac{k-1}{2}$, a set S of size at most k satisfies $\frac{c(S)}{w(S)} \geq \alpha$ if and only if S is a k-clique. Hence, the MinSBCC problem is NP-hard even with unit node weights. However, the approximation hardness of *clique* does not carry over, as the reduction requires the size k to be known.

Proof. The required condition can be rewritten as $c(S) - \alpha w(S) \geq 0$. As

$$2\big(c(S) - \alpha w(S)\big) \;\; = \;\; 2c(V) - \big(c(S, \overline{S}) + \sum_{v \in \overline{S}} d(v) + 2\alpha w(S)\big),$$

we find that S is an α-community iff $c(S, \overline{S}) + \sum_{v \in \overline{S}} d(v) + 2\alpha w(S) \leq 2c(V)$. The quantity on the left-hand side is exactly the capacity of the cut $(S, \overline{S} \cup \{t\})$, which proves the theorem. □

It should be noted, however, that in order to take advantage of these reductions and to use an unbalanced cut algorithm for finding dense subgraphs and graph communities, a $(1, \beta)$-approximation algorithm for MinSBCC is required, i.e. one that does not compromise on the capacity of the cut.

CHAPTER 3

MIN-MAX MULTIWAY CUT

3.1 Introduction

In this chapter we study a multi-terminal graph cut problem called min-max multiway cut. The *min-max multiway cut* problem is defined by an undirected graph $G = (V, E)$ with edge capacities $c(e) \geq 0$, and a set $X = \{x_1, ..., x_k\} \subseteq V$ of distinguished nodes called terminals. A *multiway cut* is a partition of V into disjoint sets $S_1, ..., S_k$ ($\bigcup_i S_i = V$), so that for all $i \in \{1, ..., k\}$, $x_i \in S_i$. For a partition we use $\delta(S_i)$ to denote the capacity of the cut separating S_i from the other sets $\bigcup_{j \neq i} S_j$, and the goal of the min-max multiway cut problem is to minimize the maximum capacity $\max_i \delta(S_i)$.

The min-max multiway cut problem models the data placement problem in a distributed peer-to-peer database system. In a peer-to-peer database, the information is stored on many servers. When a user query is issued, it is directed to the appropriate server. A request for some data item v can lead to further requests for other data. One important issue in such peer-to-peer databases is to find a good distribution of data that minimizes requests to any single server. We model this by a graph in which the non-terminal nodes represent the data items and the terminals represent the servers. Nodes in the partition S_i correspond to the data that will be stored on server i. Edges in the graph correspond to the expected communication patterns, i.e., the edge (x_i, v) represents the number of queries that users at server i issue for the data v, and the edge (v, w) represents the expected number of times that a request for data v will result in an induced request for data w. Communication costs are incurred when a query from one

server is sent to another. The goal then is to distribute the data among the servers so as to minimize the communication cost incurred by any one of them.

The min-max multiway cut problem is closely related to the traditional multiway cut problem of [15]. The difference is in the objective function. Unlike the min-max multiway cut, in which we seek to minimize the maximum capacity $\max_i \delta(S_i)$, the multiway cut problem evaluates a partition by the sum of the capacities of all edges that connect the parts, thus minimizing the average capacity $\delta(S_i)$. Multiway cut has been used to model similar applications of storing files on a network, as well as other problems such as partitioning circuit elements among different chips [15]. In many situations, however, the min-max objective function may be a better representation of the solution quality. Although the multiway cut minimizes the average communication cost of the terminals, this cost may not be distributed uniformly among them, resulting in a very heavy load on some terminals and almost no load on others. The objective of minimizing the maximum load tries to alleviate this problem by ensuring that no terminal is overloaded.

Multiway cut problem is NP-hard, but there are very good approximation algorithms for it [15, 6, 44]. However, they do not translate directly into good approximations for min-max multiway cut, because even the optimal solution to one problem can be up to a factor of $k/2$ worse than the optimum for the other.

Our results For two terminals, min-max multiway cut reduces to the well-studied minimum s-t cut problem, and hence it can be solved in polynomial time. However, as we show, it is already NP-hard for the case of 4 terminals. As a result, we focus on designing approximation algorithms. In Section 3.2, we present an $O(\alpha \cdot \log n)$-approximation algorithm for min-max multiway cut, where

$\alpha = O(\log^{3/2} n)$ for general graphs, and $\alpha = O(\log n)$ for graphs excluding any fixed graph as a minor. The algorithm uses the (α,1) bicriteria approximation algorithm for the MaxSBCC problem, presented in Section 2.3.3. We use it as a subroutine in a procedure that resembles the greedy set cover algorithm, incurring an additional factor of $O(\log n)$ in the approximation guarantee for the min-max multiway cut problem. One of the features of our algorithm is that it is able to exhibit flexibility when assigning graph nodes to terminals: if the cut that is found for one terminal is later discovered to be bad for another terminal, then the nodes are reassigned in a way that is good for both.

We extend our algorithm to a generalization of the problem, in which there is a separate bound B_i for each terminal x_i, and the goal is to find a partition in which $\delta(S_i)$ does not exceed B_i. This generalization is useful when the different peers corresponding to the terminals have different communication capabilities, and can withstand different loads.

Turning to special cases of min-max multiway cut, we show that it is strongly NP-hard even on trees, but develop a $(2 + \epsilon)$-approximation algorithm for the case of trees. What makes the problem hard on trees is that an optimal solution does not necessarily assign connected components of the tree to each terminal (see Figure 3.2, in which the black nodes are the terminals, and the optimal solution must assign the white node in the middle to one of the leaves). As a result, even if we know which edges should be cut, it may be hard to determine how to divide the resulting components among the terminals. The key idea of our $(2 + \epsilon)$ approximation algorithm is to separate the stage of finding connected pieces of the graph from the stage of partitioning them among the terminals. Then, in the first stage, the problem of finding "good" pieces is solved optimally, and in the second

stage these pieces are combined to form a 2-approximate solution. To make the dynamic programming algorithm of the first stage run in polynomial time, the edge capacities are rounded, leading to an overall $(2+\epsilon)$-approximation.

3.2 Min-max multiway cut in general graphs

3.2.1 Approximation algorithm

Our main goal in this section is to provide an approximation algorithm for the min-max multiway cut problem and its extension with nonuniform bounds on the capacities.

First we briefly recall the 2-approximation algorithm of Dahlhaus et al. [15] for the multiway cut problem, as it is useful to understand why it does not work for the min-max version. This algorithm finds, for each terminal x_i, a minimum capacity cut (S_i, T_i) separating this terminal from all other terminals. If among all minimum cuts, the one with the smallest size of S_i is chosen, then the sets S_i for different i are disjoint. The algorithm then cuts all edges in these min-cuts, which results in a 2-approximation[1] for the multiway cut. However, this procedure may leave some nodes in the graph which do not belong to any S_i component, which is not an acceptable solution to the min-max multiway cut problem. But if this set of left-over nodes, which may have many edges coming out of it, is included with one of the terminals, then the min-max objective function suddenly increases to a value up to the sum of all but one of the min-cuts.

The idea of our algorithm is to take cuts around each terminal that are larger in size than the minimum cut, in the hope that no unassigned nodes will remain.

[1]It can be improved to a $2(1-1/k)$ approximation by taking all except the largest min-cut.

Assume that we are given a bound B, and assume that there is a multiway cut where each side has capacity at most B. We use a binary search scheme to optimize B. For a given value of B, we use a subroutine for the MaxSBCC problem from Definition 2.1.1. We show how to use any (α, β)-bicriteria approximation algorithm for MaxSBCC (see Definition 2.1.2) as a subroutine for solving the min-max multiway cut problem. Recall that in Section 2.3.3, a specific $(O(log^{3/2}n), 1)$ algorithm was presented.

The idea is analogous to the greedy $\log n$-approximation for the set-cover problem. Starting from the set V of unassigned nodes of the graph, our algorithm iteratively finds (approximate) maximum size bounded capacity cuts around each terminal, and temporarily assigns nodes to terminals, until no unassigned nodes remain. One important difference is that our algorithm is not greedy, in the sense that assignment made to terminals in one iteration can be revised in later iterations if that becomes useful. The full algorithm is shown in the figure as Algorithm 2.

Theorem 3.2.1 *If there is an (α, β)-approximation algorithm for MaxSBCC, then algorithm 2 is an $O(\alpha \log_{1+\beta} n)$-approximation for the min-max multiway cut problem.*

The key to the analysis is to see that each iteration of the `while` loop assigns a constant fraction of the remaining nodes. By assumption there is a multiway cut $(S_1^*, ..., S_k^*)$ with maximum capacity B. For each terminal x_i, we use the approximation bound to claim that the application of the MaxSBCC algorithm assigned at least as many new nodes to x_i as a β fraction of the remaining nodes in S_i^*.

Lemma 3.2.2 *If there is a multiway cut with maximum capacity at most B, then in any iteration of the while loop, if U is the set of unassigned nodes in the beginning*

Algorithm 2 Algorithm for the min-max multiway cut problem

1: Initialize $S_i = \{x_i\}$ for $i = 1, ..., k$

2: Initialize weights $w(v)$ for all $v \in V$ by setting $w(x_i) = 0$ for all i, and $w(v) = 1$ for all other nodes

3: **while** $\bigcup_i S_i \neq V$ **do**

4: **for all** terminals $x_i \in X$ **do**

5: Construct a graph G' labeling x_i as a sink t and contracting all other terminals into a single source s.

6: Find an (α, β)-approximate MaxSBCC $(S, \overline{S})$ in graph G' with bound B and weights $w(v)$, trying to maximize the weight of $\overline{S}$. Note that the set $\overline{S}$ does not have to contain S_i and does not have to be disjoint from the other sets S_j for $j \neq i$.

7: Consider the intersection $I_j = \overline{S} \cap S_j$ for each $j \neq i$. We need to delete this intersection either from S_j or from $\overline{S}$. If $c(I_j, S_j \setminus I_j) < c(I_j, \overline{S} \setminus I_j)$, then let $S_j = S_j \setminus I_j$; otherwise let $\overline{S} = \overline{S} \setminus I_j$.

8: Let $S_i = S_i \cup \overline{S}$, and set the weights of all $v \in \overline{S}$ to $w(v) = 0$.

9: **end for**

10: **end while**

11: Return $S_1, ..., S_k$.

of the iteration, and U' is the set of unassigned nodes at the end of this iteration, then $|U'| \leq \frac{1}{1+\beta}|U|$.

Proof. Let N_i be the set of previously unassigned nodes added to the set S_i in this iteration. Notice that step (7) of the algorithm only reassigns nodes with zero weight, so N_i has the same weight as the solution to MaxSBCC $\overline{S}$ obtained in step (6).

Consider some optimal solution $(S_1^*, ..., S_k^*)$ to the min-max multiway cut instance. Now partition U' into sets $U_1', ..., U_k'$, such that $U_i' = S_i^* \cap U'$. We claim that $w(N_i) \geq \beta \cdot |U_i'|$. To see this, notice that the nodes in U' have weight 1 throughout this iteration of the `while` loop, and since S_i^* is a piece of the optimal partition, $\delta(S_i^*) \leq B$. Therefore, in the i^{th} iteration of the `for` loop, $(V \setminus S_i^*, S_i^*)$ is a feasible solution to the MaxSBCC problem, and $w(S_i^*) \geq w(U_i') = |U_i'|$. By the (α, β)-approximation guarantee, the algorithm for MaxSBCC must find a set with $w(N_i) \geq \beta \cdot |U_i'|$. Summing over all i, we obtain that $|U| - |U'| = \sum_{i=1}^{k} w(N_i) \geq \beta \cdot |U'|$, which proves the lemma. □

Proof of Theorem 3.2.1 : By using binary search, we can assume that a bound B is given, and our algorithm will either prove that no multiway cut of maximum capacity at most B exists, or it will find a multiway cut with maximum capacity at most $O(\alpha \log_{1+\beta} n)B$.

Throughout the algorithm, $x_i \in S_i$ for all i, and the sets S_i are always disjoint. So the algorithm finds a multiway cut, as required. By Lemma 3.2.2 the algorithm terminates in at most $\log_{1+\beta} n$ iterations of the `while` loop, if a min-max multiway cut of capacity at most B exists. If given an infeasible bound $B < B^*$, it may not stop after $\log_{1+\beta} n$ iterations, which proves that $B < B^*$. This shows that

the algorithm runs in polynomial time. We will also use this bound to give an approximation guarantee for the algorithm.

We claim that for each S_i returned by the algorithm, $\delta(S_i) \leq \alpha \log_{1+\beta} n \cdot B$. To see this, notice that for each application of the MaxSBCC subroutine in (6), the capacity of the set $\overline{S}$ returned is at most $\delta(\overline{S}) \leq \alpha B$. By the choice made in step (7), the transfer operation does not increase either $\delta(S_j)$ or $\delta(\overline{S})$. So in each iteration of the `while` loop, the capacity of each S_i increases by at most αB. Combined with the bound on the number of iterations, this observation concludes the proof. □

Combined with Theorem 2.3.4, we also get the following result.

Corollary 3.2.3 *There is a $O(\log^{5/2} n)$-approximation algorithm for min-max multiway cut.*

It is interesting to note that the algorithm can also be used for a version of the multiway cut problem in which there is a separate bound B_i for each $\delta(S_i)$. To obtain the extension, we use the MaxSBCC algorithm in each iteration i of the `for` loop with bound B_i rather than B.

Theorem 3.2.4 *Assume that we are given a graph G with k terminals, edge capacities, and k bounds $(B_1, \ldots, B_k)$. If there is a multiway cut $(S_1, \ldots, S_k)$ such that $\delta(S_i) \leq B_i$ for each i, then in polynomial time we can find a multiway cut $(S'_1, \ldots, S'_k)$ such that $\delta(S'_i) \leq O(\log^{5/2} n)B_i$, and the bound improves by a factor of $\sqrt{\log n}$ for graphs excluding any fixed graph as a minor.*

Remark Calinescu, Karloff and Rabani [6] and subsequently Karger et al. [44] gave improved approximation algorithms for the multiway cut problem based on

linear programming and rounding. It appears that this technique does not yield a good approximation for our problem. To see this, consider the graph which is a star with k terminals and a single additional node at the center, and assume the capacity of each edge is 1. There is no multiway cut where each part has capacity at most $B = 2$, or even approximately 2. By assigning the center of the star to terminal x_i, we can get a multiway cut where the capacity of each part S_j for $j \neq i$ is 1, while the capacity of S_i is $k-1$. A linear programming relaxation would allow us to take a "linear combination" of these cuts, and thereby have each side have capacity at most 2.

3.2.2 NP-completeness of min-max multiway cut

We prove using a reduction from *bisection* that the min-max multiway cut problem is NP-hard already on graphs with 4 terminals.

Theorem 3.2.5 *Min-max multiway cut is NP-hard for any fixed $k \geq 4$ even with unit-capacity edges.*

Proof. We show that it is NP-hard for $k = 4$ using a reduction from the graph bisection problem [29]. Our construction uses capacities, but we can replace each edge with multiple parallel paths. An instance of the bisection problem consists of a graph $G = (V, E)$ with an even number of vertices n, and an integer C. The question is whether or not there exists a partition of V into two sets Y and Z, each of size $n/2$, such that the capacity of the cut $c(Y, Z) \leq C$. Given G and C, we construct, in polynomial time, a graph H with 4 terminals and a bound B, so that H has a multiway cut with maximum capacity at most B if and only if G has a bisection with capacity at most C.

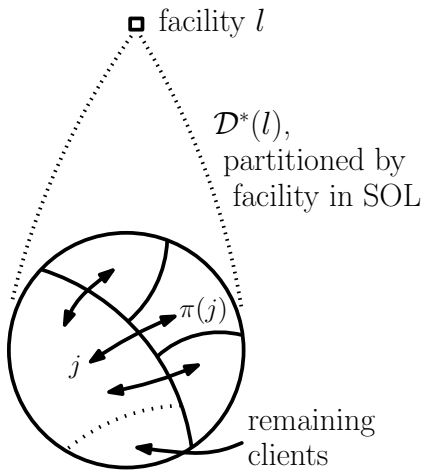

Figure 3.1: Reduction from *bisection* to *min-max multiway cut*

We obtain the graph $H = (V', E')$ by adding 4 new terminal nodes $X = \{u, d, l, r\}$ to G, and adding edges that connect nodes of G to the terminals (see Figure 3.1). E' includes E and the following additional edges, where a is chosen such that $2a > C$:

- Edge (u, d) of capacity na
- Edges (v, u) and (v, d), each of capacity a, for each $v \in V$.
- Edges (v, l) and (v, r), each of capacity $b = a - \frac{C}{n} > 0$, for each $v \in V$.

The bound is set to $B = 2na$.

Suppose H has a min-max multiway cut $(U \cup \{u\}, D \cup \{d\}, L \cup \{l\}, R \cup \{r\})$ where each part has capacity at most B. Then U and D must be empty, as $B = 2na \geq \delta(U \cup \{u\}) \geq 2na + 2b|U|$, just counting the edges to the terminals. So (L, R) is a cut of G, and let $C' = c(L, R)$ denote its capacity. The next observation is that $|L| = |R| = n/2$. To see this, suppose, for contradiction, that $|L| = k \geq \frac{n}{2} + 1$ (or similarly for $|R|$). Then

$$\delta(L \cup \{l\}) = 2ka + nb + C' \geq 2(\frac{n}{2} + 1)a + n(a - \frac{C}{n}) = 2na + (2a - C) > B,$$

where the last inequality follows from the choice of a. We conclude that the capacity C' of the bisection (L, R) must be at most C. This follows as the capacity of the cut $L \cup \{l\}$ is $na + nb + C' \leq B = 2na$, and by the choice of b this inequality implies $C' \leq C$. To show the opposite direction, given a bisection (Y, Z) in G of capacity $C' \leq C$, we produce a min-max multiway cut $(\{u\}, \{d\}, Y \cup \{l\}, Z \cup \{r\})$ of H, with each component's capacity at most B. □

3.3 Min-max multiway cut on trees

Recall from the Introduction that in an optimal solution to the min-max multiway cut problem on trees the sets of nodes assigned to the terminals do not have to be connected. This can be seen in the example of Figure 3.2. All nodes except for the middle one are terminals, and all edges have capacity 1. The optimal solution cuts all the edges incident on the middle node and assigns it to one of the leaf (degree-one) terminals, achieving a value of 4. On the other hand, any solution that assigns connected parts of the graph to each terminal would leave the middle node connected to one of its neighbors, incurring a cost of 5.

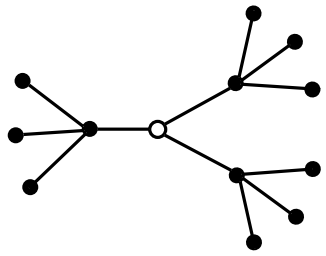

Figure 3.2: Example showing that in an optimal min-max multiway cut on a tree, the sets assigned to the terminals need not form connected components. The only non-terminal is the middle node

In Section 3.3.1 we use this observation to prove that the min-max multiway cut problem in NP-hard on trees. Then we provide a $(2+\epsilon)$-approximation for the case that the graph is a tree.

3.3.1 NP-hardness of min-max multiway cut on trees

Theorem 3.3.1 Min-max multiway cut *is strongly NP-hard when the graph is a tree with weighted edges.*

Proof. We use a reduction from *3-partition*, which is known to be strongly NP-complete [28]. In 3-partition, given a set $A = \{a_1, ..., a_{3m}\}$, a weight w_i for each $a_i \in A$, and a bound B, such that $\forall i \ \frac{B}{4} < w_i < \frac{B}{2}$ and $\sum_{i=1}^{3m} w_i = mB$, we want to know if A can be partitioned into disjoint sets $S_1, ..., S_m$, such that for each j,

$$\sum_{a_i \in S_j} w_i = B.$$

Given an instance (A, B) of 3-partition, we construct an instance of min-max multiway cut as follows. The tree T consists of separate subtrees connected with zero-capacity edges. There will be $3m$ subtrees T_i, one for each element a_i, and m isolated terminals $x_1, ..., x_m$, one for each of the desired sets. Each T_i consists of six terminals and one non-terminal v_i, with edge capacities as in Figure 3.3.

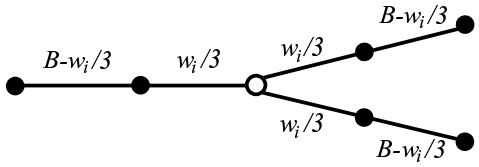

Figure 3.3: Component T_i used in the NP-completeness reduction for min-max multiway cut on trees

We claim that a min-max multiway cut of maximum capacity at most B exists if and only if the 3-partition instance is solvable. Notice that any min-max multiway cut with capacity at most B must cut all edges of T_i and assign all v_i's to the terminals $x_1, ..., x_m$, creating a partition of A. If a set of nodes S'_j is assigned to terminal x_j, then the capacity of the resulting part is $\sum_{v_i \in S'_j} w_i$. This implies that such a cut exists if and only if the 3-partition does. □

3.3.2 Algorithm for min-max multiway cut on trees

In this section we give a $(2+\epsilon)$-approximation algorithm for the min-max multiway cut on trees that have edge capacities.

The algorithm consists of two stages. In the first stage we consider a variant of the problem where we allow the algorithm to create extra parts in the partition that do not contain terminals. More precisely, we consider the following problem.

Definition 3.3.2 *The* tree cutting *problem (T, X, B) for a tree $T = (V, E)$, terminals $X = \{x_1, ..., x_k\} \subseteq V$, and a bound B is to find a partition of V into connected subtrees $T_1, ..., T_h$, subject to the following constraints: (1) no two terminals are in the same connected component; and (2) for each connected component T_i, $\delta(T_i) \leq B$. The objective is to minimize $\sum_i \delta(T_i)$.*

In the next subsection we give a pseudo-polynomial time algorithm for this problem. Here we show how to use such an algorithm to get a $(2+\epsilon)$-approximation for the min-max multiway cut on trees.

Theorem 3.3.3 *Using a pseudo-polynomial time algorithm for the tree cutting problem as a subroutine, we can give a polynomial time $(2+\epsilon)$-approximation for*

min-max multiway cut on trees.

Proof. First we give a 2-approximation for min-max multiway cut that uses a pseudo-polynomial exact algorithm for tree cutting. Given a tree T with terminals X, we will binary search for a lower bound B^* to the min-max multiway cut optimum. Observe that connected components in a feasible solution to min-max multiway cut instance (T, X) of value B give a feasible solution to the tree cutting instance (T, X, B) of value at most kB. Therefore, if our optimal tree cutting solution $T_1, ..., T_h$ does not satisfy $\sum_{i=1}^{h} \delta(T_i) \leq kB$, then $B < B^*$. The algorithm groups the components T_i into k sets $S_1, ..., S_k$ of nodes greedily, first assigning the terminals to different sets, and then assigning each next component to the set with lowest sum of $\delta(T_i)$ over components already in that set. Observe that if several components, say $T_1, ..., T_j$, are combined into a set S, then $\delta(S) \leq \sum_{i=1}^{j} \delta(T_i)$. Because $\sum_{i=1}^{h} \delta(T_i) \leq kB$, and for all j, $\delta(T_j) \leq B$, for no i will $\delta(S_i)$ exceed $2B$.

Recall that our tree cutting algorithm runs in pseudopolynomial time. We obtain a polynomial-time $(2 + \epsilon)$-approximation algorithm via rounding. For a given ϵ, capacity bound B, and $m = |E|$, let

$$\alpha = \frac{\lceil m/\epsilon \rceil}{B}.$$

For each edge $e \in E$, scale the capacity so that $c'(e) = \lfloor \alpha c(e) \rfloor$. Also set $B' = \alpha B = \lceil m/\epsilon \rceil$. If there is a multiway cut with maximum capacity B in the original problem, then there is one of maximum capacity at most B' after rounding. Now we obtain a 2-approximate multiway cut $S_1, ..., S_k$ for the graph with capacities $c'(e)$ and bound B'. The running time is polynomial in $B' = \lceil m/\epsilon \rceil$ and n. The capacity of a part S_i of this partition is at most $\delta(S_i) \leq (2 + \epsilon) \cdot B$ using the original capacities. □

3.3.3 Algorithm for the tree cutting problem

We now describe an algorithm that solves optimally, in time polynomial in B and the size of the tree n, the tree cutting problem (which we use as a subroutine for the min-max multiway cut on trees). To simplify the presentation of the algorithm, assume, without loss of generality, that (1) T is rooted at a node r and all edges are directed away from the root; (2) T is binary. (To make the tree binary without affecting the solution, replace each node u that has $d > 2$ children with a $\lceil \log_2 d \rceil$-height complete binary subtree U with edge capacities $B+1$, and attach u's children to the leaves of U, at most 2 per leaf.)

The tree cutting problem will be solved using dynamic programming. We construct a dynamic programming table $p(v, A, t)$ for all nodes $v \in V$, integers $0 \leq A \leq B$, and two values of the binary variable $t \in \{0, 1\}$. The entry $p(v, A, t)$ is the minimum total capacity of edges in the subtree of T rooted at v that can be cut such that the total capacity of edges coming *out* (i.e., toward descendants) of v's component is at most A. The variable t indicates whether the component containing v contains a terminal in v's subtree ($t = 1$) or not ($t = 0$). We have the separate bound A because the remaining $B - A$ capacity will be used to cut the edges that are incident on v's component, but lie outside of its subtree. The values $p(v, A, t)$ can be computed in a single pass up the tree.

To initialize the values at the leaf nodes, for all values of A we set $p(v, A, 1) = 0$ and $p(v, A, 0) = \infty$ for a leaf node $v \in X$ which is a terminal, and $p(v, A, 1) = \infty$, $p(v, A, 0) = 0$ for a leaf node $v \notin X$ which is not a terminal. Table entries for the internal nodes are computed using the entries for their children nodes. We describe

the case when the internal node v is a terminal, and it has one child node v_1. In this case we set $p(v, A, 0) = \infty$ for all A, because a terminal node v cannot be in a component without a terminal. For $A < c(v, v_1)$, we set $p(v, A, 1) = p(v_1, A, 0)$, because in that case the edge (v, v_1) cannot be cut as that would violate the constraint imposed by A on the capacity of edges coming out of v's component. For $A \geq c(v, v_1)$, we set

$$\begin{aligned} p(v, A, 1) \;=\; \min \;\{\; & p(v_1, A, 0), \\ & c(v, v_1) + p(v_1, B - c(v, v_1), 1), \\ & c(v, v_1) + p(v_1, B - c(v, v_1), 0) \;\}, \end{aligned}$$

where the first option corresponds to not cutting the edge (v, v_1) and using a component below v_1 without a terminal (since we cannot allow two terminals in the same component); the second and third options correspond to cutting the edge (v, v_1) and leaving v_1 in a component with a terminal or without a terminal, respectively. Cutting (v, v_1) implies that the component containing v_1 can have at most $B - c(v, v_1)$ capacity of edges leaving it below v_1, which leads to the above expression $c(v, v_1)+p(v_1, B-c(v, v_1), \cdot)$ for the total capacity of edges in the subtree below v that are cut.

Analogous recurrences can be derived for the case when v is not a terminal, and the case when v has two children, v_1 and v_2. In the case of two children, the capacity A available for cutting edges below v has to be partitioned between the edges that belong to the left subtree (including, possibly, the edge (v, v_1)), and the ones that belong to the right subtree (possibly including (v, v_2)). The algorithm tries all possibilities for such a partition $A_1 + A_2 = A$. Then, given A_i, it considers the possibilities for each child node v_i of cutting or not cutting (v, v_i),

using expressions similar to the ones above. In the case that v is a terminal, it can only be connected to subtrees not containing terminals, and in the case that v is not a terminal, it can be connected to one subtree with and one without terminals, or to two subtrees without terminals.

The final partition of the tree is derived from the minimum of two table entries $p(r, B, 1)$ and $p(r, B, 0)$, where r is the root of the tree.

Theorem 3.3.4 *The optimal solution to the* tree cutting *problem can be computed in time polynomial in the size of the graph and the bound B.*

CHAPTER 4

FACILITY LOCATION WITH HIERARCHICAL FACILITY COSTS

4.1 Introduction

In the basic facility location problem, we are given a set of clients and a set of possible facilities. A solution consists of opening some facilities and assigning each client to an open facility, with the objective of minimizing the sum of the facility opening cost and the client connection cost. In the most basic and well-studied version of the problem, the metric uncapacitated facility location, there is a metric of distances between the clients and the facilities, and the connection cost is the sum of distances between clients and the facilities to which they are assigned. Each facility has a cost given as part of the input, and the total facility opening cost is the sum of costs for the facilities that are opened. Facility location problems have been used to model a wide range of practical settings, including location problems, supply chain management, Web server locations, etc. While even the basic uncapacitated metric facility location problem is NP-complete, there are many good approximation algorithms known for it, using the whole range of techniques including local search, linear programming and the primal-dual method.

In this chapter, we study a variant of the facility location problem in which the cost of a facility depends on the specific set of clients assigned to that facility. We propose a general model in which the facility costs are submodular functions of the set of clients assigned to the facility, and give an $O(\log n)$-approximation algorithm for it, where n is the number of clients. Submodularity of the cost functions models a natural economy of scale. We then focus on a subclass of submodular cost functions which we call hierarchical costs. Shmoys, Swamy and

Levi [61] and Ravi and Sinha [58] introduced the problem of facility location with service installation costs. In their model, each client requests a certain service, which has to be installed at the facility where this client is assigned. There are costs for opening facilities and for installing services. The hierarchical facility location problem is an extension of this model to many levels of service costs. Instead of allowing only two levels (facility opening and service installation), we allow an arbitrarily deep hierarchy that describes the costs. Such a hierarchy can be used to model a richer set of cost structures. Our main result is a local search algorithm that gives a $(4.237 + \epsilon)$-approximation (independent of the number of levels in the hierarchy) in the case that the costs are identical at all facilities.

Our models and methods We introduce the general problem of facility location with submodular facility costs, and then focus on a special case, called facility location with hierarchical facility costs. A *submodular* function $g : 2^{\mathcal{D}} \rightarrow \mathbb{R}$ on a set $\mathcal{D}$ is one that satisfies the inequality $g(A) + g(B) \geq g(A \cup B) + g(A \cap B)$ for any $A, B \subseteq \mathcal{D}$. Equivalently, for any $A \subseteq B \subset \mathcal{D}$ and an element $x \notin B$, it satisfies $g(A \cup \{x\}) - g(A) \geq g(B \cup \{x\}) - g(B)$. Submodularity models decreasing marginal costs and is a natural property of functions in many settings.

Definition 4.1.1 Facility location with submodular facility costs problem *has as input a set of facilities $\mathcal{F}$, a set of demands or clients $\mathcal{D}$, a distance metric dist on the set $\mathcal{F} \cup \mathcal{D}$, and a monotone non-decreasing submodular function $g_i : 2^{\mathcal{D}} \rightarrow \mathbb{R}^+$ for each facility $i \in \mathcal{F}$. The goal is to find a facility $f(j) \in \mathcal{F}$ for each client $j \in \mathcal{D}$ so as to minimize the sum of the connection cost, $\sum_{j \in \mathcal{D}} dist(j, f(j))$, and the facility cost, $\sum_{i \in \mathcal{F}} g_i(\mathcal{D}(i))$, where $\mathcal{D}(i) = \{j \in \mathcal{D} : f(j) = i\}$ denotes the set of clients assigned to facility i.*

We give an $O(\log|\mathcal{D}|)$-approximation algorithm for facility location with submodular facility costs in Section 4.2, using a greedy algorithm with a submodular function minimization subroutine.

In the rest of the chapter we focus on a special case of this problem, in which the form of the functions g_i is restricted to what we call hierarchical cost functions. It is not hard to see that these hierarchical functions are in fact submodular. Moreover, we assume that the cost functions of all facilities are the same, i.e. $g_i = g$ for all $i \in \mathcal{F}$. Note that the problem with independent cost functions on different facilities is *set cover*-hard, as was shown in [61]. The hierarchical facility cost function g is specified by a rooted cost tree T, whose set of leaves is $\mathcal{D}$ (the set of clients). The leaves of the tree can also be thought of as the services that the corresponding clients request. Each node k of T has a non-negative cost, $cost(k)$. The facility cost function g is defined using the tree T as follows. For a set $\mathcal{D}(i) \subseteq \mathcal{D}$ of clients assigned to facility i, we use the notation $T_{\mathcal{D}(i)}$ to denote the subgraph of T induced by the nodes that lie on a path from the root to some leaf (client) in $\mathcal{D}(i)$ (see Figure 4.1). Then the facility cost of i is $\sum_{k \in T_{\mathcal{D}(i)}} cost(k)$. We also use $cost(S)$ as a shorthand for the facility cost of a set of clients $S \subseteq \mathcal{D}$, i.e. $cost(S) = \sum_{k \in T_S} cost(k)$. The cost of an empty facility is zero.

Our main result is a local search method, where the locally optimal solutions are 5-approximations to the problem, in the case that the costs are identical at all facilities. Then we improve this bound to 4.237 by scaling. The algorithm starts with an arbitrary assignment of clients to facilities, and performs two types of local improvement moves, *aggregate* and *disperse*, while the objective function of the solution improves. The aggregate move is analogous to the move *open*, which opens a new facility, used in local search algorithms for the traditional facility

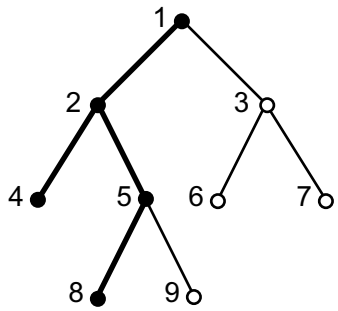

Figure 4.1: Example of a cost tree. If clients corresponding to leaves 4 and 8 form the set $\mathcal{D}(i)$ which is assigned to a facility i, then the cost of i is $F(i) = cost(1) + cost(2) + cost(4) + cost(5) + cost(8)$. Shaded nodes and thick edges form the subgraph $T_{\mathcal{D}(i)}$.

location problem (e.g. [8, 3]). In that case, however, once a facility is open, it is easy to decide which clients should be assigned to it. In the case of our hierarchical service cost model, this is less easy to decide. In fact, there is no clear meaning of opening a facility, as the cost may depend dramatically on which clients are assigned to it. We describe the move in Section 4.3, where we also prove that if there are no improving aggregate moves, then the connection cost of the solution can be bounded.

To bound the facility costs, we use a different operation, *disperse*, which is the analog of the traditional *close* operation. The high-level idea is to disperse the clients of one currently open facility to other facilities, and hence to save on facility cost. Implementing such a disperse move optimally can be used to solve the original problem in a single step. In Section 4.4 we describe an approximate disperse move that can be implemented in polynomial time. We then use the aggregate and disperse moves to show that a locally optimal solution is a 5-approximation for the problem.

Using scaling we improve the bound to 4.237, and accepting only sufficiently large improvements, we turn this into a polynomial-time $(4.237+\epsilon)$-approximation algorithm for the hierarchical facility location problem.

Motivation Our hierarchical facility location problem can be used to model practical scenarios, such as the costs of multiple levels of service installation [17]. For example, one may consider opening some stores and wonder about which items to carry in each of them. The cost of opening a store would be represented by the root of the tree, the cost of carrying a particular category of items (e.g., those supplied by a particular vendor) would be at a node one level down, that for a subcategory still lower, and so on. A client may then be identified with the item he wants to buy.

Another possible application is for data storage. Then facilities are the file servers, and clients are the users (or contributors) of data. The connection cost is the price of transferring data over the network, and facility cost is the price for storing the data. The hierarchy may represent how similar the data items are, with the assumption that similar files can be compressed together to save storage cost.

The generalization of our problem in which the facility cost functions are allowed to differ from each other by constant factors can model data gathering in sensor networks (see, for example, [68]). In that case, instead of representing storage expenses, the facility costs would model the cost of delivering compressed data to the destination.

Related work There has been much work on approximation algorithms for the uncapacitated facility location problem. Small constant factor approxima-

tion algorithms exist that use essentially every known approximation algorithmic technique, including local search [48, 8, 3], primal-dual method [42], and linear programming and rounding [62, 10, 11, 63]. The current best approximation guarantee is 1.52 [54], and no better than 1.463 approximation is possible unless $NP \in DTIME[n^{O(\log \log n)}]$ [32].

The most well-studied extension of the facility location problem considers facilities with capacities, where the capacity of a facility bounds the number of clients it can serve if opened. Many of the uncapacitated approximation algorithms extend to the version of the problem with soft capacities [42, 8, 3, 11], where soft capacities allow us to open a facility multiple times, paying the opening cost each time, in order to support more clients. Similarly many of the approximation algorithms can be used to derive bicriteria approximations for the capacitated case, by allowing the algorithm to violate the capacities to some extent. Local search is the only known algorithmic technique that extends to handle hard capacities, i.e., it gives an approximation algorithm that neither violates the capacities, nor opens the facilities multiple times. The most general such problem formulation is the universal facility location [34, 53, 30], where the cost of a facility is an arbitrary monotone function of the number of clients assigned to it.

In contrast, in the hierarchical facility location problem we consider, the facility cost depends on the *set of clients* assigned to the facility and not just their number. Shmoys, Swamy and Levi [61] and Ravi and Sinha [58] introduced the facility location problem with service installation costs (called multicommodity facility location in [58]), where the costs can be represented by a tree of height two. Ravi and Sinha consider the version of the problem where the facility costs can be arbitrarily different for different facilities, and they develop a logarithmic-factor

approximation, which is the best possible in that case. Shmoys et al. restrict the model by assuming that facilities have an ordering where each earlier facility costs less than a later one for all services, which allows them to design a constant-factor approximation. We make an even more restrictive assumption of facility costs being identical on all facilities. However, in contrast to the two-level model, our algorithm works for arbitrary number of levels and yields a bound of 5, independent of the number of levels in the tree.

The group facility location problem of Hayrapetyan, Swamy and Tardos [37] contains a similar two-level cost structure, but for connection costs. For facility costs they use the uncapacitated model.

Notation Finally, we define some further notation that will be useful in the rest of the chapter. At each step of the local search, the algorithm has some assignment of clients to the facilities which constitutes its current solution. We call this solution SOL. For the analysis of the algorithm, we assume that SOL is a locally optimal solution, and we compare it to a fixed globally optimal one, which we call OPT. Let C and F denote the connection and facility costs of SOL, respectively, and C^* and F^* denote the same quantities for OPT. For a client j, let $f(j)$ be the facility where j is assigned in SOL and $f^*(j)$ be the facility where j is assigned in OPT. For a facility i, $\mathcal{D}(i)$ denotes the set of clients assigned to i in SOL, and $\mathcal{D}^*(i)$ is the set assigned in OPT. If a set of clients S is moved to a facility i that already contains some clients $\mathcal{D}(i)$, then $cost_i(S)$ is the additional facility cost that has to be paid, and is equal to $cost(S \cup \mathcal{D}(i)) - cost(\mathcal{D}(i))$.

4.2 Approximation for submodular costs

In this section we present a logarithmic approximation for facility location with submodular facility costs. The procedure that we present works for the case that the submodular facility cost functions are different for different facilities, and does not require the distances to form a metric. The guarantee is the best possible for this case, because either one of these two generalizations makes the problem *set cover*-hard. However, for the case of identical facility costs and metric distances, we do not know of any lower bounds that would rule out the possibility of a constant-factor approximation.

Our algorithm is obtained by a reduction to the set cover problem, with a polynomial-time subroutine for implementing the greedy algorithm on this instance with exponentially many sets. Such an approach has been used for the plain uncapacitated facility location problem by Hochbaum [38], and is mentioned in [58] as a possibility for obtaining a logarithmic approximation for the facility location with service installation. Thus, our main contribution is to show how to find the best facility and a set of clients for one step of the greedy set cover algorithm.

Facility location can be viewed as an instance of set cover in the following way. The set of clients $\mathcal{D}$ is the set of elements to be covered. For each facility $i \in \mathcal{F}$ and each possible subset of clients $S \subseteq \mathcal{D}$, there is a set $A(i, S)$ in the set cover instance, covering the elements in S, whose cost is $g_i(S) + \sum_{j \in S} dist(i, j)$, the facility and connection cost of assigning clients S to facility i. Because of monotonicity of the functions g_i, any set cover solution can be transformed into one in which the selected sets are disjoint, thus forming a feasible facility location solution, without increase in cost.

A greedy algorithm that repeatedly selects a set minimizing the ratio of the

set's cost to the number of newly-covered elements is well-known to be a $O(\log n)$-approximation for the set cover problem [67], where n is the number of elements, which, in our case, is the number of clients. However, in our case such a set cannot be found by simple enumeration, because the instance of set cover that we describe has a number of sets exponential in the number of clients of the corresponding facility location instance. So we describe a procedure that finds a set $A(i, S)$ minimizing the ratio of its cost to the number of newly-covered clients in polynomial time, using submodular function minimization [40].

At each step of the greedy algorithm, in order to keep track of which clients have already been covered and which have not, let us assign a weight $w_j = 0$ for all covered clients j, and a weight $w_j = 1$ for all clients that have not been covered yet. To find the best facility i for the set $A(i, S)$, we simply try all of them, finding the best set S for each one, and then select the one with the best ratio. So for a given facility i, the task is to find a set $S \subseteq \mathcal{D}$ minimizing $\frac{g_i(S)+\sum_{j\in S} dist(i,j)}{\sum_{j\in S} w_j}$, where the denominator is just the number of clients in S that have not been covered yet. To minimize this ratio, we can do a binary search for the minimum value α for which there exists a set S such that $\frac{g_i(S)+\sum_{j\in S} dist(i,j)}{\sum_{j\in S} w_j} \leq \alpha$, or, equivalently, $g_i(S) + \sum_{j\in S} dist(i, j) - \alpha \cdot \sum_{j\in S} w_j \leq 0$. The left-hand side of this last expression is a submodular function, as g_i is submodular by assumption, and the last two terms are modular functions (i.e. ones for which the submodular inequality holds with equality). Thus it can be minimized in polynomial time.

4.3 Aggregate move and the connection cost

In this section we consider one of the two local improvement moves that are performed by the local search algorithm for facility location with hierarchical costs.

When this move, *aggregate*, is performed on a facility i, it transfers some clients from other facilities to i. This change in assignment of clients affects the cost of the solution in three ways: the clients that change assignments have a changed connection cost; facility i has an increased facility cost; and other facilities have decreased facility costs as some clients move away from them. As we show at the end of this section, it is possible to find, in polynomial time, the optimal set of clients to be transferred to facility i, even for the case of general submodular facility cost functions.

First, we present a simpler approximate way of evaluating an aggregate move, and a much more efficient algorithm for finding the best set of clients according to this approximate evaluation. Rather than exactly computing the cost of the new assignment, we ignore the savings in cost at the other facilities, and we find a set of clients $S \subseteq \mathcal{D}$ minimizing the change in connection cost plus the added facility cost incurred at facility i. This estimated change in cost can be expressed as

$$\sum_{j \in S} dist(j, i) - \sum_{j \in S} dist(j, f(j)) + cost_i(S). \tag{4.1}$$

We call (4.1) the *value* of the aggregate move (despite the fact that expression (4.1) does not take into account the possible cost-savings at other facilities), and we call the move *improving* if it has negative value. Note that if a move is performed, then the real cost of the solution decreases by at least as much as estimated using the move's value, which means that the solution is indeed improving when we make improving moves. In the next subsection we show how to find the optimal set S with respect to expression (4.1) for a given facility i. Then we show that if a solution has no improving aggregate moves, its connection cost can be bounded.

4.3.1 Finding the aggregate move with optimal value

Consider the problem of finding a set of clients S that minimizes expression (4.1) for a particular facility i. The change in distance is equal to $dist(j,i) - dist(j,f(j))$ for each client j in the set S that we choose, and the added facility cost at i depends on S as well as the set of clients $\mathcal{D}(i)$ already at i: $cost_i(S) = \sum_{k \in T_S \setminus T_{\mathcal{D}(i)}} cost(k)$. Let us now construct a modified tree T' from T which will be useful for designing the algorithm. The structure of T' is the same as that of T; the only difference is in the costs associated with the nodes. We incorporate the change in connection cost associated with each client j into the cost tree, so that the algorithm that makes a pass over this tree would be able to take into account both the changes in the connection costs and in the facility costs simultaneously. In particular, for every client $j \in \mathcal{D}$, set the cost of its leaf to $cost_{T'}(j) = cost_T(j) + dist(j,i) - dist(j,f(j))$, where $cost_T(j)$ is the original cost of this leaf in the tree T. Note that this new cost may be negative. Also, we discount the tree nodes which are already paid for at i: for these nodes $k \in T_{\mathcal{D}(i)}$, set $cost_{T'}(k) = 0$. For all other nodes k we keep the original cost $cost_{T'}(k) = cost_T(k)$. The reason that the construction of T' is helpful is explained in the following lemma.

Lemma 4.3.1 *There exists an improving aggregate move for facility i if and only if there is a connected subgraph U of T', containing the root, such that the total cost of nodes in U is negative, i.e. $\sum_{k \in U} cost_{T'}(k) < 0$.*

Proof. Suppose there is a set of clients S such that moving S to i constitutes an improving aggregate move. Then the subgraph of T' which consists of all leaves that correspond to clients in S and the union of their paths to the root has total node cost exactly equal to expression (4.1), and therefore negative.

Conversely, suppose that there is a connected subgraph U of T', containing the root, with negative total node cost. Since the only nodes with negative cost are the leaves, we can assume without loss of generality that U consists of a union of paths from the root to a subset of leaves (any internal nodes of T' that are in U but do not have descendants in U can be removed without increasing the cost). Then we take S to be the clients corresponding to the leaves in U and observe that the value of moving S to i is equal to the total cost of nodes in U. □

Given this lemma, the problem of finding the set of clients S minimizing expression (4.1) reduces to the problem of finding a minimum-cost connected component of T' containing the root. The latter problem can be solved by a simple dynamic programming procedure. In a bottom-up pass through the tree, for each node k we find the cheapest subgraph of the subtree rooted at k which contains k. For a leaf node k, this subgraph is the node itself, and its cost is $cost_{T'}(k)$. For an internal node k, the cheapest subgraph contains k itself, as well as those subgraphs of its children whose costs are negative.

The above algorithm, together with Lemma 4.3.1, yields the following results.

Lemma 4.3.2 *The subset $S \subseteq \mathcal{D}$ that minimizes expression (4.1), and hence defines the aggregate move of minimum value for a given facility i, can be found in time $O(|T|)$. The aggregate move of minimum value over all facilities can therefore be found in time $O(|\mathcal{F}| \cdot |T|)$.*

4.3.2 Bounding the connection cost

Now consider a solution with no improving aggregate moves. We bound the connection cost of this solution in a similar way as used in local search algorithms for

other facility location problems.

Lemma 4.3.3 *The connection cost C of a locally optimal solution can be bounded by the optimal cost as $C \leq C^* + F^*$.*

Proof. If there are no improving aggregate moves, then expression (4.1) is non-negative for moving *any* set of clients to *any* facility i. We consider expression (4.1) for the set of clients $\mathcal{D}^*(i)$ that are assigned to i in OPT. We have that

$$\sum_{j \in \mathcal{D}^*(i)} dist(j,i) - \sum_{j \in \mathcal{D}^*(i)} dist(j, f(j)) + cost_i(\mathcal{D}^*(i)) \geq 0.$$

Using the fact that $cost_i(\mathcal{D}^*(i)) \leq cost(\mathcal{D}^*(i))$ and adding the inequalities for all facilities i, we get

$$\sum_j dist(j, f^*(j)) - \sum_j dist(j, f(j)) + \sum_i cost(\mathcal{D}^*(i)) \geq 0,$$

or $C^* - C + F^* \geq 0$, which implies that $C \leq C^* + F^*$. □

4.3.3 Aggregate move for general submodular functions

In this section we show that finding the optimal aggregate move for the facility location problem with submodular facility costs can be done in polynomial time. We do this by showing that for a particular facility i, finding a set of clients S which would minimize the cost of the resulting solution when transferred to i can be done using submodular function minimization. This optimization is done using exact measure of the change in solution cost, without making the simplification of ignoring costs at facilities other than i, as done in expression 4.1.

If $\mathcal{D}(i)$ is the set of clients currently at the facility i, then the goal is to find a set $S \subseteq \mathcal{D} \setminus \mathcal{D}(i)$ of clients which are currently at other facilities to be transferred to facility i so as to minimize the cost of the resulting solution, or, equivalently, to minimize the difference between the cost of the new solution and the cost of the current solution. This change in cost can be expressed as

$$\begin{aligned}\Delta(S) &= g_i(\mathcal{D}(i) \cup S) - g_i(\mathcal{D}(i)) + \sum_{j \in S} [dist(j,i) - dist(j,f(j))] \\ &+ \sum_{i' \neq i} [g_{i'}(\mathcal{D}(i') \setminus S) - g_{i'}(\mathcal{D}(i'))],\end{aligned}$$

where g_i is the facility cost of facility i, the sum over clients in S is the change in the connection cost, and the last sum is the change in facility costs of facilities other than i. We show that $\Delta(S)$ is a submodular function over the set $\mathcal{D} \setminus \mathcal{D}(i)$, which allows us to use a polynomial-time submodular function minimization algorithm [40, 59] for finding the optimal set S of clients to be transferred.

Lemma 4.3.4 *$\Delta(S)$ for $S \subseteq \mathcal{D} \setminus \mathcal{D}(i)$ is a submodular function.*

Proof. To show that $\Delta(S)$ is submodular, it suffices to show that it consists of a sum of several submodular functions. To verify the submodularity of the first term, $g_i(\mathcal{D}(i) \cup S)$, we observe that

$$\begin{aligned}&g_i(\mathcal{D}(i) \cup (A \cup B)) + g_i(\mathcal{D}(i) \cup (A \cap B)) = \\ &= g_i((\mathcal{D}(i) \cup A) \cup (\mathcal{D}(i) \cup B)) + g_i((\mathcal{D}(i) \cup A) \cap (\mathcal{D}(i) \cup B)) \\ &\leq g_i(\mathcal{D}(i) \cup A) + g_i(\mathcal{D}(i) \cup B),\end{aligned}$$

where the inequality follows by submodularity of g_i.

The second term, $-g_i(\mathcal{D}(i))$, is just a constant, as it does not depend on S.

The sum $\sum_{j \in S} [dist(j,i) - dist(j,f(j))]$ representing the change in distances is a modular function. The terms $-g_{i'}(\mathcal{D}(i'))$ are constant as well.

For terms of the form $g_{i'}(\mathcal{D}(i') \setminus S)$, we get

$$\begin{aligned} & g_{i'}(\mathcal{D}(i') \setminus (A \cup B)) + g_{i'}(\mathcal{D}(i') \setminus (A \cap B)) = \\ &= \; g_{i'}((\mathcal{D}(i') \setminus A) \cap (\mathcal{D}(i') \setminus B)) + g_{i'}((\mathcal{D}(i') \setminus A) \cup (\mathcal{D}(i') \setminus B)) \\ &\leq \; g_{i'}(\mathcal{D}(i') \setminus A) + g_{i'}(\mathcal{D}(i') \setminus B), \end{aligned}$$

by submodularity of $g_{i'}$, showing that $\Delta(S)$ is a submodular function. □

The ability to find the optimal aggregate move allows us to bound the connection cost of a solution which is locally optimal with respect to this move for the facility location problem with submodular facility costs. This is done in the same way as in Lemma 4.3.3 for hierarchical costs. Unfortunately, though, we do not know of an analogue to the disperse move that would work for this more general problem.

4.4 Disperse move and the facility cost

Next we consider the *disperse* move, which reassigns clients from one facility i to other facilities, decreasing the facility cost at i. We use this move to bound facility costs. The outline of this section is analogous to that of the previous one. First we define a certain class of disperse moves, then we show that the optimal move in this class can be found in polynomial time, and then we exhibit a particular set of moves that allows us to bound the facility cost of a locally optimal solution.

4.4.1 Definition of a disperse move

The idea of the disperse move is to move some of the clients from a particular facility i to other facilities. If we could find the optimal such move, then one disperse move would solve the whole problem: just start with a solution that assigns all clients to one facility, and do the optimal disperse move on that facility. As a result, we do not consider the most general version of a disperse move with the exact evaluation function, but instead restrict our attention to a subclass of moves and an approximate evaluation. We use approximate evaluations both for the change in connection costs, which we bound in the usual way with a triangle inequality, and for the change in facility costs, for which we have a more complex scheme. For both the connection and the facility costs, the estimated change in solution cost that we use is an upper bound on the true change in cost.

We consider removing all clients from facility i and distributing them among all the facilities, possibly putting some clients back on the (now empty) facility i. This operation affects the cost of the solution in the following ways: there is a change in connection cost for clients that are moved; facility cost decreases at i and increases at the facilities where the clients are placed. Using the triangle inequality, we upper bound the change in connection cost for each client by the distance between i and that client's new facility. The decrease in facility cost at i is just its whole cost in the current solution, $F(i)$. For the estimation of increase in facility costs we define the notion of a tree-partition.

A *tree-partition* $\mathcal{S}(\mathcal{D}(i)) = \{S_h\}$ of a set of clients $\mathcal{D}(i)$ is any partition that can be obtained by cutting a subset of edges of the tree $T_{\mathcal{D}(i)}$ and grouping the clients by the resulting connected components of their tree leaves (see Figure 4.2). For example, if no edges are cut, then all clients are in the same set; if all edges

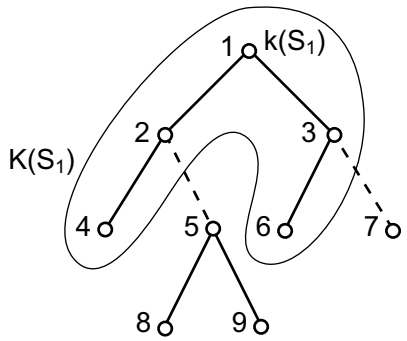

Figure 4.2: Example of a tree-partition. Cutting the two dashed edges produces three subsets: $S_1 = \{4, 6\}$, $S_2 = \{8, 9\}$, and $S_3 = \{7\}$. The marked component is $K(S_1)$.

incident on leaves are cut, then each client is in a separate set. Let us refer to the connected component associated with a set S_h as $K(S_h)$. A *disperse move* for facility i can be specified as a tuple $(i, \mathcal{S}, f')$, where $\mathcal{S}(\mathcal{D}(i))$ is a tree-partition of the set of clients from i, and $f' : \mathcal{S}(\mathcal{D}(i)) \to \mathcal{F}$ is an assignment specifying the facility $f'(S_h)$ to which each set S_h should be reassigned. It is possible that $f'(S_h) = i$ for some sets.

Returning to the evaluation of a disperse move $(i, \mathcal{S}, f')$, we estimate the facility cost of moving a set S_h to its new facility $f'(S_h)$ using the incremental cost incurred by adding clients in S_h to the clients already at this facility $f'(S_h) = i'$, which is $cost_{i'}(S_h) = cost(\mathcal{D}(i') \cup S_h) - cost(\mathcal{D}(i'))$. For different sets added to the same facility, we simply sum their incremental costs, which upper bounds the true increase in facility costs. More precisely, the true increase in cost at a facility i' is

$$cost\left(\mathcal{D}(i') \cup \bigcup_{h:f'(S_h)=i'} S_h\right) - cost(\mathcal{D}(i')),$$

whereas we estimate it as

$$\sum_{h:f'(S_h)=i'} \left[cost(\mathcal{D}(i') \cup S_h) - cost(\mathcal{D}(i'))\right].$$

This is an upper bound on the true increase by submodularity of the cost function. Facility i, from which the clients were just removed, is treated as empty, i.e. $cost_i(S_h) = cost(S_h)$. Thus, the overall upper bound on the change in solution cost resulting from the disperse move $(i, \mathcal{S}, f')$ can be expressed as

$$value(i, \mathcal{S}, f') = \sum_{S_h \in \mathcal{S}(\mathcal{D}(i))} cost_{f'(S_h)}(S_h) + \sum_{S_h \in \mathcal{S}(\mathcal{D}(i))} |S_h| \cdot dist(i, f'(S_h)) - F(i), \quad (4.2)$$

where the first term is the estimate of the increase in facility costs, the second term is the estimate of the change in connection cost, and $F(i)$ is the cost of facility i which is saved when clients are removed from it. This expression defines the *value* of a disperse move. Any move with negative value is called an *improving* move. Next we show how to find a disperse move with minimum value in polynomial time.

4.4.2 Finding the disperse move with optimal value

We begin by proving a lemma that is useful for deriving the algorithm.

Lemma 4.4.1 *There exists a disperse move $(i, \mathcal{S}, f')$ of minimum value such that for all sets S_h of the tree-partition $\mathcal{S}$, none of the nodes of the subgraph $K(S_h)$ are paid for at this set's new facility $f'(S_h)$.*

Proof. Given an optimal disperse move that does not satisfy the required condition, we transform it into one that does, without increasing the cost. If any node of the subgraph $K(S_h)$ is paid for at facility $f'(S_h)$, then so is its top node, call it $k(S_h)$, since it lies on the path from a paid node to the root. Since each client corresponds to a different leaf of the tree T, the node $k(S_h)$ has to be an internal node (there are at least two clients in its subtree: one belonging to S_h, and

one at the facility $f'(S_h)$). In this case we transform the tree-partition by cutting additional edges of the tree, namely the ones connecting $k(S_h)$ to its children (see

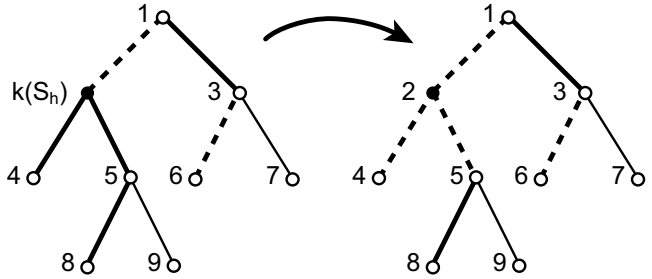

Figure 4.3: Operation performed in the proof of Lemma 4.4.1. Suppose that S_h consists of clients 4 and 8, and that node 2 is paid for at the facility $f'(S_h)$.

Figure 4.3). This may split S_h into multiple sets, all of which we assign to the same facility $f'(S_h)$ as S_h was assigned to. The new disperse move has value no greater than the old one: the connection cost is clearly the same, and since the node $k(S_h)$ and ones above it are already paid for at $f'(S_h)$, the facility cost does not increase. This procedure can be repeated in a top-down manner, eliminating each set that does not satisfy the condition. □

The algorithm for finding the optimal disperse move satisfying the conditions of Lemma 4.4.1 is based on dynamic programming. Recall that $T_{\mathcal{D}(i)}$ is the subgraph of the cost tree T for which the tree-partition has to be found, because it represents the facility costs for clients which are currently at i and which have to be dispersed. Since the dynamic programming will proceed by considering various subtrees of this tree, let the notation $T_{\mathcal{D}(i)}(k)$ stand for the subtree of $T_{\mathcal{D}(i)}$ rooted at node k.

For any node $k \in T_{\mathcal{D}(i)}$, we define three parameters, $N(k)$, $Y(k)$, and $P(k)$, that describe the part of the solution related to the subtree $T_{\mathcal{D}(i)}(k)$. First we describe what the values of these parameters are for a given disperse move $(i, \mathcal{S}, f')$

and a given node k, and then we use these parameters to construct a dynamic programming table for determining the optimal disperse move. For a given node k and a disperse move with its tree-partition $\mathcal{S}$, the components of this tree-partition that are relevant to determining the values of the parameters are the components which are entirely contained in the subtree rooted at k, and at most one special component which contains the edge between k and its parent. For example, if k is node 2 in Figure 4.2, then its special component is $K(S_1)$, as it contains the edge between node 2 and its parent, node 1. Let us call this special component with respect to the tree node k, if it exists, $K(S^k)$, and the set of clients contained in it S^k. Then the three parameters are

- $N(k) = |S^k \cap T_{\mathcal{D}(i)}(k)|$ is the number of clients from node k's special component $K(S^k)$ that are in the subtree rooted at k. For example, for node $k = 2$ in Figure 4.2, $N(k) = 1$, as only client 4 satisfies this property. In the algorithm, this parameter is used to keep track of the connection cost that will have to be paid when S^k is transferred to its new facility.

- $Y(k) = \sum_{k' \in U} cost(k')$, where $U = T_{S^k} \cap T_{\mathcal{D}(i)}(k)$. $Y(k)$ is the part of the facility cost for clients in the special component S^k which comes from the subtree rooted at k. In other words, if U is the set of nodes which lie on a path between k and some client included in $N(k)$, then $Y(k)$ is the total cost of nodes in U. For example, in Figure 4.2, the special component for node $k = 2$ is $K(S_1)$, and the only client which is included both in this component and in the subtree rooted at node 2 is client 4. So the cost for client 4 which comes from the subtree is $Y(2) = cost(4) + cost(2)$. In the algorithm, $Y(k)$ is used to keep track of the facility cost that will have to be paid when k's special component $K(S^k)$ is transferred to its new facility.

Note an important property that $N(k) = 0$ implies that $Y(k) = 0$.

- $P(k)$ is the connection and facility costs of reassigning sets S_h of the tree-partition, that are contained entirely in the subtree rooted at k, to their new facilities. Formally,

$$P(k) = \sum_{S_h : K(S_h) \subseteq T_{\mathcal{D}(i)}(k)} \left[|S_h| \cdot dist(i, f'(S_h)) + cost_{f'(S_h)}(S_h) \right].$$

  In Figure 4.2, $P(2)$ includes the connection cost and the facility cost of sending clients 8 and 9 to their new facility.

The dynamic programming algorithm for computing the optimal disperse move for facility i is shown in figure as Algorithm 3. It constructs a table A whose entries, $A_k(x)$, indexed by the nodes k of $T_{\mathcal{D}(i)}$ and integers x, contain the minimum values of $Y(k) + P(k)$ over all disperse moves on i that satisfy Lemma 4.4.1 and have $N(k) = x$. The reason that values $A_k(x)$ are interesting is that the overall minimum value of a disperse move on facility i is equal to $A_r(0) - F(i)$, where r is the root of T. We assume without loss of generality that the cost tree T is binary: if it is not, then any node with high degree can be expanded into a binary subtree without increasing the overall size of the tree by much.

The algorithm considers the nodes of the tree starting from the leaves, and for each node considers the possibilities of cutting or not cutting the edge between this node and its parent. The idea is that for constructing the disperse move at a node one level higher in the tree, it suffices to know the two parameters, N and $Y + P$, about the solutions at the subtrees of its children. The costs of assignments for components entirely contained lower in the tree are accounted for by P, the facility cost for the unassigned special components of the children nodes is accounted for by Y, and their connection costs are determined by the number of clients in them,

Algorithm 3 Finding optimal disperse move for facility i

Initialize $A_k(x) \Leftarrow \infty$ for all $k \in T_{\mathcal{D}(i)}$ and all $x \in \{0 \dots |\mathcal{D}(i)|\}$

for all leaves k of $T_{\mathcal{D}(i)}$ **do**

  $A_k(1) \Leftarrow cost(k)$ // case when edge $(k, parent(k))$ is not cut

  Let j be the client associated with k

  $A_k(0) \Leftarrow \min_{i' \in \mathcal{F}}(dist(i, i') + cost_{i'}(\{j\}))$ // edge cut, j sent to i'

end for

for all internal nodes $k \in T_{\mathcal{D}(i)}$, bottom-up **do**

  Let $k_1, \dots, k_l$ be the children of k in $T_{\mathcal{D}(i)}$

  $A_k(0) \Leftarrow \sum_{c=1}^{l} A_{k_c}(0)$ // no clients in children's special components

  for all $x_1, \dots, x_l$ such that $1 \leq \sum_{c=1}^{l} x_c \leq |\mathcal{D}(i)|$ **do**

    Let $x = \sum_{c=1}^{l} x_c$ // number of clients from children's components

    $A_k(x) \Leftarrow \min(A_k(x), \sum_{c=1}^{l} A_{k_c}(x_c) + cost(k))$

    // case when edge $(k, parent(k))$ is not cut

    for all $i' \in \mathcal{F}$ such that $k \notin T_{\mathcal{D}(i')}$ **do**

      Let $U_{i'} = \text{Path}(k, root) \setminus T_{\mathcal{D}(i')}$

      // nodes on the path from k to the root which are not paid for at i'

      $A_k(0) \Leftarrow \min(A_k(0), \sum_{c=1}^{l} A_{k_c}(x_c) + x \cdot dist(i, i') + cost(U_{i'}))$

      // edge $(k, parent(k))$ is cut, clients sent to i'

    end for

  end for

end for

N. After initializing these values for the leaves (see lines 3-5 of Algorithm 3), the algorithm considers an internal node k with $l = 1$ or 2 children. To determine $A_k(x)$ for a value $x > 0$, the corresponding solution cannot cut the edge connecting k to its parent. To get such a solution, we need to partition the number of clients x between the components corresponding to the children's subtrees. Let $k_1, \ldots, k_l$ denote the children of k. We consider $A_{k_c}(x_c)$ for every partition $x = \sum_c x_c$. The facility costs of the solutions in the subtrees are combined to get the solution for the tree rooted at k, so the cost is $\sum_c A_{k_c}(x_c) + cost(k)$, and we select the partition in line 12 that minimizes this sum.

For the case of $x = 0$ there are two possibilities for the solution that achieves the minimum value $A_k(0)$. Either $N(k_c) = 0$ for all children k_c of k, or, if not, then the edge above k must be cut to combine the special components of the children into a new component containing k. We get the minimum value of the first case by taking $\sum A_{k_c}(0)$ (as done in line 9). An alternate way is to combine the special components of the subtrees into a single (not special) component by cutting the edge connecting k to its parent. For this case we must consider all values x_c and let $x = \sum_c x_c$. The set S_h corresponding to the component of the tree-partition containing k must be sent to some facility, say i'. The connection cost for this is $|S_h| \cdot dist(i, i')$, but notice that $|S_h| = x$, and we now have to find the lowest-cost facility i' where the new component (containing k) should be sent. Because of Lemma 4.4.1, the facility cost is $\sum_{c=1}^{l} Y(k_c) + cost(\text{Path}(k, root) \setminus T_{\mathcal{D}(i')})$, where $\text{Path}(k, root) \setminus T_{\mathcal{D}(i')}$ is just the set of nodes on the path between k and the root (inclusive) which are not paid for at the facility i'. As a result, we have that $Y(k) = 0$ and $P(k)$ is the sum of all $P(k_c)$ plus the new connection and facility

costs of moving the set S_h to i'. That is,

$$\begin{aligned} Y(k) + P(k) = & \\ = \; & 0 + \sum_{c=1}^{l} P(k_c) + x \cdot dist(i, i') + \sum_{c=1}^{l} Y(k_c) + cost(\text{Path}(k, root) \setminus T_{\mathcal{D}(i')}) \\ \geq \; & \sum_{c=1}^{l} A_{k_c}(x_c) + x \cdot dist(i, i') + cost(\text{Path}(k, root) \setminus T_{\mathcal{D}(i')}). \end{aligned}$$

The minimum of these values are computed in line 15 of the algorithm.

We now give two lemmas regarding the correctness and running time of Algorithm 3.

Lemma 4.4.2 *For all $k \in T_{\mathcal{D}(i)}$ and $x \in \{0 \ldots |\mathcal{D}(i)|\}$, $A_k(x)$ found by Algorithm 3 is the minimum value of $Y(k) + P(k)$ over all disperse moves on i that satisfy Lemma 4.4.1 and have $N(k) = x$.*

Proof. We use induction on the height of the subtree rooted at k. The base case is when k is a leaf of $T_{\mathcal{D}(i)}$, which means that it corresponds to some client in $\mathcal{D}(i)$, say j. Since there is only one client in $T_{\mathcal{D}(i)}(k)$, $N(k)$ can only take values 0 or 1. All disperse moves for which $N(k) = 1$ do not cut the edge above k, and have $P(k) = 0$ and $Y(k) = cost(k)$, so $A_k(1) = cost(k)$ (as set in line 3 of the algorithm) is indeed the minimum value of $Y(k) + P(k)$ for moves with $N(k) = 1$. Only the move that cuts the edge between k and its parent has $N(k) = 0$. If the resulting set $S_h = \{j\}$ is sent to a facility i', then $Y(k) = 0$ and $P(k) = dist(i, i') + cost_{i'}(\{j\})$, so the minimum of $Y(k) + P(k)$ is obtained by choosing i' that achieves the minimum in line 5 of the algorithm.

For an internal node k we have argued that $A_k(x) \leq \min\{Y(k)+P(k) : N(k) = x\}$ for all integers x while constructing the algorithm.

To prove the induction step of the other direction, $A_k(x) \geq \min\{Y(k) + P(k) : N(k) = x\}$, we show that for all k and x such that $A_k(x) < \infty$, there exists a disperse move on i, satisfying Lemma 4.4.1, for which $Y(k) + P(k) = A_k(x)$ and $N(k) = x$. For an internal node k, let us consider several cases depending on which line of the algorithm produced the final value of $A_k(x)$. If the value was produced by line 9, then combining the solutions (which exist by induction hypothesis) corresponding to $A_{k_c}(0)$ for all children k_c of k, and leaving the edge above k intact, gives the desired result. If the value of $A_k(x)$ was produced by line 12 in the iteration of the loop corresponding to $x_1, \ldots, x_l$, then combine the solutions corresponding to $A_{k_c}(x_c)$ from the induction hypothesis, and leave the edge between k and its parent intact. Since in this case $N(k) = \sum_c N(k_c)$, $P(k) = \sum_c P(k_c)$ and $Y(k) = \sum_c Y(k) + cost(k)$, $Y(k) + P(k)$ is equal to $A_k(x)$ as produced by line 12. The last case is if the value of $A_k(x)$ resulted from line 15 when executed in the loop for facility i'. In this case we again combine solutions corresponding to $A_{k_c}(x_c)$ of k's children, but this time cut the tree edge above k and send the resulting set of clients to the facility i'. It can be verified that the value of $Y(k) + P(k)$ will be equal to $A_k(x)$ as produced by line 15 of the algorithm. □

Lemma 4.4.3 *Algorithm 3 runs in time $O(|\mathcal{D}|^3 \cdot |\mathcal{F}|)$.*

Proof. The assumption that T is binary implies that $l \leq 2$, which means that the `for` loop on line 10 executes at most $|\mathcal{D}|^2$ times each time it is entered. The loops on lines 7 and 13 execute at most $|T|$ and $|\mathcal{F}|$ times respectively, and $|T| = O(|\mathcal{D}|)$. □

Combining the above results and observing that the minimum value of a disperse move for a facility i is equal to $A_r(0) - F(i)$, where r is the root of T, we get the following theorem.

Theorem 4.4.4 *The disperse move that minimizes expression 4.2 for a given facility can be found in polynomial time $O(|\mathcal{D}|^3 \cdot |\mathcal{F}|)$, and the one with minimum value over all facilities can be found in time $O(|\mathcal{D}|^3 \cdot |\mathcal{F}|^2)$.*

4.4.3 Bounding the facility cost: a specific set of disperse moves

The way we bound facility cost of a solution SOL which is locally optimal with respect to the disperse move is by focusing on one specific disperse move for each facility and noticing that these moves (like all moves in a locally optimal solution) have non-negative values. In this section we describe these disperse moves, which consist of a tree-partition of clients at each facility and a destination facility for each set of clients in the partition.

Our technique involves finding a mapping on the set of clients, as was also done by Arya et al. [3], who use it to analyze local search with *swap* moves for the k-median problem and local search with *open*, *close* and *swap* moves for the facility location problem. However, in their case, no cost trees are involved, and all clients are the same from the point of view of the facility cost, so the definition and the use of the mapping is much more straight-forward. The idea in our case is to find for each client j a "partner" client $\pi(j)$, close to j in the cost tree, such that the two are at the same facility in OPT but at different facilities in SOL. Then, when the current facility $f(j)$ of client j is dispersed, j can be sent to facility

$f(\pi(j))$, the current facility of $\pi(j)$. This is good in two ways: first, because $\pi(j)$ is close to j in T, the additional facility cost that we have to pay for j at $f(\pi(j))$ is not too big; second, the connection cost for reassigning j can be bounded using the connection costs of j and $\pi(j)$ in OPT and in SOL (as shown in Figure 4.4). However, because of our inexact estimate of the facility cost in expression (4.2),

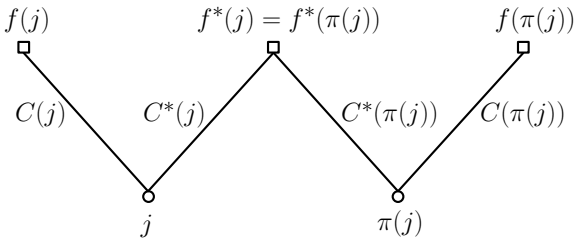

Figure 4.4: Clients j and $\pi(j)$ are assigned to the same facility in OPT, but to different facilities in SOL (unless $\pi(j) = j$). The marked distances are used in the proof of Lemma 4.4.7.

if we reassign each client separately, then we may be paying certain facility costs multiple times. To avoid this, the clients are grouped into sets (in particular, ones forming a tree-partition), and each set is moved as a unit to the current facility $f(\pi(j))$ of the partner of one of its members. To compensate for the fact that a client j is not necessarily an immediate neighbor of $\pi(j)$ in the tree T, we have a scheme for allocating credit to groups of clients in such a way that on one hand, this credit can be used to pay the extra facility costs at their new facilities, and on the other hand, the total amount of credit given out is no more than the optimal facility cost, F^*. This idea is made precise in Lemma 4.4.6.

Defining the mapping

We present a procedure (which is only used for analysis, and is never performed by the local search algorithm) that defines a mapping $\pi : \mathcal{D} \to \mathcal{D}$ on the clients. Also, for each facility $i \in \mathcal{F}$, it defines a set $H(i)$ of edges and nodes of T. These sets are used later for defining the tree-partition and for distributing credit among groups of clients.

The mapping π maps clients from $\mathcal{D}$ to other clients of $\mathcal{D}$ in a one-to-one and onto fashion. Usually a client j is mapped to a different client $\pi(j)$, but it could also be that $\pi(j) = j$. In either case, it is always true that j and $\pi(j)$ are at the same facility in the optimal solution OPT. Except for the case when $\pi(j) = j$, it is also true that j and $\pi(j)$ are at different facilities in the locally optimal solution SOL. The purpose of the sets $H(i)$ is to partition the facility cost paid by OPT among groups of clients, so as to enable them to pay the additional facility costs at their new facilities to which they are reassigned. The facility cost of OPT is partitioned by including tree nodes in the sets $H(i)$, ensuring that every time a tree node is paid for in OPT, it is placed in at most one set. Then the total cost of nodes placed in sets $H(i)$ does not exceed the facility cost paid by OPT. We summarize these properties below, with addition of two more, which will also be useful for the analysis.

1. $\pi : \mathcal{D} \to \mathcal{D}$ is 1-1 and onto.

2. $f^*(j) = f^*(\pi(j))$ for all $j \in \mathcal{D}$. That is, j and $\pi(j)$ are at the same facility in the optimal solution.

3. For all $j \in \mathcal{D}$, either $j = \pi(j)$ or $f(j) \neq f(\pi(j))$. That is, unless $\pi(j) = j$, j and $\pi(j)$ are assigned to different facilities in SOL.

4. Each node of the tree T is included in the sets $H(i)$ at most as many times as it is paid for by OPT.

5. If an edge $(k, parent(k))$ is included in set $H(i)$, then so is its lower endpoint k.

6. For j such that $\pi(j) \neq j$, let $lca(j)$ be the least common ancestor of j and $\pi(j)$ in T. Then the path between j and $lca(j)$, except for the node $lca(j)$ itself, is included in the set $H(f(j))$ of j's facility in SOL.

   For j such that $\pi(j) = j$, the path between j and the root of T, including the root itself, is included in $H(f(j))$.

To define π and $H(i)$, we consider in turn each facility $l \in \mathcal{F}$ used in the optimal solution, and the set $\mathcal{D}^*(l)$ of clients assigned to it by OPT. We assign partners $\pi(j)$ to clients j within this set, thus satisfying property 2. We perform a bottom-up pass through the cost tree of these clients. For each node $k \in T_{\mathcal{D}^*(l)}$ of this tree, we consider a set of clients $S(k) \subseteq \mathcal{D}^*(l)$ that we think of as being "at node k", starting with the leaves of the tree and their corresponding clients (the sets $S(k)$ for internal nodes k are defined as the procedure progresses).

For each node k and its set $S(k)$, we partition the clients in $S(k)$ according to their facility in SOL, forming subsets $S(k) \cap \mathcal{D}(i)$ for all facilities i used in SOL (see Figure 4.5). We aim to assign a partner $\pi(j) \in S(k)$ for all clients $j \in S(k)$ satisfying the condition that j and $\pi(j)$ are assigned to different facilities in SOL. It may not always be possible to find a mapping for all clients in $S(k)$, but we assign as many of them as possible, leaving others to be assigned at higher levels of the tree.

More formally, let i be a *majority* facility, one for which $|S(k) \cap \mathcal{D}(i)|$ is the

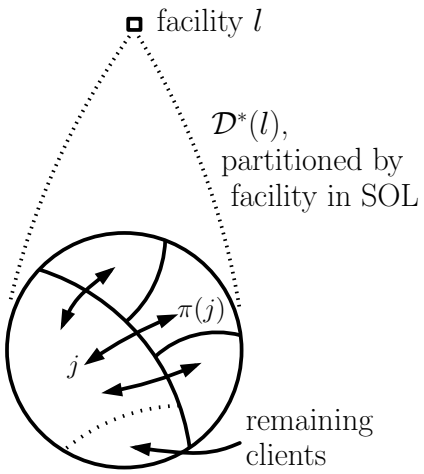

Figure 4.5: Example of how π is defined on the set $S(k)$ at one node of the tree. If there are extra clients from the majority facility, then they propagate up the tree.

largest. In the simpler case, at most half of the clients belong to this majority facility, $|S(k) \cap \mathcal{D}(i)| \leq \frac{1}{2}|S(k)|$. Then we can assign $\pi(j) \in S(k)$ for all clients $j \in S(k)$, with $\pi(j) \neq j$, satisfying $f(j) \neq f(\pi(j))$ (condition 3), and none of the clients are propagated up the tree. For example, this assignment can be done by numbering the members of $S(k)$ from 0 to $|S(k)| - 1$ so that clients from the same facilities in SOL form continuous stretches, and then assigning $\pi(j) = j + \lfloor \frac{|S(k)|}{2} \rfloor \pmod{|S(k)|}$, as is done in [3]. Note that such an assignment satisfies condition 1.

In the other case, if the majority facility has more than half the clients, $|S(k) \cap \mathcal{D}(i)| > \frac{1}{2}|S(k)|$, it is impossible to find a mapping π satisfying our condition among the clients at k. So we take the maximum number of clients from the majority facility that can be assigned, $|S(k) \setminus \mathcal{D}(i)|$ clients from $\mathcal{D}(i)$, and assign the mapping π between them and the clients from the other facilities $S(k) \setminus \mathcal{D}(i)$. The remaining clients from the majority facility, which were not assigned, are added to the set $S(parent(k))$, thus propagating up the tree. Also, in this case we add the node k

and the edge $(k, parent(k))$ to the set $H(i)$ of the majority facility i. This satisfies condition 5. Since node k is paid for at facility l in OPT, and we add it to at most one set $H(i)$ for each such facility l, condition 4 is satisfied as well. If k is the root, then the remaining clients from $\mathcal{D}(i)$ are assigned to themselves, i.e. for them we set $\pi(j) = j$. Also, in this case we add the root of the tree to $H(i)$. Thus, at the end of this process on a facility l, for each client $j \in \mathcal{D}^*(l)$ there is a client $\pi(j) \in \mathcal{D}^*(l)$. See Figure 4.6 for an example of the process as it proceeds through different levels of the tree.

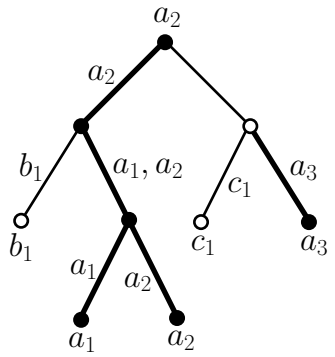

Figure 4.6: Example of the mapping. The leaves $\{b_1, a_1, a_2, c_1, a_3\}$ represent clients assigned to facility l in OPT. The different letters represent different facilities to which they are assigned in SOL. The labels on edges indicate which clients pass through them as they propagate up the tree. The result is the mapping $a_1 \leftrightarrow b_1$ and $c_1 \leftrightarrow a_3$, with a_2 assigned to itself. The thick edges and shaded nodes are included in $H(a)$.

Lemma 4.4.5 *The mapping π and the sets $H(i)$ defined by the above procedure satisfy properties (1)-(6).*

Proof. We have argued that properties 1-5 are satisfied in the description of the mapping procedure.

For the proof of property 6, assume first that $\pi(j) \neq j$, and let k be the node at which $\pi(j)$ was assigned. Then k must be the least common ancestor of j and $\pi(j)$ in T. This is because j and $\pi(j)$ belong to different facilities in SOL, which means that they could not have propagated up the tree to node k from the same child of k (since only clients from the same facility in SOL propagate up any given edge of $T_{\mathcal{D}^*(l)}$), so they must have reached k from different children of k. Now, every node and edge that j crosses as it propagates up the tree from its corresponding leaf to k is included in $H(f(j))$. This is because j only propagates from k' to $parent(k')$ when $f(j)$ is the majority facility of $S(k')$, which is also when the node k' and the edge $(k', parent(k'))$ are added to $H(f(j))$. As a result, we get that the whole path from j to k (except k itself) is included in $H(f(j))$. If $\pi(j) = j$, then j must have propagated all the way to the root of T, and $f(j)$ was still the majority facility at the root, which again implies that the whole path, as well as the root, are included in $H(f(j))$. □

Defining the tree-partition and the facility assignment

Recall that a disperse move for facility i consists of a tree-partition and a facility assignment for each set of this partition. The set of disperse moves that we use for the analysis consists of one such move, call it $(i, \mathcal{S}^i, f')$, for each facility i that is used in SOL. We now define $\mathcal{S}^i$ and f' for each i.

To define the tree-partition $\mathcal{S}^i$ for facility i, we use the edges from the set $H(i)$ constructed while defining the mapping π. Recall that edges could have been added to the set $H(i)$ for our facility i which is used in SOL while π was being defined on different facilities l which are used in OPT. In the tree $T_{\mathcal{D}(i)}$ of clients

currently at facility i, we retain the edges that are in $H(i)$ and cut all the other edges. This produces a set of connected components of the tree, and therefore defines a tree-partition of the clients $\mathcal{D}(i)$. This is the partition $\mathcal{S}^i$ that we use.

To define the facility assignment $f'(S_h^i)$ for a set $S_h^i \in \mathcal{S}^i$, we make use of the mapping π. Recall that the idea of the mapping π was to find for each client j assigned to a facility $i = f(j)$ in SOL a "partner" client $\pi(j)$, so that we can reassign j to the facility $f(\pi(j))$ when dispersing facility i. If we assign each client separately, then we may be paying certain facility costs multiple times. To avoid this, we want to assign all clients in a set S_h^i to one facility $f(\pi(j))$ for some $j \in S_h^i$. In particular, among these facilities we choose one which is closest to i. So the set of clients S_h^i in the tree-partition is assigned to the facility $f'(S_h^i) = i'$ from the set $\{f(\pi(j)) : j \in S_h^i\}$ that minimizes $dist(i, i')$. For example, if $\pi(j) = j$ for any $j \in S_h^i$, then we set $i' = i$.

4.4.4 Bounding the facility cost: analysis

We now give two lemmas that bound the facility and connection costs incurred when the sets of clients S_h^i in the partitions $\mathcal{S}^i$ defined above are transferred to their assigned facilities, $f'(S_h^i)$.

Lemma 4.4.6 *The sum of incremental facility costs incurred when each set S_h^i is transferred to its new facility $f'(S_h^i)$ is at most the optimal facility cost,*

$$\sum_{i \in \mathcal{F}} \sum_{S_h^i \in \mathcal{S}^i} cost_{f'(S_h^i)}(S_h^i) \;\leq\; F^*.$$

Proof. For each facility $i \in \mathcal{F}$, and each set $S_h^i \in \mathcal{S}^i$ that arises from a component of the tree-partition of $\mathcal{D}(i)$, we assign a budget B_h^i and show two inequalities:

1. for each $i \in \mathcal{F}$ and $S_h^i \in \mathcal{S}^i$, $cost_{f'(S_h^i)}(S_h^i) \leq B_h^i$

2. $\sum_{i \in \mathcal{F}} \sum_{S_h^i \in \mathcal{S}^i} B_h^i \leq F^*$

which together imply the lemma.

The way we define B_h^i is as follows. For a component $K(S_h^i)$ of the tree-partition, give it the amount of credit equal to the cost of the nodes which are included both in this component and the set $H(i)$. Formally,

$$B_h^i = \sum_{k \in K(S_h^i) \cap H(i)} cost(k).$$

Let us prove inequality 2 first. By property 4 in Section 4.4.3, each node of T is included in sets $H(i)$ at most as many times as it is paid for by OPT. Moreover, notice that each node in $H(i)$ can belong to at most one component of the tree-partition $\mathcal{S}^i$. Consequently, its cost was added to at most one budget B_h^i, proving the inequality.

To show that inequality 1 holds, consider a set of clients S_h^i and its tree-partition component $K(S_h^i)$. The facility cost for this set of clients, which is the cost of the union of their paths to the root, can be divided into two parts: the cost of the nodes that are in the component $K(S_h^i)$ and the cost of ones which are on the path between the highest node of this component, call it $k(S_h^i)$, and the root of T. We show that the cost of the nodes in the component (except for its highest node) is accounted for in the budget B_h^i, and the cost of the nodes on the path is already paid for at the new facility. The first part follows because the edges of the component are from the set $H(i)$, by property 5, and because the budget B_h^i is allocated for the nodes which are in the set $H(i)$. To show that the path between the component's highest node $k(S_h^i)$ and the root is already paid for at the new facility $f'(S_h^i)$, let $j \in S_h^i$ be the client whose partner's facility was chosen, i.e. such that $f'(S_h^i) = f(\pi(j))$. For the case that $j \neq \pi(j)$, by property 6, the path between

j and $lca(j)$ is in $H(i)$, and therefore also in the component S_h^i. This means that in T, the client $\pi(j)$ is somewhere under the node $k(S_h^i)$, and since the path between $\pi(j)$ and the root of T is paid for at the new facility, so is the path between $k(S_h^i)$ and the root (inclusive). For the case that $j = \pi(j)$, by properties 5 and 6, the whole facility cost of the set S_h^i, including the root, is accounted for in B_h^i. □

Lemma 4.4.7 *For the tree-partition $\{S_h^i\}$ and facilities $f'(S_h^i)$ defined above, the connection cost of transferring the sets of clients S_h^i to the facilities $f'(S_h^i)$ can be bounded as*

$$\sum_{i \in \mathcal{F}} \sum_{S \in \{S_h^i\}} |S| \cdot dist(i, f'(S)) \;\leq\; 2C + 2C^*.$$

Proof. When defining $f'(S_h^i)$ we choose $f(\pi(j))$ with minimum distance to i, where $i = f(j)$ for all $j \in S_h^i$. So the left-hand side of the inequality is at most

$$\sum_{j \in \mathcal{D}} dist(f(j), f(\pi(j))).$$

To bound this expression recall that $f^*(\pi(j)) = f^*(j)$. Then by triangle inequality (see Figure 4.4) we get that

$$dist(f(j), f(\pi(j))) \leq C(j) + C^*(j) + C^*(\pi(j)) + C(\pi(j)).$$

Note that this bound is also valid when $\pi(j) = j$ as then $dist(f(j), f(\pi(j))) = 0$. Since the mapping π is 1-1 and onto, when this expression is summed over all j, we obtain

$$\sum_{j \in \mathcal{D}} dist(f(j), f(\pi(j))) \leq 2C + 2C^*,$$

proving the lemma. □

We conclude the analysis of the algorithm by combining the results obtained so far.

Theorem 4.4.8 *A locally optimal solution SOL with no* aggregate *or* disperse *moves with negative value has cost at most 5 times the cost of the optimal solution.*

Proof. If we consider the disperse move $(i, \mathcal{S}^i, f')$ defined above for a facility i in SOL, then the cost of the solution will change by an amount upper-bounded by expression (4.2). Because SOL is a locally optimal solution, the value of this move is non-negative:

$$\sum_{S_h^i} cost_{f'(S_h^i)}(S_h^i) + \sum_{S_h^i} |S_h^i| \cdot dist(i, f'(S_h^i)) - F(i) \geq 0.$$

Summing these inequalities over all i and applying Lemmas 4.4.6 and 4.4.7, we get that $F^* + 2C + 2C^* - F \geq 0$, or $F \leq F^* + 2C^* + 2C$. Combining this with the bound on C (Lemma 4.3.3), we get $F \leq 3F^* + 4C^*$, and $F + C \leq 4F^* + 5C^*$. □

Using standard scaling techniques, we improve the bound to $2 + \sqrt{5} < 4.237$. To do that, scale the original facility costs by λ and run the algorithm, obtaining a solution with the guarantees $C \leq \lambda F^* + C^*$ (by Lemma 4.3.3), and $\lambda F \leq 3\lambda F^* + 4C^*$ (see proof of Theorem 4.4.8). Combining these gives

$$C + F \leq (3 + \lambda)F^* + (1 + 4/\lambda)C^*,$$

which yields the claimed result when λ is set to $\sqrt{5} - 1$.

Further, by taking only *aggregate* and *disperse* moves with large negative values (as in, for example, [3]), we obtain a polynomial time $(4.237 + \epsilon)$-approximation algorithm.

Theorem 4.4.9 *There is a polynomial time* $(4.237 + \epsilon)$*-approximation algorithm for the facility location problem with hierarchical facility costs that is based on local search and uses* aggregate *and* disperse *moves.*

CHAPTER 5

LOAD-BALANCED FACILITY LOCATION

5.1 Introduction

The load-balanced facility location (LBFL) problem that we consider in this chapter of the thesis is an extension of the uncapacitated metric facility location (FL), as it includes an extra set of constraints. In particular, in addition to all the elements of a regular facility location instance, an instance of a load-balanced version of the problem specifies a lower bound B, which is the minimum number of clients that can be assigned to a facility if it is opened. Obviously, if B is equal to zero, then the problem reduces to the original facility location. The LBFL problem was introduced simultaneously and independently by Karger and Minkoff [45], who use it as a subroutine for solving the *maybecast* network design problem, and by Guha, Meyerson and Munagala [33], who use it as a subroutine for solving the *access network design* problem. Both papers propose bicriteria approximation algorithms for LBFL, which violate both the lower bound constraints and the optimality of the objective function by constant factors, but are sufficient for their purposes.

As demonstrated by the algorithms of [45] and [33], LBFL can be a useful subroutine for solving various network design problems. Undoubtedly, the two problems presented in those papers are not the only ones for which the solution of LBFL would be useful. In addition, the LBFL problem formulation has direct applications. For example, Lim, Wang, and Xu [52] present a transportation problem faced by a real-world company that has to decide on an allocation of cargo from customers ('clients') to carriers ('facilities'), who then ship it overseas. There is a transportation cost per unit demand assigned from each customer to each carrier,

which can be modeled by the connection cost. But the main difficulty arises from the fact that there is a regulation enforcing a "minimum quantity commitment", i.e. a rule that the total amount of cargo delivered by each carrier, if any, must be at least a certain minimum quantity. So the problem becomes exactly LBFL, but without facility costs (which seems to be as hard as the general LBFL). Other example applications of LBFL include the location of stores, with the requirement that each individual store serve a given minimum number of customers to remain profitable [33], and a clustering problem in which each cluster has to be at least a certain size, while the average distance of data points to cluster centers is minimized [45].

Related work There has been much work on designing approximation algorithms for the uncapacitated facility location problem. The first constant-factor approximation algorithm was proposed by Shmoys, Tardos, and Aardal [62], and is based on LP rounding. Subsequently, other constant-factor approximation algorithms were designed, based on various techniques, including the primal-dual method and local search (e.g. [3, 8, 11, 42, 48, 63]). Currently the best approximation guarantee is 1.52 [54].

The load-balanced facility location problem was introduced by Guha, Meyerson, and Munagala [33], who use it for solving the *access network design* problem, which is a special case of the single-sink buy-at-bulk problem. Simultaneously, LBFL was also introduced by Karger and Minkoff [45], who call it the *r-gathering* problem and use it to solve the *maybecast* problem, which models network design under uncertainty about demands. Both papers present essentially the same bicriteria approximation algorithm for LBFL, which, for any given constant $\alpha \in [0, 1)$, finds

a solution which assigns at least $\alpha \cdot B$ clients to each open facility (where B is the lower bound on the number of clients) and costs at most $\frac{1+\alpha}{1-\alpha}\rho \cdot OPT$, where ρ is the approximation ratio for the FL problem, which is used as a subroutine, and OPT is the cost of the optimal solution to LBFL that respects the lower bound constraints. Thus, this algorithm provides a trade-off between the cost of the solution and the amount by which the lower-bound constraints are violated, but it is unable to find a truly feasible solution with a non-trivial guarantee on the cost. The LBFL problem is also considered by Lim et al. [52], who formulate it as a mixed-integer program and solve it using a branch-and-cut scheme. They also analyze a greedy heuristic for LBFL without facility costs and show that it is a $2B$-approximation.

An extension of the facility location problem which in some sense is the opposite of LBFL is the capacitated facility location (CFL) problem. In CFL, each facility has a capacity, which is the maximum number of clients that can be assigned to it. This problem is significantly harder than the uncapacitated version. For example, all known LP relaxations for it have unbounded integrality gaps. However, there are several known constant-factor approximation algorithms for CFL, all of which are based on the local search technique. Korupolu, Plaxton and Rajaraman [48] gave a constant-factor approximation for the special case of uniform capacities, which was later improved by Chudak and Williamson [12]. The first constant-factor algorithm for non-uniform capacities, providing a $(8.53+\varepsilon)$-approximation, was given by Pál, Tardos and Wexler [56]. Currently the best bound is $3+2\sqrt{2}+\varepsilon \leq 5.83+\varepsilon$ [69]. A variant of the capacitated problem is facility location with soft capacities, in which facilities can be opened multiple times for extra cost, thus serving more clients than their capacity. This version of the problem is generally

easier to solve than CFL, as it does not suffer from large integrality gaps, and can be reduced to the regular FL problem. A number of constant-factor approximation algorithms have been proposed for it [3, 9, 41, 42, 55].

A formulation that generalizes CFL with either hard or soft capacities, as well as a number of other problems, is known as the universal facility location problem. In it, instead of capacities, each facility has a cost function which depends on the number of clients that are assigned to it. For example, CFL can be modeled by a cost function that starts out as constant, but then goes to infinity when the number of clients exceeds the capacity. This formulation was introduced by Hajiaghayi, Mahdian and Mirrokni [34], who focus on the special case of concave functions and give a constant approximation based on a reduction to the uncapacitated problem. Subsequently, Mahdian and Pál [53] gave an algorithm that works for arbitrary monotone non-decreasing facility cost functions. Their algorithm is an extension of the local search technique of [56] for CFL, and gives a $7.88+\varepsilon$ approximation. It was later improved by Garg, Khandekar and Pandit [30], who achieve a $3+2\sqrt{2}+\varepsilon$ approximation ratio, bridging the gap between known guarantees of universal facility location and CFL.

Our results and techniques We present the first constant-factor true approximation algorithm for the load-balanced facility location problem, thus resolving an open question of Karger and Minkoff [45]. Our algorithm is a true approximation in the sense that the produced solution is feasible for the original problem, satisfying the lower-bound constraints exactly. This is in contrast to bicriteria algorithms, which violate these constraints by constant factors. Whether or not a bicriteria approximation algorithm is an acceptable solution depends on the specific appli-

cation. For example, in the contexts in which LBFL was originally introduced [33, 45], the bicriteria algorithms are sufficient for their purposes, and their violation of the constraints does not present major difficulties. However, in other cases, either in real-world applications or in reductions for other problems, a true approximation for the problem may be needed. For example, in the transportation application mentioned above, a bicriteria solution would not be satisfactory.

The main technical idea that we use for solving LBFL is to create an instance of the capacitated facility location problem by reversing the roles played by the clients and the facilities. To give a rough description lacking many details, we can say that a group of clients at a given location becomes a facility whose capacity is the number of those clients. Conversely, a facility that has not yet been filled to the bound B becomes a client whose demand is the number of "slots" that still have to be filled in order for this facility to reach B. Then the task becomes to make an assignment which would use the clients to fill the "slots" in such a way that each open facility has at least B clients assigned to it. We use a CFL subroutine to make such an assignment, taking advantage of the known constant-factor approximation algorithms for it. Our actual algorithm for LBFL also involves a pre-processing step, in which we compute a bicriteria solution to our input instance, as well as a post-processing step, in which we assign some remaining left-over clients.

Overview Our algorithm consists of three main stages: first, we find a bicriteria-approximate solution and use it to transform the instance, taking care that the value of the optimal solution does not increase too much; then we use this modified instance to define a CFL problem, and solve it using one of the known algorithms; finally, based on the solution to the CFL instance, we transfer clients between facil-

ities in a way that transforms the bicriteria solution into an approximate solution that does not violate the lower bound constraints. In the following sections, we begin with the formal problem definition and a review of the bicriteria algorithm in Section 5.2. Then Sections 5.3, 5.4 and 5.5 describe the three stages of the algorithm respectively.

5.2 Problem definition and the bicriteria algorithm

We begin with a precise statement of the problem.

Definition 5.2.1 *An instance $\mathcal{I}$ of the* lower-bounded facility location problem *consists of a set of clients $\mathcal{D}$, a set of facilities $\mathcal{F}$, a non-negative facility cost $f(i)$ for each facility $i \in \mathcal{F}$, a distance metric $c(i,j)$ on the set $\mathcal{D} \cup \mathcal{F}$, and a bound B. A feasible solution consists of a subset $\mathcal{O} \subseteq \mathcal{F}$ of facilities to open, and an assignment of each client to an open facility, so that each open facility has at least B clients assigned to it. For a given solution, we use $j \to i$ to denote the fact that client $j \in \mathcal{D}$ is assigned to facility $i \in \mathcal{F}$, and $i(j)$ to denote the facility to which $j \in \mathcal{D}$ is assigned. The objective then is to minimize*

$$\sum_{i \in \mathcal{O}} f(i) + \sum_{j} c(j, i(j)), \quad \textit{subject to} \ \ |\{j : j \to i\}| \geq B \quad \textit{for all } i \in \mathcal{O}.$$

Let OPT($\mathcal{I}$), or just OPT, denote the value of the optimal solution to $\mathcal{I}$, with $C^* = \sum_j c(j, i(j))$ being its connection cost and $F^* = \sum_{i \in \mathcal{O}} f(i)$ being its facility cost. Let $j \to^* i$ and $i^*(j)$ represent the assignments made by the optimal solution.

Our algorithm for LBFL uses the bicriteria approximation algorithm of [33, 45] as a first step, as described in more detail in the next section. Here, for the sake of completeness, let us review this algorithm and its analysis. The algorithm takes

a parameter $\alpha \in [0, 1)$ and returns a solution which assigns at least αB clients to each open facility.

For each facility $i \in \mathcal{F}$, let $\mathcal{D}(i) \subseteq \mathcal{D}$ be the set of the closest B clients to i. We now construct an instance $\mathcal{I}'$ of the FL problem by dropping the lower bounds from $\mathcal{I}$ and setting facility costs to

$$f'(i) = f(i) + \lambda \sum_{j \in \mathcal{D}(i)} c(i, j).$$

Here $\lambda = \frac{2\alpha}{1-\alpha}$ is just a constant used for scaling. The idea behind the term $\sum_{j \in \mathcal{D}(i)} c(i, j)$ is that if a facility i is opened in a solution to LBFL, then, since it serves at least B clients, the connection cost of its clients will be at least this much. Once the instance $\mathcal{I}'$ is constructed, we solve it using a ρ-approximation algorithm, ensuring that each client is assigned to the nearest open facility (this only improves the objective). Call the resulting solution $\mathcal{S}'$.

The second step of the algorithm is to perform a reassignment of clients from open facilities that are serving less than αB clients in $\mathcal{S}'$. For each such facility i, in arbitrary order, do the following: find the nearest to i open facility i', reassign all clients from i to i', and close i. Clearly, at the end of this procedure, each open facility is serving at least αB clients.

To analyze the algorithm, we make the following observations.

Lemma 5.2.2 $OPT(\mathcal{I}') \leq (\lambda + 1)OPT(\mathcal{I})$.

Proof. Suppose the optimal solution to the LBFL instance $\mathcal{I}$ opens a set of facilities $\mathcal{O}$, has facility cost F^* and connection cost C^*. This same solution is feasible for the FL instance $\mathcal{I}'$. Its connection cost for $\mathcal{I}'$ is the same as it is for $\mathcal{I}$,

C^*. Its facility cost for $\mathcal{I}'$ is

$$\sum_{i \in \mathcal{O}} f'(i) = \sum_{i \in \mathcal{O}} \left[f(i) + \lambda \sum_{j \in \mathcal{D}(i)} c(i,j) \right] \leq F^* + \lambda C^*.$$

So the overall cost of this solution, which serves as an upper bound on $OPT(\mathcal{I}')$, is at most $F^* + (1+\lambda)C^* \leq (1+\lambda)OPT$. □

Since the FL instance was solved using a ρ-approximation, we get the following.

Corollary 5.2.3 *The cost of the solution $\mathcal{S}'$ is at most $(\lambda+1)\rho \cdot OPT(\mathcal{I})$.*

Now we analyze the additional cost incurred by the second step of the algorithm.

Lemma 5.2.4 *The additional connection cost incurred by transferring clients from any facility i in the second step of the algorithm is at most $f'(i)$, the facility cost of i in $\mathcal{I}'$.*

Proof. To bound this cost, we observe that for any facility i with less than αB clients assigned to it by $\mathcal{S}'$, there must be at least $(1-\alpha)B$ clients that are included in the set $\mathcal{D}(i)$ but are not assigned to i. Since the total distance of all clients in $\mathcal{D}(i)$ to i is $\sum_{j \in \mathcal{D}(i)} c(i,j)$, the average distance of those $(1-\alpha)B$ clients is at most $\frac{\sum_{j \in \mathcal{D}(i)} c(i,j)}{(1-\alpha)B}$, and so is the minimum distance between i and one of these clients, say j. Because j is assigned to its nearest open facility, which is not i, this means that there must be another open facility at a distance of at most $2 \cdot \frac{\sum_{j \in \mathcal{D}(i)} c(i,j)}{(1-\alpha)B}$ from i (by using the triangle inequality on the distances from i to j and from j to its assigned facility). Therefore, the additional connection cost that we pay for reassigning clients from i is at most

$$2\frac{\alpha B}{(1-\alpha)B} \sum_{j \in \mathcal{D}(i)} c(i,j) = \lambda \sum_{j \in \mathcal{D}(i)} c(i,j) \leq f'(i).$$

□

Overall, we get the following approximation guarantee.

Theorem 5.2.5 *The solution found by the bicriteria algorithm for the LBFL instance $\mathcal{I}$ has cost at most $\frac{1+\alpha}{1-\alpha}\rho \cdot OPT(\mathcal{I})$.*

Proof. The cost of the final solution consists of the following parts: the original connection cost, which is equal to the connection cost of $\mathcal{S}'$; the facility cost of facilities that remain open, which is at most the facility cost of these facilities in $\mathcal{S}'$; and the additional connection cost for reassignments, which is at most the facility cost of the facilities that were closed. So the total cost is at most that of $\mathcal{S}'$, and substituting the definition of λ into Corollary 5.2.3, we get the result. □

5.3 Transforming the instance

In order to apply the main step of our algorithm, which uses a CFL subroutine, we simplify the problem in a few ways, ensuring that the new instance has some useful properties. In particular, it does not have facility costs, and has clients clustered in relatively large groups (a constant fraction of B) at each location. To do this, we employ the bicriteria approximation algorithm described in the previous section. We consider two modified instances of the LBFL problem, instance $\mathcal{I}_1$ obtained by modifying the original problem according to the bicriteria solution, and instance $\mathcal{I}_2$ obtained by further modifying $\mathcal{I}_1$ (see Figure 5.1). In this section we define these instances and bound the values of their optimal solutions in terms of the optimum for the original problem.

The bicriteria algorithm is applied to the original problem instance $\mathcal{I}$, with a parameter $\alpha > \frac{1}{2}$ to be specified later. Let $j \rightarrow^b i$ and $i^b(j)$ denote the assignments made by the obtained solution. Also, let C^b and F^b denote its connection and

facility costs, respectively. We now define the first modified instance, $\mathcal{I}_1$.

Definition 5.3.1 *Let $\mathcal{I}_1$ be an instance of LBFL, whose elements $\mathcal{D}$, $\mathcal{F}$, and B are the same as in $\mathcal{I}$, but the metric of distances and the facility costs are different. The distances are modified as follows. Intuitively, every client is "moved" to the location of the facility to which it is assigned by the bicriteria solution. Formally, for any two clients $j, j' \in \mathcal{D}$ and two facilities $i, i' \in \mathcal{F}$, the distances become: $c_1(j,i) = c(i^b(j), i)$; $c_1(j,j') = c(i^b(j), i^b(j'))$; and the distance between facilities remains the same, $c_1(i,i') = c(i,i')$. The facility costs are modified so that all the facilities that were opened by the bicriteria solution become free, and the costs of others remain the same: $f_1(i) = 0$ if there exists $j \in \mathcal{D}$ such that $j \to^b i$, and $f_1(i) = f(i)$ otherwise.*

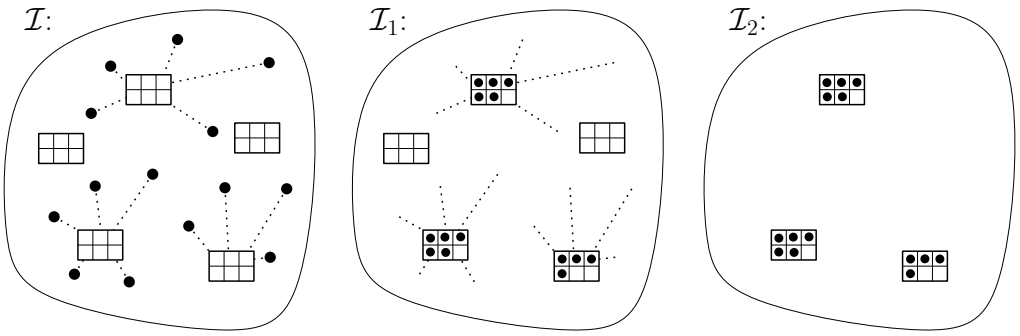

Figure 5.1: An example of defining the instances $\mathcal{I}_1$ and $\mathcal{I}_2$. The circles represent the clients, and the large rectangles represent the facilities, whose lower bound is $B = 6$. The dotted lines show the assignment of clients to facilities made by the bicriteria algorithm for the original instance, with $\alpha B = 4$.

It's not hard to see that the new distances c_1 also form a metric. The cost of the optimal solution to $\mathcal{I}_1$ can be bounded as follows.

Lemma 5.3.2 $OPT(\mathcal{I}_1) \leq \frac{1+\alpha}{1-\alpha} \cdot \rho \cdot OPT + OPT$.

Proof. One feasible solution to $\mathcal{I}_1$ is to assign each client j to his optimal facility $i^*(j)$. The facility costs of this solution will be at most those in OPT, F^*, and the connection cost for a client j will be $c_1(j, i^*(j)) = c(i^b(j), i^*(j)) \leq c(i^b(j), j) + c(j, i^*(j))$ by the triangle inequality. Intuitively, j can be first moved back to his original location, and then moved from there to his optimal facility. Summing the connection costs over all clients, we get that the connection cost of this solution is $\sum_{j \in \mathcal{D}} c_1(j, i^*(j)) \leq C^b + C^*$. Since $C^b \leq \frac{1+\alpha}{1-\alpha} \cdot \rho \cdot OPT$ by the guarantee of the bicriteria algorithm, we get the result. □

The second transformation that we make is to produce a LBFL instance $\mathcal{I}_2$ out of instance $\mathcal{I}_1$ by removing the facilities which are not used by the bicriteria solution that we found.

Definition 5.3.3 *Let $\mathcal{I}_2$ be the same as $\mathcal{I}_1$, except for the set of facilities, which becomes $\mathcal{F}_2 := \{i \in \mathcal{F} \ : \ j \rightarrow^b i \text{ for some } j \in \mathcal{D}\}$.*

Next we bound the cost of the optimal solution to $\mathcal{I}_2$ in terms of $OPT(\mathcal{I}_1)$.

Lemma 5.3.4 $OPT(\mathcal{I}_2) \leq 2 \cdot OPT(\mathcal{I}_1)$

Proof. Consider the optimal solution to $\mathcal{I}_1$, and suppose it uses some facility $i \notin \mathcal{F}_2$. Then, instead, transfer all clients from i to its closest facility $i' \in \mathcal{F}_2$. This is a feasible solution, since i' now has at least B clients. The facility cost did not increase, because i' is free (by the definition of facility costs in $\mathcal{I}_1$). To bound the possible increase in connection costs, observe that in $\mathcal{I}_1$, each client is co-located with (i.e., is at distance 0 from) some facility in $\mathcal{F}_2$. Now for the facility $i \notin \mathcal{F}_2$, let j be the closest client assigned to i. It must therefore be that $c_1(i, i') \leq c_1(i, j)$.

As a result, the total cost of transferring clients from i to i' is at most

$$\sum_{j' \to i} c_1(i, i') \leq \sum_{j' \to i} c_1(i, j) \leq \sum_{j' \to i} c_1(i, j'),$$

where the second inequality follows because j was defined as the closest client assigned to i. Since the additional connection cost incurred for transferring clients from facility i is at most their original connection cost, the overall connection cost at most doubles, implying the result of the lemma. □

In the following sections we show how to obtain a constant-factor approximation to $\mathcal{I}_2$. The next lemma summarizes its relation to the original problem.

Lemma 5.3.5 *The cost of a β-approximate solution for $\mathcal{I}_2$ is at most*

$$\left[(2\beta + 1)\frac{1+\alpha}{1-\alpha} \cdot \rho + 2\beta\right] \cdot OPT(\mathcal{I}).$$

Proof. To transform a solution to $\mathcal{I}_2$ back to a solution of the original problem $\mathcal{I}$, we may need to pay the connection cost of the clients for the distance between their original locations and their new locations of $\mathcal{I}_1$, which in total is at most C^b. We may also need to pay the facility costs of facilities in $\mathcal{F}_2$, which cost at most F^b. So the total cost of this solution becomes at most

$$\begin{aligned} C^b + F^b + \beta \cdot OPT(\mathcal{I}_2) &\leq \frac{1+\alpha}{1-\alpha} \cdot \rho \cdot OPT + 2\beta \cdot OPT(\mathcal{I}_1) \\ &\leq \frac{1+\alpha}{1-\alpha} \cdot \rho \cdot OPT + 2\beta \cdot \left(\frac{1+\alpha}{1-\alpha} \cdot \rho \cdot OPT + OPT\right) \\ &= \left[(2\beta + 1)\frac{1+\alpha}{1-\alpha} \cdot \rho + 2\beta\right] \cdot OPT, \end{aligned}$$

using Lemmas 5.3.4, 5.3.2, and Theorem 5.2.5. □

5.4 Reduction to capacitated facility location

At this point, we have an instance $\mathcal{I}_2$ of the LBFL problem which has special structure. It consists of a set of facilities, each of which with at least αB clients at distance 0 from it. Let us say that these clients, whose number is $n_i \geq \alpha B$, are *at* this facility i. The instance does not have facility costs, so its solution requires that the clients be somehow reassigned, possibly closing some of the facilities, so that the remaining facilities have at least B clients each, while minimizing the connection cost of the reassignments. Since with $\alpha > \frac{1}{2}$, the number of clients from any two facilities is sufficient to reach the bound of B, an initial idea of how to solve this problem might be to find some kind of a matching on the set of facilities. However, a simple example shows that this can be far from optimum. Consider a set of B facilities, each with $B-1$ clients, located in a uniform metric space (with all distances equal to 1). Then the optimal solution is to close one of the facilities, reassigning one client from it to each of the other facilities, which costs $B-1$ in connection cost. However, if the facilities are paired up by a matching, then the connection cost incurred is $\frac{B}{2}(B-1)$.

The way we solve the special case of the LBFL problem presented by the instance $\mathcal{I}_2$ is by using a reduction to the capacitated facility location problem. The general idea is that the clients from those locations that should be closed would correspond to facilities that have an amount of supply to give out. On the other hand, the empty slots from those facilities that should be opened but do not have enough clients to reach B would correspond to clients in CFL, which have to be satisfied by the supply from other facilities. Of course, we do not know in advance which facilities should be opened and which should be closed, but the reduction does not require this knowledge. In order to avoid a confusion of terminology

arising from the reversal of the client-facility roles, we say that the instance of CFL has *supply points* (facilities), each with some *total supply* (capacity), and *demand points* (clients), each with some amount of demand. The goal is to *select* (open) some supply points, paying a *selection cost* (facility opening cost), and to assign each demand point to a selected supply point, paying a connection cost, so that each supply point serves at most the amount of demand equal to its total supply.

The CFL instance $\mathcal{I}_{cap}$ that we create is defined as follows (see Figure 5.2). For each facility $i \in \mathcal{F}_2$ that has $n_i \leq B$ clients, create a supply point at its location with total supply B and selection cost $\delta \cdot n_i \cdot l(i)$, where $l(i)$ is the distance between i and its closest other facility $i' \in \mathcal{F}_2$, $i' \neq i$, and δ is a constant to be optimized later. In addition, create a demand point at this location, with demand $B - n_i$. This is the additional number of clients that this facility would need in order to reach B. If a facility i has more than B clients, $n_i > B$, then $\mathcal{I}_{cap}$ will have two supply points at this location, and no demand points. The first supply point has cost 0 and total supply $n_i - B$, and the second supply point has total supply B and cost $\delta \cdot B \cdot l(i)$, analogously to the previous case. The distances of $\mathcal{I}_{cap}$ are the same as in $\mathcal{I}_2$.

We now bound the cost of the optimal solution to $\mathcal{I}_{cap}$ in terms of the optimal solution to $\mathcal{I}_2$.

Lemma 5.4.1 $OPT(\mathcal{I}_{cap}) \leq (1 + \delta) \cdot OPT(\mathcal{I}_2)$

Proof. Let us examine the form of the optimal LBFL solution to $\mathcal{I}_2$, and then use it to construct a specific solution for $\mathcal{I}_{cap}$, whose cost is then an upper bound on $OPT(\mathcal{I}_{cap})$. We can assume without loss of generality, by using the triangle

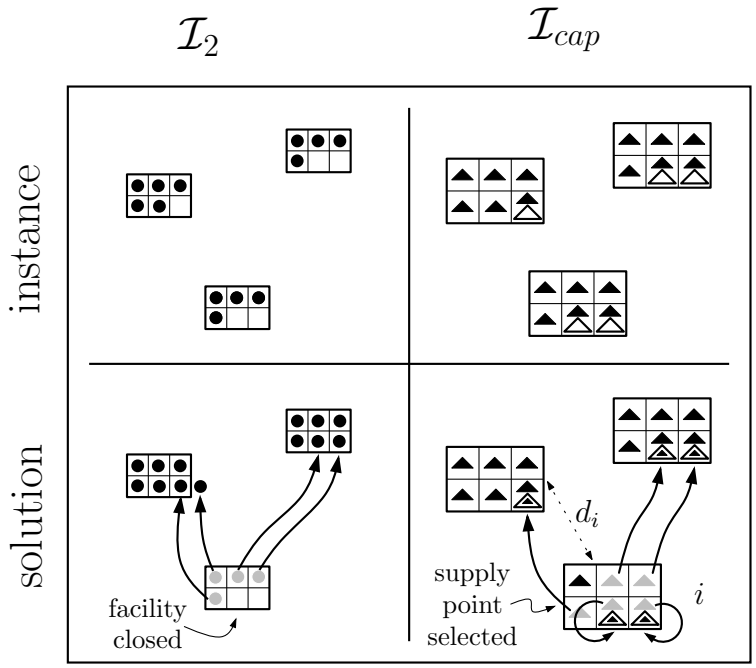

Figure 5.2: The top row shows the correspondence between the instance $\mathcal{I}_2$ of LBFL (with $B = 6$) and the constructed instance of CFL, $\mathcal{I}_{cap}$. The circles represent the clients in the LBFL problem. The black triangles represent the amount of supply at a supply point, and the white triangles represent the amount of demand. The bottom row shows the correspondence between the solutions to these instances. The location i which is closed in the solution to $\mathcal{I}_2$ is selected in the solution to $\mathcal{I}_{cap}$. Three units of its supply satisfy the demand of other locations, and two units of supply satisfy the demand of the same location.

inequality, that in this solution there is no facility i such that some clients are assigned from another facility to i, and other clients are assigned from i to another facility.

The solution that we propose for the CFL instance $\mathcal{I}_{cap}$ exactly corresponds to this LBFL solution. We select the supply points corresponding to the facilities which are closed in the LBFL solution, as well as the free supply points of the facilities with $n_i > B$. Then whenever k clients are assigned from facility i to facility i' by the LBFL solution, we say that k amount of supply is sent from the supply point i to the demand point i'. The first observation is that all the demand of the CFL instance is satisfied in this way: if a facility i with demand $B - n_i$ is opened by the LBFL solution, then there must be at least $B - n_i$ additional clients assigned to it in order to satisfy the lower bound requirement; if i is closed, then it will be selected, and will be able to satisfy its own demand of $B - n_i$ using part of its total supply of B. The second observation is that all the supply points have enough supply required from them by this solution. If a facility is closed, then it is sending n_i clients elsewhere, which is equal to its total supply in the case that $n_i > B$, or otherwise is equal to its total supply, B, minus the $B - n_i$ amount that it uses to satisfy its own demand. If the facility is open, then the only case in which it is sending out clients is if it started with $n_i > B$, and is sending out at most $n_i - B$, which is equal to the total supply of its corresponding free supply point.

Now we bound the cost of the constructed solution to $\mathcal{I}_{cap}$. Its connection cost is at most $OPT(\mathcal{I}_2)$, since we only moved supply that corresponds to clients that are assigned in the solution to $\mathcal{I}_2$. The solution's selection cost is at most δ times its connection cost, because the selection cost of $\delta \cdot l(i) \cdot \min(n_i, B)$ is paid for each

supply point i that corresponds to a closed facility, and the LBFL solution has to pay at least $n_i \cdot l(i)$ in connection cost in order to move the n_i clients from the closed facility i to other facilities, whose distance from i is at least $l(i)$. Thus the selection cost of our solution is at most $\delta \cdot OPT(\mathcal{I}_2)$. □

The next step of the algorithm is to solve the CFL instance $\mathcal{I}_{cap}$, obtaining a solution $\mathcal{S}_{cap}$, by using one of the known constant approximation algorithms for it (e.g. [69]). Say that the approximation ratio for this algorithm is γ. Then we get the following corollary to Lemma 5.4.1.

Corollary 5.4.2 *The cost of the solution $\mathcal{S}_{cap}$ found for the instance $\mathcal{I}_{cap}$ is at most $(1+\delta)\gamma \cdot OPT(\mathcal{I}_2)$.*

5.5 Reassignment of clients

Once the CFL instance $\mathcal{I}_{cap}$ is solved, we reassign clients from their locations in $\mathcal{I}_2$ according to the obtained solution, in a way that we explain and analyze in this section. Without loss of generality, we make the assumption that whenever a supply point at a location i is selected by the solution $\mathcal{S}_{cap}$, it serves all the demand of its own demand point at i.

The first type of reassignment of clients that we perform is exactly as proposed by the solution $\mathcal{S}_{cap}$: if the demand at some location i is satisfied by the supply from another location i' in the solution $\mathcal{S}_{cap}$, then we move the number of clients equal to this demand from i' to i. It is always possible to perform this reassignment because the total amount of supply exported from i' is never more than its number of clients, $n_{i'}$.

The reason that we do not yet have a feasible solution to the LBFL problem is the following. The specification of the CFL problem requires that any feasible solution satisfy all of the clients (demands); however, it does not require that an opened facility (selected supply point) use *all* of its capacity (supply). As a result, we may now have facilities, whose supply points were selected, but not all of whose clients were reassigned elsewhere. For example, in Figure 5.2, out of four clients that were at facility i, three were reassigned to other facilities, but one is left. The rest of this section explains how our algorithm deals with these clients that remain at the selected facilities. Let us summarize the two types of facilities that result after the first reassignment.

- There are some facilities, call this set $\mathcal{A} \subseteq \mathcal{F}_2$, which now have at least B clients. This set includes all facilities whose corresponding supply points were not selected by $\mathcal{S}_{cap}$ (and therefore whose demand amount of $B - n_i$, if positive, was fulfilled by supply from other locations).

- There are other facilities, $\overline{\mathcal{A}}$, which now have less than B clients. The way this happens is that their corresponding supply points were selected by $\mathcal{S}_{cap}$, and (possibly) some of their clients were reassigned to other locations. But note that for each such facility $i \in \overline{\mathcal{A}}$, the selection cost of $\delta \cdot l(i) \cdot \min(n_i, B)$ was paid by the solution $\mathcal{S}_{cap}$.

Facilities in the set $\mathcal{A}$ constitute the easy case, as we just open them and let them serve the clients currently assigned to them, satisfying the lower bound requirement. For the other facilities, however, we have to do a little more work.

Let us construct a directed graph G whose nodes are the facilities of $\mathcal{F}_2$. For each facility $i \in \overline{\mathcal{A}}$ of the second type, include an edge (i, i'), where $i' \in \mathcal{F}_2$ is the

nearest neighbor of i (remember that the distance between i and i' is $l(i)$). When constructing this graph, we use some ordering on the facilities to break ties and avoid cycles in the graph. As a result, G will consist of two types of connected components:

1. A tree, whose root is in $\mathcal{A}$, and whose other edges are directed toward the root.

2. A tree containing exactly one double edge (i.e. the pair of closest nodes with edges in both directions between them), with other edges of the tree directed toward this double edge.

Note that the facilities from $\mathcal{A}$ are always roots of type-1 trees, or singletons (which is a special case). Facilities from $\overline{\mathcal{A}}$ make up the non-root nodes of type-1 trees and the type-2 trees entirely. In particular, they are always in components of size at least two, which is important for our algorithm.

We now use the graph G to make some more reassignments of clients, to make sure that the lower bound constraints are satisfied. For each component of type 1, we do the following procedure on each facility i in this component, bottom-up (see Figure 5.3). If i has at least B clients, then open facility i and cut the tree edge going up from i. If i has less than B clients, then send all of these clients from i to its parent facility in the tree. Since the root is in $\mathcal{A}$, it will always have at least B clients, and already be open. Thus at the end of this procedure, each facility in the processed component will have either 0 or at least B clients, satisfying the lower bound constraints. Also notice that during this process, we send strictly less than B clients on each edge of the component.

For the second type of component, we perform the same bottom-up procedure

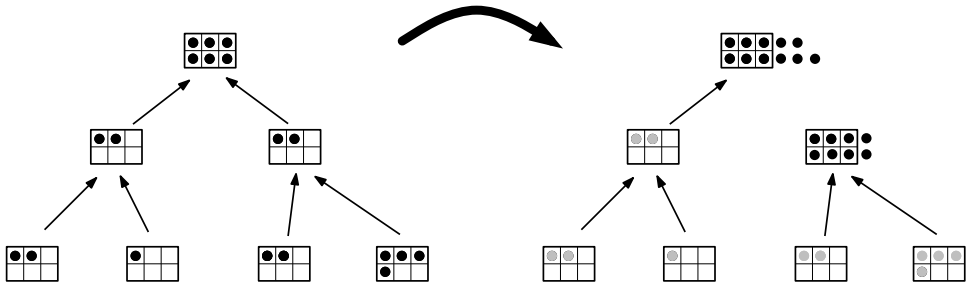

Figure 5.3: The outcome of the bottom-up process of client reassignment, with $B = 6$, on connected components of type 1.

on the parts of the tree directed toward the double edge. The only difference is in what to do with the double edge itself, whose endpoints we call i_1 and i_2. Here we consider several cases. If each of i_1 and i_2 has at least B clients, then open both of them. If one of them, say i_1, has at least B clients, and i_2 has less than B, then transfer all clients from i_2 to i_1 and open i_1. If each of them has less than B, but in total the two of them have at least B, then we transfer all clients from i_2 to i_1 and open i_1. In the case that the total number of clients at i_1 and i_2 is less than B, we find the closest facility $i \in \mathcal{A}$ to either one of the two endpoints (i.e., one minimizing $\min(c(i, i_1), c(i, i_2))$). Let us say without loss of generality that i is closer to i_1. Then we send clients from i_2 to i_1, and then all of them from i_1 to i. Since $i \in \mathcal{A}$, it already has at least B clients and is open, so the procedure overall produces a feasible solution to LBFL, which is the final solution that we output.

What remains to be done is to bound the cost incurred by all the transfers of clients that are performed after the solution of $\mathcal{I}_{cap}$. We bound it in terms of the connection cost, C^{cap}, and the selection cost, F^{cap}, of our solution $\mathcal{S}_{cap}$.

Lemma 5.5.1 *The cost of the solution found by our algorithm for $\mathcal{I}_2$ is at most*

$$\frac{2\alpha}{2\alpha - 1} \cdot C^{cap} + \frac{1}{\delta\alpha} \cdot F^{cap} \;\leq\; \max(\frac{2\alpha}{2\alpha - 1}, \frac{1}{\delta\alpha}) \cdot cost(\mathcal{S}_{cap})$$

Proof. After solving $\mathcal{I}_{cap}$, the algorithm makes three types of client reassignments, for which we bound the costs separately:

1. Reassign clients according to the supply and demand assignments of the solution $\mathcal{S}_{cap}$.

2. Reassign at most B clients for each edge of the graph G.

3. In case that facilities i_1 and i_2 forming a double edge in G don't have a total of B clients, reassign at most B clients from i_1 to the closest open facility i.

Reassignment of type 1 costs at most C^{cap}, as connection costs of $\mathcal{I}_{cap}$ are the same as those of $\mathcal{I}_2$.

For the second type of reassignment, we notice that for each edge in G which starts at a facility i and has length $l(i)$, the solution $\mathcal{S}_{cap}$ has paid $\delta \cdot l(i) \cdot \min(n_i, B)$ as a selection cost for the supply point i. But since $\mathcal{I}_2$ came from a bicriteria solution with parameter α, we know that $n_i \geq \alpha B$. So for each edge in G, the selection cost F^{cap} includes an amount of at least $\delta \cdot l(i) \cdot \alpha B$, whereas we pay at most $l(i) \cdot B$ for transferring clients on this edge. Thus, the total cost of reassignments of type 2 is at most $F^{cap}/\delta\alpha$.

For the third type of reassignment, we bound its cost against the connection cost of $\mathcal{S}_{cap}$. In particular, we make the following observation about the facilities i_1 and i_2 forming the double edge in G. As a result of the bicriteria algorithm, each of them has at least αB clients in $\mathcal{I}_2$, and so together they have at least $2\alpha B > B$ (since $\alpha > \frac{1}{2}$). However, after the execution of the CFL and reassignments of types 1 and 2, they have less than B. Since the bottom-up reassignment on the edges of G could have only added clients to i_1 and i_2, it must be that at least $(2\alpha - 1)B$ clients were moved to facilities in $\mathcal{A}$ (which are all at least as far as i) by the first

kind of reassignment. Therefore, for each such pair i_1 and i_2 that sends clients to their closest open facility i, the solution $\mathcal{S}_{cap}$ to our CFL instance must have paid at least $(2\alpha - 1)B \cdot c(i_1, i)$ in connection cost. So the total cost of type-3 reassignments is at most $C^{cap}/(2\alpha - 1)$. Adding the bounds, we get the result. □

By combining Lemma 5.5.1, Corollary 5.4.2, and Lemma 5.3.5, we get the following final result.

Theorem 5.5.2 *There is a constant-factor approximation algorithm for the load-balanced facility location problem.*

Proof. Setting $\delta = \frac{2\alpha-1}{2\alpha^2}$ and using it in Lemma 5.5.1 shows that our solution costs at most $\frac{2\alpha}{2\alpha-1}$ times the solution to the CFL instance $\mathcal{I}_{cap}$. Then applying Corollary 5.4.2 we get that it is a $\beta = \frac{2\alpha}{2\alpha-1}(1+\frac{2\alpha-1}{2\alpha^2})\gamma$ factor approximation for the instance $\mathcal{I}_2$, which can then be used in Lemma 5.3.5. Using the value of $\alpha = 0.68$, the $\rho = 1.52$ approximation algorithm for FL [54], and $\gamma = (5.83 + \varepsilon)$ approximation algorithm for CFL [69], the overall approximation ratio becomes $558+\varepsilon$. □

CHAPTER 6

CONCLUSIONS

In this thesis, we have designed approximation algorithms for problems arising from epidemiology, distributed databases, sensor networks and transportation. But many open questions remain in all these areas. We conclude by mentioning some of the possible directions for future research.

6.1 Graph cuts

The main open question raised by the work on unbalanced graph cuts is how well the MinSBCC and MaxSBCC problems can be approximated in a single-criterion sense. At this time, we are not aware of any non-trivial upper or lower bounds for their approximability. Section 2.3.3 presents a $(O(\log^{3/2} n), 1)$ approximation – however, it approximates the capacity instead of the size, and thus cannot be used for approximating dense subgraphs or communities.

Obtaining better approximation algorithms for the unbalanced cut problems will likely require using techniques different from the ones used in this thesis. Specifically, the linear program that is used in two of the bicriteria approximation algorithms for MinSBCC has a large integrality gap, as evidenced by a graph consisting of an isolated sink t, and a very large dense subgraph connected to the source s with one edge of capacity $1 + \epsilon$. The only s-t cut of capacity at most 1 is then the trivial one separating t from the rest of the graph, but the optimal fractional solution will fractionally cut the edge from s to the large subgraph, and have only an ϵ fraction of the cost of the optimum integral solution.

Further open directions involve more realistic models of the spread of diseases or disasters. The implicit assumption in our node cut approach is that each social

contact will always result in an infection. If edges have infection probabilities, for instance based on the frequency or types of interaction, then the model becomes significantly more complex.

For the multi-terminal cuts, it would be interesting to investigate the min-max objective for other graph cut problems besides the multiway cut. One example could be the *k-cut* problem, in which the graph does not have terminals, but just has to be partitioned into k components. Another direction would be to consider the combination of the min-max objective with the consideration of the sizes of the components of the cut. For example, in the data partitioning application of the min-max multiway cut, it may be desirable to also perform some load-balancing of the data among the servers. For the min-max multiway cut problem itself, it would be good to obtain some lower bounds on its approximability, as well as improved approximation guarantees.

6.2 Facility location

One extension of the facility location problem with hierarchical facility costs that we consider would be to remove the assumption that all facilities have identical cost functions. If this assumption is removed completely, i.e., facilities are allowed to have arbitrary different hierarchical cost functions, then the problem becomes hard to approximate to a factor of better than $\Omega(\log n)$ [61]. However, a milder relaxation would be to have cost functions for different facilities which are scalar multiples of one another. The analysis of our local search algorithm does not extend in a straight-forward manner to this case, so it remains an open question whether there is a constant-factor approximation for this setting.

A more fundamental question is about the general problem of facility location

with submodular cost functions. Finding a constant-factor approximation algorithm for it would be a big step forward in this area of research. Furthermore, other combinatorial optimization problems, which in their classical versions have simple objective functions, such as the sum of costs of a set of elements, can be considered in modified versions involving submodular functions in the objective or the constraints. I believe that such generalizations will find many useful applications.

For the load-balanced facility location problem, we have presented the first constant-factor true approximation algorithm. The constant in the approximation guarantee is of course not practical, so the main contribution of our work is a theoretical demonstration that there exist polynomial-time constant-factor approximation algorithms which solve the LBFL problem without violating the constraints. It would be interesting to find algorithms with much better guarantees, which may be useful in practice, and we leave it for future work.

Our algorithm can be extended to work for the case of clients with non-unit demands, in which each client has a non-negative demand, and the lower-bound constraints now require that the total *demand* served by a facility is at least B. However, the known solutions for capacitated facility location all allow the splitting of a client's demand, with parts of it being assigned to different facilities. So because we make use of the algorithms for CFL, our algorithm would also have to allow this kind of splitting of demand. Unfortunately, our algorithm for LBFL does not extend to another useful generalization of the problem, in which each facility has its own lower bound for the number of clients that it has to serve if opened. We leave the solution of LBFL with non-uniform bounds to future work. In fact, we have a simple reduction that shows how to use the solution to the non-uniform LBFL in order to solve a variant of the universal facility location

problem with monotone *non-increasing* facility costs (as opposed to the monotone non-decreasing costs which have been considered so far), without any loss in the approximation guarantee. This version of universal facility location generalizes LBFL. The reduction just involves creating multiple facilities in place of each original facility, with appropriate costs and lower bounds, but requires that the LBFL problem be solved with a true approximation, and not in the bicriteria sense. Another interesting related open problem is the universal facility location with *non-monotone* costs.

BIBLIOGRAPHY

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, 1993.

[2] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proc. 36th ACM Symp. on Theory of Computing*, 2004.

[3] V. Arya, N. Garg, R. Khandekar, K. Munagala, and V. Pandit. Local search heuristic for k-median and facility location problems. In *Proc. 33rd ACM Symp. on Theory of Computing*, pages 21–29, 2001.

[4] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34, 2000.

[5] N. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications.* Hafner Press, 1975.

[6] G. Calinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 48–52, New York, NY, USA, 1998. ACM Press.

[7] M. Charikar. Greedy approximation algorithms for finding dense components in graphs. In *Proc. 3rd APPROX*, 2000.

[8] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 378–388, 1999.

[9] F. Chudak and D. Shmoys. Improved approximation algorithms for a capacitated facility location problem. In *Proc. 10th ACM Symp. on Discrete Algorithms*, pages 875–876, 1999.

[10] F. A. Chudak. Improved approximation algorithms for uncapitated facility location. In *IPCO*, pages 180–194, 1998.

[11] F. A. Chudak and D. B. Shmoys. Improved approximation algorithms for the uncapacitated facility location problem. *SIAM J. Comput.*, 33(1):1–25, 2003.

[12] F. A. Chudak and D. P. Williamson. Improved approximation algorithms for capacitated facility location problems. In *IPCO*, pages 99–113, 1999.

[13] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd ACM Symp. on Theory of Computing*, pages 151–158, 1971.

[14] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[15] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[16] M. Develin and S. G. Hartke. Fire containment in grids of dimension three and higher, 2004. Submitted.

[17] Z. Drezner. *Facility Location: A Survey of Applications and Methods*. Springer, 1995.

[18] S. Eubank, H. Guclu, V.S.A. Kumar, M.V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429:180–184, 2004.

[19] S. Eubank, V.S.A. Kumar, M.V. Marathe, A. Srinivasan, , and N. Wang. Structural and algorithmic aspects of massive social networks. In *Proc. 15th ACM Symp. on Discrete Algorithms*, pages 711–720, 2004.

[20] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. In *Proc. 25th ACM Symp. on Theory of Computing*, 1993.

[21] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. on Computing*, 31(4):1090–1118, 2002.

[22] U. Feige, R. Krauthgamer, and K. Nissim. On cutting a few vertices from a graph. *Discrete Applied Mathematics*, 127:643–649, 2003.

[23] U. Feige and M. Seltser. On the densest k-subgraph problem. Technical report, The Weizmann Institute, Rehovot, 1997.

[24] G. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self-organization of the web and identification of communities. *IEEE Computer*, 35, 2002.

[25] G. Flake, R. Tarjan, and K. Tsioutsiouliklis. Graph clustering techniques based on minimum cut trees. Technical Report 2002-06, NEC, Princeton, 2002.

[26] L. Ford and D. Fulkerson. Maximal flow through a network. *Can. J. Math*, 8:399–404, 1956.

[27] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. on Computing*, 18:30–55, 1989.

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[29] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[30] N. Garg, R. Khandekar, and V. Pandit. Improved approximation for universal facility location. In *Proc. 16th ACM Symp. on Discrete Algorithms*, pages 959–960, 2005.

[31] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM J. on Computing*, 25:235–251, 1996.

[32] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *J. Algorithms*, 31(1):228–248, 1999.

[33] S. Guha, A. Meyerson, and K. Munagala. Hierarchical placement and network design problems. In *Proc. 41st IEEE Symp. on Foundations of Computer Science*, page 603, 2000.

[34] M. T. Hajiaghayi, M. Mahdian, and V. S. Mirrokni. The facility location problem with general cost functions. *Networks*, 42:42–47, 2003.

[35] E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In *Proc. 11th ACM Symp. on Discrete Algorithms*, pages 329–337, 2000.

[36] A. Hayrapetyan, D. Kempe, M. Pal, and Z. Svitkina. Unbalanced graph cuts. In *Proc. 13th European Symposium on Algorithms*, 2005.

[37] A. Hayrapetyan, C. Swamy, and E. Tardos. Network design for information networks. In *Proc. 16th ACM Symp. on Discrete Algorithms*, pages 933–942, 2005.

[38] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Math. Programming*, 22(1):148–162, 1982.

[39] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[40] S. Iwata, L. Fleischer, and S. Fujishige. A combinatorial strongly polynomial algorithm for minimizing submodular functions. *J. ACM*, 48(4):761–777, 2001.

[41] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. In *Proc. 34th ACM Symp. on Theory of Computing*, pages 731–740, 2002.

[42] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.

[43] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[44] D. R. Karger, P. Klein, C. Stein, M. Thorup, and N. E. Young. Rounding algorithms for a geometric embedding of minimum multiway cut. In *Proc. 31st ACM Symp. on Theory of Computing*, pages 668–678, 1999.

[45] D. R. Karger and M. Minkoff. Building steiner trees with incomplete global knowledge. In *Proc. 41st IEEE Symp. on Foundations of Computer Science*, page 613, 2000.

[46] S. Khot. Ruling out PTAS for graph min-bisection, densest subgraph and bipartite clique. In *Proc. 45th IEEE Symp. on Foundations of Computer Science*, 2004.

[47] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[48] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. *J. Algorithms*, 37(1):146–188, 2000.

[49] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *8th International World Wide Web Conference*, 1999.

[50] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehard and Winston, 1976.

[51] F.T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46, 1999.

[52] A. Lim, F. Wang, and Z. Xu. A transportation problem with minimum quantity commitment. *Transportation Science*, 40(1):117–129, 2006.

[53] M. Mahdian and M. Pál. Universal facility location. In *European Symposium on Algorithms*, pages 409–421, 2003.

[54] M. Mahdian, Y. Ye, and J. Zhang. Improved approximation algorithms for metric facility location problems. In *Proc. 5th APPROX*, pages 229–242, 2002.

[55] M. Mahdian, Y. Ye, and J. Zhang. A 2-approximation algorithm for the soft-capacitated facility location problem. In *Proc. 6th APPROX*, pages 129–140, 2003.

[56] M. Pal, E. Tardos, and T. Wexler. Facility location with nonuniform hard capacities. In *Proc. 42nd IEEE Symp. on Foundations of Computer Science*, page 329, 2001.

[57] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *Proc. Natl. Acad. Sci. USA*, 101(9):2658–2663, 2004.

[58] R. Ravi and A. Sinha. Multicommodity facility location. In *Proc. 15th ACM Symp. on Discrete Algorithms*, pages 342–349, 2004.

[59] A. Schrijver. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *J. of Combinatorial Theory, Ser. B*, 80(2):346–355, 2000.

[60] D. Shmoys. Cut problems and their application to divide-and-conquer. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard problems*, pages 192–235. PWD Publishing, 1995.

[61] D. Shmoys, C. Swamy, and R. Levi. Facility location with service installation costs. In *Proc. 15th ACM Symp. on Discrete Algorithms*, pages 1088–1097, 2004.

[62] D. B. Shmoys, E. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Proc. 29th ACM Symp. on Theory of Computing*, pages 265–274, 1997.

[63] M. Sviridenko. An improved approximation algorithm for the metric uncapacitated facility location problem. In *IPCO*, pages 240–257, 2002.

[64] Z. Svitkina. Load-balanced facility location. 2007. submitted.

[65] Z. Svitkina and E. Tardos. Min-max multiway cut. In *Proc. 7th APPROX*, 2004.

[66] Z. Svitkina and E. Tardos. Facility location with hierarchical facility costs. In *Proc. 17th ACM Symp. on Discrete Algorithms*, pages 153–161, 2006.

[67] V. V. Vazirani. *Approximation algorithms*. Springer-Verlag New York, Inc., 2001.

[68] P. von Rickenbach and R. Wattenhofer. Gathering correlated data in sensor networks. In *DIALM-POMC '04: Proceedings of the 2004 Joint Workshop on Foundations of Mobile Computing*, pages 60–66, 2004.

[69] J. Zhang, B. Chen, and Y. Ye. A multiexchange local search algorithm for the capacitated facility location problem. *Math. Oper. Res.*, 30(2):389–403, 2005.