FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Verfahren zur Platzierung mehrerer Senken in ungerichteten Flußnetzwerken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Oliver Hubmer, BSc
Matrikelnummer 0842138

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

Wien, 3. Oktober 2017

Oliver Hubmer   Martin Nöllenburg

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Techniques for Multiple Sink Placement in Undirected Flow Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Oliver Hubmer, BSc
Registration Number 0842138

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

Vienna, 3rd October, 2017

Oliver Hubmer

Martin Nöllenburg

Erklärung zur Verfassung der Arbeit

Oliver Hubmer, BSc
Breitenfurter Straße 436
1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Oktober 2017

Oliver Hubmer

Danksagung

Ich möchte hiermit die Möglichkeit nützen, um mich bei allen Beteiligten zu bedanken, ohne die, die Erstellung der Diplomarbeit nicht möglich gewesen wäre.

Einen besonderen Dank gilt an erster Stelle meinem Betreuer, Herrn Assistant Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg für die sehr gute Betreuung während des gesamten Prozesses der Erarbeitung meiner Diplomarbeit.

Ein herzliches Dankeschön richte ich an meine Freundin, meine Eltern, und meinem Bruder die immer große Geduld mit mir gehabt haben, und für die große Unterstützung während der Zeit der Erstellung der Diplomarbeit, als auch während meines gesamten Studiums.

Acknowledgements

I would like to take the opportunity to thank all those involved, without whom the preparation of the thesis could not have been possible.

A special thank you goes to my supervisor, Assistant Professor Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg for the very good support during the entire process of the development of my diploma thesis.

A hearty thanks to my girlfriend, my parents, and my brother, who have always had great patience with me, and for the great support during the time of writing the thesis, as well as throughout my study.

Kurzfassung

In Zeiten von Smartphones und mobilen Technologien ist Telekommunikation ein wichtiger Teil des modernen Lebens und unentbehrlich für private und professionelle Kommunikation. Mit der Entstehung von Social Media, Musik und Video-Streaming werden immer mehr Daten über diese Netzwerke gesendet. Diese besitzen jedoch nur eine begrenzte Bandbreite. Um eine stabile und zuverlässige Kommunikation von mobilen Geräten zu gewährleisten, muss eine solide Infrastruktur in Form eines Netzwerks sorgfältig geplant und überwacht werden, sodass für alle Kunden eine ausreichende Bandbreite gewährleistet wird. Um eine flächendeckende Abdeckung sicherstellen zu können, und um der ständig wachsenden Nachfrage nach mehr Bandbreite gerecht zu werden, müssen mehr Funktürme errichten werden, oder neue Technologien wie Glasfasern genutzt werden, um bestehende Funktürme und deren Verbindungen zu erweitern. Dadurch wird die Datenmenge erhöht, die über ein Netzwerk gesendet werden kann. Diese Diplomarbeit konzentriert sich auf die letztgenannte Option, auf die Modernisierung bestehender Sendemasten, und welche Sendemasten am besten für Upgrades geeignet sind. Dies ist abhängig von unterschiedlichen Kriterien, wie der Bandbreitenverfügbarkeit, den Kosten und anderen Faktoren wie zukünftigen potenziellen Märkten. Trotz diesem speziellen Usecase sind die entwickelten Methoden und Algorithm mit wenig bis keinem Aufwand für andere, ähnliche, Problemstellungen adaptierbar.

Diese Arbeit umfasst sechs Kapitel. Kapitel eins enthält eine Einführung mit allgemeinen und spezifischen Hintergrundinformationen, eine allgemeine Problemformulierung, und getroffene Annahmen über das Problem. Kapitel 2 besteht aus der Literaturrecherche und analysiert Artikel über ähnliche Probleme. In Kapitel 3, der Hauptteil der Arbeit, wird eine formale Problembeschreibung mit exakten und heuristischen Algorithmen zur Lösung vorgestellt. Im folgendem Kapitel 4 wird ein exakter Algorithmus zum lösen des Problems für Baum-Struktur Netzwerke, zusammen mit einem Korrektheitsbeweis beschrieben. Kapitel 5 behandelt die effiziente Berechnung von Maximalen Flüssen, ein wichtiger und entscheidender Teil der vorgestellten Algorithmen in Kapitel drei. Schlussendlich werden in Kapitel 6 die Algorithmen getestet und analysiert. Darüber hinaus wurden die Algorithmen zusätzlich auf einem echten praktischen Problem getestet.

Die dargestellten Lösungs-Techniken sind in der Lage, gute Lösungen innerhalb von 85% der optimalen Lösungen zu produzieren, wobei jedoch nur ein Bruchteil der Zeit zum generieren der Lösungen benötigt wird. Darüber hinaus produzieren die Algorithmen vielversprechende Ergebnisse für reale Instanzen.

Abstract

In times of smartphones and mobile technologies, telecommunication is a crucial part of modern life and indispensable for private and professional communication. With the emergence of social media, music and video streaming, evermore data is send over these networks, which have a limited bandwidth capacity. In order to ensure a stable and reliable communication of mobile devices, a solid infrastructure in form of a network must be planned and monitored carefully, to ensure enough available bandwidth for all customers. A comprehensive network of radio towers and their connections have to ensure an area wide coverage. In order to satisfy the ever increasing demand for more bandwidth more radio towers can be constructed, or new technologies, such as fiberglass, can be used to upgrade existing radio towers and their connections to increase the amount of data which can be send through a telecommunication network at any given time. This thesis focuses on the latter option, in upgrading existing radio towers, and in which radio towers are best suitable for upgrades depending on different criteria, such as the bandwidth availability, the costs, and other factors like future potential markets. Furthermore, despite the specific use case the methods and algorithms can be used for other problems with little or no adaption.

This thesis comprises six main chapters. Chapter one gives an introduction, with general and specific background information, general problem formulation, and made assumptions about the problem. Chapter two consists of the literature review, analysing articles about similar problems. In chapter three, the main part, a formal problem description with exact and heuristic solving algorithms are presented. Followed by chapter four, an exact algorithm for upgrading radio towers in a tree structured network, together with a proof is introduced. Chapter five treats with the efficient calculation of maximum flows, an important and crucial part of the presented algorithms in chapter three. Finally, in chapter six, the computational performance of the algorithms, and the graph generation process is presented and analysed. Furthermore, the algorithms were tested on a real life instance.

The presented solution techniques are able to produce solutions within 85% of the optimal solutions, however, needing only a fraction of the time of the exact solution techniques. In addition, the algorithms produce promising results for the real world instances.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Assumptions and Problem Statement 2

1.3 Outline 4

2 Literature Review **7**

3 Multiple Sink Placement in Undirected Flow Networks **11**

3.1 Formal Problem Description 11

3.2 Integer Linear Programming Formulations 12

3.3 Greedy Heuristic 17

3.4 GRASP 19

3.5 Greedy-Net 20

3.6 Clustering 21

3.7 Post Optimization 24

4 An Exact Algorithm for Trees **27**

4.1 Background and Formal Problem Description 27

4.2 Integer Linear Programming Formulation 27

4.3 Greedy Tree Solver 28

4.4 Greedy Tree Solver Proof 30

5 Maximum Flow Calculation **35**

5.1 Maximum Flow in an Undirected Network 35

5.2 Maximum Flow Algorithm 36

6 Computational Performance Evaluation **39**

6.1 Graph Generation 39

6.2 Testing System and Parameter Settings 43
6.3 Performance Evaluation 44
6.4 Performance Evaluation on an Austrian Network 59

7 Conclusion **65**

List of Figures **67**

List of Tables **69**

List of Algorithms **71**

Bibliography **73**

CHAPTER 1

Introduction

This chapter gives a brief introduction, with general and specific background information, general problem formulation, and made assumptions about the problem.

1.1 Background and Motivation

Mobile telecommunication networks are a crucial and important part of modern life, which allow phone and Internet access in a vast amount of areas around the world. With the emergence of social media, music and video streaming, evermore data is sent over these networks, which have a limited bandwidth capacity. Therefore, networks used for mobile telecommunication have to be planned and monitored carefully, to ensure enough available bandwidth for all customers. In order to satisfy the ever increasing demand for more bandwidth more radio towers can be constructed, or new technologies, such as fiberglass, can be used to upgrade existing radio towers and their connections to increase the amount of data which can be sent through a telecommunication network at any given time. This thesis focuses on the latter option, in upgrading existing radio towers to be capable of handling more data. However, upgrading or replacing old technology is cost intensive, therefore, the network designer has to plan the upgrades depending on different criteria, such as the bandwidth availability, the costs, and other factors like future potential markets. Furthermore, the possible applications of the presented methods are not limited to telecommunication networks, but also for use cases in general logistics.

The data is sent over one or more radio towers to the backbone, usually high bandwidth fiberglass connections, of the telecommunication network which further distributes the data. Radio towers can be directly connected to the backbone, the more radio towers are connected to the backbone the better, since more data can be directly sent to the fiberglass wires of the backbone for further distribution, and therefore, increasing the performance of the telecommunication network. Upgrading an existing radio tower with a connection to the backbone network pose a significant cost factor, depending on the distance to the backbone network and geographical restrictions. In addition to the cost factor, customer potential and customer value represent other factors to be considered. Customer potential indicates the market saturation in specific areas together with the probability that customers switch their provider. In short, how many new potential customers are in a specific area. Customer value represents the importance of existing customers, and their fragmentation in groups like business customers or power users. Consequently, installing new connections between radio towers and the backbone is a crucial part in designing telecommunication networks and highly influences the performance (how much data can be sent over the network at the same time) of the network. This thesis focuses on methods and algorithms to find the best, depending on different factors, candidates for upgrades.

1.2 Assumptions and Problem Statement

It is assumed that the telecommunications network consist of a backbone network with unlimited bandwidth (or at least enough to handle all current peaks of occurring data), and a network of radio towers. Both networks are connected and the data within the radio tower network is sent to the backbone network for further treatment. Therefore, radio towers connected to the backbone are considered as sinks. Since backbone networks have enough bandwidth to handle large amounts of data, they act as a data highway for connections between distant places, for example between two large cities, and all radio towers try to route their data to the backbone network. Figure 1.1 shows an example telecommunications network. The radio towers (or vertices) X, Y, Z are part of the backbone network depicted in black. The radio tower network is represented with grey radio towers (vertices) with dotted lines as connections, in addition, the vertices X, Y, Z are also part of the radio tower network (they are normal radio towers with a fiberglass connection to the backbone). How the backbone network further handles the data is not within the scope of this work, and therefore, only the radio tower network is considered. In the following passages the radio towers are also denoted as vertices.

The accruing data of each radio tower must be sent to the backbone network for further processing. Since not all radio towers are directly connected to the backbone, the data must be sent over other radio towers until a connection to the backbone is found. Figure 1.2 illustrates the possible flow path of data from vertex X to the backbone network. In 1.2a the data has to perform four hops to a backbone connected vertex Z. After upgrading vertex Y with a connection to the backbone network, only two hops need to be performed. In general, every vertex produces certain amount of data and all the data must be sent to backbone connected vertices, consequently, the telecommunications network can be considered as a flow network, where all vertices are sources (emitting data), and vertices connected to the backbone are sinks. The target is to find the best locations for installing new sinks. Each vertex has assigned four specific values influencing this decision.

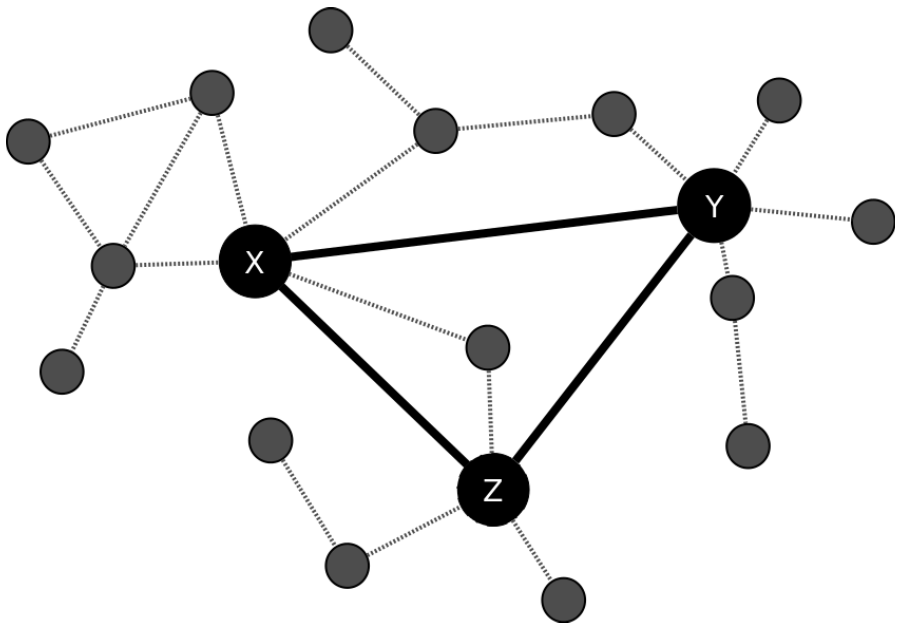


Figure 1.1: A telecommunications network, with the vertices X, Y, Z as part of the backbone network, and therefore acting as sinks. The remaining grey vertices represent ordinary radio towers, which need to route their data to the sinks.

Given is a directed graph $G(\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$. If there is a connection between two vertices v and u then there exist two directed edges, one going from v to u and one going from u to v. The vertices and edges have the following attributes.

Vertex Attributes:

1. **Accruing Data**, the amount of data which accrues at each vertex, i.e. the data which is "produced" at each vertex, not concerning forwarded data of other vertices. In general this value represents the peak load of a specific time period, or an average over a certain amount of time.
2. **Upgrade Cost**, represents the financial expense needed for upgrading a specific vertex with a connection to the backbone network. This cost is determined by geographical conditions and, mainly, by the distance from the vertex to the existing backbone network.
3. **Customer Value**, indicates the importance of customers, depending on their status, e.g. business customers usually have a higher customer value than private customers. The customer value of a vertex is defined by the average and normalized customer value of all customers served by this vertex.
4. **Customer Potential**, an artificial value defining the potential of finding new customers which will be served by this particular vertex. A higher value indicates a higher likelihood, a lower value the opposite.

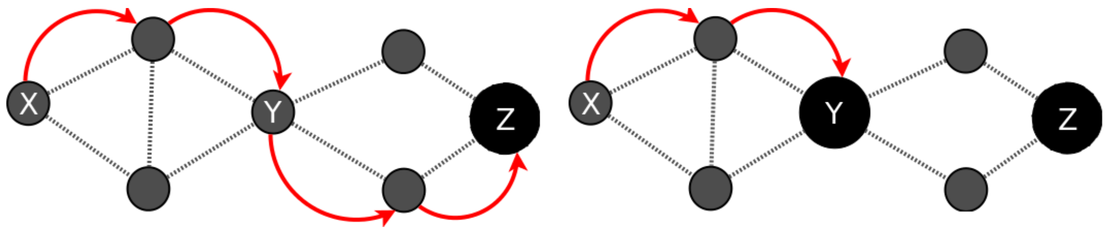
Furthermore, the location coordinates of the vertices are also given, however, the coordinates are solely needed for plotting tasks.

Radio towers are connected via directional radio or by other means, further denoted as edges. Finding a suitable vertex for a sink is also greatly determined by its incident edges, over which the data is sent and distributed. Each edge has assigned one specific value.

Edge Parameters:

1. **Edge Capacity**, represents the capacity, the maximum amount of data which can be sent over this edge at a specific time.

In addition, the vertices connected by the edge are also given.



(a) One connection (Z) to the backbone. (b) Two connections (Y and Z) to the backbone.

Figure 1.2: Possible flow of data from vertex X to a backbone connection.

The network can be considered as a flow network with all vertices as sources and vertices connected to the backbone as sinks. Sinks have to be placed in such a manner that the possible maximum flow is as high as possible, regarding the given parameters, like accruing data, of each vertex. All accruing data of the vertices should be handled. Furthermore, the utilization of each edge capacity should be as low as possible. Depending on where the sinks are placed, the optimal flow direction of an edge can change. In general, a static moment in time is considered for calculating solutions, usually the moment of maximum utilization of the network, or an average of all values for a certain time period, in order to reduce the complexity of the computation. Due to this facts, in this paper this problem is considered as Multiple Sink Placement in Undirected Flow Networks problem.

1.3 Outline

This thesis comprises six main chapters. Chapter two consists of the literature review, analysing articles about similar problems. In chapter three, the main part, a formal problem description with exact and heuristic solving algorithms are presented. Followed by chapter four, an exact algorithm for upgrading radio towers in a tree structured network, together with a proof is introduced. Chapter five treats with the efficient calculation of maximum flows, an important and crucial part of the presented algorithms in chapter three. Finally, in chapter six, the computational performance of the algorithms, and the graph generation process is presented and analysed. Furthermore, the algorithms were tested on a real life instance.

CHAPTER 2

Literature Review

The literature research for the Multiple Sink Placement problem in Undirected Flow Networks showed, that the available literature is limited. Most of the found literature considers the sink placement of new sinks in a network, whereas, the upgrade of existing nodes to new sink nodes is rarely considered.

The article [KWS$^+$11] by Kim et al. propose a new multiple-sink positioning problem in wireless sensor networks, formally defined as the k-Sink Placement Problem (k-SPP). k-SPP deals with the problem how to minimize the maximum data latency from a node to its nearest sink, in addition, the goal is to minimize the maximum hop distance between a node and its nearest sink, given k available sinks. The authors formally define the k-SPP and prove that k-SPP is APX-complete by showing there is no $(2 - \epsilon)$-approximation algorithm, where ϵ is a small positive constant (unless $P = NP$). Furthermore, the authors show that an existing greedy approximation algorithm for the k-center problem, GREEDY-k-CENTER, is also an approximation algorithm for k-SPP. This greedy approximation algorithm is used as a basis for a simple greedy algorithm (GREEDY-k-SPP), which uses a larger, but still polynomial-size, feasible solution space, in order to produce better quality solutions. It is shown that even in the worst case, the cost of an output of this algorithm is within three times from the cost of an optimal solution. In the performance analysis the average performance of GREEDY-k-SPP is compared with the alternative, GREEDY-k-CENTER. For simulations the authors use a 100x100 space grid and randomly deploy a number of nodes. On average, the GREEDY-k-SPP outperforms its competitor, and its approximation ratio is very near to the best achievable, 2.

In [CQJM04b] Qiu et al. address the efficient integration of multi-hop wireless networks with the internet. In this case, few Internet Transit Access Points (ITAP) serve as gateways, and low-cost antennas form a multi-hop wireless network, in order to route traffic to the internet through the ITAPs. The authors deal with the effiecient placement of ITAPs in the network, which in its simplest form, is to place a minimum number of ITAPs to serve a given set of nodes. These nodes are served through paths, which are allowed to pass through other nodes. The goal is to place the minimum number of ITAPs to serve the node's demands, considering all capacity constraints. Qiu et al. present several ways to model wireless interference, with the use of a conflict graph or alternate activation of paths. Furthermore, the $throughput_l$ defines the throughput on a link of length l, assuming each link has a capacity of 1. In addition, $g(l)$ defines the amount of link capacity consumed on a path of length l with a *throughput* of 1. In this paper, the authors study two models separately. The ideal link model defines the $throughput_l = 1$ and $g(l) = 1$, and the general link model defines the $throughput_l$ and $g(l)$ as a linear function of l. In the paper placement algorithms for the ideal and the general link model are considered. In detail, the authors present a LP formulation for both problems, the ideal and general link model. Furthermore, the LP formulations get extended with constraints ensuring fault tolerance. A theorem, proofed in [CQJM04a], states the ITAP placement problem has no polynomial approximation algorithm with an approximation ratio better than $\ln n$, where n is the size of the given nodes. Consequently, the authors present a greedy placement algorithm, which iteratively picks an ITAP that maximizes the total demands satisfied when opened in conjunction with the ITAPs chosen in the previous iterations. Several alternative algorithms are mentioned, augmented placement, clustering-based placement, and random placement. In order to validate the models, the authors ran simulations using Qualnet, a commercial network simulator. In addition, the authors evaluated the performance of the greedy algorithms, using different test instances of diverse size and with distinct constraints. It is shown that the greedy algorithms give close to optimal solutions over a variety of scenarios the authors have considered.

In [OE04] Oyman and Ersoy address the Multiple Sink Network Design Problem, which considers the placement of multiple sinks across a network. In a large-scale networks with a large number of sensor nodes, multiple sink nodes should be deployed to increase the manageability of the network, and to reduce the energy dissipation at each node, in order to prolong the network lifetime in wireless sensor networks. In this paper the authors define the Best Sink Locations (BSL) problem, the number of sink nodes is known, and therefore, the number of clusters which are formed around the sink nodes. As clustering algorithms the authors propose k-means clustering, and mention self-organizing maps. If the Euclidean distance is used as the clustering metric then the center of mass of the nodes within a cluster represents the location of the sink nodes. Other distance metrics can be used depending on the priorities of the routing algorithm. Furthermore, in this paper the authors define further problems, the Minimization of the Number of Sink Nodes for a Predefined minimum Operation Period (MSPOP) problem, and the Minimization of the Number of Sink Nodes while Maximizing the Network Lifetime (MSMNL) problem. The authors perform computational experiments for the BSL problem, with a test setup consisting of 200 nodes, and the placing of three sinks. As a result the authors present the corresponding energy and disconnected region maps on a sample sensor network for different snapshots in time.

In [ABIK06] Aoun et al. consider the placement of a minimum number of gateways in a wireless mesh network (WMN). In this topology all the traffic flows either to or from a gateway, as opposed to ad hoc networks where the traffic flows between arbitrary pairs of nodes. The authors present a polynomial time near-optimal algorithm to divide the WMN into clusters of bounded radius under relay load and cluster size constraints, while ensuring given QoS requirements. This consists in logically dividing the WMN into a set of disjoint clusters, covering all the nodes in the network. In each cluster, one node serves as a gateway, and serves the other nodes inside the cluster. Furthermore, in each cluster a spanning tree rooted at the gateway is used for traffic aggregation and forwarding. Each node is mainly associated to one tree, and would attach to another tree as an alternative route in case of path failure. As QoS restrictions the following points are considered, an upper bound on the cluster radius, an upper bound on the cluster size, and an upper bound on relay traffic. The core algorithm is a greedy approach consisting of the recursive approximation of the minimum Dominating Sets (DS). At iteration $i+1$, the algorithm computes a minimum dominating set using the resulting graph and dominating set from the previous iteration. The recursive calls stop if the cluster radius of the next iteration is larger than the upper bound on the cluster radius. In this case, the set of required gateways, satisfying the QoS requirements, is returned. However, each cluster is tested on feasibility. A cluster is defined as feasible if a spanning tree, rooted at the current node and covering all nodes in the current cover, satisfies the relay load and cluster size constraints. The authors prove in a theorem that the gateway placement algorithm can be implemented to run in time less then $\sqrt{2R} * O(V^2)$, where R is the upper bound on the cluster's radius and V is the number of nodes in the network. The performance is compared and tested to alternative algorithms, the Iterative Greedy Dominating Set and the Augmenting Placement. The performance of the different placement algorithms is evaluated using various QoS parameters in terms of cluster size, relay load, and radius size. The algorithms are evaluated according to the number of required gateways or clusters they produce. The evaluations show that one algorithm does not outperform all other algorithms, the performance is dependent on the QoS parameters.

The article [FKE11] by Flathagen et al. considers the optimal placement of a given number of sinks in Wireless Sensor Networks (WSN). The authors distinguish two different schemes in two categories, first, those that require knowledge about the geographical positions of all nodes (geo-aware), and second, those that rely on the network topology (topology-aware). In total four different sink deployment strategies are presented, two for each category. First, K-means placement (KSP): it is used to find the cluster centroids for a predetermined number of sinks. The clusters are generated and each node is assigned to the cluster with the closest Euclidean centroid. Then the k centroids are repositioned to the mass center of each cluster. After this step the iteration starts anew with the assignment of the nodes to the closest centroids. Second, K-medoid placement (KDP), is closely related to K-means placement, however, instead of using cluster centroids, K-medoid builds on the concept of medoids. A medoid is defined as the most central object in a cluster. Third, Shortest path placement (SPP) builds on KDP and differs mainly in the distance measure employed. A shortest path matrix between all nodes is created using Dijkstras algorithm, these shortest path distances now constitutes the distance measure replacing the Euclidian distance measure used in the KDP algorithm.

SPP then finds k nodes that minimize the average number of hops in respect to the remaining nodes in the network. Fourth, Routing Metric placement (RMP) provides an extension to the SPP algorithm, that uses a metric for each edge before performing the shortest path calculation. The sink placement will then be optimized according to the chosen metric, instead of being optimized according to a separate measure, such as the Euclidean distance. Experiments show that SPP, and especially RMP perform well under all network conditions, and outperform the geo-aware methods, KSP and KDP.

In [SEHZ12] Safa et al. solve the multiple sink placement problem by proposing an efficient and robust approach based on Particle Swarm Optimization (PSO) with local search (LS), called Discrete Particle Swarm Optimization (DPSO). The authors define a fitness function based on the DISCO network calculator, which calculates the network delay and throughput and uses curves to describe the network traffic and services. The objective function in the sink placement problem is to minimize the maximum worst case delay for each sensor. After initializing a population of particles, at every generation, an exchange operator is applied where two distinct locations are picked randomly and swapped with a certain probability. Then, with a specific probability, one-cut crossover is applied to all particles, with the global and personal best as parents. Next, two-cut crossover with a certain probability is applied, again with the global and personal best as parents. In experimental simulations the authors show that DPSO succeeded to get better results than current alternative approaches, such as Genetic Algorithm-based Sink Placement (GASP) presented in [PS08], in a shorter time.

CHAPTER 3

Multiple Sink Placement in Undirected Flow Networks

In this chapter various techniques for solving the multiple sink placement problem in undirected graphs are explained. First, a formal problem description and linear programming formulations are explained in Sections 3.1 and 3.2, with the objective function described in Section 3.2. Second, the heuristic algorithms are explained, the LP Relaxation in Section 3.2.1, the Greedy Heuristic in Section 3.3, another greedy heuristic, Greedy-Net in Section 3.5, a GRASP method in Section 3.4, and a clustering method in Section 3.6. Finally, post optimization local search techniques are introduced in Section 3.7.

3.1 Formal Problem Description

In the sections 1.1 and 1.2 general introductions and assumptions were given, followed by a formal problem description in this section. The problem is modelled such that a directed graph $G = (\mathcal{V}, \mathcal{E})$ of the network is given. The vertices $\mathcal{V}$ represent the locations of radio towers, and the edges $\mathcal{E}$ the connections of the vertices. For each vertex $v \in \mathcal{V}$ a customer value $\texttt{val}(v)$ is given, where $\texttt{val}(v)$ is a function of the customer value of the customers who use this vertex, with $\texttt{val} : v \to [0, 1]$. A higher value indicates a higher customer value. We denote by $\texttt{load}(v)$ the accrued data load at vertex v, where $\texttt{load}(v) \geq 0$. In addition, the customer potential $\texttt{pot}(v)$ is given for each vertex v, which is a function of the number of potential new attracted customers in this area in the future, it holds that $\texttt{pot} : v \to [0, 1]$. A higher value indicates a higher potential to gain new customers. The degree of each vertex is given by $\texttt{deg}(v)$ and the maximum degree of the graph by $\triangle(G)$. For each edge $e = (v, u) \in \mathcal{E}$ (the edge between the two vertices v and u) information about the maximum capacity (the maximum amount of data that can be send over this edge) $\texttt{cap}(e)$ is available. The variable $\texttt{cur}(e)$ holds the current load for each edge $e \in \mathcal{E}$, the flow of this edge. Some of the vertices $v \in \mathcal{V}$ are fibreglass vertices, which we denote by the set $\mathcal{F} \subset \mathcal{V}$. These fibreglass vertices $v \in \mathcal{F}$ act as sinks and the whole data load must be directed to a vertex $v \in \mathcal{F}$. Furthermore, a number k is given, which denotes the number of vertices $v \in \mathcal{F}$ that can be upgraded to a new fibreglass vertex. The fibreglass vertices are also denoted as sinks. For each vertex v a specific cost $\texttt{cost}(v)$ for upgrading this vertex to a sink is given. Furthermore, all other vertices $v \in \mathcal{V} \setminus \mathcal{F}$ are sources emitting $\texttt{load}(v)$ as load (in fact all vertices $v \in \mathcal{V}$ are sources emitting $\texttt{load}(v)$ as load, however since vertices $v \in \mathcal{F}$ are also sinks, their load $\texttt{load}(v)$ can be omitted). The solutions are generated for a static moment in time, usually the moment of maximum utilization of the network, or an average of all values for a certain time period.

3.2 Integer Linear Programming Formulations

The presented integer linear programming formulations are able to produce an exact and optimal solution for a given problem instance and represent the mathematical formulation of the problem. The problem gets formulated in a mathematical model whose requirements are represented by linear relationships. Linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints (for further information see [Dan16]). In the following sections two different linear programming formulations are presented, and one linear programming relaxation, 3.2.1. The objective function is presented after the two integer linear programming formulations, 3.2.

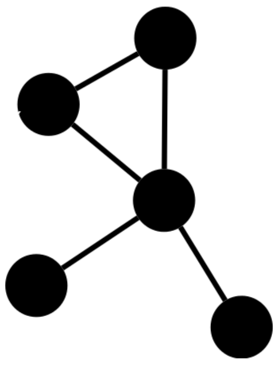
Integer Linear Programming Formulation 1

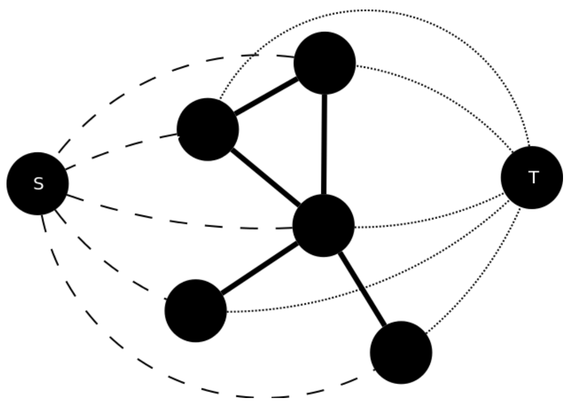
In this formulation, the existing network is extended by two further artificial vertices. A supersink T, which will be the only sink in the network, and a supersource S, which will be the only source in the network. In addition, further edges are introduced connecting the supersink and the supersource with all vertices respectively. Figure 3.1 illustrates the changes for a given network. In 3.1b the vertices S and T correspond to the supersource and supersink respectively. The dashed edges are connecting the supersource with all vertices whereas the dotted edges are connecting all vertices with the supersink. In addition, the capacity of each dashed edge $e = (S, v)$ from the supersource S to a vertex v is equal to the produced load $\texttt{load}(v)$ of the connected vertex. Furthermore, the capacity of each dotted edge $e = (v, T)$ from a vertex v to the supersink T is equal to the sum of the capacities of all incident edges of vertex v, therefore, $\texttt{cap}((v, T)) = \sum_{u \in \mathcal{V} \setminus \{S,T\}} \texttt{cap}(u, v)$.

The first integer linear programming formulation uses the following additional variable.

The binary variable $\texttt{used}((v, T))$ holds the information whether edge $e = (v, T)$, going from the vertex v to the supersink T, is used or not:

$$\texttt{used}((v,T)) = \begin{cases} 1 & \text{if edge } e = (v,T) \in \mathcal{E} \text{ is used, and therefore } v \in \mathcal{V} \text{ is a sink. } (v \in \mathcal{F}) \\ 0 & \text{otherwise} \end{cases}$$


(a) Example graph with vertices and edges.



(b) Example graph with added supersource S and supersink T, and their corresponding edges.

Figure 3.1: Example graph and its extension with a supersource and supersink.

The incoming flow has to be equal to the outgoing flow for all vertices $v \in \mathcal{V}$ except the supersource S and supersink T.

$$\sum_{u\in\mathcal{V}} \texttt{cur}((u,v)) = \sum_{w\in\mathcal{V}} \texttt{cur}((v,w)) \qquad \forall v \in \mathcal{V} \setminus \{S,T\}$$

The correct number of k sinks has to be placed. The vertices v connected to an edge $e = (v,T)$ with value $\texttt{used}((v,T)) = 1$ are the sinks in the final solution.

$$\sum_{v\in\mathcal{V}} \texttt{used}((v,T)) = k$$

The edge capacities $\texttt{cap}((v,u))$, $v,u \in \mathcal{V}$ can not be exceeded. The edges $e = (v,T)$ going from a vertex v to the supersink T are a special case. These edges can only be used if $\texttt{used}((v,T)) = 1$.

$$\begin{aligned} \texttt{cur}((v,u)) &\leq \texttt{cap}((v,u)) && \forall e = v,u \in \mathcal{E} \setminus \{e = (v,T)\} \\ \texttt{cur}((v,T)) &\leq \texttt{cap}((v,T)) * \texttt{used}((v,T)) && \forall e = (v,T) \in \mathcal{E} \end{aligned}$$

Finally, the following constraints have to be considered.

$$\texttt{cur}((v,u)) \geq 0 \qquad \forall e = (v,u) \in \mathcal{E}$$

The presented formulation is correct. Two artificial vertices are added to the network, representing a supersource S and a supersink T. The supersource is connected to all vertices by edges with a capacity equal to the produced load $\texttt{load}(v)$ of the corresponding vertex v. All vertices are connected to the supersink by edges with a capacity equal to the sum of the capacities of all incoming edges of the corresponding vertex. However, this edge can only be used if this vertex v is a sink ($\texttt{used}((v,T)) = 1$). In this network, the incoming flow is equal to the outgoing flow for each vertex, the capacities are not exceeded. The objective function seeks a maximum flow from the supersource to the supersink. The k edges (connecting a vertex v with the supersink T) maximizing the objective function are chosen. The objective function is presented at the end of the next section, 3.2.

Integer Linear Programming Formulation 2

In the second integer linear programming formulation no additional vertices like the supersink and supersource are introduced. However, big M and N, which are large integer values, and following additional variables are used.

The binary variable $\texttt{sink}(v)$ holds the information whether vertex v is a sink or not:

$$\texttt{sink}(v) = \begin{cases} 1 & \text{if vertex } v \in \mathcal{V} \text{ is a sink, therefore } v \in \mathcal{F} \\ 0 & \text{otherwise} \end{cases}$$

The variable $\texttt{uload}(v)$ holds the percentage of how much of the produced load $\texttt{load}(v)$ of vertex v can be processed. Therefore, $0 \leq \texttt{uload}(v) \leq 1$.

The variable $\texttt{dir}((v,u))$ holds the information about the flow direction of an edge $e = (v,u)$:

$$\texttt{dir}((v,u)) = \begin{cases} 1 & \text{if the load flows from vertex } v \text{ to vertex } u \\ 0 & \text{if the load flows from vertex } u \text{ to vertex } v \end{cases}$$

The following equations ensure that the data flow is handled correctly and given constraints are not exceeded.

The correct number of k sinks has to be placed.

$$\sum_{v \in \mathcal{V}} \texttt{sink}(v) = k$$

The edge capacities $\texttt{cap}((v,u))$ can not be exceeded.

$$\texttt{cur}((v,u)) \leq \texttt{cap}((v,u)) \qquad \forall e = v, u \in \mathcal{E}$$

The incoming flow has to be equal to the outgoing flow. Sinks are an exception where the outgoing flow has to be 0, to ensure this behaviour a big M formulation is used.

$$\sum_{u \in \mathcal{V}} \texttt{cur}((u,v)) + \texttt{uload}(v)\texttt{load}(v) \leq \texttt{sink}(v)M + \sum_{w \in \mathcal{V}} \texttt{cur}((v,w)) \qquad \forall v \in \mathcal{V}$$

$$\sum_{u \in \mathcal{V}} \texttt{cur}((u,v)) + \texttt{uload}(v)\texttt{load}(v) \geq (-1)\texttt{sink}(v)M + \sum_{w \in \mathcal{V}} \texttt{cur}((v,w)) \qquad \forall v \in \mathcal{V}$$

$$\sum_{u \in \mathcal{V}} \texttt{cur}((v,u)) \leq (1 - \texttt{sink}(v))M \qquad \forall v \in \mathcal{V}$$

The load on each edge $e = (v, u)$ can only flow in one direction.

$$\begin{aligned} \texttt{cur}((v,u)) &\leq \texttt{dir}((v,u))N && \forall e = (v,u) \in \mathcal{E} \\ \texttt{cur}((u,v)) &\leq \texttt{dir}((u,v))N && \forall e = ((u,v)) \in \mathcal{E} \\ \texttt{dir}(v,u) + \texttt{dir}(u,v) &= 1 && \forall e = (v,u), c = (u,v) \in \mathcal{E} \end{aligned}$$

Finally, the following constraints have to be considered.

$$\texttt{cur}((v,u)) \geq 0 \qquad \forall e = (v,u) \in \mathcal{E}$$

Objective Function

The objective function is a sum maximization of five sums, each weighted by constant factors ($\omega_1^{obj}...\omega_5^{obj}$). The five terms are:

1. First, the total cost `cost`(v) of all used sinks v is minimized, and normalized by the `topX` value, which determines the cost of the top x most expensive sinks. In this case x is always equal to the number of placed sinks k.

2. Second, the accruing load `load`(v) of each vertex v is satisfied. For each vertex the difference of the outgoing flow and the incoming flow, normalized by the accrued load of vertex v, is calculated. Only the edges from the original network are considered, therefore without edges to the artificial supersource S and supersink T.

3. Third, the accruing load of the vertices is satisfied as in the second term normalized by the customer potential of this vertex, therefore, the term is multiplied with `pot`(v).

4. Fourth, this term is equal to the third term, however, instead of the customer potential the customer value `val`(v) is considered.

5. Fifth, the usage of an edge $e = (v, u)$ is determined by its current load `cur`(v, u) normalized by the capacity `cap`(v, u) of this edge, and should be minimized for all edges. This sum of the usage of all vertices is normalized by the amount of edges $|\mathcal{V}|$.

Each term is always between (including) 0 and 1 before the multiplication with the weighting constants.

Maximize:

$$
\begin{aligned}
&\omega_1^{obj}(1 - \frac{\sum_v \mathtt{sink}(v)\mathtt{cost}(v)}{\mathtt{topX}}) \\
&+ \omega_2^{obj} \sum_v \frac{(\sum_{u \neq v} \mathtt{cur}((v,u)) - \sum_{w \neq v} \mathtt{cur}((w,v)))}{\mathtt{load}(v)} \\
&+ \omega_3^{obj} \sum_v \frac{(\sum_{u \neq v} \mathtt{cur}((v,u)) - \sum_{w \neq v} \mathtt{cur}((w,v)))\mathtt{pot}(v)}{\mathtt{load}(v)} \\
&+ \omega_4^{obj} \sum_v \frac{(\sum_{u \neq v} \mathtt{cur}((v,u)) - \sum_{w \neq v} \mathtt{cur}((w,v)))\mathtt{val}(v)}{\mathtt{load}(v)} \\
&+ \omega_5^{obj}(1 - \frac{\sum_v \sum_{u \neq v}(\frac{\mathtt{cur}((v,u))}{\mathtt{cap}((v,u))})}{|\mathcal{E}|})
\end{aligned}
\tag{3.1}
$$

3.2.1 Linear Programming Formulation Relaxation

The linear programming relaxation of the presented 0-1 integer program is the problem that arises by replacing the constraints that each variable must be 0 or 1 by a weaker constraint, that each variable belongs to the interval between 0 and 1. In the presented linear programming relaxation the variable $\mathtt{sink}(v)$, which defines whether vertex v becomes a sink, gets relaxed. Instead of the strict enforcement $\mathtt{used}((v,T)) \in \{0,1\}$, the variable $\mathtt{used}((v,T))$ can take values in the interval $0 \leq \mathtt{used}((v,T)) \leq 1$. Therefore, the relaxed integer program formulation consists of the following equations, based on the formulation 1 in section 3.2.

$$\sum_{u \in \mathcal{V}} \mathtt{cur}((u,v)) = \sum_{w \in \mathcal{V}} \mathtt{cur}((v,w)) \qquad \forall v \in \mathcal{V} \setminus \{S,T\} \tag{3.2}$$

$$\sum_{v \in \mathcal{V}} \mathtt{used}((v,T)) = k \tag{3.3}$$

$$\mathtt{cur}((v,u)) \leq \mathtt{cap}((v,u)) \qquad \forall e = (v,u) \in \mathcal{E} \setminus \{e = (v,T)\} \tag{3.4}$$

$$\mathtt{cur}((v,T)) \leq \mathtt{cap}((v,T)) * \mathtt{used}((v,T)) \qquad \forall e = (v,T) \in \mathcal{E} \tag{3.5}$$

$$0 \leq \mathtt{used}(v,T) \leq 1 \tag{3.6}$$

In order to ensure the correct termination of the formulation the capacity $\mathtt{cap}((v,T))$ of the edges between each vertex v_i and the supersink T have to be updated. Using positive infinite, will yield unusable results, since the produced load $\mathtt{load}(v)$ of each vertex v gets immediately satisfied with every value $\mathtt{sink}(v) > 0$. In order to prevent this behaviour the capacity $\mathtt{cap}((v,T))$ gets changed to the sum of the capacity of all connecting edges, exclusive the edge connecting v to the supersink T.

$$\texttt{cap}((v,T)) = \sum_{u \in \mathcal{V}} \texttt{cap}((u,v)) \qquad \forall v \in \mathcal{V} \tag{3.7}$$

Therefore the maximum flow which can flow from v to the supersink T equals the maximum flow from all vertices u to v. In this case the relaxed $\texttt{sink}(v)$ variable indicates how much capacity of the connecting edges is used to achieve an optimal solution. The $\texttt{sink}(v)$ values of the relaxed solution can be used as probability that v is upgraded to a sink in the integer solution. The algorithm of the production of the linear programming relaxation solution and the rounding to a correct integer solution is described in the pseudo code in 3.1. A simple extension of the algorithm is that the sinks are placed using the probabilities given by the relaxed $\texttt{sink}(v)$ values. This process is repeated a defined amount of time and the best solution is taken.

Algorithm 3.1: LP Relaxation Rounding

```
Input: A graph, and a number $k$ of sinks to set
Output: A graph with $k$ new sinks
1 calculate LP Relaxation solution;
2 while not repeated a specific amount do
3     round to integer solution using the relaxed solution;
4     store as best solution if score is better than current best;
5 end
6 return best solution
```

3.3 Greedy Heuristic

A greedy heuristic selects the next sinks sequentially, which are chosen based on a sink-choosing function Z. This procedure continues until a feasible solution has been created and no more sinks can be placed. The basic idea for finding a solution is to chose the next station on seven specific properties; The customer value $\texttt{val}(v)$, the customer potential $\texttt{pot}(v)$, the average utilization of the outgoing edges, the average capacity of the outgoing edges normalized using the maximum capacity $C = \texttt{max}(\texttt{cap}(e))$ of all edges e, the degree $\texttt{deg}(v)$ of the current vertex normalized by the maximum degree of the graph $\triangle(G)$, the produced data load $\texttt{load}(v)$ of the current vertex normalized by the maximum produced data load of the graph $L = \texttt{max}(\texttt{load}(v))$, the cost to upgrade $\texttt{cost}(v)$ the current vertex normalized by the maximum cost to upgrade of the graph, $O = \texttt{max}(\texttt{cost}(v))$. Before each calculation of the Z values, the maximum flow of the network must be calculated to obtain the current correct $\texttt{cur}(e)$ values for each edge e.

$$Z(v) = \omega_1^{greedy}\texttt{val}(v) + \omega_2^{greedy}\texttt{pot}(v) + \omega_3^{greedy}\frac{\sum_{u\in\mathcal{V}}\frac{\texttt{cur}((v,u))}{\texttt{cap}((v,u))}}{|\mathcal{E}|} + \omega_4^{greedy}\sum_{u\in\mathcal{V}}\frac{\texttt{cap}((v,u))}{C}$$
$$+ \omega_5^{greedy}\frac{\texttt{deg}(v)}{\triangle(G)} + \omega_6^{greedy}\frac{\texttt{load}(v)}{L} + \omega_7^{greedy}(1-\frac{\texttt{cost}(v)}{O}) \tag{3.8}$$

The Z values are calculated for all vertices that are not sinks already, and the vertex v with the maximum $Z(v)$ value is chosen to become the next sink. Then the maximum flow of the network is calculated to update the $\texttt{cur}(e)$ values for each edge e. This procedure is repeated until all k sinks are placed. Furthermore, in order to improve the results, the vertices of the top x Z values can be consecutively upgraded to sinks, and the vertex with the best performance according to the objective function is picked. A pseudo code example is given with algorithm 3.2.

Algorithm 3.2: Greedy-Heuristic

```
   Input: A graph, and a number $k$ of sinks to set
   Output: A graph with $k$ new sinks
1  while not all sinks set do
2      for all vertices which are not sinks do
3          calculate the $Z$ value for the current vertex;
4      end
5      check the top $x$ $Z$ value vertices;
6      pick the vertex with the best performances according to the objective function;
7      upgrade this vertex to a sink;
8      calculate/update the max flow of the new graph;
9  end
10 return graph with $k$ sinks
```

The greedy heuristic can be randomized, instead of picking vertices of the top x Z values one of the top $y\%$ candidates according to the sink-choosing function Z is chosen at random at each iteration. A pseudo code example is given with algorithm 3.3.

The (random) greedy algorithm has a running time of $\mathcal{O}(k(V - \frac{k-1}{2} + \texttt{maxflow}))$, for each of the k sinks to place all remaining non-sink vertices are checked and their Z value is calculated. After each iteration the number of available non-sink vertices gets reduced by 1, and therefore, the number of vertices which need to be checked is also reduced by 1. In addition, the current max flow, with a running time of $\texttt{maxflow}$, is calculated after each iteration. In chapter 5 two used algorithms and methods for calculating the maximum flow in undirected networks are presented and discussed. The faster method, based on Dinic's algorithm ([Din70]), has a running time of $\mathcal{O}(V^2E)$, resulting in a running time of $\mathcal{O}(k(V - \frac{k-1}{2} + V^2E))$ for the (random) greedy algorithm. Since, calculating the maximum flow is the dominating part, $\mathcal{O}(k(V^2E))$ is the final running time.

Algorithm 3.3: Random Greedy-Heuristic

```
Input: A graph, and a number $k$ of sinks to set
Output: A graph with $k$ new sinks
1 while not all sinks set do
2     for all vertices which are not sinks do
3         calculate the $Z$ value for the current vertex;
4     end
5     pick a random vertex of the top $y\%$ $Z$ value vertices;
6     upgrade this vertex to a sink;
7     calculate/update the max flow of the new graph;
8 end
9 return graph with $k$ sinks
```

3.4 GRASP

The greedy randomized adaptive search procedure (GRASP) is a metaheuristic, consisting of m iterations made up from successive constructions of a greedy randomized solution and subsequent iterative improvements of it through a local search. The greedy heuristic is based on the sink-choosing function Z presented in 3.3. In order to obtain variability in the candidate set of greedy solutions one of the top $x\%$ candidates according to the sink-choosing function Z is chosen at random at each iteration when building up the initial solution. The initial solutions are further improved using a local search technique. In detail, the Exchange Sinks Random method presented in 3.9, which exchanges one used sink with a random vertex that is not used as sink in the current solution. Each solution that provides an improvement according to the objective function is picked, and the process is repeated until no further improvements are found. A pseudo code example is given with algorithm 3.4.

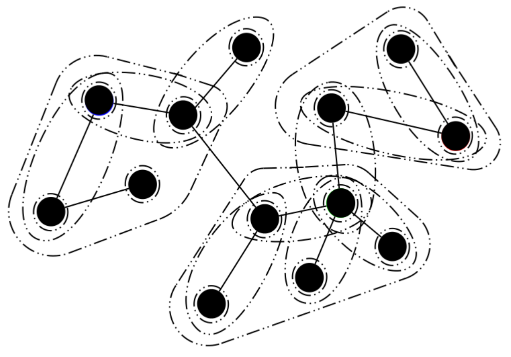
Algorithm 3.4: GRASP

```
Input: A graph, and a number $k$ of sinks to set
Output: A graph with $k$ new sinks
1 while not all $m$ iterations done do
2     calculate current greedy solution;
3     improve the current solution using a local search technique;
4     store as best solution if score is better than current best;
5 end
6 return best solution
```

The running time and explanation of the random greedy heuristic is presented in 3.3, and is $\mathcal{O}(k(V-\frac{k-1}{2}+\texttt{maxflow}))$, where `maxflow` is the running time of a maximum flow algorithm. Performing the Exchange Sinks Random method needs $\mathcal{O}(kV)$ to finish. Therefore, the running time of the GRASP algorithm is $\mathcal{O}(m(k(V-\frac{k-1}{2})+kV+k\,\texttt{maxflow}))$.

3.5 Greedy-Net

Greedy-Net, called net because the sinks are equally distributed over all vertices like a net, is an alternative greedy heuristic which also selects the sinks sequentially. Sinks are chosen depending on their potential maximum influence on the graph score, and their distance to other sinks. In general, the problem is transformed to a maximum coverage problem, where several sets are given. The set may have some elements in common, and at most k sets have to be picked such that the maximum number of elements are covered. First, the sets have to be defined. Each vertex v_i gets assigned several other neighbouring vertices. Vertex v_j gets assigned to vertex v_i, the main vertex, if the accrued (produced) load $\texttt{load}(v_j)$ of vertex v_j can be fully satisfied by a possible sink at vertex v_i. Fully satisfied means that, for at least one path from v_i to v_j (with the denotion $v_i, v_{i+1}, ..., v_{i+n}, v_j$ for the intermediate steps) the equation $\texttt{cap}(v_{j-m-1}, v_{j-m}) \geq \sum_0^m \texttt{load}(v_{j-m}) \forall 0 \leq m < n$ is valid. In other words, the sum of the accrued load of all vertices along the path from v_j to v_i can be satisfied, without violating any capacity constraints. Therefore, maximum $|\mathcal{V}|^2$ sets are defined. However, the number of sets can be reduced, such that each set which is fully contained within another set is removed. The remaining sets are denoted as $\mathcal{V}_a$, with a indicating the amount of sets. Figure 3.2a illustrates all found sets (note that each vertex on its own is already a set) and figure 3.2b gives the final sets where each set which is fully contained within another set is removed. Second, the accrued load of each vertex v gets weighted using the corresponding customer value $\texttt{val}(v)$, customer potential $\texttt{pot}(v)$, and upgrade cost $\texttt{cost}(v)$. Then the greedy algorithm at each stage chooses a, not already picked, set that contains the maximum weight of uncovered vertices.


(a) Example graph with all sets.

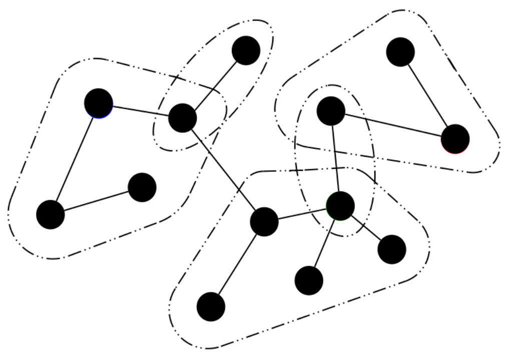

(b) Example graph with the final sets.

Figure 3.2: Example graph with all found sets, and the final remaining sets after all sets which were fully contained within another set were removed.

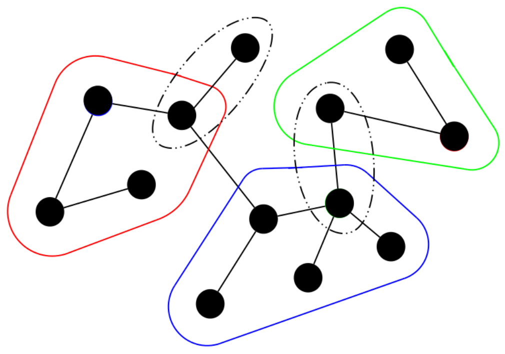
At each stage the set score $\texttt{setscore}(\mathcal{V}_a)$ of each set $\mathcal{V}_a$ is defined as the sum of the $\texttt{setscore}(v)$ of each vertex v contained in $\mathcal{V}_a$:

$$\texttt{setscore}(\mathcal{V}_a) = \sum_{v \in \mathcal{V}_a} \texttt{setscore}(v) \qquad \forall \mathcal{V}_a, a > 0$$

Whereas the $\texttt{setscore}(v)$ of a vertex v is 0 if the vertex v is already in a picked set, otherwise it is a weighted value of the corresponding customer value $\texttt{val}(v)$, customer potential $\texttt{pot}(v)$, and upgrade cost $\texttt{cost}(v)$, using the weights ω^{obj} of the objective function, indicating the importance of this vertex:

$$\texttt{setscore}(v) = \begin{cases} 0 & \text{if any set containing vertex } v \text{ has already been chosen, } v \text{ is covered} \\ \omega_3^{obj}\texttt{pot}(v) + \omega_4^{obj}\texttt{val}(v) - \omega_1^{obj}\frac{\texttt{cost}(v)}{\texttt{max}(\texttt{cost}(v))} & \text{else} \end{cases}$$

The set $\mathcal{V}_a$ with the highest set score $\texttt{setscore}(\mathcal{V}_a)$ is taken at each iteration. One advantage of this approach is that no maximum flow needs to be calculated for each intermediate step after a new sink was placed. This significantly speeds up the calculation time. Figure 3.3 shows two times the same example graph with three different picked sets each time.


(a) Example graph with picked sets.

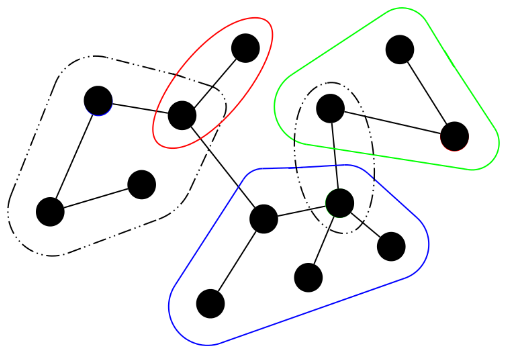

(b) Example graph with different picked sets.

Figure 3.3: Example graphs where three sets were picked.

The Greedy-Net algorithm has a running time of $\mathcal{O}(V^2 + k(V^2 - V\frac{k-1}{2}))$, for each vertex a set gets calculated, where, in the worst case, all vertices are checked, in total V^2. For the k sinks to place all remaining non-chosen sets are checked ($\mathcal{O}(V - \frac{k-1}{2})$) and their current scores are calculated needing V in the worst case. After each iteration the number of available non-chosen sets gets reduced by 1, and therefore, the number of sets which need to be checked is also reduced by 1. The final running time would be therefore $\mathcal{O}(V^2)$.

3.6 Clustering

The presented clustering method is inspired by k-medoids clustering, which aims to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. However, instead of using the Manhattan or Euclidean distance between two vertices, the shortest path with a specific distance measure is used. In detail, to determine the distance between two connected vertices v to u following asymmetric distance measure is used.

Algorithm 3.5: Greedy-Net-Heuristic

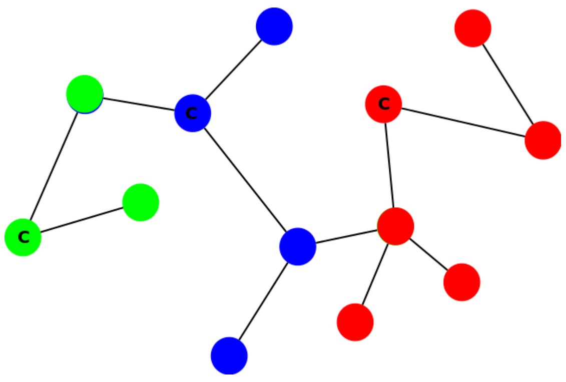
```
   Input: A graph, and a number $k$ of sinks to set
   Output: A graph with $k$ new sinks
 1 calculate the sets, one for each vertex;
 2 calculate the score for each vertex;
 3 while not all sinks set do
 4     for all sets do
 5         calculate the set score for the current set;
 6         store the best set;
 7     end
 8     mark all vertices of the best set as covered;
 9     upgrade the main vertex of the best set to a sink;
10 end
11 calculate the max flow of the new graph to obtain the correct objective function
     values;
12 return graph with k sinks
```

$$d_{v,u} = \frac{\texttt{deg}(v)}{\texttt{cap}((v,u))\texttt{load}(u)(1-\texttt{val}(u))(1-\texttt{pot}(u))} \tag{3.9}$$

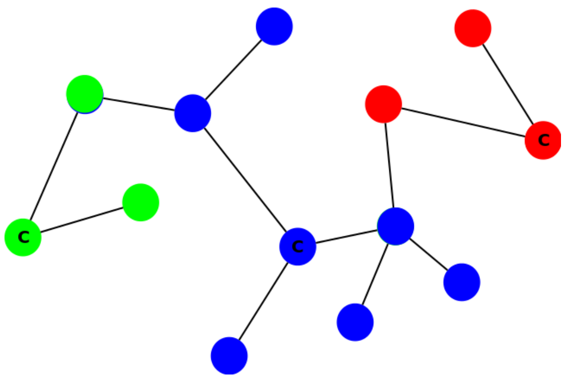
The distance between two vertices from v to u is mainly defined by the capacity $\texttt{cap}(e), e = (v,u)$ of the connecting edge and the produced load $\texttt{load}(u)$ by vertex u. A high $\texttt{cap}(e)$ together with a low $\texttt{load}(u)$ indicates a lower distance, whereas a low $\texttt{cap}(e)$ together with a high $\texttt{load}(u)$ indicates the opposite. For the distance measure the $\texttt{load}(u)$ gets fine tuned depending on the customer value $\texttt{val}(u)$ and potential $\texttt{pot}(u)$, since higher values suggest more important vertices, therefore, the produced load by such vertices gets reduced by the same amount for the distance measure. Furthermore, at vertices with a high degree $\texttt{deg}(u)$ the load can be split between different connections.

At first k random vertices are picked as initial cluster centres, or an initial random greedy solution is generated with the sinks as initial cluster centres. Then all pairs shortest paths of the graph are calculated, using the Floyd-Warshall or Johnson-Dijkstra algorithm. Furthermore, all vertices are assigned to one of the k chosen centres with the shortest path distance. These vertices form a cluster. Within these clusters, the average distance from one vertex to all other vertices is calculated using the previously computed shortest path distances. The vertex with the lowest average distance becomes the new cluster centre. This loop is repeated x times or until no more changes of the centre occur.

Figure 3.4 presents two example iterations of the clustering algorithm. In figure 3.4a the first iteration is given and in figure figure 3.4b the second iteration, where three vertices of the red cluster became members of the blue cluster. Furthermore, their cluster centre, denoted with **C** changed.



(a) First iteration.



(b) Second iteration with changed clusters.

Figure 3.4: Two example iterations, where identically coloured vertices represent clusters and vertices marked with **C** the corresponding cluster centre. In the second iteration two cluster centres and their members changed.

It is possible to randomize the process in such a way that each vertex does not get assigned to the nearest centre, but to each centre with a specific probability dependent on the shortest distance to all centres.

$$p_{v,w} = \frac{d_{v,k}}{\sum_{u \in \texttt{centres}} d_{v,u}} \qquad w \in \texttt{centres} \tag{3.10}$$

The probability $p_{v,w}$ that vertex v gets assigned to centre w equals the share of $d_{v,w}$ in the sum of all distances from v to all centres. In this case the loop is repeated a certain amount of time.

Algorithm 3.6: Clustering

```
   Input: A graph, and a number $k$ of sinks to set
   Output: A graph with $k$ new sinks
1  calculate all pairwise shortest paths;
2  calculate an initial solution, and use the sinks as cluster centres;
3  while clusters changed and not maximum iterations reached do
4      assign each vertex to the nearest cluster center;
5      calculate new cluster centers, using the average distance from one vertex to all
         other vertices;
6  end
7  return latest solution
```

The Floyd-Warshall algorithm has a running time of $\mathcal{O}(V^3)$, assigning all vertices to a cluster is done in $\mathcal{O}(V)$, and calculating the average distances within a cluster needs

Algorithm 3.7: Clustering Random

```
  Input: A graph, and a number $k$ of sinks to set
  Output: A graph with $k$ new sinks
1 calculate all pairwise shortest paths;
2 calculate an initial solution, with the sinks as cluster centers;
3 while clusters changed and not maximum iterations reached do
4     assign each vertex to a cluster center with a probability dependent on the
        distance;
5     calculate new cluster centres, using the average distance from one vertex to all
        other vertices;
6 end
7 return latest solution
```

$\mathcal{O}(V^2)$ time. The running time of the clustering algorithm is therefore $\mathcal{O}(V^3 + xV^2 + xV)$, with x as the maximum number of iterations. It is clearly visible that the running time is dominated by the all pairs shortest paths algorithm while $x < V$ resulting in $\mathcal{O}(V^3)$, with $x \geq V$ the running time is $\mathcal{O}(xV^2)$.

3.7 Post Optimization

In order to improve existing solutions, two simple local search algorithms are presented. First, the Exchange-Sinks-Depending-on-Z algorithm in 3.7.1, and second, the Exchange Sinks Random algorithm in 3.7.2 which is a random version of the first algorithm.

3.7.1 Exchange Sinks Depending on Z

The Exchange-Sinks-Depending-on-Z algorithm exchanges an existing, chosen sink with another vertex depending on their sink-choosing function Z value presented in 3.3. All placed sinks (or a subset of all sinks) are iterated. At each step the current sink gets removed and all Z values of the remaining vertices are calculated. The removed sink gets replaced by the top Z value vertex. The data flow is updated and the new graph score calculated. If the new solution is better performing the changes retain, otherwise the previous solution is restored. A pseudo code is given in 3.8.

The local search has a running time of $\mathcal{O}(V^2)$. For each sink all Z values of all vertices are calculated.

3.7.2 Exchange Sinks Random

The Exchange-Sinks-Random algorithm exchanges an existing, chosen sink with another random vertex. All placed sinks (or a subset of all sinks) are iterated. At each step the current sink gets removed, and replaced by a random vertex. The data flow is updated and the new graph score calculated. If the new solution is better performing the changes

Algorithm 3.8: Exchange-Sinks-Depending-on-Z

```
Input: A graph, an initial solution
Output: A copy of the graph with equal or better score
1  for all sinks do
2      remove current sink;
3      calculated all $Z$ values for all vertices;
4      replace the removed sink with the highest $Z$ value vertex;
5      calculate the new graph score;
6      if the new score is higher than the old score then
7          keep the current solution;
8      else
9          restore previous solution;
10     end
11 end
12 return latest solution
```

retain, otherwise the previous solution is restored. For each sink r random vertices are tested. A pseudo code is given in 3.9.

The local search has a running time of $\mathcal{O}(Vr)$. For each sink all r random vertices are tested.

Algorithm 3.9: Exchange-Sinks-Random

```
Input: A graph, an initial solution
Output: A copy of the graph with equal or better score
1  for all sinks do
2      remove current sink;
3      for $r$ times do
4          replace the removed sink with a random vertex;
5          calculate the new graph score;
6          if the new score is higher than the old score then
7              keep the current solution;
8          else
9              restore previous solution;
10         end
11     end
12 end
13 return latest solution
```

CHAPTER 4

An Exact Algorithm for Trees

In this chapter an exact algorithm for solving the multiple sink placing problem in undirected tree graphs is introduced. First, a formal description and a mathematical formulation is presented. Then the exact algorithm with an according proof is explained.

4.1 Background and Formal Problem Description

In the sections 1.1 and 1.2 general introductions and assumptions were given, followed by a formal problem description in section 3.1. In this chapter a related simpler problem is introduced, where the problem is to find the minimum amount of needed sinks, such that the whole accrued (produced) load `load`(v) of all vertices v is satisfied without violating any capacity constraints. Therefore, upgrade cost `cost`(v), customer value `val`(v), and customer potential `pot`(v) of each vertex v are not considered, and a number k of sinks to place is not needed. The notations are the same as in section 3.1.

4.2 Integer Linear Programming Formulation

This formulation, is similar to the formulation presented in 3.2, the existing network is extended by two further artificial vertices. A supersink T, which will be the only sink in the network, and a supersource S, which will be the only source in the network. In addition, further edges are introduced connecting the supersink and the supersource with all vertices respectively. Figure 3.1 illustrates the changes for a given network. In 3.1b the vertices S and T correspond to the supersource and supersink respectively. The dashed edges are connecting the supersource with all vertices whereas the dotted edges are connecting all vertices with the supersink. In addition, the capacity of each dashed edge $e = (S, v)$ from the supersource S to a vertex v is equal to the produced load `load`(v) of the connected vertex. Furthermore, the capacity of each dotted edge $e = (v, T)$ from a vertex v to the supersink T is equal to the sum of the capacity all incident edges of vertex

v, therefore, $\texttt{cap}((v,T)) = \sum_{u \in \mathcal{V} \setminus \{s,T\}} \texttt{cap}((v,u))$. The variable $\texttt{used}((v,T))$ holds the information whether edge $e = (v,T)$, going from the vertex v to the supersink T, is used or not.

The only main difference is the objective function, which is to minimize the number of placed sinks. Therefore, the sum of all $\texttt{used}((v,T))$ is minimized.

Minimize:

$$\sum_{v\,\in \mathcal{V}} \texttt{used}((v,T)) \tag{4.1}$$

4.3 Greedy Tree Solver

The Tree Solver is a special algorithm especially suitable for solving trees, however can be used to solve any graph. The method produces an optimal solution for trees, and valid solutions for non-tree graphs. In contrast to other presented algorithms the Tree Solver uses a different target function. Given is an input graph, and the Tree Solver produces a solution where all vertices are satisfied with the minimal amount of sinks set. In contrary to other methods, customer value, customer potential, and upgrade cost of each vertex are not considered and needed to produce the solution. Hence, the Tree Solver calculates a solution where all vertices are satisfied, and the number of placed sinks is minimal. Sinks are solely placed in regard of their ability to satisfy vertices, without focusing on customer value, customer potential, and upgrade cost. Since the algorithm places as much sinks as needed to satisfy all vertices no input parameter k, the amount of sinks to place, is used.

All vertices are iterated in a DFS (Depth First Search) postordering, this means the algorithm always starts at a leaf vertex with degree 1, and every currently processed vertex is connected to exactly one parent and one or more already processed vertices (see figure 4.1a). The data load which needs to be processed for each vertex is defined as $\texttt{processLoad}(v)$, $\texttt{processLoad}(v)$ must be 0 for already processed vertices, and $\texttt{processLoad}(v)$ is initialized with 0. At each iteration three steps are performed. First, in step one, the load to process $\texttt{processLoad}(v)$ of vertex v is updated, $\texttt{processLoad}(v) = \texttt{processLoad}(v) + \texttt{load}(v)$. Then as much load as possible is sent back to sinks within the already visited/processed vertices. Therefore, all paths to sinks within the visited vertices are checked. In order to decrease the running time the maximum load which can be send to a sink over a particular visited vertex can be stored and updated at each iteration. In this case only the visited neighbours of the current vertex need to be checked. In detail, at the current iteration, the maximum possible flow send able from vertex v to a sink minus the amount of flow which was actually send during the current iteration is stored as $\texttt{sendable}(v)$. During the next iteration the neighbouring vertex u only needs to check the maximum possible flow send able to v, $\texttt{min}(\texttt{sendable}(v), \texttt{cap}(u,v))$, instead of checking all paths, and calculates the same value $\texttt{sendable}(u)$ for itself in the same way. This method can be safely used since the only path, if any exists, to an already visited sink would be over the current vertex, otherwise it would not be a tree. Therefore, checking the value of the neighbouring vertices suffices. If the whole load to process, $\texttt{processLoad}(v)$, of the current vertex v is satisfied the algorithm proceeds to the next vertex u. Figure 4.1b illustrates this step, with the green framed circle representing sinks, and the red arrows the vertices which were checked. Second, in step two, if the remaining data load to process, $\texttt{processLoad}(v) = \texttt{processLoad}(v) - \texttt{loadProcessedInStepOne}$, is greater than 0, the algorithm checks if $\texttt{processLoad}(v)$ can be forwarded to the parent vertex w, if $\texttt{processLoad}(v) \leq \texttt{cap}((v, w))$. In this case, the data load of the parent vertex is updated, $\texttt{processLoad}(w) = \texttt{processLoad}(w) + \texttt{processLoad}(v)$, and the next vertex w is processed. See figure 4.1c for an illustration of this step. Third, in step 3, if the load to process $\texttt{processLoad}(v)$ can not be fully forwarded to the parent vertex, v must be upgraded to a sink, and the algorithm proceeds to the next vertex to process (see figure 4.1d). The algorithm terminates after all vertices are iterated, or the maximum amount of sinks was placed. In the latter case an optimal solution is not guaranteed. Furthermore, if more sinks to place are available after the termination of the DFS iteration, the remaining sinks are placed at non-sink vertices with the maximum sum of the (weighted) customer value and customer potential. A pseudo code example of the basic situation, where no maximum amount of sinks allowed to place is specified, is given with algorithm 4.1. Furthermore, for this situation a proof is presented in section 4.4.

Algorithm 4.1: TreeSolver

```
Input: A tree T=(V,E) with loads and capacities
Output: A minimum set of sinks for T
1  init sendable($v$) = 0 for each vertex $v$;
2  for each vertex $v_i$ in DFS postordering do
3      processLoad($v_i$) = processLoad($v_i$) + load($v$);
4      send as much current data as possible back to already visited vertices/sinks;
5      (check sendable($v_j$) of each visited neighbouring vertex $v_j$ of $v_i$ and calculate
         processLoad($v_i$) = processLoad($v_i$) − min(sendable($v_j$), cap($v_i, v_j$)) until
         processLoad($v_i$) = 0 or all neighbours $v_j$ checked
6      sendable($v_i$) = sum of the maximum possible flow send able from vertex $v_i$ to
         each neighbour $v_j$ minus the amount of flow which was actually send);
7      if processLoad($v_i$) > 0 and edge capacity from current vertex $v_i$ to its parent
         $p(v_i) \geq$ processLoad($v_i$) then
8          forward the whole current data to parent vertex;
9          (processLoad($p(v_i)$) = processLoad($p(v_i)$) + processLoad($v_i$));
10     else
11         upgrade the current vertex $v_i$ to a sink;
12     end
13 end
14 return tree with satisfied vertices and minimum amount of sinks
```

The Greedy Tree Solver algorithm has a running time of $\mathcal{O}(V + E)$, determined by the DFS, each vertex is processed once, and at each iteration all the neighbours of the current vertex are checked.

The algorithm works the same for non-tree graphs, with two minor changes. First, the load, `sendable`(v) which can be send back over a specific vertex v to already placed sinks can not be stored for each vertex, since more than one path to each sink could exist. This means, at each iteration the paths have to be checked anew, resulting in a higher running time. Second, again since the graph is not a tree, more than one neighbour could be available for forwarding the load. In this case the algorithm checks if the sum of the edge capacity to all these neighbours is greater or equal than the remaining load to process, `processLoad`(v). If not the current vertex is again upgraded to a sink, otherwise the load to process `processLoad`(v) is equally distributed, in respect to their edge capacity, between all possible neighbours. For non-tree graphs the algorithm will not produce a guaranteed optimal solution.

4.4 Greedy Tree Solver Proof

The Greedy Tree Solver algorithm produces optimal solutions for trees. The proof of this is presented in this section.

Theorem 1. *Given is a tree T=(V,E) with n vertices, denoted as $v_1, ..., v_n$. Algorithm TreeSolver computes a valid solution (all vertices are satisfied) with a minimum number of sinks.*

Proof. The Theorem is proven using induction. Claim: at each iteration i of the algorithm, the first i vertices are satisfied and processed, and $i \leq n$. Furthermore, it holds that, for these i vertices the number of set sinks is minimal, and for these i vertices $v_1, ..., v_n$ no constraints are violated.

Base State $i = 0$, all invariants are valid, 0 vertices are satisfied and 0 sinks are set. The claim obviously holds for the initial state.

Inductive Step $i > 0$, $i - 1$ vertices are satisfied with a minimal number of set sinks and no constraints of these $i - 1$ vertices are violated.

Due to DFS search postorder, the connections of the currently treated vertex can be split in two groups as presented in Figure 4.2a. First, the group of already visited vertices, denoted by X, second, the group of not processed vertices Y, with the parent vertex $p(v_i)$ of vertex v_i as only important vertex and only connection of Y to vertex v_i. v_i is the single connection of X and Y. Furthermore, there is no other path from any vertex in X to any vertex in Y, other than the sole path over v_i, otherwise it would not be a tree, since if there would be a v_j, such that a path from any vertex in X to any vertex in Y over vertex v_j ($\{X\} \rightarrow v_j \rightarrow \{Y\}$ would be possible, then together with the path from any vertex in X to any vertex in Y over vertex v_i, the path $\{X\} \rightarrow v_i \rightarrow \{Y\} \rightarrow v_j \rightarrow \{X\}$ would form a cycle and the graph would not be a tree. See Figure 4.2b for an illustration.

There are three possibilities to process the load of the current vertex v_i during the inductive step, and satisfy (adding the current vertex to the group of vertices X) this vertex without violating any constraints and placing the minimum amount of needed sinks.

First, the flow is sent to sinks located in area X using augmenting paths (see Figure 4.1b). If the whole flow located at the current vertex v_i can be satisfied in this way, the current vertex can be added to X and i vertices are satisfied, no constraints are violated and the number of sinks is minimal, since no further sinks were placed. This would satisfy the claim. However, in the case that not all of the current load at the current vertex v_i can be handled by existing sinks, as much load as possible is sent over the augmenting paths. This would reduce the remaining load as much as possible, and since no other path could use the remaining edge capacities, this is a valid step.

Second, after step 1, if there is current load left, the only possibility to further send the load is to the parent vertex $p(v_i)$ in Y (see Figure 4.1c). If $\texttt{cap}((v_i, p(v_i)))$ is greater or equal to the remaining current load, the whole current load can be forwarded to the parent vertex. In this case vertex v_i is satisfied, no constraint violated, and the number of sinks is minimal, since no further sinks were placed and the amount of sinks in for the vertices in area X is minimal. This, again, satisfies the claim.

Third, after step 1 and 2, if there is load left, there is no further option to send the load. All possibilities, sending load to existing sinks, and forwarding load to the parent vertex were exhausted. In this case the current vertex must become a sink in order to satisfy the current load (see Figure 4.1d). In this case the number of sinks is minimal since all possibilities to satisfy the current vertex were checked, no constraints were violated, and the number of sinks in step $i-1$ is minimal. Again this satisfies the claim. The load sent from v_i to $p(v_i)$ in step 2 can be returned to v_i. However, this would not influence the final result (all vertices satisfied and minimum amount of sinks placed) and further computation of the algorithm, since the edge $(v_i, p(v_i))$ will not be used in the further computation steps.

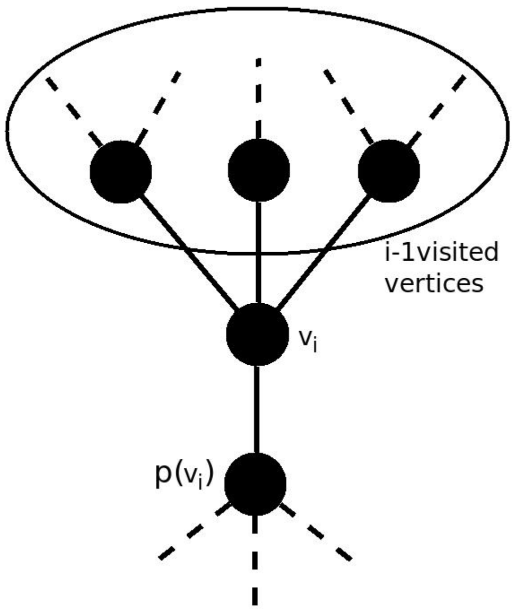
These three possibilities to process the load show that the maintenance state is always valid, and together with the valid initialization state the algorithm produces optimal solutions for trees.

□

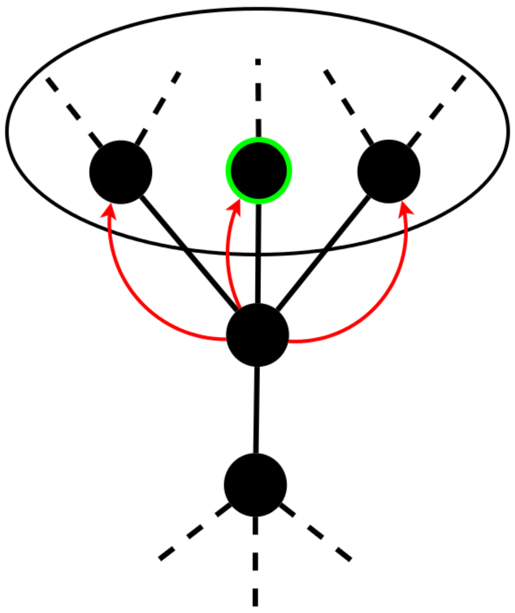
Theorem 2. *Given is a tree T=(V,E) with n vertices, denoted as $v_1, ..., v_n$. Algorithm TreeSolver has a running time of $\mathcal{O}(V + E)$.*

Proof. The tree is traversed using Depth-First-Search (DFS), which has a worst-case running time of $\mathcal{O}(V + E)$. At each iteration all the neighbours of the current vertex v_i are checked (implicitly done by DFS) and load of v_i is, if at all, sent to these neighbours (`sendable`() of visited neighbours is reduced, and/or `processLoad`() of the parent increased). This results in a running time determined by DFS, therefore, $\mathcal{O}(V+E)$.

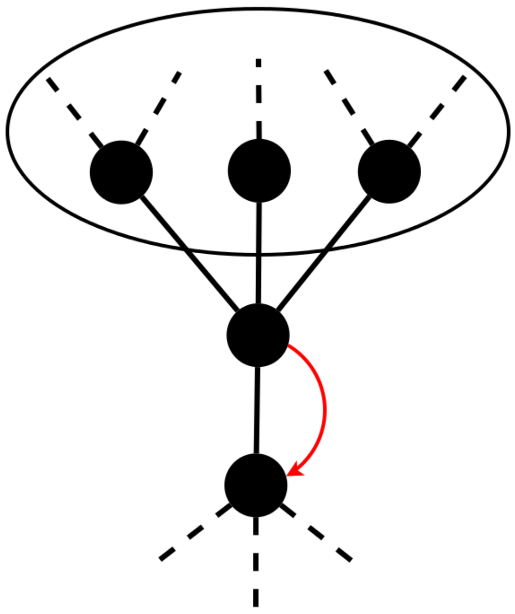
$\square$



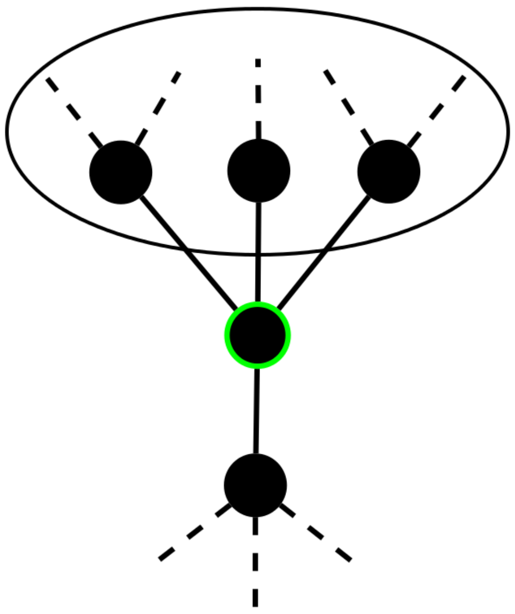
(a) Current vertex v_i, already visited vertices $(v_0, ..., v_{i-1})$, and the parent vertex $p(v_i)$ of v_i.



(b) Sending flow to already visited vertices $(v_0, ..., v_{i-1})$.

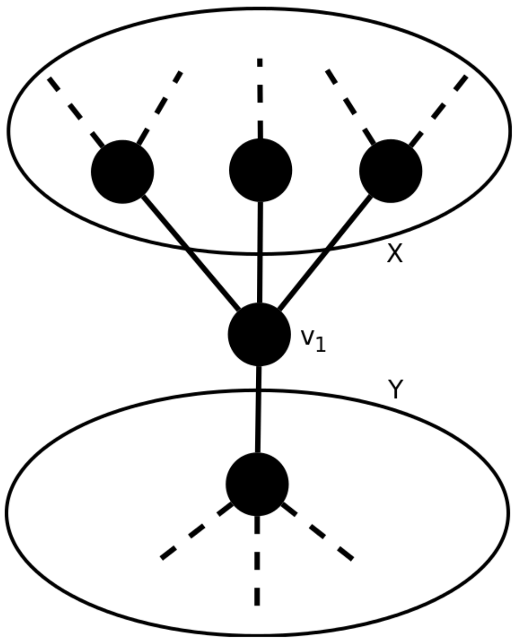


(c) Sending flow to the parent of vertex v_i.

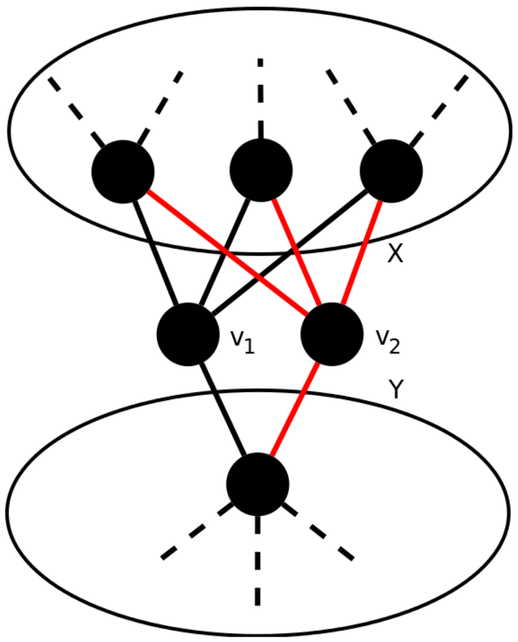


(d) Upgrading the current vertex v_i to a sink.

Figure 4.1: Possibilities how to handle the accruing (produced) data load. Green framed circles represent sinks, and red arrows show possible flow directions



(a) Valid tree, with already processed vertices X, unprocessed vertices Y, and the only connecting vertex $v1$.



(b) Invalid tree, with already processed vertices X, unprocessed vertices Y, and the connecting vertices $v1$ and $v2$.

Figure 4.2: Valid and invalid tree

CHAPTER 5

Maximum Flow Calculation

Calculating the maximum flow of a graph is an important, commonly reoccurring part of the presented algorithms. Therefore, the implementation and used method should be performant. Furthermore, since the used graphs are undirected the flow direction of each edge should be selected depending on their contribution to the maximum flow. In 5.1 a general method of calculating a maximum flow in an undirected network is presented, and in section 5.2 the used algorithms for the maximum flow calculation are introduced.

5.1 Maximum Flow in an Undirected Network

In order to obtain a maximum flow of an undirected graph, two algorithms are introduced which are based on the ideas presented in [SGJ04]. The algorithms use the concepts of the Ford–Fulkerson method, which calculates the maximum flow of a graph based on the computation of flow augmenting paths. These paths take an existing flow and construct a new flow that is greater than the original flow. The two used algorithms are equivalent to just applying the Ford-Fulkerson method to the directed graph obtained from original graph, where each edge got replaced by two directed arcs, one in each way. Figure 5.1 shows an example of an undirected graph and its corresponding directed graph. The Ford–Fulkerson method can be applied due to the fact that, considering the two arcs between a given pair of vertices, an arc is used in a flow, the other arc cannot be used, otherwise it would not satisfy the skew symmetry. In [SGJ04] a proof of correctness is presented.

Each graph edge is replaced by two arcs, as mentioned above, and then the algorithms are applied. The two algorithms are similar to the Edmonds–Karp (see [EK72]) algorithm and Dinic's algorithm (see [Din70]). The sole difference is that each time the flow between vertex u and vertex v is increased by an amount Δ, the flow between vertex v and vertex u is decreased by Δ. Instead of updating the residual graph, the reverse arc is updated. In detail, this property is called the skew symmetry property, and says that the net flow

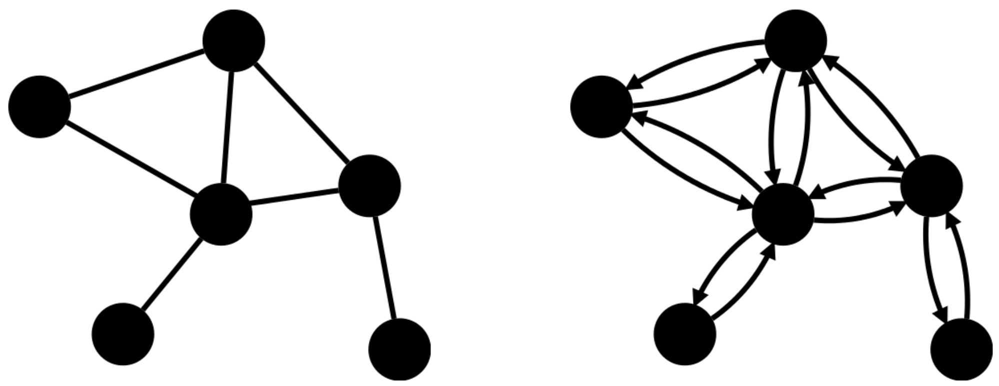

(a) Undirected Graph. (b) Directed Graph.

Figure 5.1: An undirected graph and its corresponding directed graph.

Instance	100		500		1000		2000		5000		10000	
	DI	EK	DI	EK	DI	EK	DI	EK	DI	EK	DI	EK
0.0	0.004	0.01	0.034	0.136	0.035	0.104	0.191	0.643	1.078	2.515	4.844	9.783
0.1	0.001	0.001	0.008	0.027	0.038	0.138	0.217	0.581	1.351	3.565	6.373	15.464
0.4	0.001	0.002	0.018	0.062	0.052	0.298	0.323	1.218	2.568	7.869	12.346	30.58
1.0	0.001	0.003	0.017	0.102	0.114	0.475	0.657	1.888	4.163	12.548	28.464	56.516

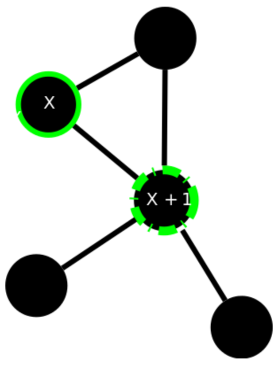
Table 5.1: Dinic's (DI) algorithm and Edmonds-Karp (EK) algorithm time (in s)

from a vertex v to a vertex u is the negative of the net flow in the reverse direction. The running time of the algorithms is determined by the maximum flow algorithms they are based on. In detail this would be a running time of $\mathcal{O}(VE^2)$ for Edmonds-Karp, and $\mathcal{O}(V^2E)$ for Dinic's algorithm.

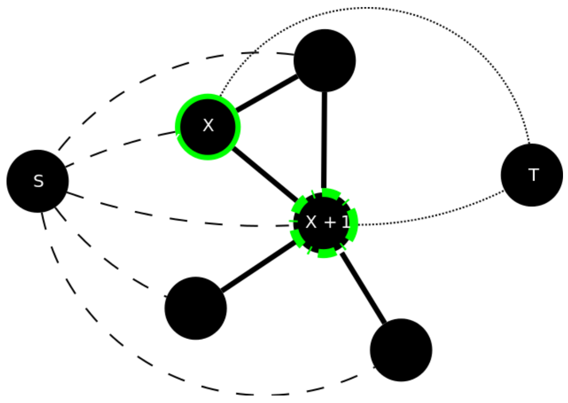
The running time of Dinic's algorithm and Edmonds-Karp algorithm is compared for instances with 100, 500, 1000, 2000, 5000 or 10000 vertices, and 10% of these vertices as sinks. Four Different versions of the same graphs were used for testing, first, the graph as a tree (with an edge-add probability of 0.0), second, with an edge-add probability of 0.1, third, with an edge-add probability of 0.4, and fourth with an edge-add probability of 1.0, which represents a Delaunay triangulated version of the graph. These instances were created using a graph generator, for furthermore and detailed descriptions about the graph creation process see section 6.1. In the table 5.1 the running time of both algorithms in seconds for the different instances is presented. Dinic's algorithm is faster in all test instances with a factor of ~2.

5.2 Maximum Flow Algorithm

The Edmonds-Karp and Dinic's algorithm calculate the maximum flow from one source s to one target sink t. In order to get correct maximum flow results with more than one source and one sink, artificial vertices have to be added to the graph. One artificial supersource (in figure 5.2b denoted as S) and one artificial supersink (in figure 5.2b denoted as T). The supersource is connected to all vertices with edges, whose capacity is equal to the produced load of their corresponding connecting vertex (the dot-like lines connected to vertex S in 5.2b). Every vertex produces a certain amount of load, therefore every vertex is considered as a source. On the other hand, the supersink is connected to all sinks of the graph with edges whose capacity is the sum of the capacity of all edges connected to the corresponding sink in the graph (the dotted lines connected to vertex T in 5.2b). After these updates the algorithms for calculating the maximum flow between the supersource and supersink can be applied. Finally, in order to obtain the final maximum flow of the graph the added artificial supersource and supersink with their connecting edges are removed. Figure 5.2 shows an illustration of the calculation of a maximum flow with more than one source and sink.



(a) Given graph with one sink X (green border), one more sink $X + 1$ is added (dotted green border).



(b) Calculating the new maximum flow, with adding a supersource S and a supersink T, with corresponding edges.

Figure 5.2: Maximum flow calculation with more than one source and sink.

The presented algorithms often use an iterative approach, for example the presented greedy algorithm in 3.3, which means that one sink is added to a solution at a time. Calculating the maximum flow each iteration is time consuming. Therefore, in order to decrease the needed time a specific concept of the Ford–Fulkerson method can be exploited. The Ford-Fulkerson method calculates the maximum flow of a graph based on the computation of all flow augmenting paths between the source and the sink. If a vertex is added or removed as sink in step $y + 1$, the already calculated maximum flow of step y can be used to calculate the new maximum flow in shorter time.

First, in the case that a new sink is added in step $y + 1$ the already calculated maximum flow of step y can be used. In detail, the maximum flow graph (as seen in figure 5.2b) of step y is used with one more edge added from the new sink to the supersink. As mentioned before, this edge needs to have a capacity which is equal to the sum of the capacity of all edges connected to the corresponding sink in the graph. On this new graph the maximum flow algorithms can be applied, and since most of the augmenting

Instance	100		500		1000		2000		5000		10000	
	EX	AN	EX	AN	EX	AN	EX	AN	EX	AN	EX	AN
0.0	0.023	0.036	0.274	0.697	0.387	5.011	1.242	43.893	9.404	893.312	30.909	2983.662
0.1	0.002	0.010	0.088	0.704	0.354	4.535	1.431	63.901	11.100	1148.085	39.111	4245.618
0.4	0.006	0.010	0.141	1.093	0.575	7.317	2.460	87.959	16.134	1518.826	65.390	5908.233
1.0	0.007	0.018	0.221	1.502	0.930	12.847	3.987	143.321	26.905	2455.647	107.496	9503.353

Table 5.2: Time for a maximum flow calculation using an existing calculated flow graph (EX) and starting anew for each set sink (AN) (Dinic's algorithm as used method in both cases) (in s)

paths were already found in step y the time for finding the maximum flow (equal to finding all augmenting paths) is drastically reduced.

Second, in the case that a sink is removed the already calculated maximum flow graph (as seen in figure 5.2b) of step y is used with the edge connecting the removed sink and the supersink removed. Then the capacity of the edges connected to the former sink is set to zero, and a maximum flow from the former sink to the supersource is calculated. This would reduce the flow that was send to the supersink over the former sink to zero. After this step the original capacities are restored and the maximum flow from the supersource to the supersink is computed. Again, since most of the augmenting paths were already found in step y the time for finding the maximum flow (equal to finding all augmenting paths) is drastically reduced.

The running time of the presented iterative algorithm (using Dinic's algorithm for the maximum flow calculation) is compared to a simple naive algorithm which calculates the maximum flow anew for at each step (using Dinic's algorithm for the maximum flow calculation). The needed time is tested for instances with 100, 500, 1000, 2000, 5000 or 10000 vertices, and 10% of these vertices are randomly upgraded to sinks one after another. After each upgrade the maximum flow is calculated. The iterative algorithm reuses already calculated flow networks of previous iterations, whereas the simple naive algorithm calculates the maximum flow anew at each step. As mentioned before, four different versions of each graph were generated (with an edge-add probability of 0.0, 0.1, 0.4, 1.0) using the graph generator presented in section 6.1. In the table 5.2 the running time of both algorithms in seconds for the different instances is presented. Reusing already calculated flow networks drastically reduces the running time, needing only a fraction of the time of the simple naive algorithm.

In all presented algorithms the iterative maximum flow algorithm is used.

CHAPTER 6

Computational Performance Evaluation

In this chapter the computational performance of the presented algorithms is tested and evaluated. This chapter is structured in the following way. First, in section 6.1 the graph generation process of the test instances is explained. Second, in section 6.2 the test system and the parameter settings of the algorithms are described. Third, in section 6.3 the computational results are presented and analysed. Finally, in section 6.4 the algorithms are tested on a created real world instance of an Austrian network system.

6.1 Graph Generation

6.1.1 Random Graph Generator

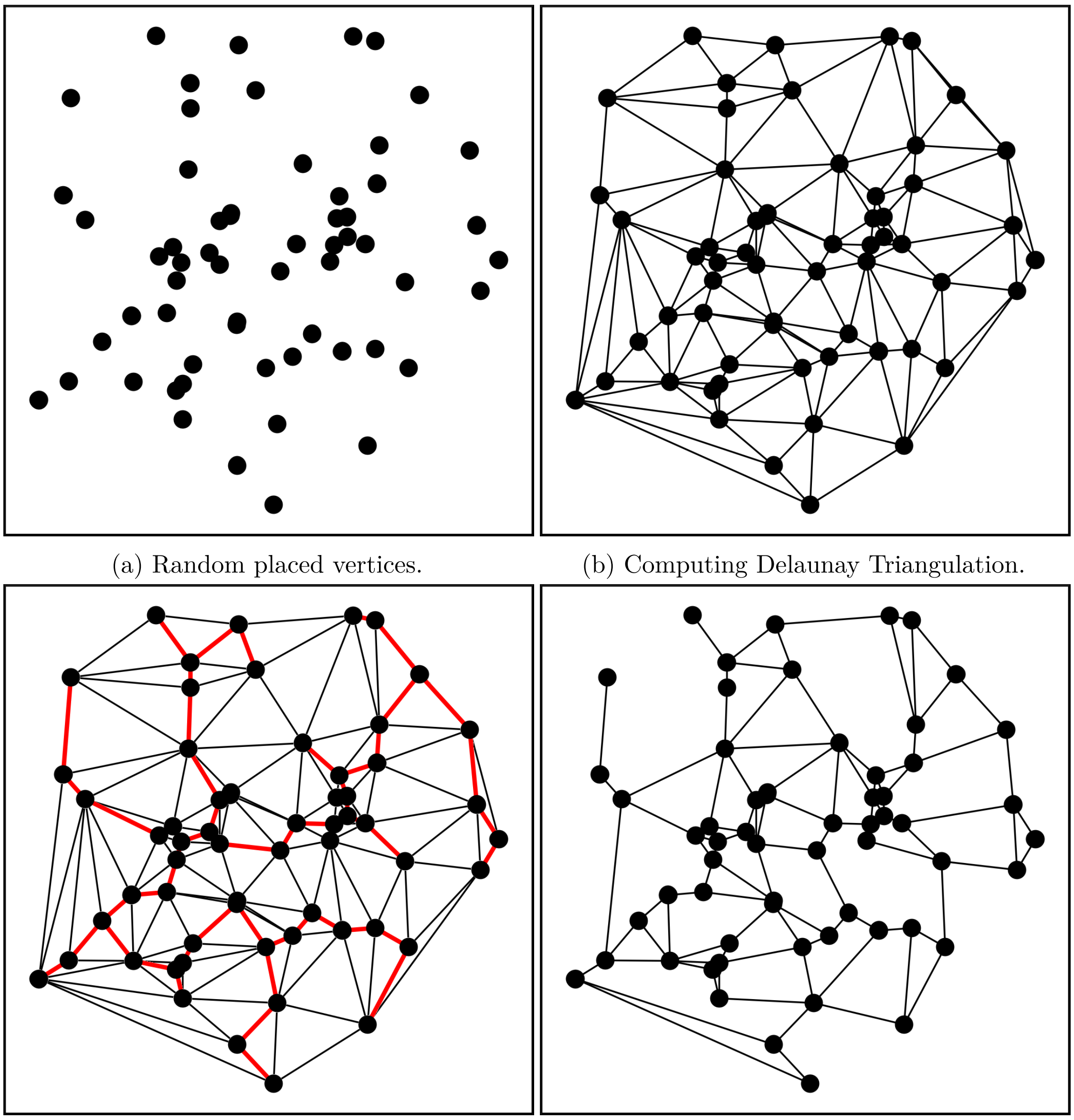
The graph generation for testing purposes is a crucial part to ensure well tested algorithms. Each generated graph should resemble a real world instance, in order to test the algorithms for practical use. The image shown in Figure 6.1 illustrates the concept of the graph generation. In a first step, presented in Figure 6.1a, a rectangular area with and edge length l is created, and a defined amount of vertices are semi-randomly placed in this area. Depending on an input probability p and a number c, more or less vertices are placed closer together in one or more clusters, mimicking cities, and the rest of vertices are spread across the area representing the countryside. The number c represents the number of clusters to be placed, which are randomly selected points (with x and y coordinates) as cluster center on the area. The probability p defines whether a vertex is placed near a cluster center or randomly on the area. For the placement near the cluster center a specific method is used. Outgoing from the random chosen cluster center a straight line with a maximum random length $h < \frac{l}{2}$ with a random angle is temporarily placed. The length of the line is normal distributed with 0 as mean and h as standard deviation.

The new vertex gets placed at the end of this line, which is afterwards removed. This results in vertices which are close to the cluster center and become less likely further away from the center. Each placed vertex gets a random load, which is produced at this vertex, a random customer potential, a random customer value, and a random cost value, determining the costs to upgrade this vertex. Furthermore, these random values are in general higher at vertices within city-cluster than at vertices located on the countryside. In detail, they get increased by a specific percentage. During the second step a Delaunay Triangulation over the placed vertices is computed, which is a triangulation for a given set of vertices such that no vertex is inside the circumcircle, which is a circle that passes through all the vertices of a triangle, of any triangle of the triangulation. The motivation to use this method is to obtain a planar graph. Figure 6.1b illustrates this step. The implementation computes the Delaunay Triangulation using randomized incremental construction presented in [dBCvKO08] with a running time of $\mathcal{O}(n \log n)$. Each placed edge gets a random capacity, whereas the capacities of edges connecting vertices in city-clusters is in general higher than of edges connecting countryside vertices (get multiplied by a specific number). In the third step, presented in Figure 6.1c with the red edges, a minimum spanning tree, using the Euclidean distances of the edges inserted in step two, is computed. These edges are fixed and ensure that the graph remains connected in step four. During the last step, illustrated in Figure 6.1d edges are deleted with a specified probability. Again, this probability is higher for edges connecting vertices in the countryside and lower for edges connecting vertices in city-clusters. Furthermore, the edges of the minimum spanning tree created in step three can not be deleted. In conclusion, with careful adjustment of the input parameters it is possible to create instances containing zero or more city-clusters, or solely consisting of countryside vertices. Furthermore, the amount of edges can be influenced and additionally tree-like graphs can be created.

6.1.2 Graph Test Instances

The presented algorithms were tested using several graph instances created by the previously introduced method. Graphs with 100, 500, 1000, 2000, 5000, 10000 vertices were created to measure the performance of the algorithms on instances of different size. The variable p was set to 0.33, and the the variable c was set to 2 for the 100, 500, 1000 instances and to 4 for the 2000, 5000, 10000 instances. Furthermore, for each generated graph four distinct versions with a different amount of edges were created. In step four of the graph generation process in 6.1.1, illustrated in Figure 6.1d, edges are deleted with a specified probability *edgeDel*, or equivalent, edges are kept with a probability $edgeProb = 1 - edgeDel$, which is used to identify the versions. Edges of the minimum spanning tree created in step three illustrated in Figure 6.1c are excluded from the deletion process. Four *edgeProb* values were used to create four versions of the same graph.

1. $edgeProb = 0.0$ and $edgeDel = 1.0$, all edges, except the minimum spanning tree edges were deleted, resulting in a tree version of the graph (see Figure 6.2a).



(a) Random placed vertices.

(b) Computing Delaunay Triangulation.

(c) Calculating a minimum spanning tree.

(d) Removing edges (except the MST edges).

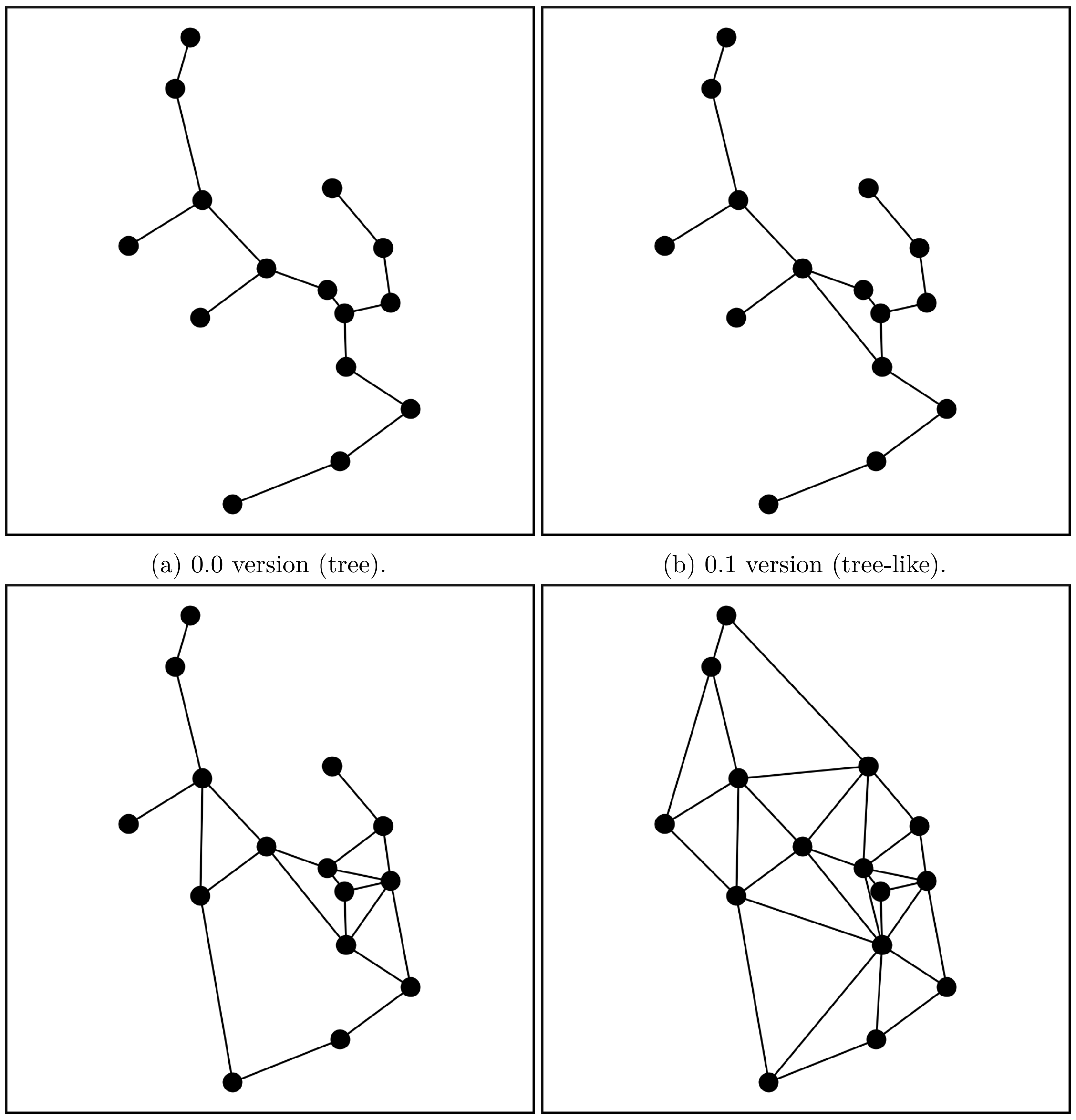
Figure 6.1: Graph Generation Concept

2. $edgeProb = 0.1$ $edgeDel = 0.9$, 90% of the edges are deleted, resulting in a tree-like version of the graph (see Figure 6.2b).

3. $edgeProb = 0.4$ and $edgeDel = 0.6$, 60% of the edges are deleted (see Figure 6.2c).

4. $edgeProb = 1.0$ $edgeDel = 0.0$, no edges are deleted, resulting in a planar Delaunay triangulated version of the graph (see Figure 6.2d).

In Figure 6.2 an example graph with 15 vertices is shown in its four different versions.



(a) 0.0 version (tree).

(b) 0.1 version (tree-like).

(c) 0.4 version.

(d) 1.0 version (Delaunay triangulated).

Figure 6.2: Four versions of an example graph with 15 vertices.

In conclusion, the problem instances consist of 6 different sets, reaching from 100 to 10000 vertices. Each set contains 10 different instances. A problem instance describes the produced load, customer value, customer potential, upgrade costs for each vertex, and the capacity of the connecting edges. These values are randomly generated. Each instance exists in four different versions as described above, with an *edgeProb* of 0.0, 0.1, 0.4, and 1.0. All instances and versions were tested and the average results calculated.

6.2 Testing System and Parameter Settings

This section presents the testing system and the parameters used for the computational analysis. The test system consisted of an Intel Core I5-4460 CPU with up to 3.4 GHz, 8 GB RAM and using the Linux distribution Ubuntu 16.04 as operation system.

The weights of the objective function, $\omega_1^{obj}, \omega_2^{obj}, \omega_3^{obj}, \omega_4^{obj}$ and ω_5^{obj}, were set to 1 each, to equally weight all the terms.

IBM ILOG CPLEX, in version 12.6.1, and Gurobi, in version 7.0, were used for solving the linear programs. Both solvers were set up to use 4 cores, and solve to 100% optimality, without a time limit. Java was used to describe the model and access the APIs.

The heuristic methods were coded in Java with no use of multi-threading or parallel computing. The following paragraphs describe the chosen parameters in detail.

Linear Programming Formulation Relaxation Settings

The LP Relaxation was solved using CPLEX, and the rounding process was repeated as long as no more improvement occurred five times in a row.

Greedy Heuristic Settings

The Greedy algorithm has several parameters which must be chosen before running the calculation. These parameters influence, and greatly determine the quality of the produced solution. Therefore, the parameters must be carefully chosen in order to obtain high performing solution given the objective function. To obtain useful values for the terms $\omega_1^{greedy}, \omega_2^{greedy}, \omega_3^{greedy}, \omega_4^{greedy}, \omega_5^{greedy}, \omega_6^{greedy}$, and ω_7^{greedy}, of the sink-choosing function Z, 1000 random generated instances with 1000 vertices and 100 sinks to place were solved using different combinations, values between 0 and 1, of settings for the weights. The top 10% of the best performing parameters were picked and their average rounded value selected. This results in the greedy parameters shown in table 6.1.

ω_1^{greedy}	1
ω_2^{greedy}	1
ω_3^{greedy}	0.5
ω_4^{greedy}	0.5
ω_5^{greedy}	0.5
ω_6^{greedy}	1
ω_7^{greedy}	1

Table 6.1: Greedy Heuristic Parameters

The Greedy Heuristic was tested using the deterministic and random approach. For the deterministic test run, the best performing vertex, according to the Z function was chosen, whereas for the random test run, a random vertex of the top 15% according to the Z function was picked.

GRASP Settings

GRASP was tested using the random Greedy approach with the same parameters as explained before. Furthermore, within each GRASP test run, 10 solutions were generated and further improved by the local search technique Exchange Sinks Random.

Greedy-Net, Greedy Tree Solver, and Cluster Settings

Greedy-Net, and Greedy Tree Solver, and Cluster did not have any further parameters to define.

Post Optimization Settings

The two local search techniques for post optimization, Exchange Sinks Depending on Z, further denoted as Exchange Sinks Z, and Exchange Sinks Random, were checking 50 random sinks, and for each checked sink, 100 vertices were tested. For a better comparison random solutions were created, where the sinks were chosen randomly.

6.3 Performance Evaluation

The tables 6.2, 6.5, 6.8, and 6.11 present the average results of the algorithms tested on the generated test instances, as described in section 6.1.2. The algorithms were tested using the same parameters and settings explained in 6.2. In the main test instance 10% of the vertices were upgraded to sinks, with no initial sinks set. This value was obtained empirically by the Greedy Tree Solver algorithm which was run on random instances (created with the same settings and properties as described), and fully satisfied these instances. On average 10% of the vertices were upgraded to sinks. Furthermore, for better comparison, the algorithms were also tested with 5% and 2% of the vertices upgraded to sinks. The first value indicates the performance in % of the optimal maximum, optimal values achieved by the exact solution techniques have 100%. The value in the parenthesis shows the running time in seconds for solving the instance. The same data is shown for the post optimization local search techniques in table 6.14. The values are averages of 100 test runs on randomly generated solutions with Greedy Random, using random 0.4 instances.

6.3.1 Performance 0.0 Instances (Trees)

From the optimal, 100%, integer linear programming models Model 1, in general, had a lower running time of a factor of 2 than Model 2. In most instances CPLEX produced the optimal solution in half the time of Gurobi.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.3	100.0	0.8	100.0	2.3	100.0	6.4	100.0	42.2	100.0	124.8
CPLEX Model 2	100.0	0.4	100.0	2.2	100.0	5.9	100.0	17.2	100.0	86.0	100.0	250.0
Gurobi Model 1	100.0	0.1	100.0	1.9	100.0	4.5	100.0	24.5	100.0	72.1	100.0	131.9
Gurobi Model 2	100.0	0.3	100.0	2.5	100.0	6.0	100.0	35.3	100.0	232.7	100.0	665.8
Greedy	72.3	0.0	68.2	0.1	70.6	0.5	71.5	0.8	71.3	5.1	71.9	22.9
Greedy Random	**100.0**	0.0	86.8	0.1	86.1	0.7	87.9	0.9	87.1	5.2	87.9	24.8
GRASP	**100.0**	0.0	86.9	0.9	86.1	11.0	87.3	18.1	87.7	168.3	88.0	673.1
LP Relaxation	95.6	0.2	89.9	0.3	89.9	0.9	88.2	2.6	89.5	32.1	88.2	86.2
Greedy-Net	94.7	0.0	**93.6**	0.1	**93.8**	0.1	**93.2**	0.2	**94.7**	1.5	**93.8**	5.9
Greedy Tree Solver	94.8	0.0	84.7	0.0	85.6	0.0	86.5	0.2	87.4	1.2	86.9	4.7
Clustering	90.0	0.0	86.9	0.3	89.0	0.8	86.5	2.4	88.7	13.5	88.3	57.7
Random	68.7	0.0	61.3	0.0	61.1	0.0	60.3	0.1	62.5	1.0	61.3	4.4

Table 6.2: Results for the 0.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	1.5	100.0	4.0	100.0	8.5	100.0	59.1	100.0	231.7	100.0	1042.6
CPLEX Model 2	100.0	1.4	100.0	10.0	100.0	28.6	100.0	153.9	100.0	507.6	100.0	2484.2
Gurobi Model 1	100.0	0.7	100.0	8.3	100.0	24.0	100.0	139.9	100.0	628.6	100.0	2901.1
Gurobi Model 2	100.0	0.9	100.0	19.4	100.0	47.1	100.0	155.7	100.0	1110.0	100.0	6281.6
Greedy	64.5	0.0	70.5	0.2	72.6	0.3	73.1	0.9	75.2	4.0	75.1	16.0
Greedy Random	80.1	0.0	91.1	0.2	90.9	0.4	92.7	0.9	94.4	3.6	94.8	7.2
GRASP	80.9	0.0	91.7	0.6	91.3	8.5	93.2	13.9	95.0	110.9	95.2	390.9
LP Relaxation	**100.0**	0.2	**95.0**	0.6	**97.4**	1.5	94.3	4.2	95.1	32.0	94.5	213.3
Greedy-Net	90.5	0.0	93.2	0.0	93.4	0.1	**95.3**	0.3	**96.3**	1.5	**96.1**	3.4
Greedy Tree Solver	87.5	0.0	90.1	0.0	92.0	0.1	91.0	0.2	93.0	1.4	92.9	2.1
Clustering	76.8	0.1	88.9	0.4	90.9	0.7	90.0	2.9	92.9	11.9	93.0	33.7
Random	49.8	0.0	59.8	0.1	61.0	0.1	61.3	0.2	59.8	0.9	60.1	1.9

Table 6.3: Results for the 0.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	1.3	100.0	27.8	100.0	62.8	100.0	174.2	100.0	523.2	100.0	3056.4
CPLEX Model 2	100.0	1.1	100.0	55.9	100.0	112.0	100.0	825.2	100.0	3498.1	100.0	20382.8
Gurobi Model 1	100.0	0.9	100.0	38.5	100.0	444.7	100.0	6687.9	100.0	28383.9	-	-
Gurobi Model 2	100.0	1.0	100.0	122.7	100.0	305.8	100.0	1124.0	100.0	3593.0	100.0	22345.1
Greedy	65.9	0.0	70.7	0.1	67.7	0.1	73.0	0.4	73.7	1.7	74.3	3.4
Greedy Random	**100.0**	0.0	91.0	0.1	90.5	0.2	91.3	0.4	89.4	1.8	90.2	3.2
GRASP	**100.0**	0.1	91.4	0.6	90.8	4.3	91.8	8.8	89.8	82.3	90.3	331.8
LP Relaxation	99.9	0.2	98.2	0.5	**100.0**	1.4	**98.3**	3.4	**97.0**	17.9	**98.3**	48.8
Greedy-Net	94.9	0.0	94.0	0.0	94.9	0.1	96.1	0.3	95.0	1.6	95.7	4.0
Greedy Tree Solver	89.1	0.0	**100.0**	0.0	94.3	0.1	95.9	0.2	95.3	1.4	95.2	3.2
Clustering	95.1	0.1	91.1	0.4	93.9	0.6	92.0	1.7	93.2	10.2	94.1	73.1
Random	61.3	0.0	59.0	0.0	59.4	0.0	60.1	0.3	60.3	1.5	60.2	3.4

Table 6.4: Results for the 0.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

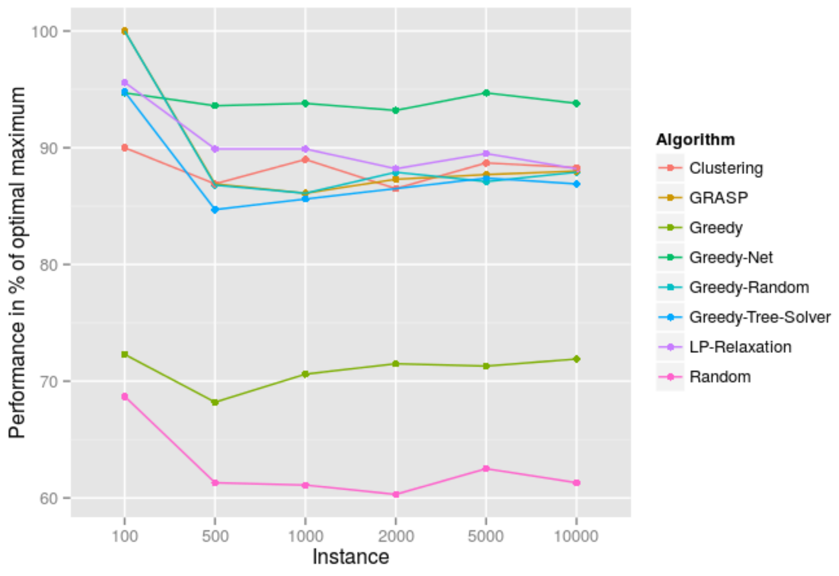


Figure 6.3: Results for the 0.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

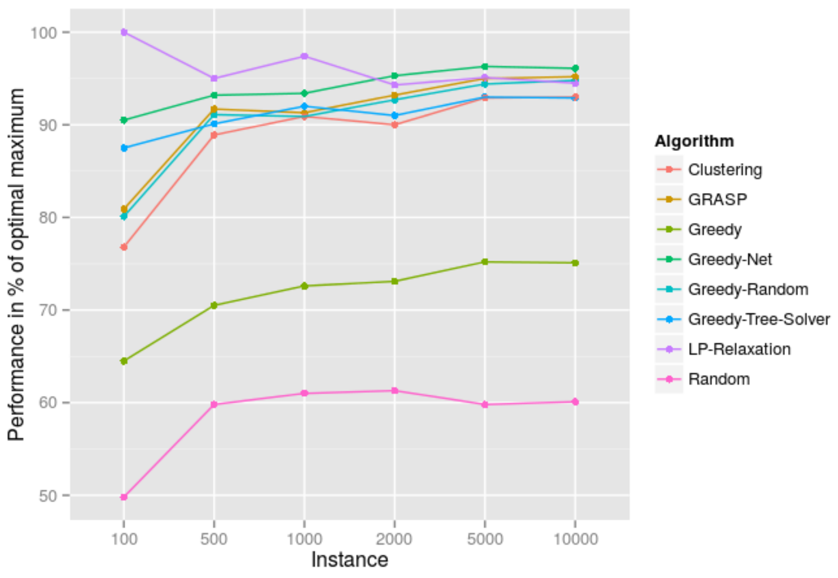


Figure 6.4: Results for the 0.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

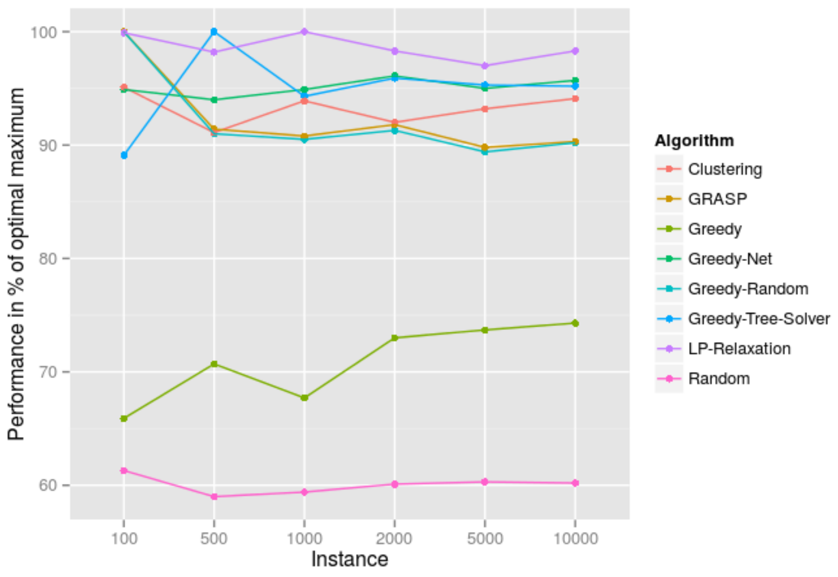


Figure 6.5: Results for the 0.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

The random Greedy Heuristic (denoted as Greedy Random) outperforms the deterministic Greedy Heuristic (denoted as Greedy), always producing over 86% results (with placing 5% sinks in 100 vertices being the only exception), whereas Greedy has an average of ~73%. The running time, with 22.9 and 24.8 for the largest instances, is mid-table compared to the other heuristics in the 10% sinks instances. The running time for the 5% and 2% sinks instances is significantly lower, since less sinks are placed.

GRASP was the slowest heuristic for placing 10%, 5%, and 2% sinks, while producing solutions which were maximum 1% point better than Greedy Random. However, this can be explained with the low performance of the local search techniques shown in table 6.14.

LP Relaxation was the second slowest heuristic tested, and in general, was not producing better results than faster algorithms like Greedy Random for 10% sinks instances. However, the solutions produced for the 5% and 2% sinks instances are among the best and second best results.

The Greedy-Net heuristic was the second fastest calculating heuristic, and producing stable, ~94% solutions for all 10% sinks instances.

The Greedy Tree Solver produced results, which were 1% to 6% points worse than Greedy Random, within the fastest running time for the 10% sinks instances. The Greedy Tree Solver did not produce optimal solutions since customer value, customer potential and upgrade cost are not considered in the algorithm, but in the objective function.

The Clustering algorithm calculated similar performing solutions as Greedy Random for the 10% sinks instances, however, needing the second longest running time of all heuristics.

The results of Greedy-Net, Greedy Tree Solver, and Clustering for the for the 5% and 2% sinks instances are performing equally with results of ~93%.

As expected, picking sinks randomly yields the worst performing results, being ~25 percent points lower than the best performing heuristic.

6.3.2 Performance 0.1 Instances

The performance of the 0.1 instances is similar to the 0.0 instances, however, in general needing longer for calculating a solution, and with slightly less good performing results for the heuristics. CPLEX with Model 1 is the fastest exact solution method. Greedy Random outperforms the deterministic Greedy Heuristic, and GRASP was the slowest heuristic, while producing solutions which were hardly better than Greedy Random. LP Relaxation was the second slowest heuristic tested, and in general, was not producing better results than faster algorithms like Greedy Random. The Greedy-Net heuristic was the fastest calculating heuristic, and producing stable ~90% - 95% solutions for all instances. The Greedy Tree Solver produced results, which were ~1% - 2% points worse than Greedy Random, within the second fastest running time, and finally, the Clustering algorithm calculated slightly better, ~1% - 2% points, performing solutions as Greedy Random, however, needing the second longest running time of all heuristics.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.2	100.0	0.5	100.0	2.2	100.0	6.9	100.0	35.9	100.0	185.3
CPLEX Model 2	100.0	0.4	100.0	2.4	100.0	9.1	100.0	31.0	100.0	92.4	100.0	346.7
Gurobi Model 1	100.0	0.3	100.0	2.6	100.0	9.4	100.0	12.4	100.0	300.7	100.0	1243.5
Gurobi Model 2	100.0	0.6	100.0	5.2	100.0	17.5	100.0	44.7	100.0	241.4	100.0	1014.1
Greedy	72.7	0.0	67.8	0.1	69.9	0.2	68.1	0.9	70.8	6.1	70.1	27.0
Greedy Random	85.3	0.0	85.7	0.1	87.0	0.2	83.5	1.0	85.1	6.6	85.4	31.1
GRASP	85.7	0.0	86.5	1.0	87.2	14.1	83.9	22.3	84.9	199.6	85.5	782.7
LP Relaxation	**94.2**	0.0	90.3	0.4	84.6	1.0	84.8	5.7	85.6	42.4	85.8	97.5
Greedy-Net	86.5	0.0	**91.5**	0.0	**90.0**	0.0	**90.0**	0.2	**90.0**	1.8	**91.0**	6.8
Greedy Tree Solver	85.5	0.0	83.8	0.0	81.9	0.0	81.9	0.2	82.7	2.0	83.6	10.8
Clustering	90.7	0.0	87.9	0.2	82.1	0.6	85.5	2.7	87.2	23.4	86.3	164.0
Random	58.9	0.0	61.4	0.0	58.8	0.0	58.0	0.2	59.2	1.2	59.7	5.2

Table 6.5: Results for the 0.1 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.2	100.0	4.9	100.0	5.0	100.0	11.1	100.0	75.8	100.0	589.3
CPLEX Model 2	100.0	0.5	100.0	15.6	100.0	33.6	100.0	61.3	100.0	692.9	100.0	7821.7
Gurobi Model 1	100.0	0.1	100.0	7.2	100.0	24.4	100.0	30.8	100.0	471.0	100.0	5501.4
Gurobi Model 2	100.0	0.2	100.0	14.5	100.0	58.8	100.0	98.3	100.0	458.9	100.0	8081.6
Greedy	74.7	0.0	71.8	0.0	69.3	0.1	71.1	0.6	72.0	3.8	72.1	16.9
Greedy Random	90.9	0.0	91.7	0.0	88.4	0.1	89.6	0.6	87.9	4.2	88.3	17.1
GRASP	92.3	0.1	92.0	0.1	89.1	0.2	89.9	0.9	88.5	4.9	89.0	19.2
LP Relaxation	94.5	0.0	**95.8**	0.5	89.7	1.6	89.5	4.2	92.2	31.2	91.7	182.8
Greedy-Net	91.2	0.0	95.2	0.0	**92.1**	0.0	**92.9**	0.2	**94.3**	1.6	**93.2**	2.5
Greedy Tree Solver	**96.4**	0.0	86.7	0.0	85.7	0.0	86.3	0.2	86.7	1.7	86.9	2.6
Clustering	87.2	0.0	94.6	0.2	88.2	1.0	89.4	2.4	91.6	21.5	90.5	71.4
Random	48.4	0.0	57.0	0.0	58.4	0.1	56.1	0.2	57.2	1.2	56.1	2.7

Table 6.6: Results for the 0.1 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.3	100.0	4.1	100.0	5.6	100.0	42.7	100.0	200.0	100.0	701.3
CPLEX Model 2	100.0	0.7	100.0	23.8	100.0	52.5	100.0	470.3	100.0	1786.1	100.0	19573.8
Gurobi Model 1	100.0	0.6	100.0	7.5	100.0	20.2	100.0	127.8	100.0	777.9	100.0	8391.5
Gurobi Model 2	100.0	0.9	100.0	23.8	100.0	61.0	100.0	247.9	100.0	885.6	100.0	9891.3
Greedy	69.1	0.0	66.2	0.0	70.4	0.1	70.2	0.3	71.7	1.7	70.7	3.8
Greedy Random	81.1	0.0	88.3	0.0	87.0	0.1	91.3	0.3	91.4	2.0	91.2	4.5
GRASP	81.8	0.1	88.7	0.2	87.6	0.2	91.8	0.4	91.9	2.7	91.8	5.1
LP Relaxation	**100.0**	0.0	**100.0**	0.3	**94.5**	1.3	**95.6**	3.2	94.8	21.7	95.3	71.3
Greedy-Net	71.5	0.0	96.9	0.0	94.5	0.0	95.9	0.1	**96.0**	1.0	**95.4**	2.0
Greedy Tree Solver	97.4	0.0	87.5	0.0	92.8	0.0	89.4	0.2	88.7	1.9	89.2	3.9
Clustering	85.3	0.0	88.3	0.3	88.1	0.9	90.1	3.0	95.7	23.0	94.2	89.2
Random	49.1	0.0	58.9	0.0	56.1	0.1	60.3	0.2	60.7	1.3	60.1	3.2

Table 6.7: Results for the 0.1 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

6.3.3 Performance 0.4 Instances

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	1.4	100.0	3.9	100.0	13.0	100.0	40.4	100.0	177.8
CPLEX Model 2	100.0	0.1	100.0	2.9	100.0	44.5	100.0	65.6	100.0	348.5	100.0	3980.9
Gurobi Model 1	100.0	0.1	100.0	3.7	100.0	7.2	100.0	68.3	100.0	171.1	100.0	1992.5
Gurobi Model 2	100.0	0.1	100.0	8.0	100.0	38.5	100.0	87.6	100.0	377.6	100.0	1485.6
Greedy	67.2	0.0	71.1	0.1	72.0	0.3	68.8	1.4	69.3	9.3	68.8	40.9
Greedy Random	87.9	0.0	84.7	0.1	83.2	0.3	83.4	1.5	83.0	10.1	81.9	44.9
GRASP	88.0	0.0	84.9	1.1	83.8	18.3	83.5	25.3	83.0	221.7	81.9	881.2
LP Relaxation	91.9	0.0	**88.9**	0.5	85.3	2.3	84.2	8.5	83.7	27.2	82.9	189.3
Greedy-Net	**93.5**	0.0	86.5	0.0	**87.5**	0.1	**86.4**	0.3	**88.6**	2.1	**87.1**	10.5
Greedy Tree Solver	81.6	0.0	77.1	0.0	77.8	0.1	77.0	0.4	78.1	4.2	78.1	18.5
Clustering	89.3	0.0	81.5	0.2	84.0	1.0	83.5	7.0	84.7	120.1	84.4	1373.4
Random	63.1	0.0	58.3	0.0	58.8	0.0	57.2	0.3	58.2	1.8	58.2	7.2

Table 6.8: Results for the 0.4 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

The performance of the 0.4 instances similar to the previous instances, again with in general longer running times and with slightly less good performing results for the heuristics. CPLEX with Model 1 stays the fastest exact solution method. Greedy Random outperforms the deterministic Greedy Heuristic, and GRASP was the slowest heuristic, and again producing solutions which were hardly better than Greedy Random. LP Relaxation was the second slowest heuristic tested, and again in general, was not producing better results than faster algorithms like Greedy Random. The Greedy-Net heuristic was the fastest calculating heuristic, and the Greedy Tree Solver the second fastest. Again, the Clustering algorithm calculated slightly better, ~1.5% points,

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	1.8	100.0	1.8	100.0	5.0	100.0	20.7	100.0	196.6
CPLEX Model 2	100.0	0.3	100.0	7.1	100.0	9.9	100.0	37.3	100.0	107.9	100.0	1083.4
Gurobi Model 1	100.0	0.1	100.0	3.8	100.0	3.3	100.0	15.1	100.0	67.0	100.0	633.6
Gurobi Model 2	100.0	0.1	100.0	8.6	100.0	8.5	100.0	57.2	100.0	159.6	100.0	2455.1
Greedy	68.7	0.0	68.5	0.1	71.7	0.2	69.5	0.8	70.4	5.5	70.1	24.7
Greedy Random	78.4	0.0	79.4	0.1	84.7	0.3	85.6	1.0	87.3	5.8	86.7	25.9
GRASP	79.2	0.1	79.9	0.2	85.4	0.4	86.1	1.2	87.9	6.1	87.2	27.3
LP Relaxation	81.8	0.0	**90.8**	0.5	87.9	1.3	87.1	6.1	88.6	30.8	88.1	86.5
Greedy-Net	**90.9**	0.0	88.1	0.0	**89.0**	0.0	**88.5**	0.2	**89.7**	1.6	**89.6**	3.0
Greedy Tree Solver	85.4	0.0	78.1	0.0	80.8	0.1	79.9	0.4	81.7	3.7	81.1	5.3
Clustering	82.8	0.0	85.1	0.2	84.9	0.9	85.5	6.6	87.8	128.2	88.2	421.3
Random	54.8	0.0	57.9	0.0	59.4	0.0	57.6	0.2	61.2	1.7	60.9	2.5

Table 6.9: Results for the 0.4 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	1.4	100.0	1.9	100.0	4.3	100.0	17.6	100.0	183.4
CPLEX Model 2	100.0	0.2	100.0	6.3	100.0	5.9	100.0	19.0	100.0	71.7	100.0	671.2
Gurobi Model 1	100.0	0.0	100.0	1.2	100.0	3.7	100.0	3.3	100.0	14.0	100.0	154.9
Gurobi Model 2	100.0	0.2	100.0	7.2	100.0	13.5	100.0	41.2	100.0	133.6	100.0	1834.7
Greedy	53.3	0.0	66.4	0.0	72.7	0.1	68.7	0.4	71.3	2.6	69.4	5.3
Greedy Random	95.7	0.0	90.3	0.0	90.2	0.2	93.3	0.4	92.6	2.6	91.3	4.3
GRASP	96.1	0.1	90.8	0.2	90.8	0.3	**93.8**	0.5	92.9	3.0	91.8	5.2
LP Relaxation	**98.9**	0.0	**96.5**	0.4	**97.0**	0.8	92.8	4.5	**93.4**	26.0	**92.7**	79.3
Greedy-Net	75.5	0.0	90.5	0.0	86.6	0.0	85.3	0.1	92.9	0.9	91.1	2.0
Greedy Tree Solver	87.4	0.0	86.2	0.0	85.4	0.1	84.1	0.4	90.0	3.9	89.4	8.2
Clustering	67.7	0.0	82.3	0.2	90.5	1.0	89.9	7.4	91.4	148.5	90.5	481.4
Random	57.0	0.0	55.1	0.0	57.1	0.1	55.3	0.2	58.1	2.0	58.3	3.1

Table 6.10: Results for the 0.4 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

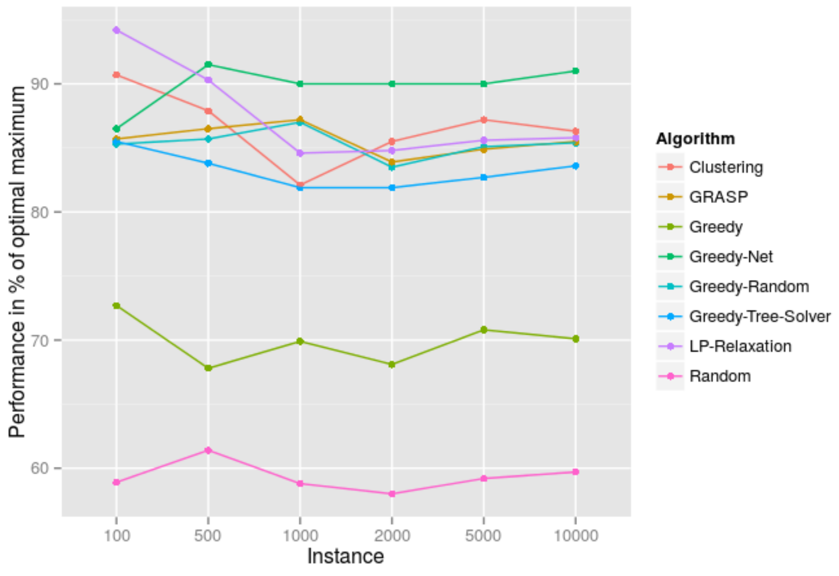


Figure 6.6: Results for the 0.1 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

performing solutions as Greedy Random, however, needing the second longest running time of all heuristics. Interestingly, the result of the deterministic Greedy heuristic for 100 vertices with 2% sinks is worse than the according Random result.

6.3.4 Performance 1.0 Instances

The performance of the 1.0 instances similar to the previous instances, and again with in general longer running times and with slightly less good performing results for the heuristics. CPLEX with Model 1 stays the fastest exact solution method. Greedy Random outperforms the deterministic Greedy Heuristic, and GRASP was the slowest Greedy heuristic. The Greedy-Net heuristic was the fastest calculating heuristic, and producing the best results overall. Greedy Tree Solver the second fastest, however, the performance decreased even further compared to the previous instances. Again, the Clustering algorithm was the slowest algorithm producing the second best results.

6.3.5 Performance Local Search

Table 6.14 shows the performance of the local search techniques. The running time is over 34.1 seconds for the random local search and over 38.5 seconds for the deterministic version. The results of Exchange Sinks Random are slightly better, around 0.1 percent points, than of Exchange Sinks Z.

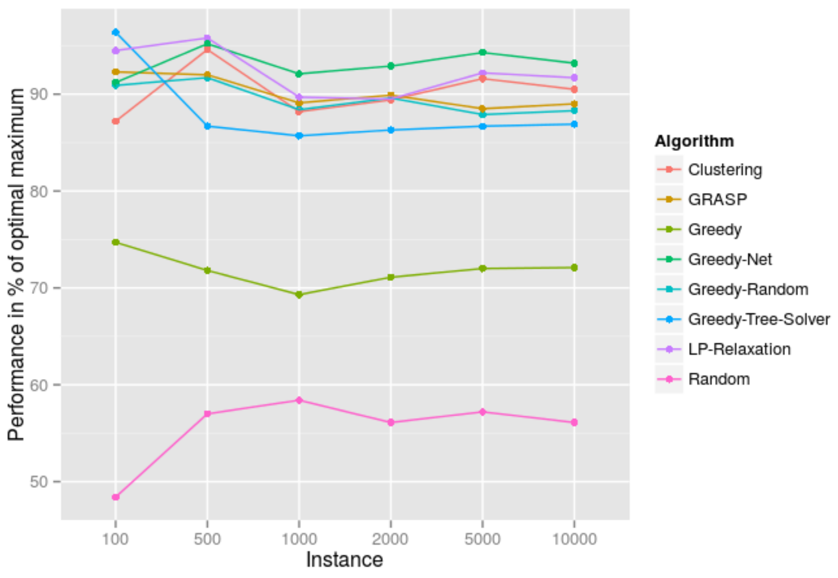


Figure 6.7: Results for the 0.1 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

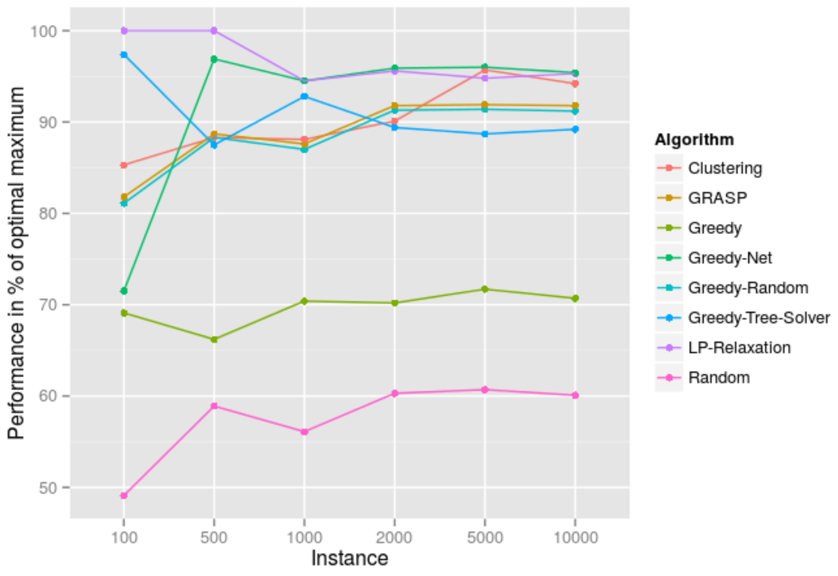


Figure 6.8: Results for the 0.1 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

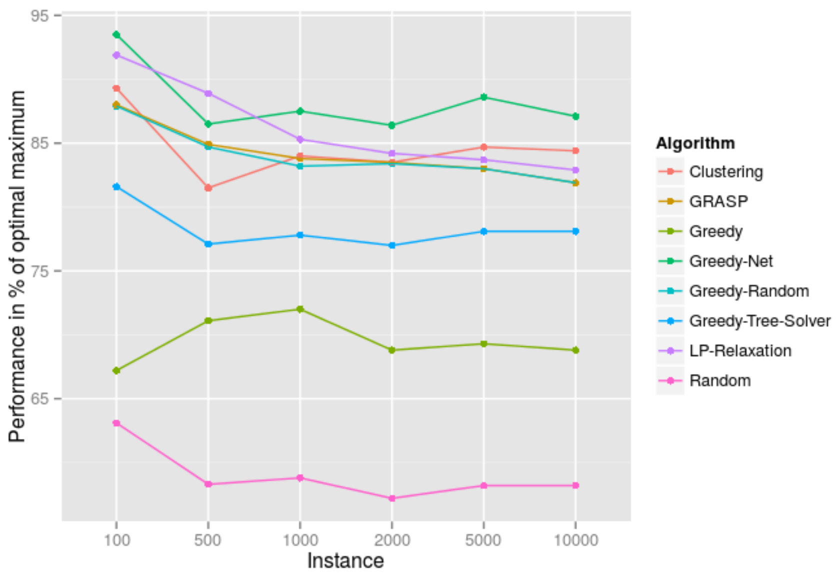


Figure 6.9: Results for the 0.4 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

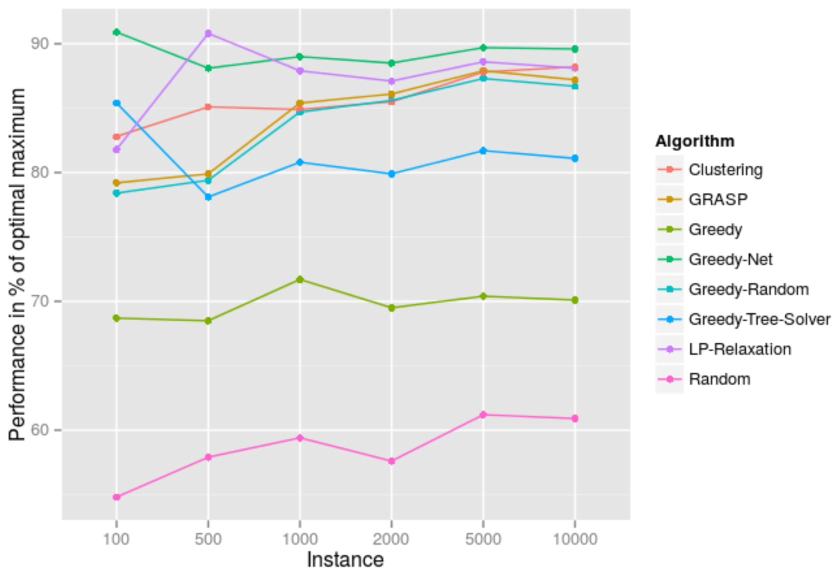


Figure 6.10: Results for the 0.4 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

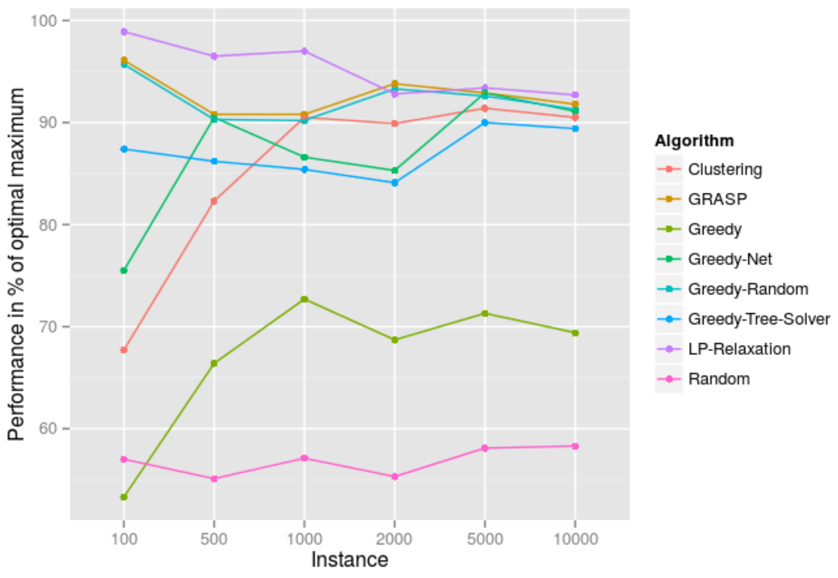


Figure 6.11: Results for the 0.4 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	0.8	100.0	2.9	100.0	6.8	100.0	21.3	100.0	250.7
CPLEX Model 2	100.0	0.4	100.0	7.0	100.0	17.6	100.0	88.6	100.0	311.8	100.0	780.5
Gurobi Model 1	100.0	0.1	100.0	1.0	100.0	6.6	100.0	18.5	100.0	259.6	100.0	1225.9
Gurobi Model 2	100.0	0.3	100.0	1.8	100.0	7.8	100.0	91.9	100.0	1008.7	100.0	4522.4
Greedy	66.6	0.0	67.5	0.1	70.4	0.5	69.1	2.2	69.8	14.5	69.4	62.2
Greedy Random	79.2	0.0	81.4	0.1	79.2	0.5	80.8	2.3	82.4	15.9	80.2	69.1
GRASP	79.6	0.0	81.5	1.4	79.0	22.2	80.7	32.1	82.4	261.9	80.2	1034.5
LP Relaxation	**90.5**	0.0	80.2	0.3	81.0	1.1	79.5	5.1	81.2	70.3	80.5	149.8
Greedy-Net	85.4	0.0	**82.0**	0.0	**86.0**	0.1	**82.2**	0.4	**84.4**	3.5	**84.5**	13.0
Greedy Tree Solver	77.4	0.0	73.0	0.0	74.6	0.1	73.0	0.8	75.5	4.8	74.5	29.6
Clustering	87.2	0.0	81.8	0.3	81.4	1.9	80.4	16.4	83.3	349.5	81.8	3088.6
Random	59.3	0.0	56.9	0.0	56.3	0.0	56.1	0.3	57.2	2.3	57.0	10.4

Table 6.11: Results for the 1.0 instances with 10% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	0.6	100.0	2.9	100.0	5.4	100.0	21.0	100.0	181.9
CPLEX Model 2	100.0	0.3	100.0	1.0	100.0	7.2	100.0	15.7	100.0	77.1	100.0	589.3
Gurobi Model 1	100.0	0.0	100.0	0.6	100.0	3.1	100.0	10.7	100.0	42.3	100.0	299.2
Gurobi Model 2	100.0	0.2	100.0	4.2	100.0	5.5	100.0	17.4	100.0	64.9	100.0	501.1
Greedy	66.1	0.0	70.7	0.1	76.7	0.3	68.2	1.3	72.0	8.3	71.7	40.0
Greedy Random	90.3	0.0	84.6	0.1	87.4	0.3	85.6	1.3	86.4	9.5	87.3	51.2
GRASP	90.7	0.1	85.1	0.2	87.9	0.3	86.2	1.8	86.9	9.9	87.8	55.1
LP Relaxation	**97.0**	0.0	84.6	0.4	**92.1**	1.6	**87.7**	7.5	87.8	34.7	**88.2**	132.4
Greedy-Net	84.3	0.0	83.0	0.0	87.8	0.1	87.5	0.2	**89.6**	1.8	87.2	2.4
Greedy Tree Solver	78.2	0.0	73.4	0.0	82.8	0.1	79.7	0.7	79.6	6.6	78.8	28.1
Clustering	91.0	0.0	**85.9**	0.3	89.8	1.8	87.6	15.9	88.9	374.7	86.2	281.2
Random	53.1	0.0	53.4	0.0	57.3	0.0	54.7	0.1	55.9	1.5	54.0	2.0

Table 6.12: Results for the 1.0 instances with 5% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	0.1	100.0	0.5	100.0	1.8	100.0	5.2	100.0	28.8	100.0	210.2
CPLEX Model 2	100.0	0.6	100.0	0.7	100.0	3.9	100.0	11.1	100.0	52.1	100.0	352.1
Gurobi Model 1	100.0	0.1	100.0	0.8	100.0	2.0	100.0	4.9	100.0	36.6	100.0	262.6
Gurobi Model 2	100.0	0.1	100.0	1.5	100.0	4.1	100.0	29.8	100.0	48.8	100.0	302.9
Greedy	58.4	0.0	69.0	0.0	77.1	0.1	68.8	0.5	72.0	3.5	71.4	7.2
Greedy Random	87.5	0.0	93.1	0.0	94.0	0.1	89.1	0.6	93.2	3.9	92.6	8.1
GRASP	87.9	0.1	93.7	0.2	94.4	0.2	89.7	0.6	93.8	4.3	92.9	9.3
LP Relaxation	**100.0**	0.1	**100.0**	0.5	**97.5**	1.4	**92.7**	8.0	**95.9**	23.4	94.4	81.4
Greedy-Net	77.9	0.0	81.6	0.0	95.2	0.0	89.1	0.1	93.8	1.0	92.8	2.8
Greedy Tree Solver	75.8	0.0	80.5	0.0	93.2	0.1	88.1	0.3	87.4	4.8	87.5	9.5
Clustering	77.1	0.0	90.9	0.3	96.8	1.9	85.9	18.7	95.6	451.7	**95.1**	1245.9
Random	42.4	0.0	54.9	0.0	60.9	0.0	57.5	0.2	59.8	1.6	60.1	2.4

Table 6.13: Results for the 1.0 instances with 2% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold.

Algorithm	100		500		1000		2000		5000		10000	
	opt	time	opt	time	opt	time	opt	time	opt	time	opt	time
Exchange Sinks Random	0.4	0.0	0.5	0.9	0.8	3.3	0.7	7.2	0.1	14.6	0.0	34.1
Exchange Sinks Z	0.3	0.0	0.3	1.2	0.7	3.2	0.5	8.4	0.0	18.0	0.0	38.5

Table 6.14: Results of the local search techniques for random 0.4 instances using Greedy Random solutions as a basis. The first value indicates the average performance increase in percent points (new performance − old performance), and the second value in the parenthesis indicates the average running time in seconds.

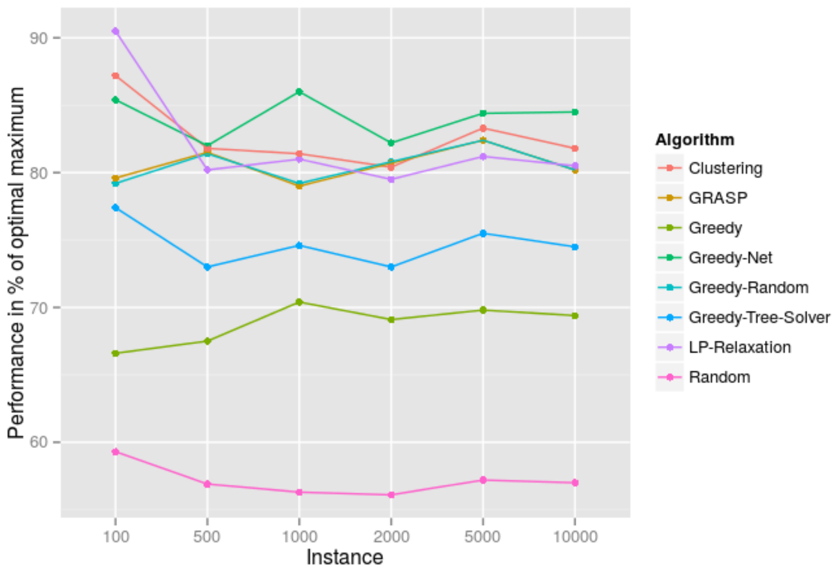


Figure 6.12: Results for the 1.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

6.3.6 Analysis

The performance results presented in section 6.3 show that some algorithms yield good results over all test instances.

Integer Linear Programming Formulations

The two integer linear programming models were tested with CPLEX and Gurobi. Obviously, all models optimally solved the test instances. In general Model 1 is faster than Model 2 in all instances, and CPLEX calculates the solution faster than Gurobi. Model 1 with CPLEX was solved in under 5 minutes for all instances, making this model suitable for non-artificial instances.

Greedy, Greedy Random and GRASP

Greedy Random produces better results than the deterministic Greedy version for all instances. The running time is mid-class for both variants. GRASP is producing equally good results as Greedy Random, however, within a much higher running time. This is due to the time needed by the local search techniques, which are hardly improving the quality of the instance. In general, the other heuristics are producing more promising results in less time.

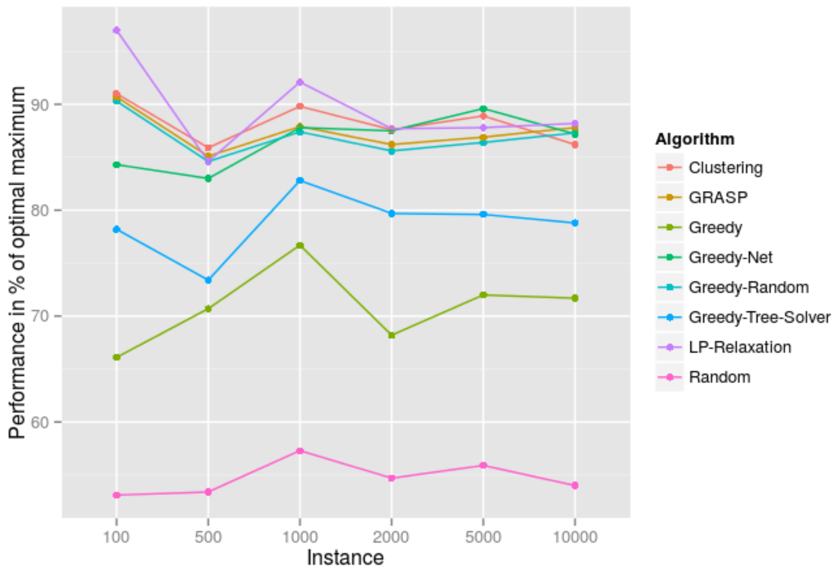


Figure 6.13: Results for the 1.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

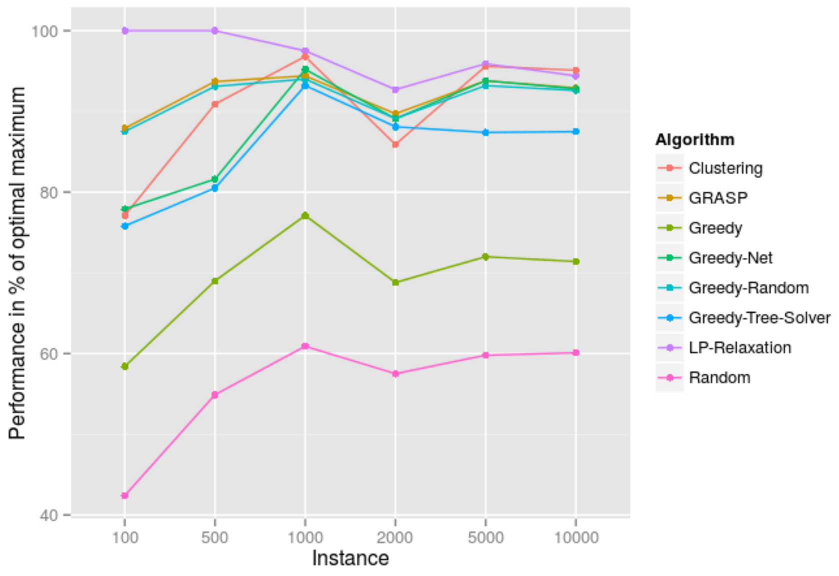


Figure 6.14: Results for the 1.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot.

LP Relaxation

The LP Relaxation is the second slowest heuristic, however, producing the second best results in general.

Greedy-Net

Greedy-Net is the best performing algorithm, both in the quality of the results and the time needed for the calculating the results. In ~90% of the instances Greedy-Net was the fastest and best performing algorithm, making it the most suitable algorithm for solving the instances.

Greedy Tree Solver

The Greedy Tree Solver is the second fastest algorithm, however, due to its characteristics it is producing especially suitable for solving trees, where it is producing the second best result. However, the results for the other instances were mediocre compared to the other heuristics.

Clustering

In general, Clustering is the second or third best performing algorithm, however, it was the slowest heuristic for all instances, making it only limited usable for solving the instances.

6.4 Performance Evaluation on an Austrian Network

The algorithms were tested on a created real world instance representing an Austrian communication network. It consists of 5264 vertices and 5272 edges. Therefore, the structure of this network is similar to a tree. For all vertices and edges the needed information was given, including the accrued data load `load`(v), customer value `val`(v), and customer potential `pot`(v) for each vertex, and the capacity `cap`(e) for each edge. The upgrade costs `cost`(v) for each vertex were not available, therefore an equal cost of 1 was assumed. Some of the vertices were already treated as sinks. Figure 6.15 illustrates the network, the shape of Austria an bigger cities with a dense population of vertices are clearly visible.

The algorithms were tested using the same parameters and settings explained in 6.2. In this test instance between 50 and 500 sinks were placed, in detail four different variants were tested, adding 50, 100, 250, and 500 additional sinks. Table 6.15 shows the computational results. The first value indicates the performance in % of the optimal maximum, optimal values achieved by the exact solution techniques have 100%. The value in the second columns shows the running time in seconds for solving the instance. The same data is shown for the post optimization local search techniques in table 6.16.

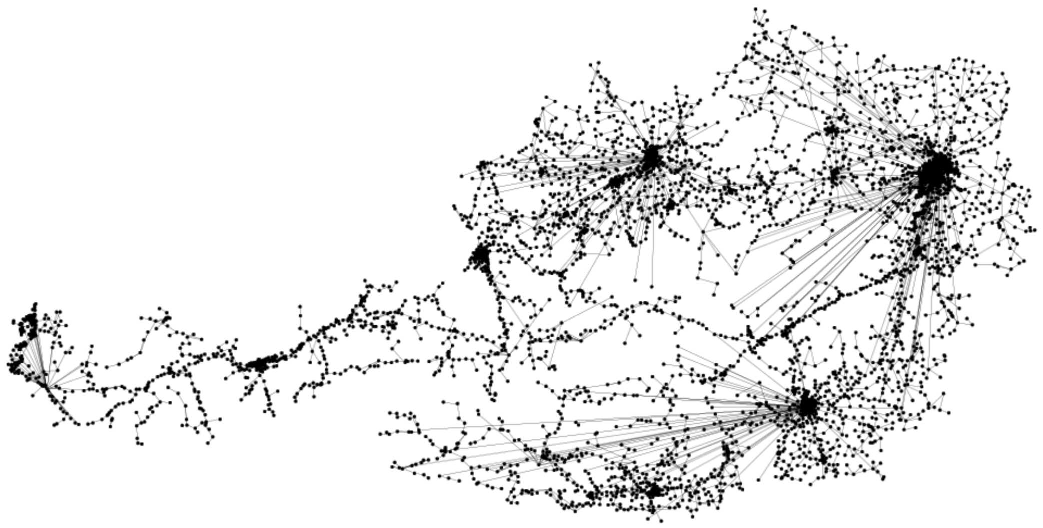


Figure 6.15: Telecommunication network in Austria.

The values are averages of 100 test runs on randomly generated solutions with Greedy Random.

Algorithm	50		100		250		500	
	opt	time	opt	time	opt	time	opt	time
CPLEX Model 1	100.0	200.4	100.0	61.4	100.0	54.9	100.0	2.0
CPLEX Model 2	100.0	3580.0	100.0	205.0	100.0	48.7	100.0	16.0
Gurobi Model 1	100.0	48.1	100.0	44.3	100.0	36.4	100.0	6.5
Gurobi Model 2	100.0	3580.0	100.0	1492.3	100.0	50.8	100.0	93.0
Greedy	50.9	3.2	52.6	2.9	68.5	5.3	89.7	8.9
Greedy Random	72.4	4.0	83.1	4.7	95.2	6.0	98.8	8.2
GRASP	73.2	141.1	84.0	148.8	95.8	171.0	98.8	196.7
LP Relaxation	82.8	51.4	88.9	27.3	97.3	47.2	99.1	29.4
Greedy-Net	62.2	1.5	77.6	2.3	96.8	3.0	99.0	2.6
Greedy Tree Solver	76.0	1.3	85.6	1.7	95.4	3.0	97.3	4.7
Clustering	**91.9**	14.0	**94.2**	14.1	**98.1**	15.9	**99.5**	18.8
Random	41.1	2.1	52.7	3.0	66.1	3.2	68.4	2.7

Table 6.15: Results for the Austrian network. The first value indicates the average performance in % of the optimal maximum, and the second value in the parenthesis indicates the average running time in seconds.

The results of the algorithms for the Austrian network instance differ from the artificial

Algorithm	50		100		250		500	
	opt	time	opt	time	opt	time	opt	time
Exchange Sinks Random	0.8	14.4	0.8	14.2	0.5	14.6	0.0	14.1
Exchange Sinks Z	0.7	18.2	0.6	18.4	0.5	18.0	0.0	18.5

Table 6.16: Results of the local search techniques for the Austrian network using Greedy Random solutions as a basis. The first value indicates the average performance increase in percent points (new performance − old performance), and the second value in the parenthesis indicates the average running time in seconds.

random instances. This could be due to a specific characteristic of the Austrian network instance, where specific vertices of the graph form the center of a (or part of a) graph with more connections to neighbouring vertices than average, as pictured in figure 6.16. This vertices are defined as stars (vertices with more than 2 neighbours), and predestined to be sinks, since due to their central position they are capable to satisfy more vertices than in general.

In the Austrian network instance 19.4% of the vertices are stars (with minimum 3 neighbours), and connected (including themselves) to 81% of all vertices. In the randomly generated instances only 5.5% of the vertices are stars (with minimum 3 neighbours) connected (including themselves) to 22% of all vertices. This characteristic could be an explanation of the slightly differing performance of algorithms between the randomly generated instances and the Austrian network instance, since algorithms which efficiently identify these star vertices as sinks are potentially capable to satisfy more vertices. See figure 6.17 presenting a diagram comparing the amount of star vertices for the artificial random instances and the Austrian network instance.

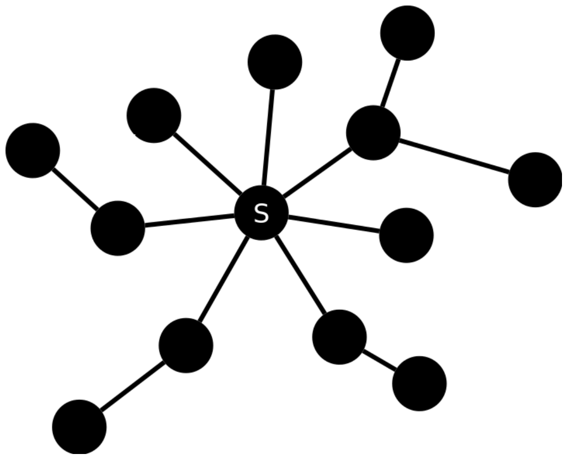


Figure 6.16: Example graph with a star vertex (S) in the center.

The integer linear programming models produce, of course, optimal solutions with 100%.

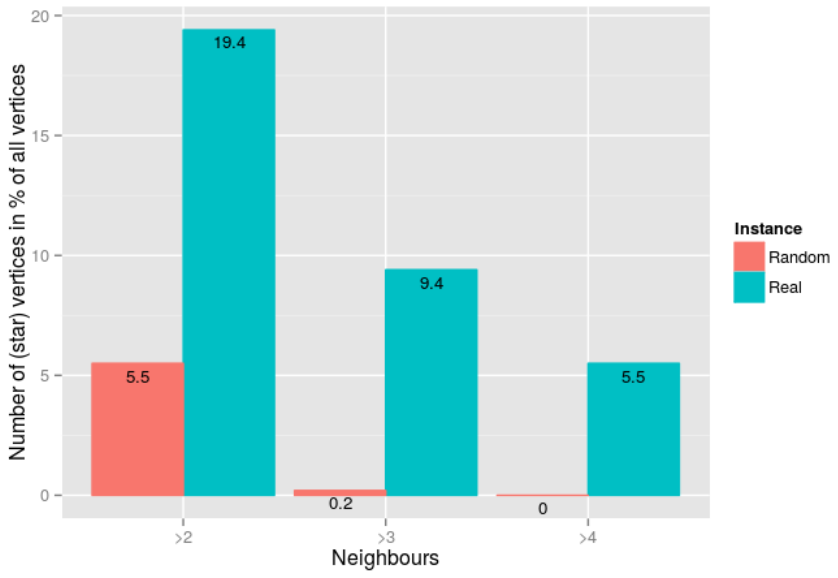


Figure 6.17: Number of star vertices with more than 2, 3 or 4 neighbouring vertices in % of all vertices for the artificial random 5000 vertices instance (Random) and the Austrian network instance (Real).

In general Model 1 had a much lower running time, about a factor of 8, than Model 2. It is interesting to see that CPLEX performed faster in Model 2, and Gurobi generally performed faster in Model 1. Placing 50 sinks optimally took the longest time, since the sinks had to be placed more carefully, whereas for placing 500 sinks the solution was calculated more than a 100 times faster than for 50 sinks. For this instance Gurobi with Model 1 is the best choice, producing the optimal solution in the shortest time. This is in contrast to the artificially created instances, in which CPLEX with Model 1 was better performing.

The Greedy Heuristic was tested using a deterministic (denoted as Greedy) and random (denoted as Greedy Random) version, where the random version outperforms the deterministic test run. However, the random results are mid-table compared to the other tested heuristics. The deterministic approach yields the worst results of all heuristics, being only around 9 to 20 percent points better than naively selecting the sinks randomly. Both test runs, the deterministic and random, nearly needed the same time to calculate the solution.

GRASP had the highest running time, while producing solutions which were maximum 1% point better than Greedy Random. This is due to the low performance of the local search techniques shown in table 6.16.

Calculating the solution with the LP Relaxation was the second slowest heuristic tested, even slower than calculating an exact solution with Gurobi and Model 1. The resulting solutions were better than average of all the heuristics, however, still up to 8% worse than the best performing heuristics.

The Greedy-Net heuristic was the second fastest calculating heuristic, however, yielding results which were only a few percent points, around 8 to 10%, better than the worst performing heuristic Greedy deterministic. This is in contrast to the results for the random instances and could be related to the specific characteristic of the Austrian network instance described above.

The Greedy Tree Solver produced the third best results within the fastest running time. The results for placing 50 and 100 sinks are mediocre, since the algorithm terminated before iterating all vertices. However, for placing 250 and 500 sinks the results are near the optimal value. This good performance is unexpected, since the Greedy Tree Solver is not taking specific parameters, such as customer potential, customer value, and upgrade cost into account for calculating the solution. A possible explanation could be that vertices with high customer value and customer potential are also predestined as sinks due to their connections and location.

The Clustering algorithm calculated the best solutions for the heuristics, being maximum 8% points worse than the optimal solution. Compared to the heuristics, the Clustering method is the third slowest algorithm needing around 10 times the time of the fastest algorithm, however, producing significantly better results.

Table 6.16 shows the performance of the local search techniques. The running time is around 14 seconds for the random local search and 18 seconds for the deterministic version. The results of Exchange Sinks Random are slightly better, around 0.1 percent points, than of Exchange Sinks Z.

CHAPTER 7

Conclusion

Telecommunications networks are a crucial and important part of modern life, and therefore, have to be planned and monitored carefully, to ensure enough available bandwidth for all customers. In order to satisfy the ever increasing demand for more bandwidth existing radio towers can be upgraded to act as sink for all accruing data. The network can then be considered as an undirected graph. This thesis proposes various heuristics and exact models for solving the problem of placing multiple sinks in an undirected flow network, a problem which is, for example, of special interest for telecommunication companies in order to decide where to upgrade the existing network. Each network consists of vertices and edges, with assigned properties, such as customer value, customer potential or costs. A given amount of vertices can be upgraded to sinks (which means these vertices get a connection to the backbone network for further processing of the data). The placement of these sinks has to be carefully chosen in regard to cost efficiency, and satisfying customer value, customer potential and the accruing data load.

In this thesis several algorithms and methods for solving this problem were presented. Two integer linear programming models, five heuristics, and one exact algorithm for solving a special case of instances. The methods were tested using artificial random test instances and a created Austrian telecommunication network instance. All algorithms are producing reasonable results, however differing depending on the type of instances used. Greedy-Net is the fastest heuristic, and producing the best results for the random instances. However, for the tested Austrian network instance other heuristics proved to be better performing, especially Clustering, producing mid-class results for the random instances, is producing the best results for the Austrian network instance. This varying results could be due to a specific differing property of the two instance types. The Austrian network instance has 4 times as much star vertices than the random instances.

The Greedy Tree Solver is producing optimal solutions for trees, not taking customer value, customer potential and upgrade costs into account. An according proof was presented. Future work could determine if the algorithm can be improved to handle graphs with circles, or planar graphs in general.

With the adaption of the parameters, the presented heuristics can be used for solving other similar problems. For example, finding a logistic center to upgrade among a network of logistic centers and customer locations. In this case only logistic centers would be valid candidates for sinks.

Furthermore, calculating the maximum flow is an important and time consuming part of three of the presented algorithms. Instead of using the exact methods, approximations can be used which only need a fraction of running time. In [CKM$^+$11] a possible usable algorithm is presented, using such methods will significantly reduce the running time at the expense of accuracy of the solution.

List of Figures

1.1 A telecommunications network, with the vertices X, Y, Z as part of the backbone network, and therefore acting as sinks. The remaining grey vertices represent ordinary radio towers, which need to route their data to the sinks. . 3
1.2 Possible flow of data from vertex X to a backbone connection. 4

3.1 Example graph and its extension with a supersource and supersink. 13
3.2 Example graph with all found sets, and the final remaining sets after all sets which were fully contained within another set were removed. 20
3.3 Example graphs where three sets were picked. 21
3.4 Two example iterations, where identically coloured vertices represent clusters and vertices marked with **C** the corresponding cluster centre. In the second iteration two cluster centres and their members changed. 23

4.1 Possibilities how to handle the accruing (produced) data load. Green framed circles represent sinks, and red arrows show possible flow directions 33
4.2 Valid and invalid tree 34

5.1 An undirected graph and its corresponding directed graph. 36
5.2 Maximum flow calculation with more than one source and sink. 37

6.1 Graph Generation Concept 41
6.2 Four versions of an example graph with 15 vertices. 42
6.3 Results for the 0.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 46
6.4 Results for the 0.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 47
6.5 Results for the 0.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 47

6.6 Results for the 0.1 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 52
6.7 Results for the 0.1 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 53
6.8 Results for the 0.1 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 53
6.9 Results for the 0.4 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 54
6.10 Results for the 0.4 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 54
6.11 Results for the 0.4 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 55
6.12 Results for the 1.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 57
6.13 Results for the 1.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 58
6.14 Results for the 1.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The linear programming models were omitted from this plot. 58
6.15 Telecommunication network in Austria. 60
6.16 Example graph with a star vertex (S) in the center. 61
6.17 Number of star vertices with more than 2, 3 or 4 neighbouring vertices in % of all vertices for the artificial random 5000 vertices instance (Random) and the Austrian network instance (Real). 62

List of Tables

5.1 Dinic's (DI) algorithm and Edmonds-Karp (EK) algorithm time (in s) 36

5.2 Time for a maximum flow calculation using an existing calculated flow graph (EX) and starting anew for each set sink (AN) (Dinic's algorithm as used method in both cases) (in s) 38

6.1 Greedy Heuristic Parameters 43

6.2 Results for the 0.0 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 45

6.3 Results for the 0.0 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 45

6.4 Results for the 0.0 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 46

6.5 Results for the 0.1 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 49

6.6 Results for the 0.1 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 49

6.7 Results for the 0.1 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 50

6.8 Results for the 0.4 instances with 10% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 50
6.9 Results for the 0.4 instances with 5% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 51
6.10 Results for the 0.4 instances with 2% sinks. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 51
6.11 Results for the 1.0 instances with 10% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 55
6.12 Results for the 1.0 instances with 5% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 56
6.13 Results for the 1.0 instances with 2% vertices. The first value indicates the average performance in % of the optimal maximum (opt), and the second value (time) indicates the average running time in seconds. The best heuristics performances are bold. 56
6.14 Results of the local search techniques for random 0.4 instances using Greedy Random solutions as a basis. The first value indicates the average performance increase in percent points (new performance − old performance), and the second value in the parenthesis indicates the average running time in seconds. 56
6.15 Results for the Austrian network. The first value indicates the average performance in % of the optimal maximum, and the second value in the parenthesis indicates the average running time in seconds. 60
6.16 Results of the local search techniques for the Austrian network using Greedy Random solutions as a basis. The first value indicates the average performance increase in percent points (new performance − old performance), and the second value in the parenthesis indicates the average running time in seconds. 61

List of Algorithms

3.1 LP Relaxation Rounding 17
3.2 Greedy-Heuristic 18
3.3 Random Greedy-Heuristic 19
3.4 GRASP 19
3.5 Greedy-Net-Heuristic 22
3.6 Clustering 23
3.7 Clustering Random 24
3.8 Exchange-Sinks-Depending-on-Z 25
3.9 Exchange-Sinks-Random 25

4.1 TreeSolver 29

Bibliography

[ABIK06] Bassam Aoun, R. Boutaba, Youssef Iraqi, and Gary Kenward. Gateway placement optimization in wireless mesh networks with qos constraints. *IEEE Journal on Selected Areas in Communications, (Volume:24, Issue:11, Pages:2127 - 2136)*, 2006.

[CKM$^+$11] Paul Christiano, Jonathan A. Kelner, Aleksander Mądry, Daniel Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. *STOC11 Proceedings of the forty-third annual ACM symposium on Theory of computing, (Pages:273-282)*, 2011.

[CQJM04a] Ranveer Chandra, Lili Qiu, Kamal Jain, and Mohammad Mahdian. On the placement of integration points in multi-hop wireless networks. *MSR Technical Report MSR-TR-2004-43*, 2004.

[CQJM04b] Ranveer Chandra, Lili Qiu, Kamal Jain, and Mohammad Mahdian. Optimizing the placement of integration points in multi-hop wireless networks. *IEEE International Conference on Network Protocols (ICNP), (Pages:271-282)*, 2004.

[Dan16] George Dantzig. *Linear programming and extensions.* Princeton university press, 2016.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, 3rd ed. edition, 2008.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 1970.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM. Association for Computing Machinery. (Volume:19 (2), Pages:248–264*, 1972.

[FKE11] Joakim Flathagen, Oivind Kure, and Paal E. Engelstad. Constrained-based multiple sink placement for wireless sensor networks. *Eighth IEEE International Conference on Mobile Ad-Hoc and Sensor Systems, (Pages:783-788)*, 2011.

[KWS$^+$11] Donghyun Kim, Wei Wang, Nassim Sohaee, Changcun Ma, Weili Wu, Wonjun Lee, and Ding-Zhu Du. Minimum data-latency-bound k-sink placement problem in wireless sensor networks. *IEEE/ACM Transactions on Networking (Volume:19, Issue:5)*, 2011.

[OE04] E. Ilker Oyman and Cem Ersoy. Multiple sink network design problem in large scale wireless sensor networks. *2004 IEEE International Conference on Communications, (Volume:6, Pages:3663-3667)*, 2004.

[PS08] Wint Yi Poe and Jens B. Schmitt. Placing multiple sinks in time-sensitive wireless sensor networks using a genetic algorithm. *Proceedings of the 14th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems, (Pages:1-15)*, 2008.

[SEHZ12] Haidar Safa, Wassim El-Hajj, and Hanan Zoubian. Particle swarm optimization based approach to solve the multiple sink placement problem in wsns. *IEEE ICC 2012 - Wireless Networks Symposium, (Pages:5445-5450)*, 2012.

[SGJ04] Jonatan Schroeder, Andre Pires Guedes, and Elias P. Duarte Jr. Computing the minimum cut and maximum flow of undirected graphs. Technical report, Federal University of Paraná - Dept. of Informatics, 2004.