Engineering Weighted Connectivity Augmentation Algorithms

Thomas Möller

December 19, 2023

3719463

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Prof. Dr. Felix Joos
Ernestine Großmann
Dr. Marcelo Fonseca Faraj

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor Prof. Christian Schulz for assisting and guiding me throughout this thesis with weekly feedback and also arousing my excitement for graph algorithms and algorithm engineering during my whole studies in the first place. Without him this work would not have been possible. I would also like to give a huge thanks to Dr. Marcelo Fonseca Faraj and Ernestine Großmann for their unwavering support and for always providing feedback and great ideas for any problem, as well as my co-supervisor Prof. Felix Joos for supporting me throughout this thesis. Last but not least, I would like to thank my family and friends who supported and encouraged me throughout my master's degree.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, December 19, 2023

Thomas Möller

Abstract

Increasing the connectivity of a graph is a fundamental problem in robust network design. The weighted connectivity augmentation problem (WCAP) is a common version of the problem that takes costs into account. A solution to the problem is a minimum cost subset of a given set of weighted links that increases the connectivity of a graph by one when added to the edge set. In this work, a first implementation of better-than-2 approximations only discovered recently is given. Furthermore, an optimal ILP and three new heuristic approaches are proposed. These include a greedy algorithm considering edge weights and the number of unique cuts covered, an approach based on minimum spanning trees and a local search algorithm that may improve a given solution by swapping links of paths. An experimental evaluation shows that the minimum spanning tree based algorithm is fastest and yields the best solutions, while the local search algorithm is still able to find small improvements on these solutions.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Many real-world coherences can be modeled as graphs, including technological, social, and biological networks. A common problem of interest is the robustness of such a graph. Particularly in technological networks this is important for creating systems that are robust and fail-safe [17]. An example is a power grid where single lines can fail, either randomly due to age, or by targeted attacks. If a line fails, alternative routes are used which is increasing the load on them and therefore the chance of failure. To obtain a fail-safe network that can survive both, random failures and targeted failures of important lines, the graph needs to be well-connected. Increasing the connectivity and therefore improving the robustness at minimum cost is known as connectivity augmentation or the survivable network problem. Another technological example is a computer network like the internet which should be designed in a fail-safe way while reliable transportation networks can avoid traffic congestion.

However, it is well-known that the weighted connectivity augmentation problem is NP-hard. Eswaran and Tarjan have shown that the decision problem, whether there is an augmentation of at most a given weight, is NP-complete [13]. Frederickson and Ja'Ja' showed that this is also true for the simpler special case where the graph is a tree, with weights being only 1 or 2 [16]. This justifies the importance of good heuristic algorithms. Furthermore, the weighted connectivity augmentation problem is APX-hard, which was also shown for the weighted tree augmentation problem by Kortsarz, Krauthgamer and Lee [27]. Despite the fact that there is no polynomial time approximation algorithm with an approximation factor arbitrarily close to 1, there has been much progress in improving the approximation ratio.

Recently, the connectivity augmentation problem has been discussed frequently in the context of approximation algorithms with approximation factors below 2 [5], [33], [39], [40]. This includes work on special cases like the tree augmentation

problem [39], as well as the general case [5], [40]. As of now, these algorithms are purely theoretical and there is no implementation or experimental results.

To the best of our knowledge, the only experimentally evaluated algorithms are by Watanabe et al. [46], [45], [44]. However, their work was done decades ago and algorithmic and technological progress may make better algorithms possible. Therefore, engineering weighted connectivity augmentation algorithms is a promising field for further research.

1.2 Contributions

The contribution of this thesis consists of three parts, namely giving the first implementation of two recent connectivity augmentation approximation algorithms with approximation factor below 2 from the literature, the proposal of new heuristic algorithms and the experimental evaluation of both.

The currently best known approximation algorithm is a $(1.5+\epsilon)$-approximation by Traub and Zenklusen [40]. The paper describes a greedy $(1 + \ln 2 + \epsilon)$-approximation and a local search based $(1.5 + \epsilon)$-approximation that use the same dynamic program as core part. For both approximations, the first implementation is given in this thesis. Furthermore, the computational complexity of $\mathcal{O}(n^{4\alpha+7}/\epsilon)$ using this implementation is given. An experimental comparison with an optimal ILP verifies the approximation ratios, but also shows the impact of the large exponent in the complexity. Although giving the best theoretical guarantees, the algorithms are irrelevant in praxis due to its computational complexity.

Next, three heuristic algorithms are proposed. The first algorithm is a simple greedy heuristic considering the cost of links and the number of cuts that a link crosses. A data structure that allows an efficient computation is provided. The next algorithm uses minimum spanning trees to find a feasible solution and greedily improves this solution by removing links. Last, a local search algorithm is presented that can improve a given solution by replacing link sets with cheaper ones. To find such sets, alternating paths with edges that are and are not in the solution are considered.

The approximation algorithms as well as the heuristic algorithms are experimentally evaluated using real-world graphs and generated graphs with randomly generated link weights. The ratios of the approximation algorithms are verified on generated instances and the limits of the exact ILP are evaluated. The heuristic algorithms are compared to each other in terms of solution quality and running time. The minimum spanning tree based algorithm is best with respect to both, quality and running time. Furthermore, different link weight distributions and link set sizes are tested. The best parameters for the local search algorithm are experimentally verified.

1.3 Structure

The remainder of this thesis is structured into six chapters. Chapter 2 states the fundamentals used in this thesis. Chapter 3 gives an overview of related work on the weighted connectivity augmentation problem itself as well as related problems. The approximation algorithms from the literature are described in Chapter 4 along with some implementation details. Our own algorithms contributed in this thesis are presented in Chapter 5. All the approximation algorithms in Chapter 4 and the algorithms described in Chapter 5 are experimentally evaluated in Chapter 6. Chapter 7 concludes this work by discussing results and possible future work.

CHAPTER 2

Fundamentals

This chapter introduces the preliminaries necessary for the weighted connectivity augmentation problem. Furthermore, the related minimum cut problem and the cactus graph representation of minimum cuts are described.

2.1 Graph and Partition

A graph $G = (V, E)$ is a structure that consists of a set of vertices V and a set of edges $E \subseteq V \times V$ connecting pairs of vertices. The number of vertices is denoted as n and the number of edges as m. For a vertex set $W \subseteq V$ the graph $G[W] = (W, \{(u, v) \in E \mid u, v \in W\})$ is called the *induced subgraph*. A graph G is *connected* if there is a path between any two vertices. The *edge connectivity* of a graph is the number of edge disjoint paths that exist between any pair of vertices. A graph is k*-connected*, if $k - 1$ arbitrary edges can be removed without disconnecting the graph. A vertex $a \in V$ of a connected graph G is an *articulation point* if the graph $G' = (V \setminus \{a\}, E)$ is disconnected. A partition of a graph is a partition of the vertex set into mutually disjoint non-empty sets. A *cut* of a graph is a partition of the vertex set into two disjoint subsets, also called a bipartition. For a more concise notation, any cut can be represented as one of its two constituent vertex sets, i.e. the complementary vertex set is implied. Every non-empty proper subset of V is a cut. The *size* or *weight* of a cut is the number of edges or the sum of the edge weights that have one endpoint in each subset.

2.2 Minimum Cut

A cut of a graph G is a *minimum cut* if there is no cut with smaller size or weight. The set of all minimum cuts is denoted as C_G and $\mathrm{cut} : C_G \times V^2 \to \{0, 1\}$ is a function that is 1 if and only if the endpoints u and v of an edge $e = (u, v) \in V^2$ lie in different sets of the

partition of a cut $c \in C_G$. To prevent different representations of the same cut using the notation where a cut is given as one set of the partition, only the representation that does not include an arbitrarily chosen root $r \in V(G)$ is used.

2.3 Cactus Graphs

Below the cactus graph is defined and some of its properties as well as the representation of minimum cuts are discussed.

Definition 1 (Cactus Graph)**.** *A cactus graph is a connected graph $C = (V, E)$, such that any two cycles have at most one vertex in common.*

Such a cactus graph consists of cycles that touch in at most one vertex and eventually vertices that do not lie in a cycle (but are connected by edges). To distinguish between edges that lie within a cycle and those that do not, they are called cycle edges and tree edges, respectively. Tree edges can also be seen as a cycle between two vertices, sometimes also defined as two parallel edges. Using this definition, a cactus graph can equivalently be defined as a connected graph where each edge lies in exactly one cycle. Vertices that belong to multiple cycles are articulation points of the cactus graph because removing the vertex disconnects the cycles it is contained in. Visually, a cactus graph is a tree of cycles where each cycle is a vertex and there is an edge between two vertices if the corresponding cycles share a vertex in the cactus. Figure 2.1 gives an example for a cactus graph.

Dinitz et al. [10] have shown that all minimum cuts of a connected graph G can be represented as a cactus graph $C = C(G)$.

Definition 2 (Cactus Graph Representation of Minimum Cuts)**.** *The cactus graph representation of the set of minimum cuts C_G of a graph $G = (V, E)$ is a cactus graph $C = C(G) = (V', E')$ with a function $\Pi : V \to V'$ and its inverse $\Pi^{-1} : V' \to 2^V$ defined as $v \mapsto \{u \in V : \Pi(v) = u\}$, such that the following conditions are fulfilled:*

$$\forall c \in C_C : \bigcup_{v \in c} \Pi^{-1}(v) \in C_G \tag{2.1}$$

$$\forall c_G \in C_G : \exists c_C \in C_C : \Pi(v) \in c_C \forall v \in c_G \tag{2.2}$$

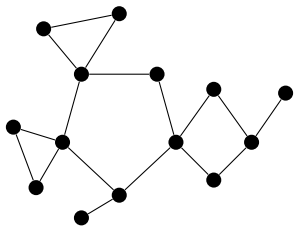


Figure 2.1: Example of a larger cactus graph

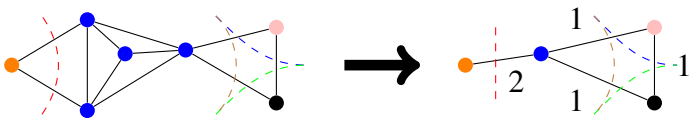


Figure 2.2: A graph and its weighted cactus graph with corresponding minimum cuts drawn as dashed lines of same color. Vertex colors encode the function Π.

The function Π ensures that each vertex of the original graph G is contained in exactly one vertex of the cactus graph C. (2.1) means that each minimum cut in C corresponds to a minimum cut in G while (2.2) ensures that each cut in G corresponds to at least one cut in C. This requires the cactus graph to have weighted edges; the weight of a tree edge corresponds to the minimum cut and the weight of a cycle edge corresponds to one half of the minimum cut, because a cut through a cycle cuts two edges. A graph with a corresponding cactus graph is illustrated in Figure 2.2. To transfer a cut from the cactus graph to the original graph (in set notation), one needs to transfer the partition with respect to Π^{-1}, i.e. a cut $c \subset V_C$ becomes $\bigcup_{v \in c} \Pi^{-1}(v)$. Note that there can be a vertex v such that $\Pi^{-1}(v) = \emptyset$ and that a minimum cut in G can correspond to multiple cuts in C if there are cactus vertices that do not contain vertices of G [32]. For instance, this is the case for a complete graph K, where each minimum cut separates an arbitrary single vertex. All minimum cuts can be represented as a star with center vertex v and $\Pi(v) = \emptyset$, and a vertex connected to the center for each vertex in K. The definition does not give a bound on the size of the cactus graph, but Dinitz et al. [10] showed that there is always a cactus graph representation with $\mathcal{O}(n)$ vertices with n being the number of vertices of G.

2.4 Weighted Connectivity Augmentation Problem

This section introduces the weighted connectivity augmentation problem (WCAP) and delimits it from variations not covered in this thesis.

Definition 3 ((Weighted Connectivity) Augmentation)**.** *Let $G = (V, E)$ be a k-connected graph, $L \subseteq V \times V$ a set of links and $c : L \to \mathbb{R}_{\geq 0}$ a cost function on the link set. A weighted connectivity augmentation is a set $S \subseteq L$ that minimizes* (2.3) *subject to* (2.4)*.*

$$\min_{S \subseteq L} \sum_{l \in S} c(l) \tag{2.3}$$

$$\text{s.t.} \quad (V, E \cup S) \text{ is } (k+1)\text{-edge-connected} \tag{2.4}$$

The goal of the weighted connectivity augmentation problem is to increase the edge connectivity of a graph. Given is a graph $G = (V, E)$ with edge connectivity k and a set $L \subseteq V^2$ of links with non-negative cost that can be added to the graph. The task is to find the cheapest subset $S \subseteq L$ that will increase the edge connectivity to $k + 1$.

A link $l \in L$ *covers* a minimum cut $c \in C_G$ if $\mathrm{cut}(c, l) = 1$, i.e. the size of the cut c in the graph $G' = (V, E \cup \{l\})$ is larger than in G. The graph $G_L = (V, L)$ that contains all the links (but not the edges of G) is called the *link graph*. The set of all sets S sufficing (2.4) is denoted as $S_{G,L}$. For the ease of notation the cost function is extended to sets, where it is the sum of the cost of all elements, i.e. $c : 2^L \to \mathbb{R}_{\geq 0}$ is defined as $S \mapsto \sum_{v \in S} c(v)$.

If the graph is disconnected, the weighted connectivity augmentation problem coincides with the minimum spanning tree problem (MST) among its components. In this thesis only connected graphs are considered, as the other case is simple to solve via well-known efficient MST algorithms.

There are many variations of the weighted connectivity augmentation problem, that are not covered in this thesis. These include modifications from the following non-exhaustive list.

- The graph G can have edge weights, such that not only the connectivity, but a weighted minimum cut is increased.
- Parallel links are allowed, i.e. links parallel to already existing edges can be added.
- There is a required connectivity demand between any pair of vertices instead of a global minimal connectivity. This is also known as the survivable network design problem.
- The link set is always complete, allowing further assumptions.

2.5 Approximation Algorithms

An approximation algorithm is an algorithm with polynomial running time that guarantees a solution close to the optimal solution. It is commonly used to obtain approximate solutions of NP-hard problems. A ρ-approximation is an approximation algorithm with approximation factor ρ guaranteeing that the solution deviates at most by the factor ρ from the optimal solution.

Let f be an objective function, I be an arbitrary input, $x(I)$ the solution of the approximation and $x^*(I)$ the optimum solution. A minimization algorithm has the approximation factor ρ if the following holds:

$$\frac{f(x(I))}{f(x^*(I))} \leq \rho \; \forall I$$

For the weighted connectivity augmentation problem, an algorithm A is a ρ-approximation if the weight of the augmentation $c(A(I))$ is at most $\rho \cdot c(x^*(I))$ for all instances I.

CHAPTER 3

Related Work

This chapter presents prior work done on the weighted connectivity augmentation problem, as well as closely related problems. First, the state of the art for computing all minimum cuts of a graph in the cactus graph representation is presented, followed by research and implementations in the field of connectivity augmentation problems and the current state of (integer) linear program solvers.

3.1 Minimum Cuts

Computing all minimum cuts is usually a fundamental step in connectivity augmentation. Near-minimum cuts can be computed in linear time based on cluster contractions [21]. Nagamochi, Nakao and Ibaraki presented an efficient algorithm to compute all minimum cuts in the cactus graph representation [32]. They observed that all minimum cuts between two vertices s and t can be computed by running a maximum s-t-flow algorithm, and edges that are cut by no minimum cut can be contracted. In each iteration of the algorithm a maximum s-t-flow is computed for an edge $e = (s, t)$, and either e is contracted, or the graph is bi-partitioned such that the cactus graph can be computed in both parts independently. A detailed description can be found in [32].

The current state of the art algorithm is VieCut by Henzinger, Noe, Schulz and Strash [22], [23]. It uses linear time edge contraction based reduction rules and an optimized version of the algorithm by Nagamochi, Nakao and Ibaraki. An edge can be contracted if the connectivity of its endpoints is larger than the minimum cut. Such edges could be found by computing k edge-disjoint spanning trees where k is the size of the minimum cut [22], [31]. Furthermore, reduction rules by Padberg and Rinaldi [35] were adapted from the problem of finding one minimum cut to the problem of finding all minimum cuts. Lastly, edges that form a trivial minimum cut are contracted and remembered. These cuts are reintroduced at the end of the algorithm. The reduction rules are used exhaustively as

long as a significant number of edges is contracted. The remaining kernel is solved based on the algorithm by Nagamochi, Nakao and Ibaraki [32].

3.2 Connectivity Augmentation

This section gives related research for the weighted connectivity augmentation problem, including approximation algorithms, randomized algorithms and experimentally evaluated heuristic algorithms.

3.2.1 Approximations

There have been several approximation algorithms for the weighted connectivity augmentation problem in the past. An early approach is using minimum cost arborescences, which was introduced by Frederickson and Ja'Ja' for bridge connectivity augmentation, the case where the graph is 1-connected but not 2-connected [16] The algorithm was generalized for the weighted connectivity augmentation problem by Watanabe et al. [44]. The idea is to compute a minimum cost arborescence in a graph that contains links as well as directed versions of edges of the original graph. The intuition is that for an augmentation a path needs to be added between vertices u and v that are not $(k+1)$-connected. However, this path must not entirely consist of new links, but can also use backward edges of the original graph rooted at an arbitrary vertex. This is shown in Figure 3.1, where edges of the original graph are drawn as solid edges and new links are drawn as dashed edges. After adding the dashed links to the graph, there is a path from u to v and vice versa using links and backward edges, and the edge connectivity is increased from one to two. The bridge connectivity algorithm results in a 2-approximation while the generalization cannot guarantee an approximation factor.

There are, however, well-known 2-approximations for WCAP. One possibility, which was discovered in 1992, reduces the problem to a directed version by replacing each undirected edge of the graph with two directed edges in opposite direction [26]. The directed version can be solved in polynomial time based on minimum-cost flows [15] or by using a linear program which has integral solutions for the cactus augmentation problem [6]. Another approach involves the LP relaxation of an ILP formulation combined with iterative rounding techniques [25].

Only recently, progress beyond an approximation factor of 2 has been made with various approximation algorithms regarding special cases of the connectivity augmentation

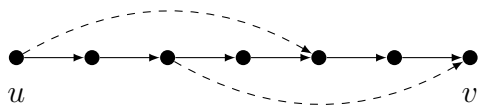


Figure 3.1: Directed augmenting path

problem, as well as the general case. For the unweighted version of the connectivity augmentation problem the first approximation with factor below 2 was found in 2020 by Byrka, Grandoni and Ameli [5], [33]. The unweighted connectivity augmentation problem is reduced to the steiner tree problem, for which a specialized approximation gives an approximation factor of 1.91.

For the tree augmentation problem, the special case where the cactus graph is a tree, and the unweighted connectivity augmentation problem an approximation factor of 1.393 was found in 2021 [6]. For the weighted tree augmentation problem, a $(1 + \ln 2 + \epsilon)$-approximation was discovered [39], which builds upon the 2-approximation which reduces the problem to a directed one, and greedily improves this solution. The algorithm was transferred to the weighted connectivity augmentation problem and refined to a $(1.5 + \epsilon)$-approximation [40], which improves an arbitrary solution through local search. However, no implementations or experimental results of those algorithms exist so far. For the weighted connectivity augmentation problem an implementation for both, the $(1+\ln 2+\epsilon)$-approximation and the $(1.5+\epsilon)$-approximation, is given in this thesis. Therefore, a detailed description follows in Chapter 4.

3.2.2 Randomized Algorithms

There has been recent work on randomized Monte Carlo algorithms for the weighted connectivity augmentation problem that give a solution with high probability on graphs with integer edge weights. Algorithms based on maximum flow computations achieve a running time of $\tilde{\mathcal{O}}(m + \sqrt{n^3})$ where $\tilde{\mathcal{O}}(f) = \mathcal{O}(f \cdot \text{polylog}(f))$ abstracts logarithmic factors [7]. They were able to solve the connectivity augmentation problem by running a logarithmic number of maximum flow computations. The state of the art is an $\tilde{\mathcal{O}}(m)$ time algorithm that gives a near-linear running time by Cen, Li and Panigrahi [8]. This shows that the connectivity augmentation problem is simpler than the maximum flow problem as there is no known $\tilde{\mathcal{O}}(m)$ time maximum flow algorithm.

3.2.3 Experimentally Evaluated Algorithms

Many theoretical algorithms lack an implementation and practical results. In praxis there can be a large gap between theoretical algorithms and efficient implementations. It is therefore important to transfer algorithms to actual hardware and do experiments with real data. This is particularly true for fixed-parameter algorithms, where Abu-Khzam, Lamm, Mnich, Noe, Schulz and Strash [1] described possible techniques. There have been fast, practically applicable heuristic algorithms for WCAP in the past, however, there has not been much progress recently. Watanabe, Mashima and Taoka [30], [44], [45], [46] proposed five different approaches, called FSA, MW, FSM, SMC and HBD, including experimental evaluation. An observation used for all algorithms is that there is a subset of all vertices of the cactus graph called leafs that must be an endpoint in any augmentation. A vertex of a cactus is a

leaf if it has degree 1 or if it is part of a cycle and has degree 2. Vertices that are not a leaf of the cactus do not necessarily have to be an endpoint in an augmentation.

The algorithm FSA uses minimum cost arborescences based on the ideas of Frederickson and Ja'Ja' mentioned in the previous section [16]. MW is a stronger algorithm also based on arborescences that guarantees a 2-approximation. FSM is based on maximum cost matchings. In the cactus graph, a link must be added to every leaf vertex v, otherwise there is still a minimum cut, namely $\{v\}$. To find a solution with as few links as possible while still minimizing their weight, a maximum weight matching algorithm along with a special cost function is used. The third approach, SMC, is a greedy strategy adding the cheapest incident link for each vertex of the cactus graph. HBD combines FSM and SMC and tries to use the best of both concepts.

Experimental results on random graphs with up to 1400 vertices showed that the solution quality of FSM is the best, followed by HBD, SMC, FSA and lastly MW [30], [44]. Regarding running time, SMC is the fastest algorithm, followed by FSA, HBD, FSM and lastly MW. HBD is considered the best general algorithm, because it prevents arbitrary bad solutions that may be produced by FSM or SMC. MW is the only algorithm with a guaranteed approximation factor, however, in practice it is slower and the solution quality of the other algorithms is better [30].

To the best of our knowledge, no other experimentally evaluated algorithms are mentioned in the literature. Furthermore, there is a variety of random graph models that have different features present in real-world graphs. Penschuck et al. [36] present aspects of generating different graph models at a large scale in a survey. There are no experimental results using such generated instances.

3.3 (Integer) Linear Programming

Linear programs (LPs) are part of the basic algorithm toolbox and have been heavily studied in the last decades. For a long time the algorithm by Vaidya [42] based on the interior point method was the fastest linear program solver for dense matrices achieving a bound of $\mathcal{O}(n^{2.5}\log(n/\delta))$ where δ is a numerical accuracy. This bound comes from $\sqrt{n}$ iterations of the multiplication of an approximate matrix with a vector, which has $\Omega(n^2)$ complexity for dense vectors. Recently, Chen et al. [9] have proposed a randomized algorithm with complexity $\tilde{\mathcal{O}}(n^\omega \log(n/\delta)) = \mathcal{O}(n^\omega \log(n/\delta)\,\mathrm{polylog}(n))$ where ω is the exponent for matrix multiplication, currently $\omega \approx 2.37$. This was achieved by sparsifying the vector through random sampling. Brand [43] gave a deterministic version of the algorithm also achieving a complexity of $\tilde{\mathcal{O}}(n^\omega \log(n/\delta))$, in particular $\mathcal{O}(n^\omega \log^2(n)\log(n/\delta))$. The key difference is that not only the approximate matrix is maintained, but also the product with the sparse vector.

Integer linear programs (ILPs) are NP-complete in general, which includes mixed integer linear programs with integer and continuous variables and binary (integer) linear programs, where variables can have the value 0 or 1. A general approach to solving those problems

is using the branch and bound paradigm examining possible solutions in exponential time and was first introduced by Land and Doig [28]. The running time of the branch and bound algorithm could be improved by finding a good heuristic solution in the beginning such that more branches can be cut off. A further improvement is a presolve step, which applies reductions eventually fixing variables if their value in an optimal solution can be known, removing redundant constraints or tightening the bounds of variables or constraints [2]. This step aims at improving the model definition before the ILP solver starts solving the problem. There are reductions considering individual constraints / variables, a set of constraints / variables or the whole problem. Most recent progress has been made using the concept of cutting planes. Cutting planes were introduced by Gomory in 1958 [19]. First, the LP relaxation is solved by dropping the integrality constraints. Then, infeasible solutions including fractional values are excluded by adding additional constraints. However, the method is not guaranteed to find an optimal solution and there are many possibilities how these constraints are selected. Selections have influence on solution quality and the number of cuts necessary. Heuristics can be used to determine good cuts. A recent approach uses deep reinforcement learning to select cuts [38]. They came to the conclusion that reinforcement learning can do better decisions than heuristics in a similar running time.

CHAPTER 4

Approximation Algorithms

This section describes two approximation algorithms with approximation factors $(1+\ln 2+\epsilon)$ and $(1.5+\epsilon)$ by Traub and Zenklusen [40] for which a first implementation is given in this thesis. As a prerequisite a 2-approximation is also given [6]. Some technical details as well as correctness proofs are left out and can be looked up in [40].

4.1 LP-based 2-Approximation

As mentioned in Section 3.2.1, one way to achieve a 2-approximation is to reduce the problem into a directed one, solve this easier problem and transfer the solution back to the original problem. For the reduction each undirected link $l = (u, v)$ is replaced by two directed links $l_1 = (u, v)$ and $l_2 = (v, u)$, resulting in a set $\vec{L}$ with $|\vec{L}| = 2 \cdot |L|$. A directed solution is transferred to the undirected problem by replacing each link in the directed solution with the undirected one while removing duplicates. To solve the reduced problem, the linear program based algorithm is implemented. This approach was first proposed by Jain [25], and Cecchetto, Traub and Zenklusen provided a definition where the solution is integral for any cactus augmentation instance [6].

To define the connectivity augmentation problem in a directed setting, let C_G be the set of cuts using the set representation not including a fixed root $r \in V(G)$. A cut $C \in C_G$ is only increased by a link $\vec{l} = (u, v)$ if $v \in C$ and $u \notin C$, i.e. the link $\vec{l}$ is entering the cut C. The function $\vec{\text{cut}} : C_G \times \vec{L} \to \{0, 1\}$ is 1 if and only if $v \in C$ and $u \notin C$. This is a weaker formulation as every incoming and outgoing link increases a cut in the undirected problem. Equation (4.1) and (4.2) give the LP using this definition. The constraints in (4.2) ensure that each cut is covered by a link in the augmentation.

$$\min_{x} \sum_{l \in \vec{L}} x_l c(l) \tag{4.1}$$

$$\text{s.t.} \quad \sum_{l \in \vec{L}} \vec{\text{cut}}(c, l) x_l \geq 1 \; \forall c \in C_G \tag{4.2}$$

$$x \in [0, 1]^{|\vec{L}|}$$

Complexity

The number of variables of the linear program is equal to the number of directed links, which is twice the number of undirected links. This can be $\mathcal{O}(n^2)$ in case of a near-complete graph. The number of constraints corresponds to the number of minimum cuts, which is bounded by $\mathcal{O}(n^2)$ in the case where the graph is a ring. Using Brand's $\mathcal{O}(N^{2.37} \log^2 N \log N/\delta)$ algorithm [43] where N is the number of variables, this results in a complexity of $\mathcal{O}(n^{4.74} \log^2 n \log n/\delta)$.

The space complexity follows from the constraint set. $\mathcal{O}(n^2)$ constraints may use up to $\mathcal{O}(n^2)$ variables, which results in a matrix with $\mathcal{O}(n^4)$ entries. However, links may not cover many cuts, which results in a sparse matrix.

4.2 Relative Greedy $(1 + \ln 2 + \epsilon)$-Approximation

Traub and Zenklusen [40] presented a greedy approach to obtain a better undirected solution from a directed solution than a 2-approximation as described in Section 4.1. This section briefly describes the algorithm. More details including correctness proofs can be found in [40]. This section first gives an outline of the algorithm and then describes the single steps in more detail.

Algorithm Outline

On a high level, the algorithm first does an exact reduction of the cactus graph of the problem to a ring graph. It starts with a directed solution of this ring graph that has at most two times the optimum weight.

In this solution, directed links are replaced by shorter versions called shadows that have the same weight as the original links and together still form an augmentation (this means a solution to the original problem can be found by selecting the original links of the same weight later). Such a shortened solution forms an arborescence whose structure is useful.

Next, sets of directed links are greedily replaced by sets of undirected links called components, resulting in mixed solutions that contain directed as well as undirected links and

still depict an augmentation. If all directed links are replaced by undirected ones, the solution is a solution to the original problem. The greedy objective is the ratio of the cost of the added undirected links and the cost of the directed links that are not needed anymore.

That is, however, difficult to compute. This is why only link sets that can be constructed iteratively by a dynamic program are considered. Furthermore, it is easier to check if a given ratio is better or worse than the optimum. The algorithm uses binary search with respect to the ratio to determine the optimum along with a set of links that achieves this ratio. For each bisection a dynamic program is run.

The individual steps of the algorithm including some implementation details are described in more detail in the next subsections.

Reduction to a Ring Graph

Gálvez, Grandoni, Ameli and Sornat first gave an approximation preserving reduction from a cactus graph to a ring graph [18]. An easy algorithm using Eulerian walks is given in [40]. As a cactus consists of cycles that are connected in at most one vertex, an Eulerian walk can easily be found. Going along this Eulerian walk, every time a vertex is visited multiple times, a new vertex is introduced along with a link of weight 0 to the original vertex. All vertices with the edges of the Eulerian walk result in a ring graph. An example is shown in Figure 4.1 where corresponding vertices have the same color and links of weight 0 that were introduced are drawn as dashed lines.

Adding a link of weight zero when visiting a vertex twice can be seen as the inverse of an edge contraction as described in [18]. Contracting all links that were added would result in the original cactus graph. Therefore, a solution of the ring graph can be transferred to a solution of the cactus graph by removing links of weight 0.

To efficiently store a ring graph, vertices are relabeled in the order they appear in the ring. This has the following advantages:

- Edges can be stored implicitly, i.e. there is an edge between two vertices u and v if $|u - v| = 1$ or if $\{u, v\} = \{0, n - 1\}$.

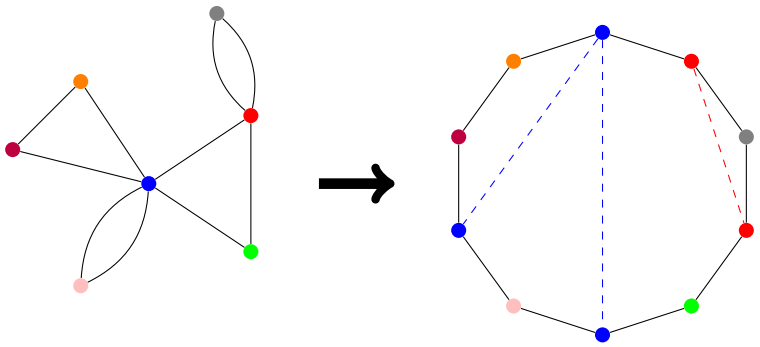


Figure 4.1: Reduction from a cactus graph to a ring graph

- It gives the ring an implicit root at vertex 0 and a direction which will be needed later. There is also no need for a BFS or DFS in the ring graph, as a path between two vertices is just the sequence of vertices between their IDs.

- Cuts can be represented as an interval on the ring, making it only necessary to store the first and last vertex while still being able to check in constant time on which side of the cut a given vertex is. In this section, a cut C is defined as the set of vertices in its interval.

Shortened Directed Solutions

A directed solution of a ring graph G with weight at most two times the optimum of the undirected problem can for instance be found using the linear program described in Section 4.1. The result is an arbitrary set of links in the ring. To obtain a more structured solution, links are shortened with respect to a root r and a direction of the ring. The direction is given by an edge $e_r = (r, w)$. In the data structure described above r is always 0 and e_r is always $(0, n-1)$. A *shortening* of a directed link (u, v) is another directed link (u', v) that has the same endpoint, and the source is a vertex on the unique uv-path in $G \setminus e_r$. Such a link is also called *shadow*. For an example see the orange link (u, v) in the top left graph and its shortening (u', v) in the top right graph in Figure 4.2. The length of the shadow is $v - u'$, i.e. the distance of the vertices on the ring. Computing a shortening of the solution is done by going over all directed links in an arbitrary order and replacing them with the shortest shadow such that the solution is still an augmentation. If the shadow has length 0, it is dropped and not necessary. To find a shortening of a link, all candidates need to be checked in increasing order by length to determine if the result is still an augmentation, using the first valid one. The result is a spanning arborescence $\vec{F}_0$, rooted at vertex 0 in the data structure described in the previous subsection. Figure 4.2 gives an example of how a directed solution is shortened.

Checking whether replacing a link $l = (s, t)$ with a shorter shadow $l' = (s', t)$ still results in an augmentation can be done in a naive way by checking if all cuts are still entered by at least one link. To keep the running time of checking a replacement linear in the number of cuts and independent of the number of links, the number of links entering each cut is stored. The replacement results in a valid augmentation if every cut that is crossed by l but not l' is covered by at least two links including l. If a link is replaced by a shadow, the counts need to be updated, which is also linear in the number of cuts.

Finding Components with Optimum Ratio

The shortened directed solution $\vec{F}_0$ is converted step by step into an undirected one by replacing a subset of the remaining directed links $\vec{F}_i$ in step i with a set K_i of undirected links. The ratio of the cost of K_i and the gain of replaced links is minimized. The formula

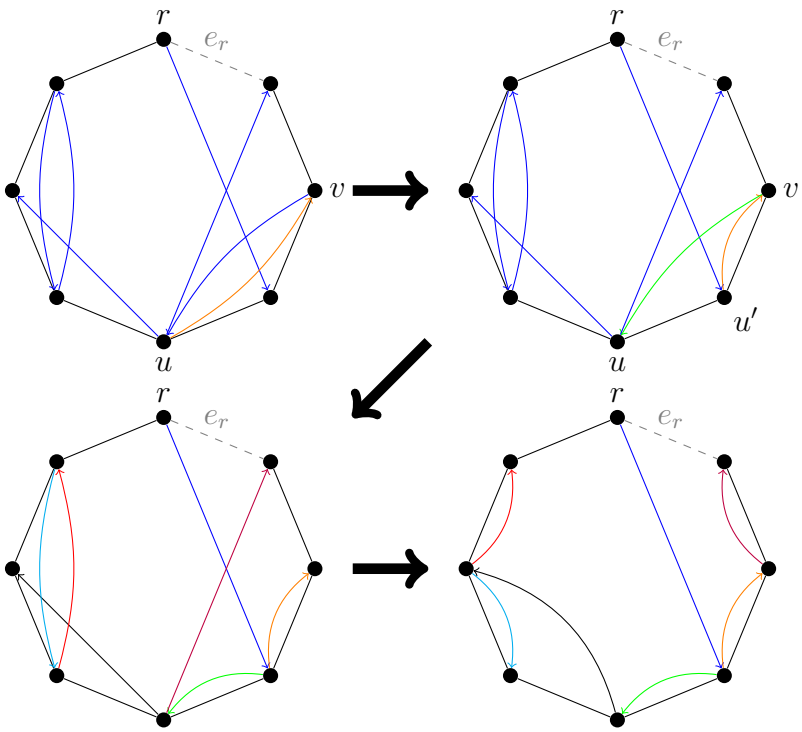


Figure 4.2: A ring graph with directed solution (top left) and a shortening of the links (u, v), (v, u) and the remaining links in three steps

is given in (4.3) where $\mathrm{Drop}_{\vec{F}_0}(K_i)$ is the set of directed links that are covered by links in K_i with respect to the shortened solution $\vec{F}_0$.

$$\rho = \frac{c(K_i)}{c(\mathrm{Drop}_{\vec{F}_0}(K_i) \cap \vec{F}_i)} \tag{4.3}$$

To ensure a polynomial running time of the algorithm, K_i is only picked from a component class $\mathfrak{K}$ that ensures K_i is α-thin as stated in Definition 4.

Definition 4 (α-thin)**.** *Let C_G be the set of minimum cuts of a graph G and $\alpha \in \mathbb{Z}_{>0}$. A set $K \subseteq L$ is α-thin if there exists an inclusion-wise maximal laminar family $\mathcal{L} \subseteq C_G$ such that at most $alpha$ links cross C for all $C \in \mathcal{L}$.*

α-thinness ensures that only a constant number of links crosses relevant cuts. A family of sets is *laminar* if every two sets are either disjoint or one is a subset of the other, i.e. $\forall C, D \in \mathcal{L} : C \cap D = \emptyset$, $C \subseteq D$ or $C \supseteq D$. Visually, because $\mathcal{L}$ is maximal, Definition 4 requires that all cuts $\bigcup_{v \in V \setminus \{r\}} \{v\}$ can iteratively be combined to the cut $V \setminus \{r\}$, such that all intermediate cuts are crossed by at most α links. The maximal laminar family $\mathcal{L}$ does not need to be known, but by construction during the dynamic program α-thinness of sets K will be ensured later. α depends on the desired approximation ratio, namely

$\alpha = 4\lceil\frac{2}{\epsilon}\rceil$. Note that the approximation ratio can only be improved if $(1 + \ln 2 + \epsilon) < 2$, requiring $\epsilon < 1 - \ln 2$ and therefore $\alpha \geq 28$.

The minimum $\rho^* = \min\left\{\frac{c(K_i)}{c(\mathrm{Drop}_{\vec{F}_0}(K) \cap \vec{F}_i)}\right\}$ is hard to compute directly, but it turns out that for a fixed ρ' it can be computed in polynomial time whether $\rho' > \rho^*$ or $\rho' \leq \rho^*$. For that, a slack function is introduced:

$$\mathrm{slack}_\rho(K) := \rho \cdot c(\mathrm{Drop}_{\vec{F}_0}(K) \cap \vec{F}_i) - c(K) \tag{4.4}$$

The slack function is maximized in (4.5) using a dynamic program described later in this section.

$$\eta := \max\{\mathrm{slack}_\rho(K) : K \in \mathfrak{K}\} \tag{4.5}$$

Note that $\rho \in [0, 1]$ because the ratio cannot be negative and there is always a K with ratio 1, i.e. replacing a directed link with its undirected version, which will become clear in the next subsection. The slack function is constructed such that η is 0 if $\rho = \rho^*$, and $\max\{\mathrm{slack}_\rho(K) : K \in \mathfrak{K}\} > 0 \Leftrightarrow \rho > \rho^*$. To compute ρ^* along with a set K, binary search with respect to ρ is applied where in each step a dynamic program is executed to solve (4.5). If the interval is sufficiently small (for integral weights, the interval size is $\frac{1}{c(F)^2}$, otherwise weights need to be scaled up), the link set K computed by the dynamic program corresponds to a minimizer of (4.3).

Computing Droppable Links

To be able to compute η of (4.5), it is necessary to compute $\mathrm{Drop}_{\vec{F}_0}(K)$ for a link set K, i.e. the set of links that are dropped by K. A link (u, v) covers entering the subtree at v in the arborescence $\vec{F}_0$. It can therefore be dropped if all sets including v and any subset of its descendants are covered by K (and do not form a 2-cut). The set of descendants is always an interval on the ring (colored in orange for the link l in Figure 4.3).

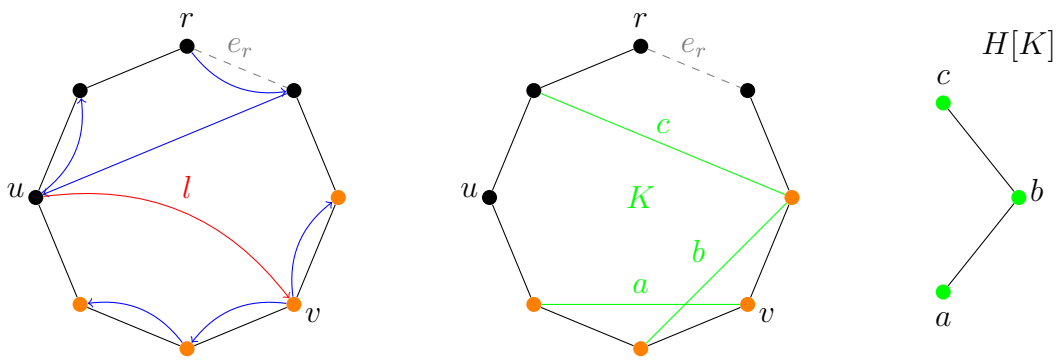


Figure 4.3: A shortened solution (left), a link set K (green in the middle) dropping link l in the shortened solution and the link intersection graph $H[K]$ (right)

To check if a link l is in the drop in a well-defined way, the link intersection graph of K is introduced. Two links are called *intersecting* if they interleave on the ring or if they share an endpoint. In Figure 4.3, a and b are intersecting and b and c are intersecting, but not a and c. The link intersection graph H has the links L as vertex set and has an edge between two vertices if they are intersecting. The link intersection graph of $K \subseteq L$ is the induced subgraph $H[K]$.

A link $l = (u, v)$ is droppable if and only if v is connected to an ancestor of itself in the link intersection graph $H[K]$, where connected means that a link incident to v is connected to a link incident to an ancestor of v. In Figure 4.3 l is dropped by K because a is incident to v, c is incident to an ancestor of v and a is connected to c in $H[K]$.

Checking if a directed link (u, v) is dropped by a link set K is reduced to a BFS in the link intersection graph $H[K]$ of size $|K|$ with multiple sources and multiple targets. The sources are links incident to v and the targets are links having an ancestor of v as endpoint.

Maximizing Slack through a Dynamic Program.

The central part of the algorithm is a dynamic program to find good components that can be added to the solution. The dynamic program optimizes (4.5), maximizing the slack function (4.4).

Entries in the dynamic programming table are addressed by a pattern $P = (C, B)$, where $C \in C_G$ is a cut and $B \subseteq L$ is a set of links crossing C. Each entry stores auxiliary data including a set S called realizer for the pattern, corresponding to the link set K, and an objective value π, corresponding to the value of the slack function for the set S. For the set S only links that have at least one endpoint in C are considered, i.e. $S \subseteq \{(u, v) \in L : \{u, v\} \cap C \neq \emptyset\}$. S consists of links crossing C and links that have both endpoints in C. B is the subset of S that has exactly one endpoint in C, i.e. the subset of links crossing C. For a shorter notation the subset of links of a set S that have exactly one endpoint in a set C is denoted as $\delta_S(C)$. In case of a set $\vec{S}$ of directed links, the notation $\delta^-_{\vec{S}}(C)$ depicts the subset of links entering the cut C.

The objective π keeps track of the cost and gain achieved regarding the cut C, considering only directed links that have their target endpoint in C. The function is defined in (4.6) and for $C = V \setminus \{r\}$ it is the same as the slack function (4.4).

$$\pi(S, C) := \rho \cdot \left(\mathrm{Drop}_{\vec{F}_0}(S) \cap \bigcup_{v \in C} \delta^-_{\vec{F}_0}(v) \right) - c(S) \tag{4.6}$$

The dynamic programming table has a 3-dimensional structure. The first dimension is the size of the cut such that all entries with a specific cut size can easily be accessed, the second dimension is the cut itself, and the third dimension is the set B. The first two dimensions are arrays indexed by the cut size and the first vertex in the cut, respectively. The third dimension is a hash map indexed by B.

The first row of the dynamic programming table contains all patterns where $|C| = 1$, i.e. patterns where the cut is cutting off exactly one vertex. The pattern must be α-thin, which means $|B| \leq \alpha$. Therefore, all possibilities of B can be enumerated for the first row, resulting in a polynomial number of patterns. Note that S is equal to B because $|C| = 1$.

To fill the following rows 2 to $(n-1)$ of the dynamic programming table, corresponding to the size of C, previous patterns are combined. Each row still has polynomial size because the number of cuts of size k is linear, and the exponent for the number of possibilities for B is bounded by the constant α. A pattern can be reached through different combinations with different realizers S, but only the one with the highest objective π is of interest. To be able to combine two patterns (C_1, B_1) and (C_2, B_2), they must be compatible, as defined in Definition 5.

Definition 5 (Compatible)**.** *Two patterns (C_1, B_1) and (C_2, B_2) are compatible if the following three conditions are fulfilled.*

1. *C_1 and C_2 are neighboring, meaning $C_1 \cap C_2 = \emptyset$ and $C_1 \cup C_2 \in C_G$*
2. *$\delta_{B_1}(C_2) = \delta_{B_2}(C_1)$, i.e. the subset of links between C_1 and C_2 must be in B_1 and B_2*
3. *$|\delta_{B_1 \cup B_2}(C_1 \cup C_2)| \leq \alpha$, ensuring α-thinness*

Combining two patterns (C_1, B_1) and (C_2, B_2) with objectives π_1, π_2 and realizers S_1, S_2 to a new pattern (C, B) with objective π and realizer S can, on a high level, be done in the following steps:

1. $C = C_1 \cup C_2$
2. $B = \delta_{B_1 \cup B_2}(C_1 \cup C_2)$, i.e. the union of B_1 and B_2 without links having both endpoints in $C_1 \cup C_2$
3. $S = S_1 \cup S_2$
4. $\pi = \pi_1 + \pi_2 + c(B_1 \cap B_2) + \rho \cdot c(R)$, where $c(B_1 \cap B_2)$ is the weight of links in B that have one endpoint in C_1 and one endpoint in C_2 and R is a set of additionally droppable links. Details on how R is computed can be found in [40].

By combining patterns and therefore cuts of size one to $V \setminus \{r\}$, the existence of a maximal laminar family $\mathcal{L}$ and therefore α-thinness and $K \in \mathfrak{K}$ is ensured. When all rows of the dynamic program are computed, the maximizer of (4.6) can be found by picking a dynamic programming table entry with maximum objective π.

Complexity

The $(1 + \ln 2 + \epsilon)$-approximation has a polynomial running time, however, it is very expensive. The most expensive part is the dynamic program for which the size of the dynamic

programming table is analyzed first. The first row of the table (cuts of size one) involves enumerating all subsets of size at most α of links outgoing from a vertex v. There can be at most $n-3$ links, one to every other vertex except the neighboring ones. This leads to up to $(n-3)^{\min(\alpha,n-3)}$ table entries for each vertex except the root. The size of the first row is therefore $(n-1) \cdot n'^{\min(\alpha,n')}$ with $n' = n-3$. For subsequent rows B can be chosen from a larger set of links crossing C. For row and cut size k there can be $k \cdot (n-k) - 2$ crossing links. The number of cuts of size k decreases with increasing k, because the root vertex is never part of a cut and the first vertex of a cut must be in $\{1, \ldots, n-k\}$. This leads to a row size of $(n-k) \cdot n'^{\min(\alpha,n')}$ with $n' = k \cdot (n-k) - 2$. The total number of entries in the dynamic programming table is:

$$\sum_{k=1}^{n-1} (n-k) \cdot n'^{\min(\alpha,n')} \text{ with } n' = k \cdot (n-k) - 2$$

Assuming a non-tiny graph with $n > \alpha + 2$ and therefore $\alpha < k \cdot (n-k) - 2 \; \forall k$, this can be simplified:

$$\sum_{k=1}^{n-1} (n-k) \cdot (k \cdot (n-k) - 2)^{\alpha} < \sum_{k=1}^{n-1} n \cdot \left(\frac{n^2}{4}\right)^{\alpha} < n^2 \cdot (n^2)^{\alpha} = \mathcal{O}(n^{2\alpha+2}) \tag{4.7}$$

Recall that the algorithm gives a $(1 + \ln 2 + \epsilon)$-approximation and $\alpha = 4\lceil \frac{2}{\epsilon} \rceil$, such that $\alpha \geq 28$ for an approximation ratio below 2. For graphs with at most α vertices the size of the dynamic programming table as well as the running time is therefore exponential in n.

Now the computational complexity is analyzed. Computing the dynamic programming table involves computing the first row by enumeration and combining all feasible entries for subsequent rows, where the combinations depict the significant part. An upper bound can be given by the number of all possible combinations, $\mathcal{O}\left(\binom{n^{2\alpha+2}}{2}\right) = \mathcal{O}(n^{4\alpha+4})$. Although for each row of the dynamic programming table entries are combined, every distinct combination is only checked for a single row due to the required size of the cut. This results in the complexity $\mathcal{O}(n^{4\alpha+4} \cdot T_{\text{combine}})$.

When combining two entries, the necessary conditions in Definition 5 need to be checked, and the combined pattern needs to be computed. The conditions can be checked in $\mathcal{O}(n + \alpha \ln \alpha)$ time; checking if cuts are neighboring is constant work, the sets B_1 and B_2 can have $\mathcal{O}(n)$ size and $\delta_{B_1}(C_2)$ and $\delta_{B_2}(C_1)$ can have size α and need to be compared. Combining two patterns involves constructing a link intersection graph with vertices $B_1 \cup B_2$ of size $\mathcal{O}(n)$ and iterating over all vertex pairs to determine edges. Details on how patterns are combined can be found in [40]. Combining two patterns has therefore the complexity $T_{combine} = \mathcal{O}(n^2)$. Note that this is the worst case and in many cases patterns are not compatible and do not need to be combined. An upper bound for the complexity of a dynamic program execution is therefore $\mathcal{O}(n^{4\alpha+4} n^2) = \mathcal{O}(n^{4\alpha+6})$.

The algorithm iteratively adds link sets K_i to the solution and drops directed links from $\vec{F}_0$. In the worst case only one directed link is removed in each step while $\vec{F}_0$ has $n-1$ links.

Finding a component K_i with optimum ratio requires a binary search with respect to the ratio $\rho \in [0, 1]$ until the interval has at most the size $1/c(F)^2$ (assuming integer or upscaled weights). Therefore, $\log_2(c(F)^2) = \mathcal{O}(\ln c(F)) \overset{c(F)\leq 2OPT}{=} \mathcal{O}(\ln(OPT))$ steps are needed, where each step maximizes the slack function (4.5) using the dynamic program. In total there are at most $\mathcal{O}(n \ln(OPT))$ dynamic program invocations.

An upper bound for the total complexity is therefore given by $\mathcal{O}(n^{4\alpha+6} n \ln(OPT)) = \mathcal{O}(n^{4\alpha+7} \ln(OPT))$ with $\alpha \geq 28$ for an approximation ratio below 2.

4.3 Local Search $(1.5 + \epsilon)$-Approximation

The state-of-the-art approximation algorithm by Traub and Zenklusen [40] is a $(1.5 + \epsilon)$-approximation where a detailed description and correctness proofs can be found in. The algorithm is based on the ideas and the dynamic program of their relative greedy $(1+\ln 2+\epsilon)$-approximation described in the previous section.

The main difference is that the algorithm does not only greedily replace all links of a directed solution with undirected links. Instead, replaced links should iteratively improve the solution and can themselves be replaced in further iterations.

The algorithm can start with an arbitrary solution. To reduce the number of iterations, it is useful to start with a solution that is already good. This can for instance be the 2-approximation from Section 4.1 that was also used for the $(1 + \ln 2 + \epsilon)$-approximation.

A high-level outline is given in Algorithm 1. First, the cactus graph is reduced to a ring graph in line 3 as described in the relative greedy $(1 + \ln 2 + \epsilon)$-approximation. Then, an initial undirected solution F is computed in line 4 using the LP-based 2-approximation from Section 4.1.

The dynamic program requires a directed, shortened solution $\vec{F}$ of F, which is computed by bidirecting F and shortening the result as described for the relative greedy approximation. The directed versions of links $l = (u, v) \in F$ (or their shadow) are called *witnesses*. The set of up to two witnesses of l is called the *witness set* W_l. If for any link $l \in F$ both directed links are removed from $\vec{F}$ and the witness set W_l becomes empty, l is also removed from F.

To keep track of the progress in reducing the weight of the augmentation F that was already made, a special objective function $\Phi : 2^L \to \mathbb{R}_{\geq 0}$ is introduced in (4.8). This function gives a higher weight to a link $l \in F$ if shadows of both directed versions are in $\vec{F}$ than to links where only one shadow of directed versions is in $\vec{F}$. This follows the intuition that it is easier to replace just one directed link than two.

$$\Phi(F) := \sum_{l \in F : |W_l| = 2} \frac{3}{2} \cdot c(l) + \sum_{l \in F : |W_l| = 1} c(l) \tag{4.8}$$

While the objective Φ decreases significantly by a factor of at least $\left(1 - \frac{\epsilon}{6n}\right)$, link sets are replaced. The dynamic program computes a link set K maximizing the gain in Φ.

Algorithm 1 $(1.5+\epsilon)$-Approximation

input Cactus graph $C=(V,E)$, L_C, $c_C : L_C \to \mathbb{R}_{\geq 0}$
output augmentation $S \subseteq L$
procedure $(1.5+\epsilon)$-Approximation(G, L, c)
  $G, L, c \leftarrow$ reduction of C, L_C, c_C to a ring graph
  $F \leftarrow$ any augmentation of G, i.e. a 2-approximation
  $\vec{F} \leftarrow$ bidirect F
  $\vec{F} \leftarrow$ shortening of $\vec{F}$
  remove links from F that have no shadow in $\vec{F}$
  do
    $K \leftarrow 4\lceil\frac{4}{\epsilon}\rceil$-thin link set through dynamic program
    remove all links from F and $\vec{F}$ that can be dropped by adding K
    add K to F and K bi-directed to $\vec{F}$
    $\vec{F} \leftarrow$ shortening of $\vec{F}$
    remove links from F that have no shadow in $\vec{F}$
  while $\Phi(F)$ decreases at least by factor $\left(1-\frac{\epsilon}{6n}\right)$
  return undirected version of $\vec{F}$ without duplicates

Further details follow below. All directed links in $\vec{F}$ that are dropped by K are removed, also removing their correspondents from F if the witness set becomes empty. Then, K is added to F and a bidirected version of K to $\vec{F}$. This results in an arbitrary solution, which can be shortened again to be able to start with the next iteration.

Dynamic Program

The dynamic program works very similar to the dynamic program of the $(1+\ln 2+\epsilon)$-approximation. The key difference is a different objective function, such that the difference of the function Φ measuring the progress is maximized. In particular, instead of the slack function $\text{slack}_\rho(K) := \rho \cdot c(\text{Drop}_{\vec{F}_0}(K) \cap \vec{F}_i) - c(K)$ given in (4.4), the Function (4.10) is maximized with an adjusted cost function $\overline{c}$ for directed links as defined in (4.9). $\overline{c}(\vec{F})$ gives the cost of an undirected version of a directed link set $\vec{F}$ without duplicates.

$$\overline{c}(\vec{F}) := \sum_{l \in \vec{F}:|W_l|=2} \frac{1}{2} \cdot c(l) + \sum_{l \in \vec{F}:|W_l|=1} c(l) \tag{4.9}$$

$$\overline{c}(\text{Drop}_{\vec{F}_i}(K)) - 1.5 \cdot c(K) \tag{4.10}$$

Using this cost function, the dynamic program is able to compute a link set reducing Φ as far as possible. Note that unlike the $(1+\ln 2+\epsilon)$-approximation this iterative approach does not require binary search.

Complexity

The main part of the algorithm, the dynamic program, is the same as for the $(1 + \ln 2 + \epsilon)$-approximation. Therefore, the size of the dynamic programming table is bounded by $\mathcal{O}(n^{2\alpha+2})$ and the running time of one dynamic program invocation is bounded by $\mathcal{O}(n^{4\alpha+6})$ with $\alpha = 4\lceil\frac{4}{\epsilon}\rceil > 32$ for an approximation ratio below 2. The maximum number of iterations after which the algorithm terminates is given in (4.11) [40].

$$\ln\left(\frac{1.5 \cdot c(F_0)}{c(OPT)}\right) \cdot \frac{6n}{\epsilon} \tag{4.11}$$

If the algorithm starts with a 2-approximation, this is bounded by $\frac{\ln(3)\cdot 6n}{\epsilon} < \frac{7n}{\epsilon}$. This results in a total running time of $\mathcal{O}(n^{4\alpha+7}/\epsilon)$. In contrast to the $(1 + \ln 2 + \epsilon)$-approximation this is independent of the (upscaled) augmentation weight OPT.

CHAPTER 5

Engineering Weighted Connectivity Augmentation Algorithms

This chapter first describes the data structures required for the algorithms. Then, an exact algorithm and heuristic approaches for the weighted connectivity augmentation problem are described. As WCAP is NP-hard, exact algorithms will only be able to solve small instances. Unlike the approximation algorithms in the previous chapter, the heuristics cannot give guarantees on the solution quality, but aim at being fast or giving good solutions in many real-world cases.

5.1 Data Structures

This chapter describes data structures that will be used in different algorithms. First, the data necessary regarding the original graph and the cactus graph is discussed. Second, a dynamic cactus graph data structure is introduced, that maintains an updated cactus graph while links are being added.

Graph Data Structure

All minimum cuts of a graph G can be represented as a (potentially significantly smaller) cactus graph C as described in Section 2.3. Therefore, it is sufficient to do computations on the cactus graph C. To be able to give an augmentation for the original graph G, or to output an augmented graph G', G must be known. To achieve this, G is stored in adjacency list representation along with an array modeling the function $\Pi : V(G) \to V(C)$ and using vertex IDs as indices.

Additionally, the link set L must be transferred to a link set L_C in the cactus graph C. A link $l = (u, v)$ is translated to a link $l_C = \Pi(l) := (\Pi(u), \Pi(v))$ in the cactus graph. To be able to reverse this function and obtain a link in G again, the endpoints of the original edge

l are stored as well. Multiple vertices of G may be mapped to a single vertex in C. This can lead to different links $g, h \in L, g \neq h$ with $\Pi(g) = \Pi(h)$, i.e. to parallel edges in the link graph G_{L_C} (with not necessarily equal weight). The following proposition shows that all parallel links in G_{L_C} can be dropped except for one of smallest weight.

Proposition 1. *Let C be a cactus graph representing all minimum cuts of a graph G, L_C a set of links in C such that L_C is an augmentation, and G_{L_C} be the link graph of L_C. Let $L'_C \subseteq L_C$ be the set $\bigcup_{(u,v)\in V(C)^2} \operatorname{arb}(\operatorname{argmin}_{l=(u,v)\in L_C} c(l))$ that contains an arbitrary link of minimum weight for each set of parallel edges of G_{L_C}. Then there is a minimum weight connectivity augmentation of C that only uses links in L'_C.*

Proof. Assume for a contradiction that every minimum weight connectivity augmentation A includes a link not in L'_C. There can be two reasons:

1. The augmentation contains two links g and h that are parallel in L_C. As they have the same endpoints in C, they are cut by the same set of minimum cuts and therefore increase the same set of cuts and $A \setminus \{h\}$ is also an augmentation. ↯

2. The augmentation contains a link $l \notin L'_C$. Because of the definition of L'_C a parallel link l' of equal or less weight must be in L'_C. This link increases the same set of cuts, which means $A \setminus \{l\} \cup \{l'\}$ is also an augmentation of equal or less cost. ↯

Applying those cases iteratively, an augmentation $A' \subseteq L'_C$ can always be found. □

Using Proposition 1 the link set L_C can be stored in an adjacency matrix only keeping an arbitrary link of minimum weight per vertex pair.

Dynamic Cactus

Some algorithms will iteratively add links to a solution and require an up-to-date set of minimum cuts. Explicitly storing and updating this set is expensive, but the cactus representation can be updated rather efficiently. A similar approach of updating the cactus was proposed by Henzinger, Noe and Schulz in [24], where a union find data structure is used to keep track of the function Π that associates each vertex of G with a vertex of the cactus C. To be able to provide more information, i.e. computing the number of minimum cuts that a given link crosses, a different data structure is implemented.

When adding a link $l = (u, v)$ that crosses minimum cuts, the cactus must shrink. Dinitz has shown that cuts on the uv-path are affected [11]. In particular, vertices that lie on every uv-path in the cactus must be contracted to update the cactus [24]. Those can be found by computing a path in the tree of cycles. That is the graph that contains all cycles as vertices and has an edge between two vertices if the corresponding cycles share a vertex. There is a unique path because the graph is a tree. For every cycle in the path, the shared endpoints need to be contracted (at the end points of the path u and v are contracted). If $u \neq v$,

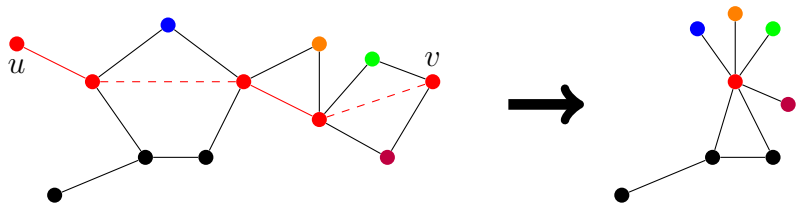


Figure 5.1: Updating a cactus graph by contracting a path (red) after inserting an edge (u, v). Dashed edges do not exist in the cactus graph

at least one contraction is done and the graph becomes smaller. An example is shown in Figure 5.1.

To be able to quickly find the path in the tree of cycles, this graph is maintained by the data structure. The cactus graph is implicitly stored and could easily be constructed. In particular the following three lists are maintained.

1. The list of vertices, `node_list`. Each vertex corresponds to a cycle in the cactus graph. The index of a vertex in the list corresponds to its ID while the value at the position is the ordered list of vertices in the cycle. This is necessary to be able to reconstruct the cactus graph and determine the cycles that emerge if two vertices are contracted.

2. The adjacency lists, `edge_lists`. Two cycles are adjacent if they share a vertex. Because this common vertex is needed to contract a path, it is stored along with the edge. If two cycles i and j are connected via a vertex v, the list with index i contains a pair of the target cycle j and the common vertex v, and vice versa.

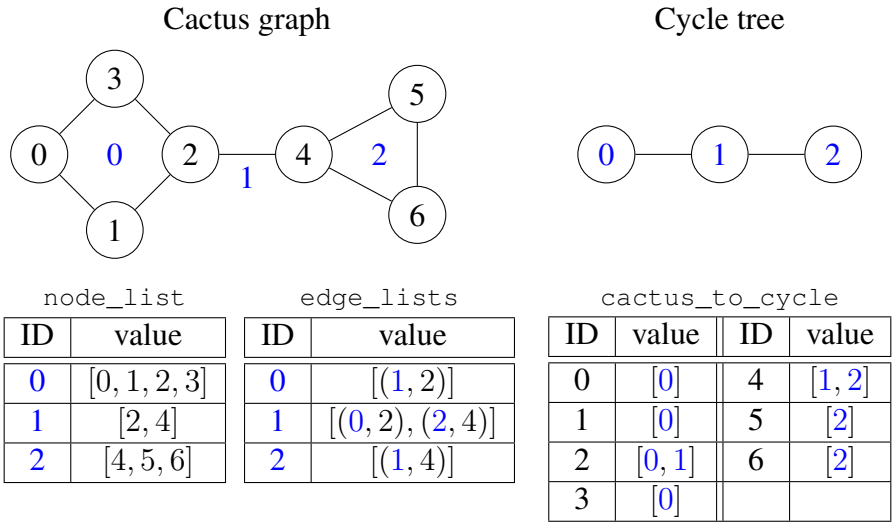
3. A mapping `cactus_to_cycle`, that stores for each cactus graph vertex the list of cycles it is contained in. This is necessary to efficiently determine which vertices need to be contracted without updating every link after a contraction.

Figure 5.2 gives an example of how the data structure looks for a small cactus graph. IDs of cactus graph edges are written in black and IDs of cycles are written in blue. Using only the first value of pairs in `edge_lists`, this is a commonly used adjacency list graph data structure.

In this graph data structure the unique path in the cycle tree can easily be computed using a DFS. Next, the edges in the path are contracted one after the other. When contracting two vertices u and v there can be three cases that need to be handled differently.

1. u and v are the only vertices in the cycle. In this case the cycle disappears and the cycle tree looses a vertex. To keep the data structure valid, the last vertex is moved to the position of the removed vertex.

2. u and v are connected by an edge in the cactus graph, but the cycle has at least three vertices. In this case the number of cycles does not change and v can be removed



`node_list`

ID	value
0	$[0, 1, 2, 3]$
1	$[2, 4]$
2	$[4, 5, 6]$

`edge_lists`

ID	value
0	$[(1, 2)]$
1	$[(0, 2), (2, 4)]$
2	$[(1, 4)]$

`cactus_to_cycle`

ID	value	ID	value
0	$[0]$	4	$[1, 2]$
1	$[0]$	5	$[2]$
2	$[0, 1]$	6	$[2]$
3	$[0]$		

Figure 5.2: Example of data structure for dynamic cactus graph

from the list of vertices in the cycle. However, edges of the cycle graph that have u or v as connecting vertex need to be updated. Note that new links are added if there is a cycle attached to u and a cycle attached to v because the cycles now share the contracted vertex uv.

3. u and v are not connected by an edge in the cactus graph. In this case a cycle is split into two cycles, where the new cycle is appended to the list of vertices. Similar to the previous case, edges between cycles containing u or v need to be updated.

The main goal of this data structure is the ability to count the number of cuts that a given link crosses and improve the $\mathcal{O}(n^2)$ complexity of checking every possible cut. A cut always cuts two edges of the same cycle, therefore each cycle can be considered separately. As a link $l = (u, v)$ affects all cuts on the uv-path [10], all cycles on the uv-path need to be considered. In Figure 5.1 each cycle containing a red link has to be considered for counting the number of cuts that the link $l = (u, v)$ crosses. These cycles can be found by doing a BFS in the cycle graph. In case of a tree edge there is only one minimum cut. An example is the leftmost edge in Figure 5.3. Otherwise, there are two vertices a_1 and a_2 that lie on every uv-path and there are two edge-disjoint a_1a_2-paths p_1 and p_2. In Figure 5.3 this is the cycle in the center with a_1 and a_2 being the articulation points and p_1 and p_2 being the upper and the lower part of the cycle. Picking any edge from p_1 and any edge from p_2 results in a minimum cut that is covered by l. The number of covered minimum cuts through the cycle containing a_1 and a_2 is therefore the product of the path lengths, $|p_1| \cdot |p_2|$. The number of minimum cuts covered by l is hence the sum of these products over all edges (cycles) in the uv-path. Figure 5.3 also shows all minimum cuts covered by l. To determine the lengths $|p_1|$ and $|p_2|$ in the cycle i, the distance d of a_1 and a_2 in the vertex list `node_list[i]` is computed. $|p_1|$ is equal to the distance d, while $|p_2| = |\texttt{node\_list[i]}| - d - 1$.

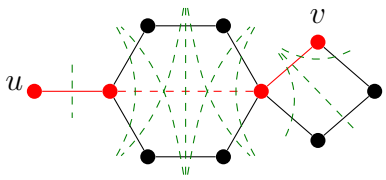


Figure 5.3: Cuts (green) covered by a link $l = (u, v)$

Complexity

The contraction of two vertices in the same cycle (i.e. an edge of the path in the cycle tree) may require updating the whole data structure in all three cases described above. In the worst case this is $\mathcal{O}(n^2)$ work in the case of a star, where every two cycles share the center vertex as common vertex. However, a contraction can be done in linear or even constant time in many cases, i.e. when contracting two vertices that are no articulation points. The total number of contractions is bounded by the number of vertices in the cactus graph because each contraction reduces the number of vertices in the cactus graph by one. In the worst case, contracting the whole cactus into a single vertex has $\mathcal{O}(n^3)$ complexity.

The main reason for this data structure is the ability to compute the number of cuts that a given link crosses. The complexity for the BFS in the cycle tree is bounded by $\mathcal{O}(n)$ because the structure is a tree and there cannot be more cycles than vertices in the cactus graph. Counting the number of cuts that the computed path crosses can be done in linear time too; for each edge in the path one needs to go through all vertices contained in the corresponding cycle to find the distance of the articulation points. As this is done at most once per cycle, the overall work required is $\mathcal{O}(n)$.

5.2 Integer Linear Program

This section gives an optimal algorithm that can be used to solve small instances. The optimal solution will later be used to measure the quality of non-optimal algorithms. Furthermore, the maximum problem size that an optimal solver is able to solve is of interest.

A commonly used approach to solve a problem to optimality is an (integer) linear program (ILP). To define an integer linear program for the weighted connectivity augmentation problem, $|L|$ binary variables are introduced that decide which links are added to the augmentation. The objective given in (5.1) sums up the weight of all selected links. For each

minimum cut in the graph G there is a constraint assuring that at least one link increasing the cut is added to the augmentation as depicted in (5.2).

$$\min_{x} \sum_{l \in L} x_l c(l) \tag{5.1}$$

$$\text{s.t.} \quad \sum_{l \in L} \text{cut}(l) x_l \geq 1 \; \forall c \in C_G \qquad x \in \{0,1\}^{|L|} \tag{5.2}$$

Improvements

A common approach for solving integer linear programs is a branch and bound algorithm. If the algorithm starts with a good initial solution, non-optimal branches can be discarded faster. Therefore, using the solution of a heuristic solver as initial solution can speed up solving the integer linear program.

Another approach used by integer linear program solvers is a presolve step which tries to eliminate redundant variables and constraints. This optional step applies different reduction rules with different computational complexity. Different configurations can lead to different results in terms of solution time and memory requirements.

Complexity

The number of variables is bounded by $\binom{n}{2} - |E| = \mathcal{O}(n^2)$ if the graph $G' = (V, E \cup L)$ is complete. The number of constraints is bounded by $\binom{n}{2} = \mathcal{O}(n^2)$ because in a cactus graph each minimum cut is cutting at most two edges. This occurs if the cactus is a cycle, where any two edges form a cut while $|E| = |V| = n$. In case of a cycle and a complete link set, on average each constraint uses $\left(\sum_{i=1}^{n-1} i(n-i)\right)/n = \mathcal{O}(n^2)$ variables. In the worst case, the ILP has therefore $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n^2)$ constraints with $\mathcal{O}(n^4)$ non-zeros in a matrix representation.

5.3 Heuristic Algorithms

In this section algorithms that do not give an exact solution are covered. These include greedy heuristics, an algorithm based on minimum spanning trees as well as a local search algorithm.

5.3.1 Greedy Heuristics

This section introdues greedy strategies that pick one link $l \in \text{argmin}_{l \in L} h(l)$ after the other with respect to a heuristic h and add it to the augmentation until all minimum cuts have been increased. Possible heuristics are explained in the following subsections.

Weight Heuristic

As the problem aims at minimizing the cost of an augmentation, it is natural to use the weight of a link as heuristic. An additional constraint is that the link must cover at least one cut, otherwise it can be left out from the solution. This leads to the heuristic $h_w : L \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ in (5.3).

$$h_w(l) = \begin{cases} c(l) & \text{if } l \text{ covers at least one cut} \\ \infty & \text{else} \end{cases} \tag{5.3}$$

A variation of this heuristic picks an arbitrary uncovered minimum cut c and only considers links that cover c. This leads to heuristic $h_w^c : L \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ in (5.4). For each link selection a new cut $c \in C_G$ is picked arbitrarily resulting in a different heuristic function.

$$h_{w,c}(l) = \begin{cases} c(l) & \text{if } l \text{ covers } c \\ \infty & \text{else} \end{cases} \tag{5.4}$$

Weight-Coverage Heuristic

This heuristic also favors links with small weight, but at the same time the number of links needed is minimized in order to prevent large solutions. To take this into account, the number of minimum cuts covered by a link can be used, i.e. the number of cuts where the link has one endpoint on each side of the cut. This leads to the heuristic $h_{wc} : L \to \mathbb{R}_{\geq 0}$ in (5.5), which computes the cost per augmented minimum cut.

$$h_{wc}(l) = \frac{c(l)}{|\{c \in C_G : \text{cut}(c, l) = 1\}|} \tag{5.5}$$

Implementation

A naive implementation computes the set C_G of all minimum cuts. Then, the heuristic h, either $h_w(l)$, $h_w^c(l)$ or $h_{wc}(l)$, is computed for every link $l \in L$. A link with the lowest objective is added to the solution and cuts covered by l are removed from C_G. This is repeated until the set of cuts is empty.

A more sophisticated algorithm for the heuristic h being h_w or h_{wc} uses the dynamic cactus data structure. Instead of storing the set of all cuts, a copy of the cactus graph is kept and updated if a link is added. Recall that the number of cuts that a link l crosses can be computed by doing a BFS in the cycle graph and computing a sum over the number of crossed cuts for all edges of the path. This has linear running time.

Iterating over all links in L is also expensive as this can be $\mathcal{O}(n^2)$ work. To improve this, one can first observe that by adding links to the graph, because the heuristics $h_w(l)$ and $h_{wc}(l)$ can only increase; $c(l)$ is constant and the set $\{c \in C_G : \text{cut}(c, l) = 1\}$ can only shrink by adding links because cuts are only removed from C_G. The heuristic value

computed in a previous iteration can therefore be used as a lower bound. If there is a link with heuristic value below this bound, it does not need to be checked. To check as few links as possible in a general case, all links are separated into a constant number of lists where each list has a fixed lower bound. The lower bound of list i is $1/2^i$, assuming weights in the interval $[0, 1]$. If weights are larger they could be rescaled first. Initially, each link is added to the last list where the lower bound holds, i.e. a link l is added to list $\mathrm{argmin}_i(\frac{1}{2^i} > h(l))$. In each iteration the algorithm goes over the lists starting from the last one and recomputes the heuristic h for links l encountered. This is necessary since the heuristic may have changed due to links added in previous iterations. In case a link does not cover any cut it is not needed for the solution and is removed from the list. If the heuristic value is larger than the lower bound of a list with a smaller index, it is moved to the corresponding list. If a list contains a link that does not need to be moved, the algorithm can stop early after processing the current list. Following lists can be skipped due to the lower bound being greater than the best link found and the link with the lowest heuristic value can be added to the solution.

Complexity

The complexity of the naive implementation is $\mathcal{O}(n \cdot n^2 \cdot n^2) = \mathcal{O}(n^5)$ because the solution has at most $\mathcal{O}(n)$ links, and in each iteration for $\mathcal{O}(n^2)$ links $\mathcal{O}(n^2)$ minimum cuts are checked.

In the dynamic algorithm the whole cactus graph needs to be contracted in the end. This has complexity $\mathcal{O}(n^3)$ as described in Section 5.1. Just like the naive algorithm the dynamic algorithm also has at most $\mathcal{O}(n)$ iterations. How far the bounds can decrease the number of links that need to be checked depends heavily on the cactus structure and link weights. In a star graph with unit link weights all links that do not have the center vertex as endpoint have the same heuristic value. In this case the bounds are of very little use and $\mathcal{O}(n^2)$ links need to be checked. This results in a total complexity of $\mathcal{O}(n^4)$. However, if the cactus graph contains larger cycles, contracting a link changes the heuristic value of all links crossing the cycle by different amounts. This ensures the use of the bounds, even if link weights happen to be in such a way, that the heuristic value is the same initially.

5.3.2 Minimum Spanning Tree

The greedy strategies described above worked by adding links to a set until this set is a valid connectivity augmentation. Another approach starts with a (possibly much larger) set of links that increases the connectivity when added to the graph, and then removes links that are not needed. The complete set of links can have $\mathcal{O}(n^2)$ size and checking if a link can be removed from a set can also get expensive. So it is of interest to find a smaller initial solution. An intuitive starting point is a minimum spanning tree $\mathrm{mst}(G_L)$ of the graph that has all links as edge set. This is clearly a valid connectivity augmentation because there is a path between any two vertices and therefore every cut is covered. Call the edges of

Algorithm 2 MinimumSpanningTreeAlgorithm

input $G = (V, E)$, L, $c : L \to \mathbb{R}_{\geq 0}$
output augmentation $S \subseteq L$
procedure MinimumSpanningTreeAlgorithm(G, L, c)
  $G_L \leftarrow (V, L)$
  $L_{MST} \leftarrow \mathsf{MST}(G_L, c)$
  // while augmentation not minimal
  while $\exists l \in L_{MST} : L_{MST} \setminus \{l\} \in S_{G,L}$ **do**
    $L_{MST} \leftarrow L_{MST} \setminus \mathrm{arb}(\{\arg\max\{c(l) : l \in L_{MST} \text{ and } L_{MST} \setminus \{l\} \in S_{G,L}\})$
  return L_{MST}

the minimum spanning tree L_{MST}. Next, links not needed for the augmentation should be removed from L_{MST}. This is done greedily, removing the heaviest unneeded link first. Pseudocode is given in Algorithm 2.

Checking Solutions

An essential part of the minimum spanning tree based algorithm is checking if removing a link $l = (u, v)$ from a valid solution $S \subseteq L$ is still an augmentation. If this is not the case, there is still a minimum cut in the graph $G' = (V, (E_C \cup S) \setminus \{l\})$ that would be covered by l. Whether such a cut exists can be checked by computing the connectivity between u and v. In particular, a maximum flow algorithm from u to v in G' can be used. This maximum flow execution is efficient, because the graph is very sparse; it is the union of a cactus graph and a minimum spanning tree which limits the number of edges to $2n + n = 3n$. The cost function for the edges of G' is altered in the following way: $c : E(G') \to \{1, 2\}$ with $c(e) = 1$ if e is a cycle edge in E_C and $c(e) = 2$ otherwise. Using these edge weights, any two vertices in the cactus graph are 2-connected, but not 3-connected. A link $l = (s, t)$ can be removed, if the s-t-flow and therefore the minimum s-t-cut in G' is larger than 2. The Ford-Fulkerson algorithm can be used to compute a maximum s-t-flow by finding augmenting paths in the residual graph [14]. The complexity for integer weights is $\mathcal{O}(m \cdot f)$ where the number of edges m comes from a pathfinding algorithm like a BFS and f is the maximum flow value coming from iterations of finding paths. Each augmenting path increases the flow by at least one because edge weights are in $\{1, 2\}$. Since the algorithm only needs to determine whether $f \leq 2$ or $f > 2$, it can be stopped after $f' \leq 3$ iterations. Therefore, the complexity of the modified algorithm with early termination is $\mathcal{O}(m \cdot f') \overset{m \leq 3n}{=} \mathcal{O}(n)$.

Complexity

A minimum spanning tree can be computed using Kruskal's algorithm with a complexity of $\mathcal{O}(m \cdot \log m)$ [29], or the improved Filter-Kruskal algorithm with complexity $\mathcal{O}(m +$

$n \log n \log \frac{m}{n})$ [34]. For each of the $n-1$ links in the minimum spanning tree the Ford-Fulkerson algorithm is run having a complexity of $\mathcal{O}(n)$. The total complexity is therefore $\mathcal{O}(n^2)$ because $m < n^2$ and $n \log n \log \frac{m}{n} < n^2$. The constant factors are much higher for the Ford-Fulkerson algorithm than for Kruskal's algorithm, such that the additional complexity of Filter-Kruskal is not crucial in praxis, even though the theoretical complexity $\mathcal{O}(m \log m) = \mathcal{O}(n^2 \log n)$ is worse.

5.3.3 Local Search

A non-optimal solution $S \subset L$ may be improved by a local search algorithm. To describe the main concepts, some definitions are introduced first.

Definition 6 (Alternative)**.** *Let S be a solution to the WCAP, L the set of links. A subset $R \subseteq L \setminus S$ is called an alternative to $F \subseteq S$, if $S' = (S \setminus F) \cup R$ also forms a solution. The objective value of S' is $c(S') = c(S) - c(F) + c(R)$.*

Definition 7 (Potential)**.** *Let S be a solution to the WCAP, L the set of links. The potential c_{pot} of a set $P \subseteq L$ is $c_{\text{pot}}(P) = \sum_{l \in P \setminus S} c(l) - \sum_{l \in P \cap S} c(l)$. Applying the set P to a solution S results in a new set $S' = (S \cup P) \setminus (S \cap P)$ which is denoted as $S' = S[P]$. The cost function can be written as $c(S') = c(S) + c_{\text{pot}}(P)$.*

The local search algorithm improves a given solution S by finding cheaper alternatives R for sets $F \subseteq S$. To find good replacements, some conditions that $P = F \cup R$ must fulfill are introduced next.

Lemma 1 (Condition Improvement)**.** *If $S[P]$ is a valid solution, P improves S only if P has a negative potential.*

Proof. Because $c(S[P]) = c(S) + c_{\text{pot}}(P)$, only sets P with $c_{\text{pot}}(P) < 0$ improve the solution. □

Lemma 2 (Condition Non-triviality)**.** *Let $C = (V, E)$ be a cactus graph representing all minimum cuts of a graph G. Let $v \in V$ be no articulation point in C. Then, v is an endpoint in any connectivity augmentation of C.*

Proof. Any vertex that is contained in more than one cycle, counting tree edges as cycles of size 2, is an articulation point. It disconnects the cycles it is contained in, because any two cycles share at most one vertex and a cactus graph is a tree of cycles, meaning there is only one unique path between cycles. Therefore, every vertex v, that is no articulation point of C, is contained in exactly one cycle. Hence, it has degree 1 if it is incident to a tree edge or degree 2 if it is incident to cycle edges. In both cases, $\{v\}$ is a minimum cut of C. To cover the cut $\{v\}$, there must be a link incident to v in any augmentation. □

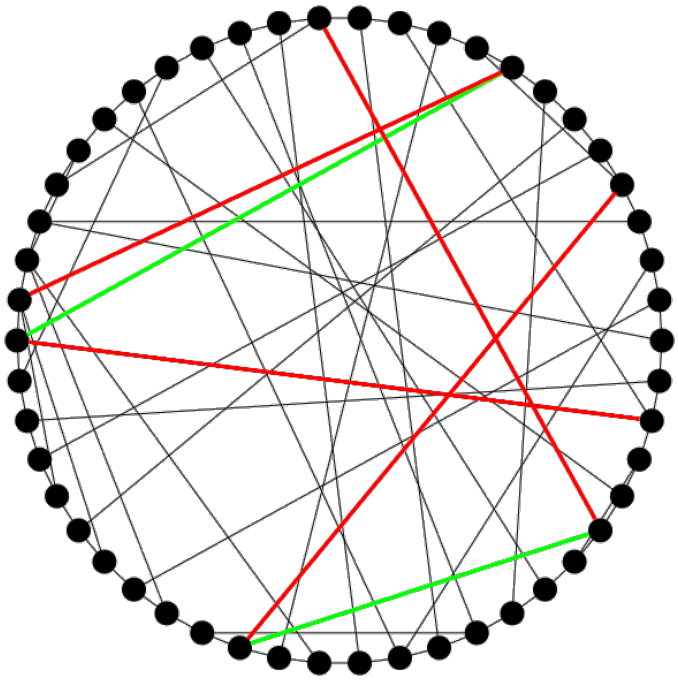


Figure 5.4: Difference between optimum solution (black and green links) and optimal solution (black and red links)

Let $u, v \in V$ be two vertices that are no articulation points of C and $F = \{(u, v)\}$. When removing the link (u, v) from S, both u and v must still be covered by either $S \setminus \{l\}$ or R by Lemma 2. If l is the only link in S covering u and v it is therefore necessary that links incident to u and v need to be added to R. Note that $F \cup R$ is a path of length 3 in this case. In another pattern two links $l_1 = (u, v)$ and $l_2 = (w, x)$ from S can be replaced by a single link. If S is optimal, i.e. $S \setminus \{l\}$ is not an augmentation for all $l \in S$, the new link l must have an endpoint from l_1 and an endpoint from l_2. Again, $\{l_1\} \cup \{l_2\} \cup \{l\}$ is a path of length 3. Figure 5.4 visualizes this by showing an optimal solution S and an optimum solution OPT in a single graph. Links that are in both solutions are drawn in black, links in OPT but not S are drawn in green and links in S but not OPT are drawn in red. The two solutions only differ in two paths of length 3, where links are alternating from S and $L \setminus S$.

This can be generalized to arbitrary path lengths, with the intuition that links in R that are not at the end of the path cover two vertices at once, one vertex of each neighboring link from S. This generalization gives rise to alternating paths, using edges alternating from F and R.

The local search algorithm enumerates alternating paths up to a length limit using a DFS from each vertex. By Lemma 1 only paths with negative potential can improve the solution are therefore stored in a list. The number of paths can be further reduced by applying Lemma 2 to the endpoints of the path. These vertices must not form a trivial minimum cut;

either they need to be covered by S' or the vertices do not need to be covered because they are articulation points in the cactus graph. However, these are only necessary conditions and not sufficient, as not every path leads to a valid solution when it is applied to S. This needs to be checked in addition as described in the next subsection. To maximize the gain in the objective, paths with the most negative potential are considered first.

The number of paths grows exponentially with the search depth with vertex degrees as basis. To keep this computable it is useful to reduce the set L of possible links. This is done by only using links from k minimum spanning trees. For instance, $k = 2$ minimum spanning trees lead to the set $L' = \mathrm{mst}(L) \cup \mathrm{mst}(L \setminus \mathrm{mst}(L))$. This way the average degree is a small constant.

Checking paths

The algorithm needs to check whether applying a path p to a solution S is still a valid augmentation, i.e. if $S[p] \in S_{G,L}$. Instead of checking all up to $\mathcal{O}(n^2)$ minimum cuts in a naive approach, a maximum flow algorithm can be used similar as in the minimum spanning tree algorithm. The maximum s-t-flow algorithm needs to be executed for each removed edge $e = (s, t)$ in the updated graph to ensure that the graph is still $(k + 1)$-connected. Other vertex pairs do not need to be checked as the connectivity between vertices of removed edges is ensured to be at least $k + 1$. The graph $S' = (V, E_G \cup S)$ can be constructed in $\mathcal{O}(n)$ time because it has $m = \mathcal{O}(n)$ edges. The maximum flow algorithm also runs efficiently as in the minimum spanning tree algorithm because the graph S' is very sparse and the maximum flow value needed is bounded. Using the modified Ford-Fulkerson algorithm from Section 5.3.2, the maximum s-t-flow computation takes $\mathcal{O}(n)$ time because of small integer weights a constant number of iterations.

Path Caching

The most expensive part of the algorithm is checking whether applying a path p is still a solution. First, the direction of the path does not matter, so only paths where the ID of the first vertex is smaller than the ID of the last vertex are considered. Second, if local search is done iteratively, the same paths with the best potential are checked every time. It is unlikely that the same path will lead to a valid solution if it was not in a previous iteration. Storing the set of paths that did not have a negative potential can avoid many expensive recomputations. To minimize memory overhead, the vertex sequence of a path is hashed and stored in a set to look up if the check can be skipped.

Early Termination

The set of paths fulfilling the necessary condition in Lemma 2 is checked in the order of their potential. Therefore, the gain of applying paths decreases with each iteration. The algorithm can be terminated early as soon as the gain drops below a threshold and

the solution only improves by a small factor. If the threshold depends on the weight of the initial solution, it is independent of the order of the objective. The potential depends heavily on the link weight distribution, i.e. for unit link weights, the potential is in $\{0, 1\}$ and an early termination is therefore not possible.

5.4 Cactus Graph Generation

Graphs where the cactus graph representing all minimum cuts has a complex structure are rare in real-world graphs like technical or social networks. A graph generator can be used to generate complex graphs of different sizes and evaluate algorithms for such cases. This section gives an algorithm that is able to generate graphs with given properties, namely the number of vertices and the number of cycles.

Generating Cactus Graphs

Given two integers $n, c \in \mathbb{N}$, $n > c$, the goal is to generate a cactus graph C with n vertices and c cycles. This is done by generating the cycles iteratively. The average number of vertices per cycle is n/c. To get a larger amount of possible graphs the number of vertices per cycle is randomly distributed around the average n/c. A Poisson distribution turned out to yield a higher variety of graphs than a uniform distribution. To ensure that the correct number of cycles will be achieved, the distribution range is bounded such that, considering cycles already generated, at least one vertex for every remaining cycle is available. The first generated cycle is used as the base graph. Each consecutive cycle must additionally use an existing vertex to connect to the existing graph. This vertex is chosen uniformly among existing vertices. Figure 5.5 shows an example of a graph generated with $n = 100$ and $c = 20$.

Graph with given Cactus Graph

Given a cactus graph C, one might ask how a graph G of which C represents all minimum cuts could look like. Trivially G could be equal to C. Different graphs could be constructed by reversing the process of edge contractions during the computation of a cactus graph. In particular, each vertex of the cactus graph C could be replaced by a dense subgraph. Let k be the desired connectivity. Then, each vertex can be replaced by an at least $(k + 1)$-connected subgraph while each link is replaced by k unweighted links in case of a tree edge or by $k/2$ links in case of a cycle edge between corresponding dense subgraphs. However, for all algorithms considered in this thesis, neither the structure of the original graph nor the connectivity k matter as they are abstracted in preprocessing steps. Therefore, only the simplest case of cactus graphs with connectivity $k = 2$ is considered.

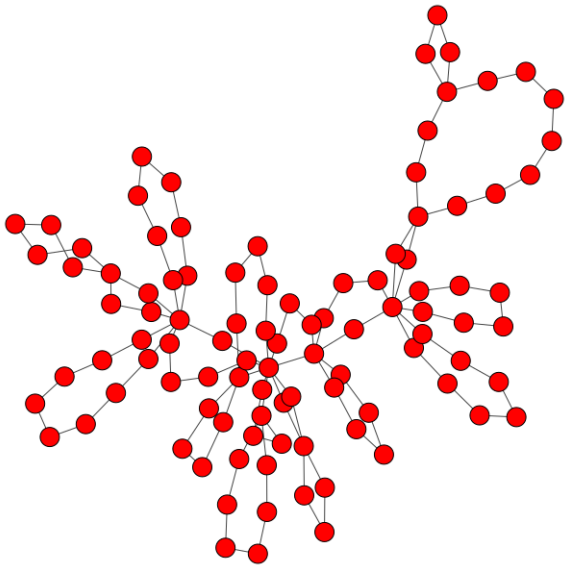


Figure 5.5: Generated cactus graph with 100 vertices and 20 cycles

CHAPTER 6

Experimental Evaluation

The main part of this chapter deals with the experimental evaluation of the algorithms described and implemented in the previous chapters. First, the methodology and the set of test instances is described. Then the approximation algorithms and heuristic algorithms are evaluated and compared in terms of quality, running time and memory consumption. Also, the impact of local search in the solution of the minimum spanning tree algorithm is evaluated.

6.1 Methodology

The experiments are run on a personal computer with an AMD Ryzen 5 2600 six core processor with 12 threads at up to 3.5 GHz and 80 GB of main memory running Linux. The C++ code is compiled using `gcc 13.2.1` with optimization level `O3`. The memory for the process is limited to 50 GB and the running time is limited to 3 hours.

By default, every instance is tested using 5 different seeds for edge weights and the weight of the augmentation, the running time and the maximum memory used is measured. The geometric mean is calculated when averaging over different seeds or instances such that every instance has a comparable influence on the result.

Different algorithms are compared using performance profiles [12]. These plots use the best algorithm as baseline for each instance and relate the other algorithms to this baseline. A performance profile can use the objective function to compare quality, and running time and memory consumption to compare resource requirements. The x-axis shows a parameter $\tau \geq 1$. On the y-axis the fraction of instances whose objective is at most $\tau \cdot \mathrm{best}$ is plotted, in particular $\#\{\mathrm{objective} \leq \tau \cdot \mathrm{best}\}/\#\mathrm{instances}$. For running time and memory usage, time and memory are used instead of the objective, respectively. For each algorithm the performance profile contains a monotone increasing, piece-wise constant function. At $\tau = 1$ the plot shows the fraction of instances where the algorithm is able to find the best solution / has the fastest running time or lowest memory consumption.

Some algorithms are not able to solve every instance due to constraints on memory and time. In performance profiles this is denoted with ✗ and ◷, respectively.

6.2 Instances

The algorithms are evaluated using four types of graph instances: cycles or ring graphs, stars, generated cactus graphs as well as real-world instances. Cycles and stars represent edge cases of cactus graphs, with $\mathcal{O}(n^2)$ being the largest amount and $\mathcal{O}(n)$ being the smallest amount of minimum cuts possible. Many real-world graphs have unique or very few distinct minimum cuts, which leads to very small cactus graphs with only a few vertices. Therefore, instances with non-trivial cactus graphs must be selected carefully. In this thesis, a subset of connected graphs with non-trivial cactus graph from the 10th DIMACS Implementation Challenge is used [3]. The instances are listed in Table 6.1 including their sizes and the sizes of the corresponding cactus graphs. Finally, to be able to test the algorithms on instances that represent more complex cactus graphs and have sizes of interest, cactus graphs generated by the algorithm described in Section 5.4 are used. Table A1 of the appendix gives a complete list of all instances used.

6.3 Objective

The objective function being minimized in the weighted connectivity augmentation problem is the sum of the weights of all links added to the graph. In all following results, the objective of an algorithm for an instance is therefore the cost of all links in the solution S, namely $\sum_{l \in S} c(l)$.

name	n	m	n cactus	m cactus	description
coAuthorsCiteseer	227 320	814 134	30 322	30 321	Social network
delauney_n20	1 048 576	3 145 686	11 740	11 739	Delauney graph
coPapersCiteseer	434 102	16 036 720	6 372	6 371	Social network
t60k	60 005	89 440	1 136	1 332	Sparse matrix
vibrobox	12 328	165 250	625	624	Sparse matrix
email	1 133	5 451	156	155	Social network
M6	3 501 776	10 501 936	132	131	Simulation
queen8_8	64	728	29	28	Chess moves
jazz	198	2 742	6	5	Social network
karate	34	78	2	1	Social network

Table 6.1: 10th DIMACS Implementation Challenge instances

6.4 Evaluation

In this section the optimal integer linear program, the approximation algorithms from the literature as well as the heuristic algorithms developed in this thesis are evaluated. Running time, memory consumption and solution quality are considered in the results where appropriate.

6.4.1 Integer Linear Program

An important decision for this algorithm is the integer linear program solver. In this thesis two solvers are evaluated, the Gurobi Optimizer [20] and Google's OR-Tools [37] using the SCIP backend [4]. However, it quickly turned out that Gurobi is able to solve larger instances because of a smaller memory footprint. Experimental results shown in Figure 6.1 depict 30 % less memory usage for cycle graphs. Therefore, the Gurobi Optimizer is used as solver in all following experiments.

The Gurobi Optimizer allows three modes for presolving a model: skipping the presolve step entirely, a conservative mode, and an aggressive mode which takes more time than the conservative mode, but may reduce the model further. Additionally, an initial solution can be set as the starting point. Figure 6.2 shows the running time of all combinations for cycle graphs with 50, 100, 150 and 200 vertices, while Figure 6.3 gives the memory consumption thereof. The geometric mean over five different seeds for the link weight random

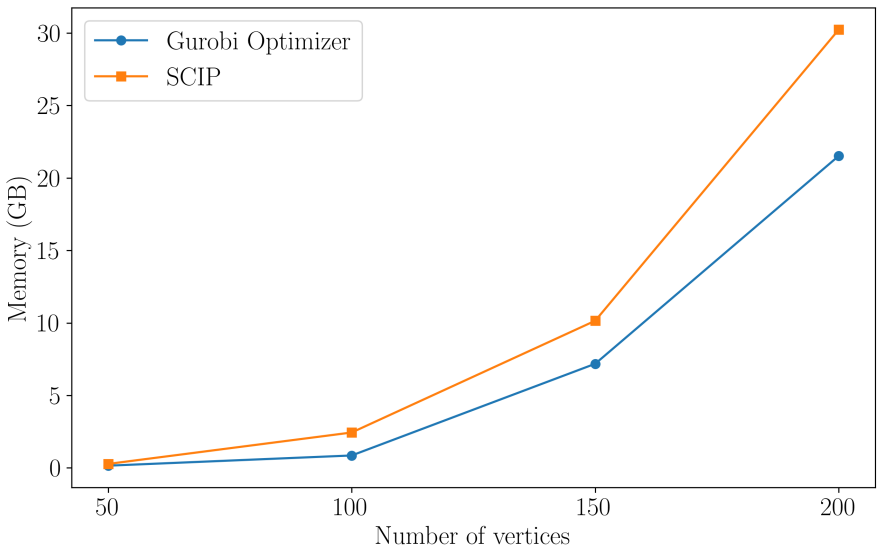


Figure 6.1: Memory consumption of the ILP solver for cycle graphs with 50, 100, 150 and 200 vertices

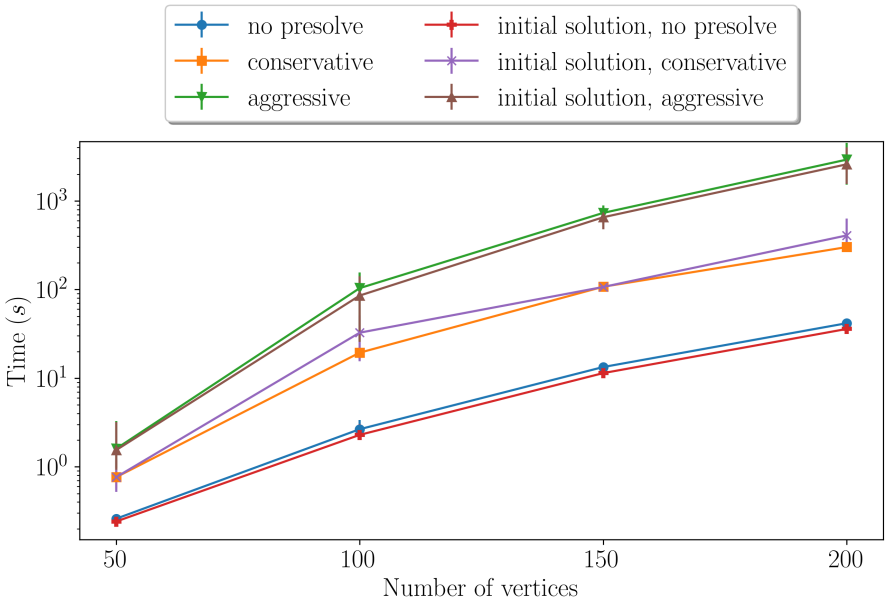


Figure 6.2: Running time for different presolve configurations and initial solution for cycle graphs with 50, 100, 150 and 200 vertices

number generator is plotted as well as the variance. One can observe that solving the problem without a presolve step is consistently much faster than a conservative or aggressive presolve step. However, the memory consumption is slightly higher without reducing the model. The running time is acceptable for all solvable instances and the limiting resource is clearly memory. Therefore, presolving the model leads to more solvable instances. If instances are small and sufficient memory is available, the presolve step can be skipped to increase performance.

Setting a good initial solution, in particular the solution of the minimum spanning tree heuristic, does not have a large impact on running time and memory. However, the memory requirement tends to be lower when using an initial solution, while the running time is slightly longer. As well as for the presolve step, more instances can be solved by setting a good heuristic as initial solution due to memory limits.

Considering the number of integer variables and constraints, the solution time of the ILP is very short. This suggests that connectivity augmentation models are easy to solve with respect to their size, i.e. they contain much redundancy. This can be explained by the fact that a link can cover many cuts, or in terms of an ILP, a single variable can satisfy many constraints.

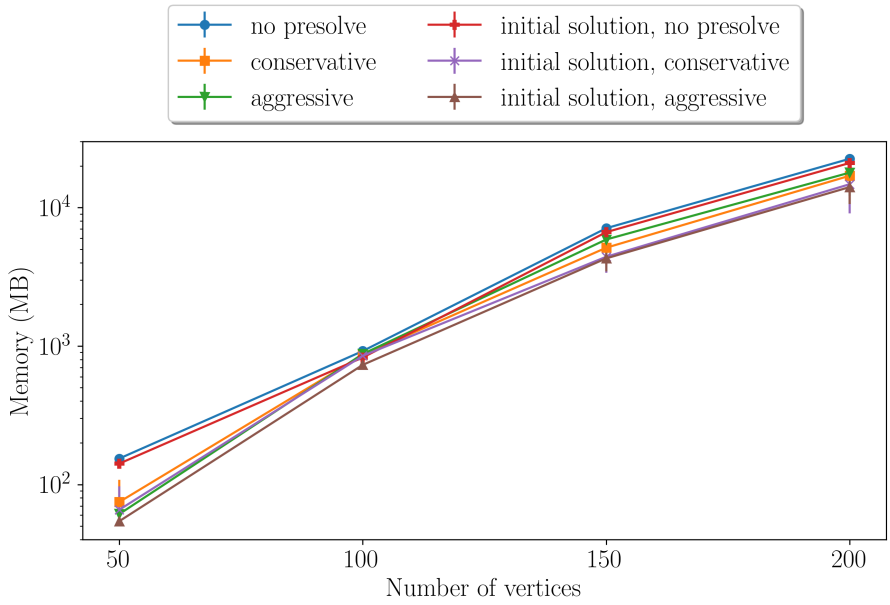


Figure 6.3: Memory usage for different presolve configurations and initial solution for cycle graphs with 50, 100, 150 and 200 vertices

6.4.2 Approximations

As the theoretical complexity already predicted, the $(1 + \ln 2 + \epsilon)$-approximation and the $(1.5 + \epsilon)$-approximation have a long running time and can only solve tiny graphs. The 2-approximation is able to solve larger graphs and will also be compared to the heuristic algorithms in the next section.

Figure 6.4 gives the running time of the approximation algorithms as well as the ILP with respect to the graph size in a logarithmic plot. The graphs are cycle graphs with a complete set of links. For every instance 10 different seeds for uniformly distributed link weights are used, except for the cycle with 9 vertices due to long runtime. The geometric mean and the variance is shown in the plot. It is clearly visible that the running time increases exponentially for the approximation algorithms based on the dynamic program. This is expected for graphs with fewer than $\alpha \geq 28$ vertices as the computational complexity is exponential in $4 \min(n, \alpha) + 7$. Both, the $(1 + \ln 2 + \epsilon)$-approximation and the $(1.5 + \epsilon)$-approximation are orders of magnitude slower than the optimal integer linear program. This gives them only theoretical value, because problems where the approximations are able to compute a solution within reasonable time can be solved to optimality.

The $(1.5 + \epsilon)$-approximation is an order of magnitude faster than the $(1 + \ln 2 + \epsilon)$-approximation. This is due to fewer invocations of the dynamic program, in particular the $(1.5 + \epsilon)$-approximation requires only one to three invocations for the tiny graphs with a 2-approximation as starting point and no binary search.

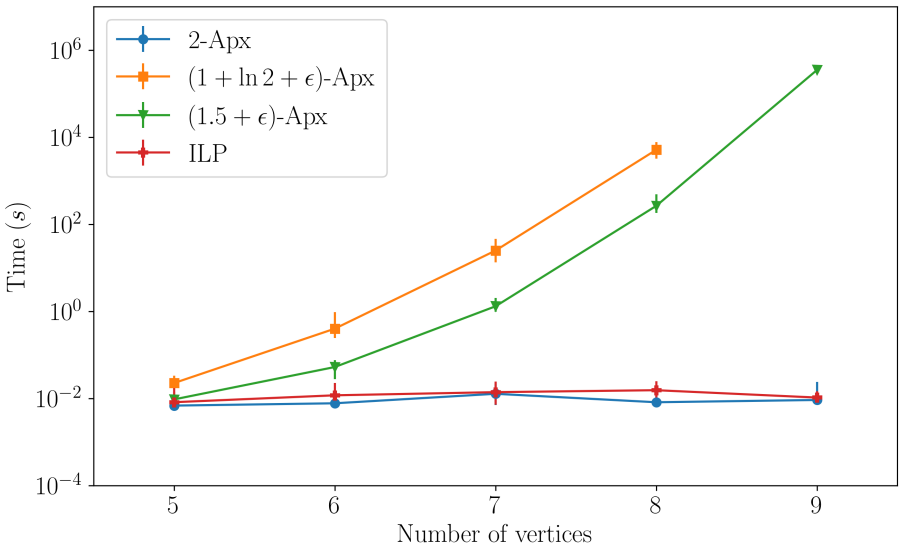


Figure 6.4: Running time of approximations on cycle graphs by size

The 2-approximation is slightly faster than the ILP, but the difference is negligible and both can easily solve tiny graphs solvable by the dynamic program based approximations. Due to the fact that the ILP is much faster than the dynamic program based approximations and performs similar to the 2-approximation, the approximations have very little relevance in practical connectivity augmentation algorithms.

Figure 6.5 shows a performance profile of the objective, i.e. the augmentation weight, for tiny graphs. As the optimal solution was computed by the ILP, the x-axis gives the approximation ratio. The first observation is that all approximation algorithms only give solutions within their claimed ratio. Furthermore, the 2-approximation consistently gives the worst solutions, which means both, the $(1 + \ln 2 + \epsilon)$-approximation and the $(1.5 + \epsilon)$-approximation, can improve this initial solution. This confirms the theoretical value of the better-than-2 approximations.

6.4.3 Heuristics

In this section an experimental evaluation of the heuristic algorithms as well as the local search algorithm developed in this thesis is given. If the graph is small enough, the results are compared to the optimal solution computed using the ILP and the LP-based 2-approximation.

Figure 6.6 shows a performance profile of the objective for the greedy weight-coverage heuristic, the minimum spanning tree algorithm and the minimum spanning tree algorithm

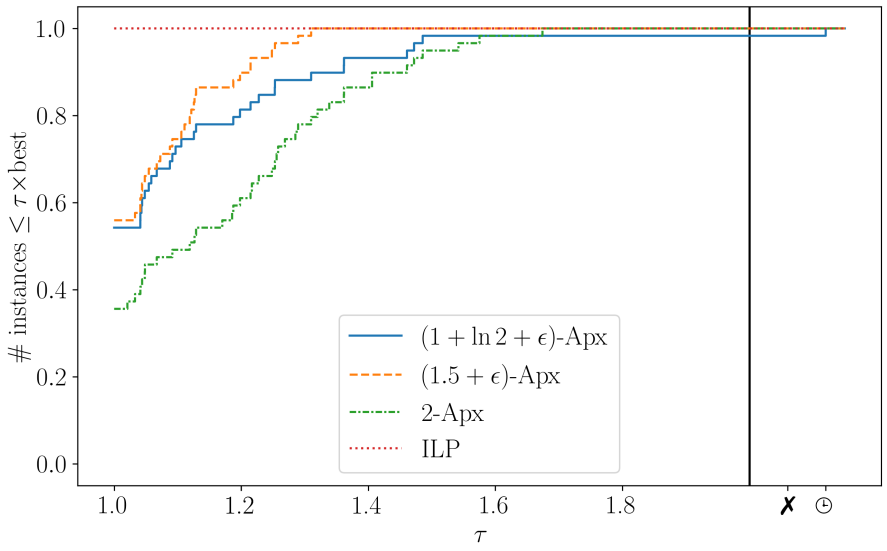


Figure 6.5: Performance profile of the approximation algorithms on tiny cycles and stars

with local search as well as the ILP and the LP-based 2-approximation. The plot includes all four types of graphs, i.e. cycles, stars, real graphs and generated cactus graphs.

The LP-based 2-approximation does not only yield the worst results, but is also the one to solve the fewest instances. That the ILP is able to solve more instances is due to the fact that the memory consumption is lower as Figure 6.7 indicates. The reason for higher memory requirements of the LP is that each undirected link is replaced by two directed ones and therefore the LP has twice as many variables compared to the ILP, and, even more important, the graph is reduced to a cycle graph with up to twice as many vertices.

The best results are achieved by the minimum spanning tree algorithm, which can be slightly improved by local search. However, local search is slower and therefore fewer graphs can be solved. Figure 6.7 shows that the minimum spanning tree algorithm requires the least memory as well, but local search and the greedy algorithm do not need significantly more memory.

Link Weight Distribution

The algorithms can perform differently based on the distribution of the link weights. To be able to compare the algorithms in different scenarios, three different distributions are used:

- Unit weights, i.e. each link has weight 1
- A uniform distribution in the interval $[0, 1]$

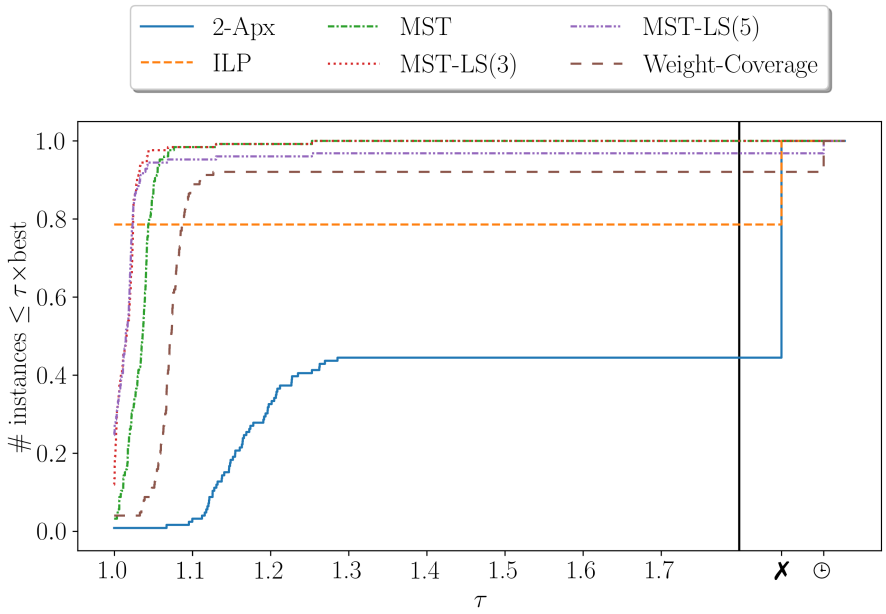


Figure 6.6: Performance profile for objective of algorithms

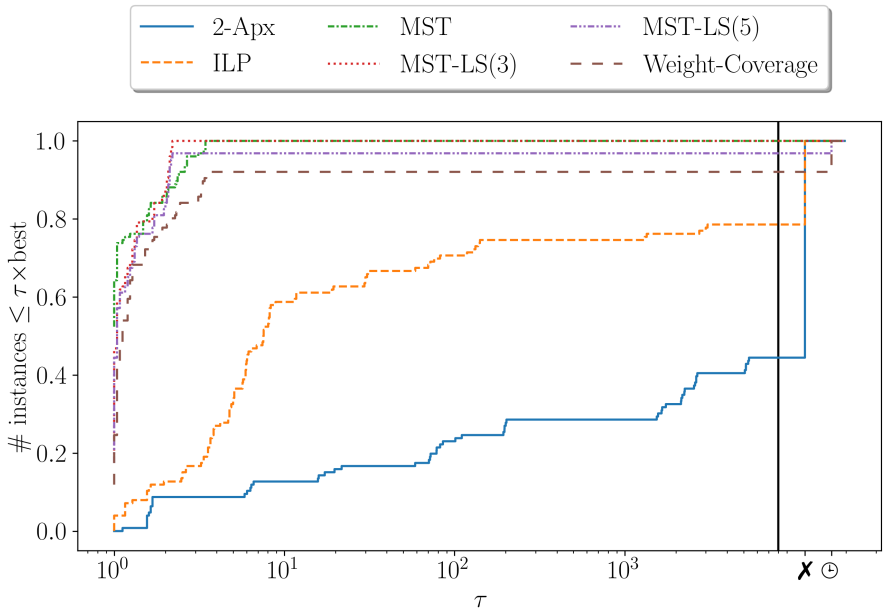


Figure 6.7: Performance profile for memory usage of algorithms

- A normal distribution with mean $\mu = 0.5$ and standard deviation $\sigma = 0.5$. However, negative link weights are not allowed. Instead, new link weights are drawn from the distribution until a non-negative weight occurs.

Performance profile plots of the objective for these three link weight distributions are shown in Figure 6.6 for a uniform distribution and Figure 6.8 and in Figure 6.9 for unit weights and a normal distribution. There is almost no difference in the solution quality for the uniform distribution and the normal distribution. For unit link weights there are two major differences: the deviation from the optimum solution is much higher, and some algorithms are able to solve fewer instances than for uniform or normal distributed link weights.

The higher discrepancy from the optimum solution can be explained by the higher impact of non-optimal decisions. Every non-optimal link that is needed for the solution has weight 1, while for the other distribution, there may be a non-optimal link that is only slightly more expensive.

The difference in solvable instances becomes clear when looking at the geometric mean of the running time over all tested instances in Figure 6.10. For the uniform distribution and the normal distribution, the running time is almost the same. For unit link weights, the local search algorithm is much slower, the minimum spanning tree algorithm is also significantly slower, and the weight-coverage heuristic and the (I)LP-based algorithms need slightly more time. The local search algorithm computes far more paths, and all of them need to be checked for feasibility. There are two reasons for the higher number of paths. First, it is more likely that an arbitrary path has a positive gain - every path that uses more new links than links from the initial solution decreases the objective. Second, Kruskal's algorithm for computing minimum spanning trees is a poor choice for unit weights. The algorithm always picks the first link it can add to the tree, which results in a star. With the initial solution being a star, and the available links being the union of two stars with different center vertices, many alternating paths exist. This can be improved by randomizing the link order in Kruskal's algorithm for computing the minimum spanning trees. The reason for the weight-coverage heuristic being slower is that the heuristic value is not as well distributed and therefore, lower bounds for the heuristic are not as useful.

Incomplete Link Set

So far, the cactus graph and the link set yielded a complete graph. Now the performance of the algorithms for instances with fewer links is evaluated for a uniform link distribution. Figure 6.11 and Figure 6.12 give the geometric mean of the running time and memory usage over all tested instances where 25 %, 50 %, 75 % and 100 % of the links are available.

The largest difference is the memory usage of the ILP and the LP-based 2-Approximation. This is because the number of variables is linearly dependent on the number of links, as well as the average number of variables in constraints. This also improves the runtime. The memory consumption of the remaining algorithms does not change significantly.

The local search algorithm does not benefit from fewer links. The links of two minimum spanning trees are the only links considered for local search - the total link set is solely used by the minimum spanning tree algorithm that does not carry weight compared to the local search. The number of links the weight-coverage heuristic needs to check depends

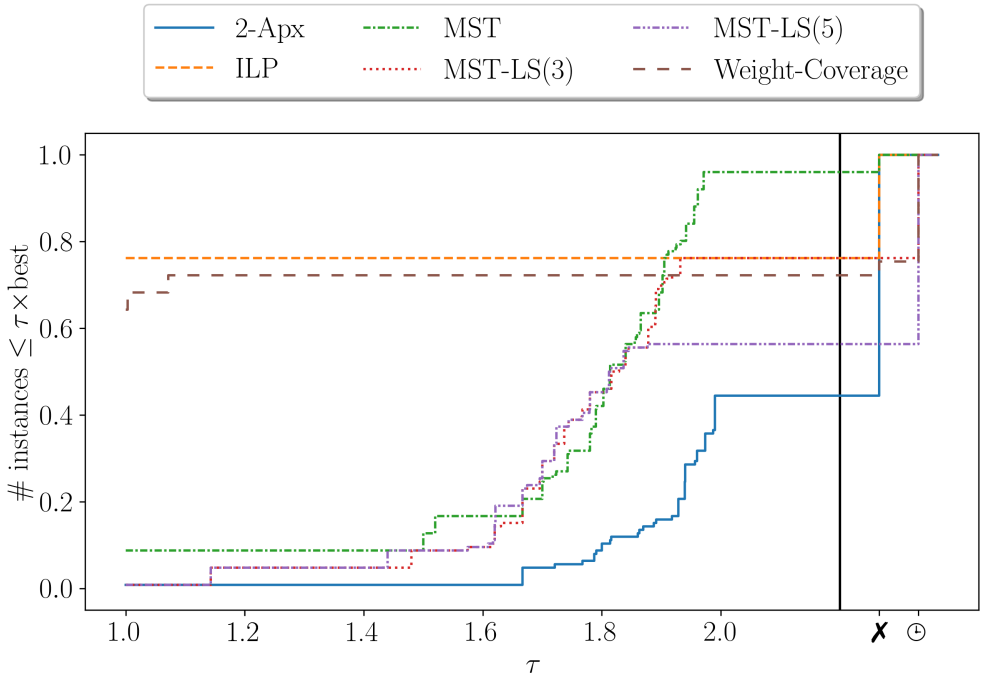


Figure 6.8: Performance profile of the objective for unit link weights

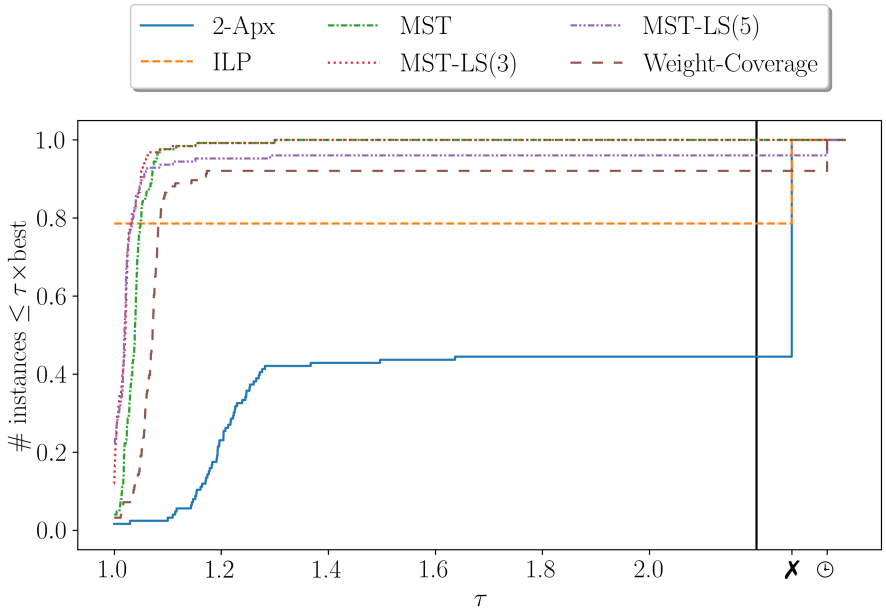


Figure 6.9: Performance profile of the objective for normal link weight distribution

linearly on the link set size. However, the performance difference is not as high as one

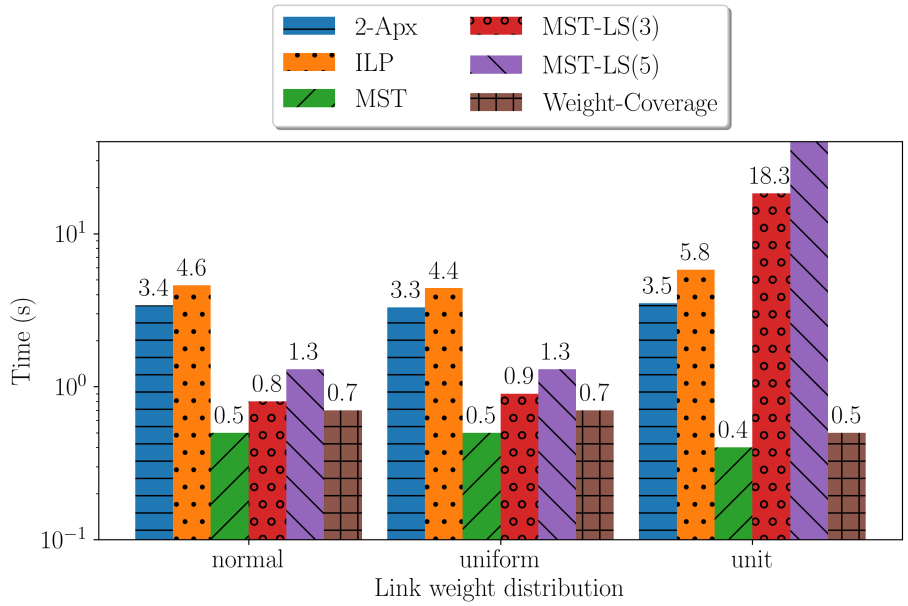


Figure 6.10: Geometric mean of running time for different link weight distributions

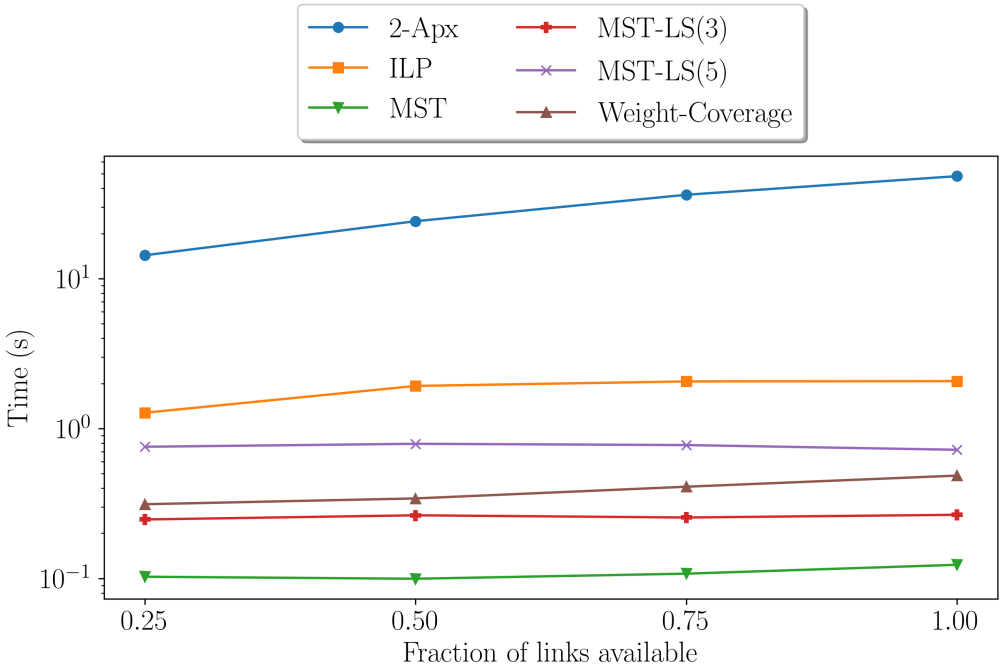


Figure 6.11: Geometric mean of running time for different sized link sets

could expect, because most links do not need to be checked due to known lower bounds from previous iterations.

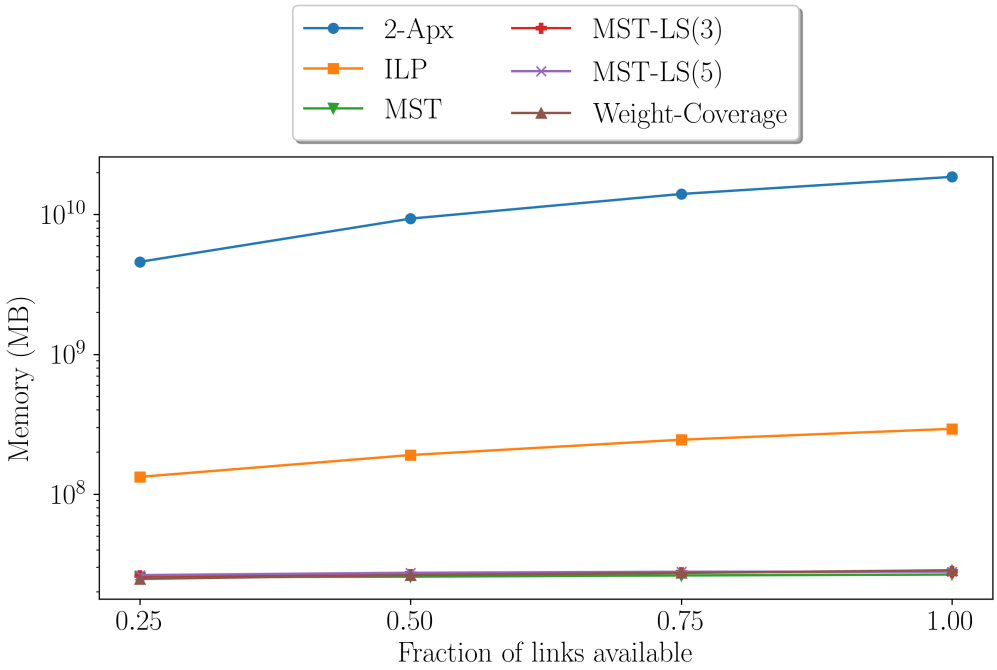


Figure 6.12: Geometric mean of memory usage for different sized link sets

Weight and Weight-Coverage Heuristic

Three greedy heuristics are described in Section 5.3.1, the weight heuristic, the alternative weight heuristic covering a specific cut, and the weight-coverage heuristic. Figure 6.13 shows a performance profile of the objective for all three heuristics. The weight heuristic always performs worst. Picking an arbitrary cut and only considering links that cover this cut results in better solutions in all cases. The best results are achieved by the weight-coverage heuristic in general. However, depending on the random link weights, the alternative weight heuristic is able to compute the best result in 17 % of the instances.

Local Search

The result of the local search algorithm on all instances has already been shown in Section 6.4.3. In this section the effect of reducing L by only using links of minimum spanning trees on the objective, and the effect of path caching and early termination on the running time is evaluated.

In Figure 6.14 the objective of the local search algorithm applied to the solution of the minimum spanning tree algorithm using links of $k \in \{1, 2, 3, 4, 5\}$ MSTs is shown. Additionally, the result is compared to the ILP using the same set of links showing the objective

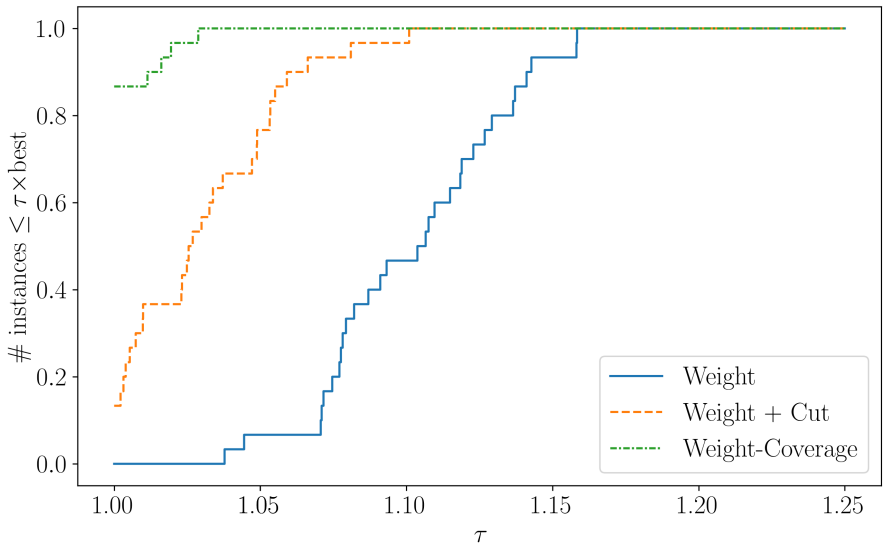


Figure 6.13: Performance profile of the objective for greedy heuristics

if the local search would be able to find all improvements (MST-ILP). Only graphs where the cactus graph has less than 300 vertices were used such that the ILP solution can always be computed. The objective for the minimum spanning tree algorithm and the optimal solution using the complete link set are drawn for reference. Using a single MST for local search does not improve the solution of the minimum spanning tree algorithm, and there is no benefit of using more than two MSTs. Therefore, using the links of two MSTs is clearly the best configuration. The ILP shows that an optimum solution may require links from further MSTs, but most progress can be done using the first two MSTs.

The most important parameter for the local search algorithm is the length of the augmenting paths that are considered. Figure 6.15 shows the geometric mean of the objective of the local search algorithm on the solution of the minimum spanning tree algorithm using path lengths $\{1, \ldots, 7\}$. For reference the solution of the minimum spanning tree algorithm is also plotted. The local search is only able to find improvements for path lengths ≥ 3. Increasing path lengths from 2 to 3 gives the most gain in solution quality. Path lengths of 4 and 5 are able to find further small improvements. For longer paths, the difference in solution quality is negligible. This makes path lengths of 3 and 5 most interesting.

The effect of caching whether applying paths results in a valid augmentation on the running time is shown in Figure 6.16. The geometric mean over all instances is improved by a factor of more than two. On larger instances the impact is even bigger because the algorithm spends a larger fraction of the time in checking paths. However, caching paths is considerable faster for all instances.

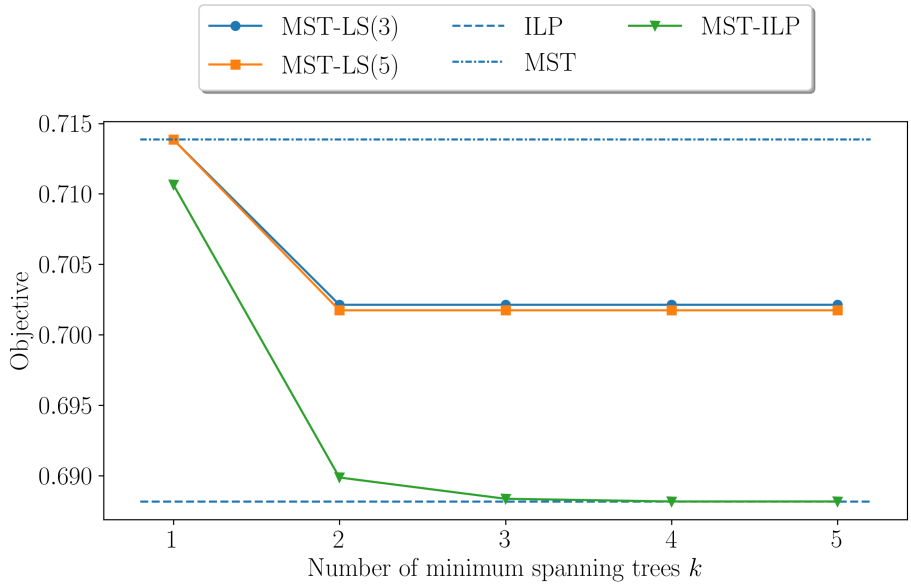


Figure 6.14: Objective with link set being restricted to k MSTs for graphs with $n < 300$

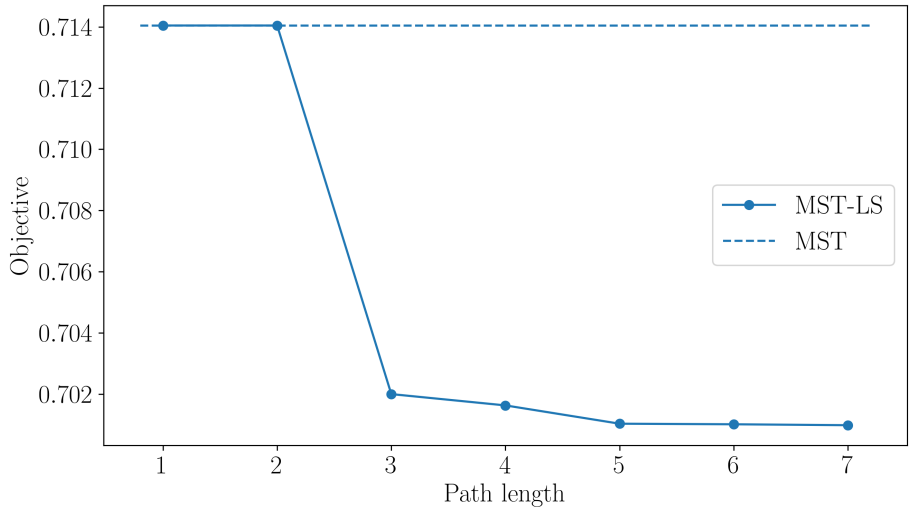


Figure 6.15: Geometric mean of the objective of local search for different path lengths

The last consideration for the local search algorithm is whether it can be terminated early without losing significant improvements of the solution. This is shown in Figure 6.17 for three graphs of similar size. In the beginning the local search algorithm needs more time to find improvements. The main reason is that there are no cached paths yet and every possible path needs to be checked for feasibility. Then, local search makes continuous progress with

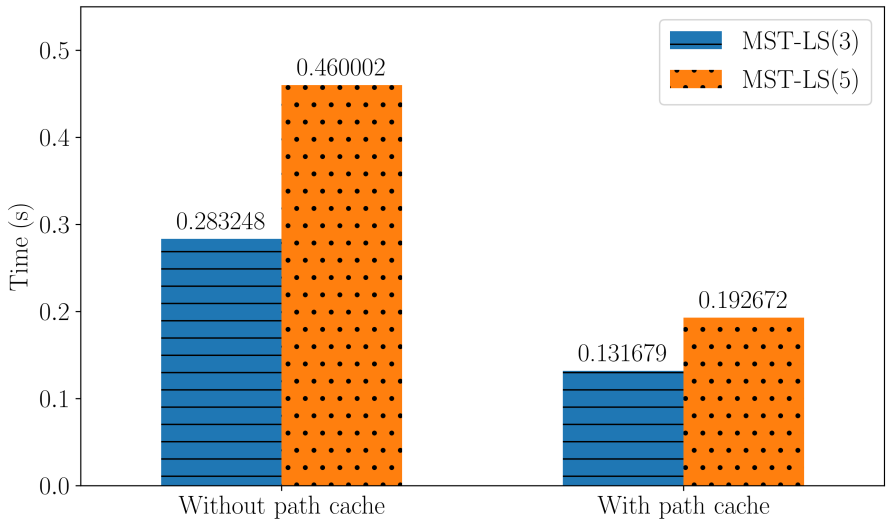


Figure 6.16: Geometric mean of the running time of local search with and without caching paths

declining improvements. Most improvements are done after half of the algorithm runtime. While skipping the last part does not sacrifice a lot of solution quality, the running time is only improved by a small constant factor. If local search can be run for the required time to find good improvements, it can most likely be run until there are no improvements left. If there is a strict limit on the running time local search can be terminated early, but otherwise there is only a small benefit in stopping the algorithm.

6.4.4 Comparison against State-of-the-Art

For the algorithms from Watanabe et al. ([44], [45], [46]), neither the instances used in their experimental evaluation nor source code or binaries for the algorithms are available. To get an idea of how the algorithms compare, we did an own implementation of their simplest greedy algorithm, SMC. However, we are unable to reproduce results from [44], where SMC is able to find the optimum solution for 52.5 % of the instances. On our instances of the same size SMC is only able to find the optimum solution in 6.7 % of the cases.

Our implementation of SMC is compared against the minimum spanning tree algorithm, the weight-coverage heuristic, the ILP and the 2-approximation in Figure 6.18. It is able to solve the same set of instances as the weight-coverage heuristic, but yields worse results in every case. Only the 2-approximation is outperformed by SMC. The algorithms FSM and HBD yielded better results than SMC [44], [46], however, we are not able to provide an implementation in the scope of this thesis. According to results in [46], SMC is able

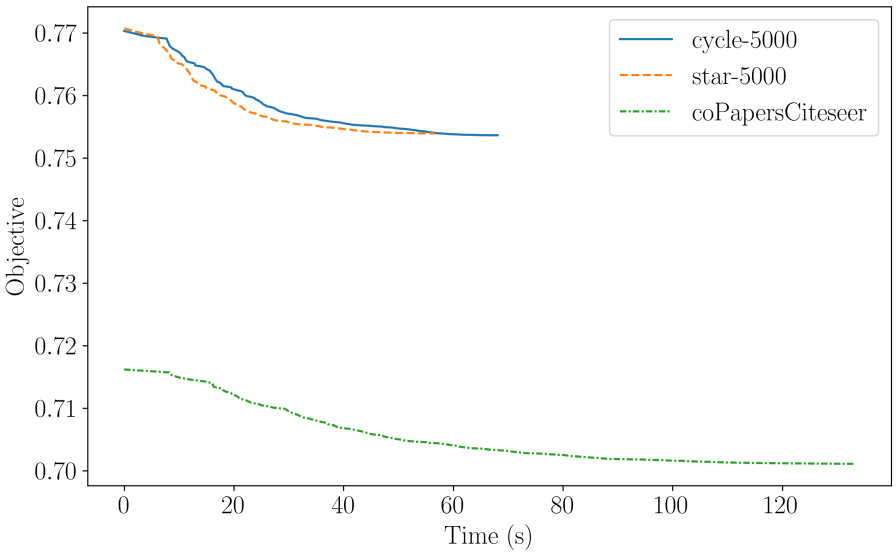


Figure 6.17: Objective over time for local search algorithm with path length 3

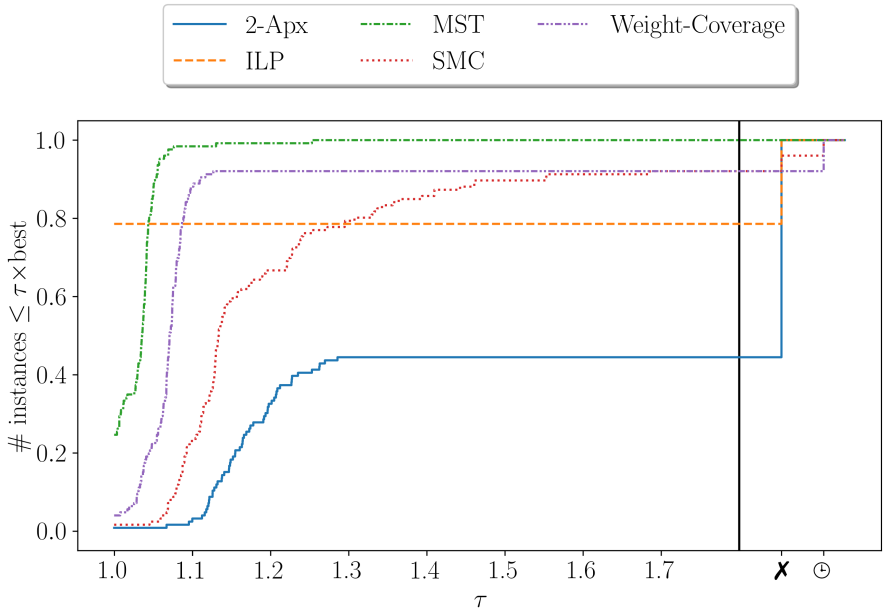


Figure 6.18: Performance profile comparing against our implementation of SMC

to compute a solution equal or better than FSM for only 20.18 % of the instances. For 77.54 %, FSM performs up to 10 % better, and in 2.29 % it is more than 10 % better.

To compare the running time of the algorithms, data from [46] is used. The experiments by Watanabe et al. were run on an Intel Pentium IV at 1.7 GHz and the average running

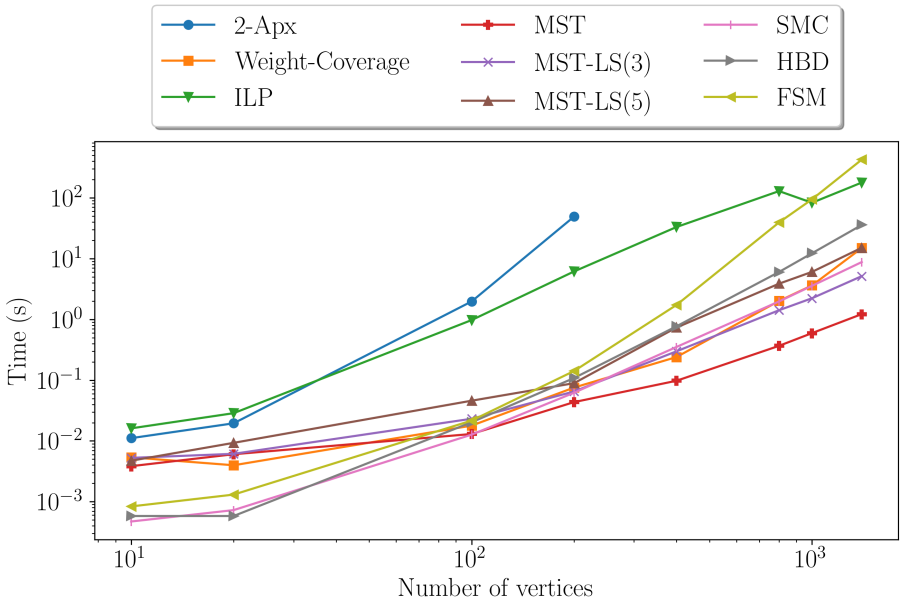


Figure 6.19: Running time comparison against SMC, FSM and HBD with data from [46]

time for increasing the connectivity by 5 by running the same algorithm 5 times is given. The running time of a single iteration of the algorithms is assumed to be a fifth of the total runtime. To get comparable results, the single core performance difference of the more recent Ryzen 2600 CPU used in our experiments is determined with a factor of 5.5 using data from UserBenchmark [41]. The running time given by Watanabe et al. is therefore divided by 27.5.

The performance of the weight-coverage heuristic is comparable to the performance of the fastest competitor, SMC. The minimum spanning tree algorithm is almost an order of magnitude faster and local search with search depth 3 can beat the algorithms as well. HBD and FSM are significantly slower.

CHAPTER 7

Discussion

7.1 Conclusion

Lately, there has been much work on theoretical approximation algorithms on the weighted connectivity augmentation problem. However, there have been no implementations of the algorithms. The first contribution of this thesis is an implementation of the state-of-the-art approximation algorithm. In praxis, this algorithm performs very bad, and a given problem can be solved much faster by an optimal ILP. The only work on fast, practical approximation algorithms is decades old. Therefore, the second contribution of this thesis, the heuristic algorithms developed, aim at introducing new algorithms after a long time.

The first approach is a greedy strategy that greedily adds links to a solution. The best heuristic found is the weight-coverage heuristic. This results in a simple yet effective greedy algorithm that requires a sophisticated data structure to be computed efficiently. Because of its simplicity it can be used as a baseline for further algorithms.

The minimum spanning tree algorithm is the best heuristic algorithm in terms of both, solution quality and runtime. In contrast to the greedy approach and algorithms from the literature, it starts with an MST as feasible solution and then improves it.

The local search algorithm can be applied to a solution computed by any other algorithm and is able to find small improvements. To the best of our knowledge it is the first algorithm of this kind in the literature that is usable in praxis. The benefit is that it can be used for as long as running time is remaining to get the best solution possible.

7.2 Future Work

In this thesis heuristic algorithms that can solve larger graphs than an optimal solver are proposed. However, the size of graphs that are solvable is still limited and there are real-world instances that are much larger. These include for instance street networks of countries and information networks like links in the World Wide Web that cannot be solved

within reasonable time. To compute a solution for such instances faster algorithms that may sacrifice quality are needed.

The algorithms developed in this thesis are not optimized for unit link weights. They have a higher running time and may produce worse results because the link weight cannot be used for prioritization. Specialized algorithms can make more assumptions if all link weights are known to be equal.

Another way to improve the existing algorithms are reductions before the actual algorithm is executed apart from the cactus graph. The size of the cactus graph can be reduced by fixing links in the solution. Rules on how to select these links must still be developed and evaluated. Another approach for reducing the problem size is reducing the set of links. This can be particularly useful for the ILP as each link corresponds to a variable. Links that are completely covered by a set of cheaper links cannot be in a solution. Using only the links of minimum spanning trees as in the local search algorithm is already an aggressive heuristic approach for reducing the link set, but there may be better heuristics that give a more useful link set containing more important links.

Parallel algorithms are not covered in this thesis. For a start, parallel algorithms could speed up the computation considerably. All heuristic algorithms have much independent work that can be parallelized. The greedy algorithms need to compute the heuristic for many links, the minimum spanning tree algorithm needs to check if every single link is required for the solution and the local search algorithm checks a list of paths for feasibility. But parallel algorithms can also enable larger graphs. A complete link set has size $\mathcal{O}(n^2)$, which limits graph sizes possible on a single machine due to main memory. In a parallel setting the link set can be split among multiple processors, reducing the memory requirements for a single machine. For these reasons, future work can include designing parallel algorithms.

Lastly, there are many variations of the connectivity augmentation problem as well as related problems that are not covered in this thesis but may benefit from similar ideas. These include the survivable network design problem, which has a target connectivity for every pair of vertices, or the vertex connectivity augmentation problem considering the vertex connectivity instead of the edge connectivity. Future work can include generalizations of the algorithms to solve similar problems.

Zusammenfassung

Ein grundlegendes Problem beim Design von rubusten Netzwerken besteht darin, die Konnektivität eines Graphen zu erhöhen. Das Problem der gewichteten Konnektivitätserhöhung (weighted connectivity augmentation problem, WCAP) ist eine verbreitete Version, die Kosten für Kanten berücksichtigt. Gegeben ist eine Menge von Kanten, die zum Graphen hinzugefügt werden können. Eine Lösung für das WCAP ist eine Teilmenge dieser Kanten mit minimalem Gewicht, welche die Konnektivität des Graphen um eins erhöht, wenn sie zum Graphen hinzugefügt wird. In dieser Arbeit wird die erste Implementierung von kürzlich entdeckten Approximationen mit einem Faktor besser als 2 vorgestellt. Außerdem werden ein optimales ILP und drei heuristische Algorithmen vorgeschlagen. Diese beinhalten einen Greedy-Algorithmus, der Kantengewichte und die Anzahl der abgedeckten minimalen Schnitte berücksichtigt, einen auf minimalen Spannbäumen basierenden Algorithmus und eine lokale Suche, die eine gegebene Lösung durch Austauschen der Kanten von Pfaden verbessern kann. Eine experimentelle Auswertung zeigt, dass der Algorithmus auf der Grundlage von minimalen Spannbäumen am schnellsten ist und die besten Lösungen liefert. Dabei ist die lokale Suche immer noch in der Lage, kleine Verbesserungen bei diesen Lösungen zu finden.

Bibliography

[1] Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. Recent advances in practical data reduction. In Hannah Bast, Claudius Korzen, Ulrich Meyer, and Manuel Penschuck, editors, *Algorithms for Big Data - DFG Priority Program 1736*, volume 13201 of *Lecture Notes in Computer Science*, pages 97–133. Springer, 2022. doi: 10.1007/978-3-031-21534-6_6. URL https://doi.org/10.1007/978-3-031-21534-6_6.

[2] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS J. Comput.*, 32(2):473–506, 2020. doi: 10.1287/IJOC.2018.0857. URL https://doi.org/10.1287/ijoc.2018.0857.

[3] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018. doi: 10.1007/978-1-4939-7131-2_23. URL https://doi.org/10.1007/978-1-4939-7131-2_23.

[4] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[5] Jaroslaw Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: A reduction to steiner tree. *SIAM J. Comput.*, 52(3):718–739, 2023. doi: 10.1137/21M1421143. URL https://doi.org/10.1137/21m1421143.

[6] Federica Cecchetto, Vera Traub, and Rico Zenklusen. Bridging the gap between tree and connectivity augmentation: unified and stronger approaches. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 370–383. ACM, 2021. doi: 10.1145/3406325.3451086. URL https://doi.org/10.1145/3406325.3451086.

[7] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Augmenting edge connectivity via isolating cuts. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3237–3252. SIAM, 2022. doi: 10.1137/1.9781611977073.127. URL https://doi.org/10.1137/1.9781611977073.127.

[8] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Edge connectivity augmentation in near-linear time. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 137–150. ACM, 2022. doi: 10.1145/3519935.3520038. URL https://doi.org/10.1145/3519935.3520038.

[9] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022. doi: 10.1109/FOCS54457.2022.00064. URL https://doi.org/10.1109/FOCS54457.2022.00064.

[10] E. Dinic, Alexander Karzanov, and M. Lomonosov. The system of minimum edge cuts in a graph. *In book: Issledovaniya po Diskretnoĭ Optimizatsii (Engl. title: Studies in Discrete Optimizations), A.A. Fridman, ed., Nauka, Moscow, 290-306, in Russian,*, 01 1976.

[11] Yefim Dinitz. Maintaining the 4-edge-connected components of a graph on-line. In *Second Israel Symposium on Theory of Computing Systems, ISTCS 1993, Natanya, Israel, June 7-9, 1993, Proceedings*, pages 88–97. IEEE Computer Society, 1993. doi: 10.1109/ISTCS.1993.253480. URL https://doi.org/10.1109/ISTCS.1993.253480.

[12] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002. doi: 10.1007/S101070100263. URL https://doi.org/10.1007/s101070100263.

[13] Kapali P. Eswaran and Robert Endre Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976. doi: 10.1137/0205044. URL https://doi.org/10.1137/0205044.

[14] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399 – 404, 1956. doi: 10.4153/CJM-1956-045-5. URL https://doi.org/10.4153/CJM-1956-045-5.

[15] András Frank and Éva Tardos. An application of submodular flows. *Linear Algebra and its Applications*, 114-115:329–348, 1989. ISSN 0024-3795. doi: https://doi.org/10.1016/0024-3795(89)90469-2. URL https://www.sciencedirect.com/science/article/pii/0024379589904692. Special Issue Dedicated to Alan J. Hoffman.

[16] Greg N. Frederickson and Joseph F. JáJá. Approximation algorithms for several graph augmentation problems. *SIAM J. Comput.*, 10(2):270–283, 1981. doi: 10.1137/0210019. URL https://doi.org/10.1137/0210019.

[17] Scott Freitas, Diyi Yang, Srijan Kumar, Hanghang Tong, and Duen Horng Chau. Graph vulnerability and robustness: A survey. *IEEE Trans. Knowl. Data Eng.*, 35(6):5915–5934, 2023. doi: 10.1109/TKDE.2022.3163672. URL https://doi.org/10.1109/TKDE.2022.3163672.

[18] Waldo Gálvez, Fabrizio Grandoni, Afrouz Jabal Ameli, and Krzysztof Sornat. On the cycle augmentation problem: Hardness and approximation algorithms. *Theory Comput. Syst.*, 65(6):985–1008, 2021. doi: 10.1007/S00224-020-10025-6. URL https://doi.org/10.1007/s00224-020-10025-6.

[19] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 77–103. Springer, 2010. doi: 10.1007/978-3-540-68279-0_4. URL https://doi.org/10.1007/978-3-540-68279-0_4.

[20] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

[21] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM J. Exp. Algorithmics*, 23, 2018. doi: 10.1145/3274662. URL https://doi.org/10.1145/3274662.

[22] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory exact minimum cuts. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 13–22. IEEE, 2019. doi: 10.1109/IPDPS.2019.00013. URL https://doi.org/10.1109/IPDPS.2019.00013.

[23] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPICS.ESA.2020.59. URL https://doi.org/10.4230/LIPIcs.ESA.2020.59.

[24] Monika Henzinger, Alexander Noe, and Christian Schulz. Practical fully dynamic minimum cut algorithms. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 13–26. SIAM, 2022. doi: 10.1137/1.9781611977042.2. URL https://doi.org/10.1137/1.9781611977042.2.

[25] Kamal Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Comb.*, 21(1):39–60, 2001. doi: 10.1007/S004930170004. URL https://doi.org/10.1007/s004930170004.

[26] Samir Khuller and Ramakrishna Thurimella. Approximation algorithms for graph augmentation. *J. Algorithms*, 14:214–225, 1993. doi: 10.1006/JAGM.1993.1010. URL https://doi.org/10.1006/jagm.1993.1010.

[27] Guy Kortsarz, Robert Krauthgamer, and James R. Lee. Hardness of approximation for vertex-connectivity network design problems. *SIAM J. Comput.*, 33(3):704–720, 2004. doi: 10.1137/S0097539702416736. URL https://doi.org/10.1137/S0097539702416736.

[28] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010. doi: 10.1007/978-3-540-68279-0_5. URL https://doi.org/10.1007/978-3-540-68279-0_5.

[29] Harry R. Lewis. *Ideas That Created the Future: Classic Papers of Computer Science*. The MIT Press, 02 2021. ISBN 9780262363174. doi: 10.7551/mitpress/12274.001.0001. URL https://doi.org/10.7551/mitpress/12274.001.0001.

[30] Toshiya Mashima and Toshimasa Watanabe. Approximation algorithms for the k-edge-connectivity augmentation problem. In *1995 IEEE International Symposium on Circuits and Systems, ISCAS 1995, Seattle, Washington, USA, April 30 - May 3,*

1995, pages 155–158. IEEE, 1995. doi: 10.1109/ISCAS.1995.521474. URL https://doi.org/10.1109/ISCAS.1995.521474.

[31] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Program.*, 67:325–341, 1994. doi: 10.1007/BF01582226. URL https://doi.org/10.1007/BF01582226.

[32] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17:245–264, 04 2012. doi: 10.1007/BF03167346.

[33] Zeev Nutov. Approximation algorithms for connectivity augmentation problems. In Rahul Santhanam and Daniil Musatov, editors, *Computer Science - Theory and Applications - 16th International Computer Science Symposium in Russia, CSR 2021, Sochi, Russia, June 28 - July 2, 2021, Proceedings*, volume 12730 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2021. doi: 10.1007/978-3-030-79416-3_19. URL https://doi.org/10.1007/978-3-030-79416-3_19.

[34] Vitaly Osipov, Peter Sanders, and Johannes Singler. The filter-kruskal minimum spanning tree algorithm. In Irene Finocchi and John Hershberger, editors, *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments, ALENEX 2009, New York, New York, USA, January 3, 2009*, pages 52–61. SIAM, 2009. doi: 10.1137/1.9781611972894.5. URL https://doi.org/10.1137/1.9781611972894.5.

[35] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, 1991. doi: 10.1137/1033004. URL https://doi.org/10.1137/1033004.

[36] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation1. In David A. Bader, editor, *Massive Graph Analytics*, pages 333–376. Chapman and Hall/CRC, 2022. doi: 10.1201/9781003033707-16. URL https://doi.org/10.1201/9781003033707-16.

[37] Laurent Perron and Vincent Furnon. OR-Tools, 2023. URL https://developers.google.com/optimization/.

[38] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR, 2020. URL http://proceedings.mlr.press/v119/tang20a.html.

[39] Vera Traub and Rico Zenklusen. A better-than-2 approximation for weighted tree augmentation. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1–12. IEEE, 2021. doi: 10.1109/FOCS52979.2021.00010. URL https://doi.org/10.1109/FOCS52979.2021.00010.

[40] Vera Traub and Rico Zenklusen. A (1.5+ϵ)-approximation algorithm for weighted connectivity augmentation. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1820–1833. ACM, 2023. doi: 10.1145/3564246.3585122. URL https://doi.org/10.1145/3564246.3585122.

[41] UserBenchmark. UserBenchmark, 2023. URL https://www.userbenchmark.com/.

[42] Pravin M. Vaidya. Speeding-up linear programming using fast matrix multiplication (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 332–337. IEEE Computer Society, 1989. doi: 10.1109/SFCS.1989.63499. URL https://doi.org/10.1109/SFCS.1989.63499.

[43] Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 259–278. SIAM, 2020. doi: 10.1137/1.9781611975994.16. URL https://doi.org/10.1137/1.9781611975994.16.

[44] Toshimasa Watanabe, Toshiya Mashima, and Satoshi Taoka. The k-edge-connectivity augmentation problem of weighted graphs. In Toshihide Ibaraki, Yasuyoshi Inagaki, Kazuo Iwama, Takao Nishizeki, and Masafumi Yamashita, editors, *Algorithms and Computation*, pages 31–40, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[45] Toshimasa Watanabe, Toshiya Mashima, and Satoshi Taoka. Approximation algorithms for minimum-cost augmentation to k-edge-connect a multigraph. In *1993 IEEE International Symposium on Circuits and Systems, ISCAS 1993, Chicago, Illinois, USA, May 3-6, 1993*, pages 2556–2559. IEEE, 1993.

[46] Toshimasa Watanabe, Satoshi Taoka, and Toshiya Mashima. Maximum weight matching-based algorithms for k-edge-connectivity augmentation of a graph. In *International Symposium on Circuits and Systems (ISCAS 2005), 23-26 May 2005, Kobe, Japan*, pages 2231–2234. IEEE, 2005. doi: 10.1109/ISCAS.2005.1465066. URL https://doi.org/10.1109/ISCAS.2005.1465066.

Test Instances

10th DIMACS Implementation Challenge	n	m
queen8	64	728
karate	34	78
jazz	198	2 742
email	1 133	5 451
vibrobox	12 328	165 250
t60k	60 005	89 440
coAuthorsCiteseer	227 320	814 134
coPapersCiteseer	434 102	16 036 720
delauney_n20	1 048 576	3 145 686
M6	3 501 776	10 501 936
Cycles		
cycle-50	50	50
cycle-100	100	100
cycle-200	200	200
cycle-500	500	500
cycle-1000	1 000	1 000
cycle-5000	5 000	5 000
Stars		
star-200	200	199
star-1000	1 000	999
star-5000	5 000	4 999
Cacti		
cactus{01, 02, 03, 04, 05}	100	109
cactus{06, 07, 08, 09, 10}	100	119
cactus{11, 12, 13, 14, 15}	200	279
cactus{16, 17, 18, 19, 20}	1000	1199

Table A1: Complete set of test instances used in the evaluation