2023

An Empirical Analysis of Approximation Algorithms for the Unweighted Tree Augmentation Problem

Jacob Thomas Restanio
West Virginia University, jtr0018@mix.wvu.edu

An Empirical Analysis of Approximation Algorithms for the Unweighted Tree Augmentation Problem

Jacob Restanio

Thesis submitted
to the Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Science

K. Subramani, Ph.D., Chair
Donald Adjeroh , Ph.D.
Piotr Wojciechowski, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2022

Keywords: Tree Augmentation

Abstract

An Empirical Analysis of Approximation Algorithms for the Unweighted Tree Augmentation Problem

Jacob Restanio

In this thesis, we perform an experimental study of approximation algorithms for the tree augmentation problem (TAP). TAP is a fundamental problem in network design. The goal of TAP is to add the minimum number of edges from a given edge set to a tree so that it becomes 2-edge connected. Formally, given a tree $T = (V, E)$, where V denotes the set of vertices and E denotes the set of edges in the tree, and a set of edges (or links) $L \subseteq V \times V$ disjoint from E, the objective is to find a set of edges to add to the tree $F \subseteq L$ such that the augmented tree $(V, E \cup F)$ is 2-edge connected. Our goal is to establish a baseline performance for each approximation algorithm on actual instances rather than worst-case instances. In particular, we are interested in whether the algorithms rank on practical instances is consistent with their worst-case guarantee rankings. We are also interested in whether preprocessing times, implementation difficulties, and running times justify the use of an algorithm in practice. We profiled and analyzed five approximation algorithms, viz., the Frederickson algorithm [16], the Nagamochi algorithm [31], the Even algorithm [12], the Adjiashivili algorithm [1], and the Grandoni algorithm [19]. Additionally, we used an integer program and a simple randomized algorithm as benchmarks. The performance of each algorithm was measured using space, time, and quality comparison metrics. We found that the simple randomized is competitive with the approximation algorithms and that the algorithms rank according to their theoretical guarantees. The randomized algorithm is simpler to implement and understand. Furthermore, the randomized algorithm runs faster and uses less space than any of the more sophisticated approximation algorithms.

Acknowledgments

I would like to thank my advisor and chair of my committee, Dr. K. Subramani, whose many insights and directions has given me inspiration and motivation in both my exploration and research into many problems in computer science. His experience in previous implementation and practical analysis allowed me to approach my implementation with more forethought. Without his help this thesis would not have been possible. Without his keen eye for detail and language this thesis would not be nearly as easy to read.

I would like to extend my sincere thanks to the Air Force Research Laboratory. The funding from the summer fellowship and opportunity to conduct research at the Air Force lab made it possible to complete my thesis and gave me the opportunity to meet and discuss with many like-minded peers.

The project would not have been possible without the help of Dr. Alvaro Velasquez, whose work on the exact integer program was a great service and allowed me to focus my time and efforts on the approximation algorithms. The project would also not have been possible without Cody Klingler, and acquaintance and friend whose collaboration on the Nagamochi algorithm was the only reason that algorithm was able to be implemented.

I extremely grateful to my family and friends, whose support has kept me in school and following my dreams and passions. I would have quit many times if it was not for the encouragement of my family. I would have gone mad many times if I did not have the occasional distraction with my friends.

Contents

1 Introduction **1**

2 Statement of Problem **4**

2.1 Preliminaries 4

2.2 Tree augmentation problem (TAP) 8

3 Motivation and Related Work **10**

4 Empirical Setup **15**

4.1 Trees Types 15

4.2 Checking for Validity of Solutions 16

4.3 Algorithms considered 17

4.3.1 The Randomized Tree Augmentation Algorithm 17

4.3.2 Exact Integer Program 18

4.3.3 Frederickson Algorithm 19

4.3.4 Nagamochi Algorithm 22

4.3.5 Even Algorithm 22

4.3.6 Adjiashvili Algorithm 23

4.3.7 Grandoni Algorithm 24

4.4 Hardware and Implementation 27

5 Results **29**

5.1 Size 10 29

5.2 Size 100 32

5.3 Size 1000 34

6 Conclusion **36**

List of Figures

1.1 A network where a single edge failure disconnects it 1
1.2 A network being disconnected by the removal of an edge 1
1.3 A network that is single fault tolerant 1
1.4 Reducing a graph to a tree 2

2.1 A 2-edge connected graph. 8
2.2 An instance of TAP and an optimal solution 9

4.1 Frederickson example 20

5.1 Box plots of each algorithm and each graph type of size 10. 30
5.2 The running time of the algorithms on all graphs of size 10. 31
5.3 The memory usage of the algorithms on all graphs of size 10. 32
5.4 Box plots of each algorithm and each graph type of size 100. 33
5.5 The running time of the algorithms on all graphs of size 100. 33
5.6 The memory usage of the algorithms on all graphs of size 100. 33
5.7 Box plots of each algorithm and each graph type of size 1000. 34
5.8 The running time of the algorithms on all graphs of size 1000. 34
5.9 The memory usage of the algorithms on all graphs of size 1000. 35

List of Algorithms

1 Randomized Tree Augmentation 18

2 Frederickson Bridge-connectivity Augmentation 20

3 Edmond's Minimum Weighted Arborescence 21

4 Nagamochi Tree Cover Algorithm 22

5 Even Tree Cover Augmentation 23

6 Adjiashvili Algorithm 24

Chapter 1

Introduction

TAP is a well-studied, fundamental networking problem [25] [19] [34]. Consider the network shown in Figure 1.1. The network becomes disconnected if any of the connections in the network are removed (Figure 1.2). A network that can become disconnected by a single point of failure is vulnerable to incidental failures or attack. It is imperative any network that serves a critical role is made single fault tolerant by adding additional connections so that any one failure will not disconnect the network (Figure 1.3). However, adding connections to a network has an associated cost. It is therefore vital to add as few connections as possible. It is valuable to know the fewest number of additional connections necessary to make the network single fault tolerant. We define a graph to be 2-edge connected when any edge can be removed without breaking connectivity. TAP takes a tree as input and the objective is to find the minimum number of edges to add to make the tree 2-edge connected.

Formally, we are interested in the fundamental graph connectivity augmentation prob-

a)

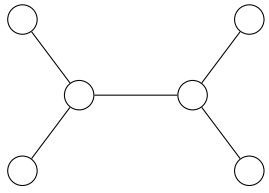


Figure 1.1: A network where a single edge failure disconnects it

b)

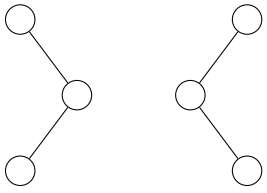


Figure 1.2: A network being disconnected by the removal of an edge

c)

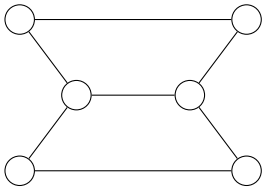


Figure 1.3: A network that is single fault tolerant

lem. Given a tree $T = (V, E)$, where V denotes the set of vertices and E denotes the set of edges in the tree, and a set of edges (or links) $L \subseteq V \times V$ that is disjoint from E, the objective is to find a set of edges to add to the tree $F \subseteq L$ such that the augmented tree $(V, E \cup F)$ is 2-edge connected. We formulate the problem by assuming the graph G we are given is a tree. We can make this assumption due to the fact that strongly connected components in the graph can be contracted into a single node (see Figure 1.4). Under this assumption, the problem is known as the Tree Augmentation Problem [11].

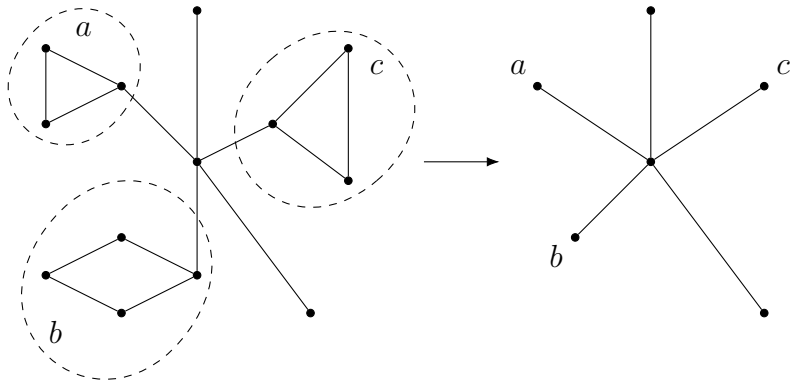


Figure 1.4: A graph being reduced to a tree by contracting the strongly connected components

We are able to assume the graph given is a tree without loss of generality. If we are given a graph, we can find all of the connected components in $O(|V| + |E|)$ time [36]. We can contract each connected component into a single node, making the graph into a tree. Assuming the graph is a tree eliminates the need to perform this search and contraction. The assumption also allows us to restrict our analysis to different types of trees instead of types of trees and graphs.

We only consider the unweighted variant of the problem, where any edge added is of unit cost. the unweighted variant can be viewed as finding the minimum number of edges to add that makes a tree into a 2-edge connected graph. In the weighted case, the optimal solution may not be the minimal edge solution. Most of the algorithms considered require only minor changes to deal with the weighted case.

The rest of this thesis is organized as follows: Chapter 2 formally specifies the problem

under consideration. Chapter 3 contains our motivation and describes the related work in the literature. The empirical setup for our experiments is described in Chapter 4. The results of our implementation are discussed in Chapter 5. We conclude in Chapter 6 by summarizing our results and identifying avenues for future research.

Chapter 2

Statement of Problem

In this chapter, we formally define the tree augmentation problem and also define the various terms that will be used in the rest of the thesis.

2.1 Preliminaries

A *decision problem* is a problem that has a *yes* or *no* solution. A problem is *decidable* if there exists a program to solve the problem in finite time. We group decision problems into a hierarchy of complexity classes.

R	problems decidable in finite time
EXP	problems decidable in exponential time $2^{n^{O(1)}}$
P	problems decidable in polynomial time $n^{O(1)}$

These sets are distinct, i.e. $\mathbf{P} \subsetneqq \mathbf{EXP} \subsetneqq \mathbf{R}$.

Definition 1. NP - *The complexity class NP is defined as the set of decision problems that can be solved by a nondeterministic Turing machine in $O(n^k)$ time. Formally,*

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k),$$

where **NTIME** *is the set of decision problems solved by a nondeterministic Turing machine and* $\mathbf{NTIME}(n^k)$ *constrains the set to only problems solvable in $O(n^k)$ time (polynomial time).*

Alternatively, **NP** can be defined using deterministic Turing machines as verifiers. **NP** is the set of decision problems in which there is a *verification* algorithm V that takes as input an instance I of a problem and a *certificate* bit string of length polynomial in the size of I, such that

- V always runs in polynomial time in the size of I,
- if I is a *yes* input, then there is some certificate c so that V outputs *yes* on input (I, c), and
- if I is a *no* input, then no matter what certificate c we choose, V always outputs *no* on input (I, c).

Definition 2. Reduction - *A reduction from problem A to problem B is a function that maps an instance x of problem A to an instance x' of problem B.*

Because problem B can be used to solve A, B is *at least as hard* as A. We often denote $A \leq_m B$ where m is the type of reduction. Problem A is **NP-hard** if every problem in **NP** has a log-space reduction to A. A problem is **NP-complete** under log-space reductions if it is **NP** and **NP-hard**. All **NP-complete** problems are reducible to each other.

Definition 3. NP-completeness - *A decision problem P is considered* **NP-complete** *under polynomial-time reductions if it is in* **NP***, and every problem in* **NP** *is reducible to P in polynomial time.*

In more practical settings, we are typically concerned with optimization problems. The goal of an optimization problem is to find the best solution to a given problem.

Definition 4. Optimization Problem - *An optimization problem consists of the following:*

- *a set of instances of the problem;*
- *for each instance, a set of possible solutions, each with a nonnegative cost; and,*

- *an objective, either minimization or maximization.*

The goal of an optimization problem is to find a solution for any instance that achieves the objective for the cost.

We use the shorthand OPT(x) to refer to the cost of the optimal solution for an instance x.

Definition 5. Linear Programming (LP) - *Linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints.*

Linear programs are problems that can be expressed in canonical form as

maximize (or minimize)	$\sum_{i=1}^{n} c_i x_i$
subject to	$\sum_{i=1}^{n} a_{j,i} x_i \leq b$
and	$\mathbf{x} \geq \mathbf{0}$.

The elements of $\mathbf{x}$ are the variables to be determined. The function whose value is to be maximized or minimized, $\mathbf{x} \mapsto \mathbf{c^T x}$, is called the objective function. The inequalities $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ are called constraints.

Definition 6. Approximation Algorithm - *An approximation algorithm is an algorithm that finds an approximate solution to optimization problems with provable guarantees on the distance of the returned solution to the optimal one.*

Definition 7. ϵ-approximation Algorithm - *Let M be an approximation algorithm. M is considered an ϵ-approximation algorithm, where $\epsilon \geq 0$:*

- *in the minimization case if for all valid instances x,* $\dfrac{M(x)}{OPT(x)} \leq \epsilon$ $(\epsilon \geq 1)$*;*

- *in the maximization case if for all valid instances x,* $\dfrac{M(x)}{OPT(x)} \geq \epsilon$ $(\epsilon \leq 1)$.

We generally only have interest in approximation algorithms that run in polynomial time. If a problem has an exact solution in exponential time, there is not as much benefit to approximating it in exponential time as well. There are many factors that influence

the quality of an approximation algorithm beyond the approximation factor ϵ, such as space usage and running time. The ϵ approximation factor for a given approximation algorithm may also not be tight. Solutions for a 2-approximation algorithm, for example, may generally give results closer to 1.5 of the optimal solution.

Definition 8. Integrality Gap - *The maximum ratio between the solution quality of an integer program and of its linear programming relaxation is defined as its integrality gap.*

Definition 9. Polynomial-time Approximation Scheme (PTAS) - *A PTAS is an algorithm which takes an instance of an approximation problem and a parameter $\epsilon > 0$ and produces a solution that is within a factor $1 + \epsilon$ of being optimal.*

The running time of a PTAS is polynomial in the problem size for every fixed ϵ, but can vary with different ϵ. So, both $O(n^{\frac{1}{\epsilon}})$ and $O(n^{2^{\frac{1}{\epsilon}}})$ would be considered PTAS in terms of running time.

Definition 10. APX - *The class* **APX** *is the set of* **NP** *optimization problems that have polynomial-time constant-factor approximation algorithms.*

Definition 11. PTAS-reduction - *A PTAS reduction is an approximability preserving reduction consisting of*

- *a pair of mappings $f(\cdot), g(\cdot)$, where f and g are computable in polynomial time.*
- *f maps an instance x of the problem A we are reducing to an instance x' of the problem B we are reducing to.*
- *g maps a feasible solution y' of an instance x' of the problem B to a feasible solution y of an instance s of the problem A.*
- *The following condition must hold:*
  - *$\forall \epsilon > 0$, $\exists \delta = \delta(\epsilon) > 0$, such that if y' is a $(1 + \delta(\epsilon))$-approximation to B, then y is a $(1 + \epsilon)$-approximation to A.*

Definition 12. APX-hard - *A problem is* **APX-hard** *if there exists a PTAS-reduction from any problem in* **APX** *to the given problem.*

2.2 Tree augmentation problem (TAP)

Before discussing TAP, we first need to define edge connectivity, since it is a fundamental property of TAP.

Definition 13. Edge-Connectivity - *A graph $G = (V, E)$ is considered k-edge-connected if a subgraph $G' = (V, E \backslash X)$ is connected for all $X \subseteq E$ where $|X| < k$. More simply stated, a graph is k-edge connected if it remains connected when fewer than k edges are removed.*

A tree, by definition, is 1-edge connected. See Figure 2.1 for an example of a graph that is 2-edge connected. Notice that the removal of any edge keeps the graph connected. In order to disconnect the graph, you must remove at least 2 edges.

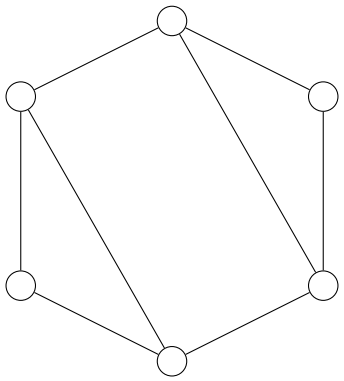


Figure 2.1: A 2-edge connected graph.

Tree Augmentation Problem

Instance: A tree $T = (V, E)$ and an edgeset $L \subseteq V \times V$ disjoint from E.

Problem: Find a minimum sized subset $F \subseteq L$ of edges such that $T = (V, E \cup F)$ is 2-edge-connected.

Note that we cannot choose any arbitrary edge to add to our tree to make it 2-edge connected. We are given a set of edges we must choose from. Figure 2.2 shows an instance of TAP. A simple lower-bound on the optimal solution is the ceiling of the number of leaves in the tree divided by 2. Since there are 5 leaves in the example, there must be at least 3 edges in the optimal solution. Since our solution contains only 3 edges, we know it

Figure 2.2: An instance of TAP and an optimal solution, the dashed lines represent the edgeset L and the red lines represent the selected edges F.

is also optimal. On any given instance, our edge set L may not contain leaf to leaf edges. So, not every instance will have an optimal solution that is equal to this lower bound. However, the given lower bound can be used to approximate $\text{OPT}(x)$ for the purpose of determining the solution quality of our approximation algorithms.

Some common definitions shared among multiple algorithms are as follows.

Definition 14. Shadow - *A link s is a shadow of another link l if the endpoints u and v of s are in the path P_l in the tree.*

Definition 15. Shadow-complete - *An instance of TAP is shadow-complete if for any link l and two vertices u and v in the path P_l, there exists a shadow link. You can assume every instance of TAP is shadow-complete. If you include a shadow s in the solution, you can change the link to l without uncovering any edges or increasing the number of links in your solution.*

Definition 16. Up-links, in-links, and cross-links - *Let r be the root of the tree. A link $l \in L_{up}$ is an up-link if one of its two endpoints u and v, say u, lies on the path from r to v. A link $l \in L_{in}$ is an in-link if the path P_l does not cross the root r, but it can end at r. A link $l \in L_{cross}$ is a cross-link if the path P_l crosses r but does not end at r.*

Chapter 3

Motivation and Related Work

In this chapter, we give some applications and motivation for studying TAP. We also give a comprehensive overview of the work in the literature.

Graph augmentation, and by extension the tree augmentation problem, is of great interest in network design. TAP is equivalent to finding a minimum sized cover of a laminar set-family [7, 26, 29]. Laminar families and augmentation problems have applications to the generalized Steiner network problem, also called the survivable network design problem. The problem consists of finding minimum-cost subgraphs such that there are at least a number n unique paths between every pair of vertices. Another way to view the problem is given any cut in a subgraph, there are at least n crossing edges. The generalized Steiner network problem has applications to the design of water and electricity supply networks, communication networks, and any large system where inoperability a single point can affect other parts of the system [42, 41, 18, 23]. A survey of connectivity problems was conducted by Kortsarz and Nutov on general connectivity problems [26]. Despite there being numerous approximation algorithms proposed for TAP, none of them have been studied in an empirical experiment. More factors than theoretical guarantee affect the choice of best algorithm in practice.

The tree augmentation problem was first proposed among many other graph augmentation problems by Eswaran and Tarjan [11]. TAP was shown to have an efficient algorithm if the link set was complete and unweighted. When the edges are weighted

(WTAP), the problem was shown to be **NP-complete** by reduction from Hamiltonian circuits. Frederickson and Ja'Ja' showed the WTAP is **NP-complete** for the more general case on graphs by reduction from 3-dimensional matching [16]. It is important to note that TAP is **NP-complete** when the edge set given is not complete.

Tarjan and Eswaran gave an efficient algorithm to 2-connect any tree given a complete unweighted edge set [11]. Extending this work, Ueno, Kajitani, and Wada show an efficient method of finding the minimum number of unweighted edges that need to be added to a tree to produce a k-connected graph when the edge set is complete [40]. The proof relies on the fact that in a k-connected graph, each vertex must have a degree of at least k. Let $d(v)$ be the degree of vertex v. Let the deficiency of a vertex be the $max(k - d(v), 0)$ It follows that the minimum number of edges needed to be added to a tree to make it k connected is at least the sum of the deficiencies of the vertices divided by 2 (since each edge added increases the overall degree of the graph by 2). Kajitani and Ueno extend their work to directed trees being augmented to k-connected directed graphs as well by using the greatest of in-degree or out-degree deficiencies [24]. Extending the work to graphs in general, Cai and Sun give an efficient method for constructing a k-connected graph from any arbitrary unweighted graph even when the initial graph is disconnected [2]. All efficient algorithms rely on being able to select any edge from a complete edge set.

The first approximation algorithm for WTAP is a 2-factor approximation by Frederickson and Ja'Ja' [16]. The algorithm consists of reducing WTAP to the minimum spanning arborescence problem. The edges in the solution to the minimum spanning arborescence problem are added to the original tree to make it 2-edge connected. The algorithm runs in $O(|V|^2)$ time. Frederickson and Ja'Ja's algorithm assumed the input was a connected tree. Khuller and Vishkin gave a 2-factor approximation for undirected graphs that were not necessarily connected [25]. The algorithm runs in $O(|E| + |V| \cdot \log |V|)$ time.

Nagamochi proposed an approximation algorithm that broke the 2-factor barrier [31]. The algorithm has an approximation factor of $(1.875 + \epsilon)$. The algorithm consists of reducing any one of four cases repeatedly on the input tree, each case contracting the

graph, until none of the cases hold. The algorithms then checks whether a certain condition holds and chooses between two subroutines. These steps alternate until the graph is reduced to a single vertex. The same principle idea was later built on by Even et al. to reduce the factor to 1.5 [12], though the initial proof was long and complex. Even et al. also devised a much simpler proof that shows a factor 1.8 solution [13] for their algorithm. Even et al. then provided another factor 1.5 approximation proof [14]. It was shown recently that there was in error in [14] but the Even et al algorithm was a 1.5 factor approximation solution in a new proof by Kortsarz and Nutov [27]. The algorithm consists of greedy contractions, rooting the subtree and contracting the tree at the rooted subtree, and using a credit system to determine which vertex to root the subtree in the next iteration. Traub and Zenklusen designed a relative greedy algorithm, which takes a weak greedy solution and iteratively improves the solution by replacement and has a $(1 + \ln 2)$-factor approximation [38]. They improve their original algorithm using local-search to achieve a $(1.5 + \epsilon)$-factor approximation [39].

Let $f \in L$ be the subset of links that will be inserted to the tree to make it 2-edge connected. Guo and Uhlmann show that TAP admits a problem kernel with $O(f^2)$ vertices and links [20]. Expanding on that, Marx and Végh show that the minimum cost augmentation of edge-connectivity from $(k-1)$ to k with at most f new edges is fixed-parameter tractable parameterized by f and admits a polynomial kernel [30].

Much work has been done with respect to LP-based algorithms. The integrality ratio was originally conjectured to be $\frac{4}{3}$ by Cheriyan, Jordán, and Ravi [7]. They also give a $\frac{4}{3}$ approximation algorithm for a special case of TAP called the tree plus cycle. Later, Cheriyan et al. showed that the integrality ratio approaches $\frac{3}{2}$ [8]. Whether the standard LP-relaxation, called the cut-LP, had an integrality gap less than 2 for unit costs was open until Nutov showed the integrality gap is at most $\frac{28}{15}$ [35]. Additionally, Nutov gives another LP-relaxation for TAP that has an integrality gap at most $\frac{7}{4}$.

Kortsarz and Nutov give two LP-relaxations with respect to the unweighted tree augmentation problem that are a 1.75-factor approximation using a primal-dual approach [28]. An LP-based approximation algorithm for TAP was given by Adjiashvili

that achieves an approximation ratio of $(\frac{5}{3}+\epsilon)$ which uses bundle constraints [1]. The algorithm consists of two phases, in the first phase, the fractional LP solution guides a simple decomposition method that breaks the tree into well-structured tree with the corresponding part of the LP solution. The second phase rounds each decomposition to an integral solution with two rounding procedures. One rounding procedure exploits the constraints of the LP, while the other exploits a connection to the edge cover problem. Adjiashivili also gives the first approximation algorithm to break the 2-factor barrier for WTAP using the bundle LP to achieve a $(1.96417+\epsilon)$-factor [1]. Fiorini et al. improves on Adjiashvili's algorithm by introducing an LP that uses $\{0,\frac{1}{2}\}$-Chvátal-Gomory cuts in addition to the bundle constraints to achieve a $(\frac{3}{2}+\epsilon)$-approximation for both WTAP and TAP [15]. Cheriyan and Gao show another $(1.5+\epsilon)$-approximation using an SDP relaxation [5, 6]. They propose a combinatorial algorithm that uses greedy contractions and adds edges based on semiclosed trees. The analysis is done using SDP relaxation to obtain their bounds, which produces fractional solutions of the Lasserre system via decomposition. Grandoni, Kalaitzis, and Zenklusen break the 1.5 barrier by improving on Fiorini et al. to achieve a 1.458-approximation [19]. They accomplish this by providing a number of new techniques. They give a new approximation algorithm for $O(1)$-wide tree instances. They use the property that wide trees naturally decompose into smaller subtrees with constant number of leaves, rather than rounding each subtree independently. They also use rewiring techniques in the rounding scheme to reduce the number of edges added to the integral solution.

Censor-Hillel and Dory extend the application of TAP and WTAP to distributed network connectivity and give a 2-approximation algorithm that is considerably faster than the other algorithms in the literature [4]. An online version of connectivity problems is studied in [21]. The work is extended to an online version of WTAP and a constant factor algorithm is given in [32]. Iglesias and Ravi use a top-down coloring algorithm to achieve a $\frac{3}{2}$-approximation for special cases of WTAP [22].

The literature contains numerous examples of related and adjacent connectivity problems (e.g. [23, 26, 33, 9, 35]). The work for TAP and WTAP is often extended to

such generalized connectivity problems. A 1.393-approximation algorithm is given for the more general connectivity augmentation problem, and therefore also TAP [3]. This is the current best known approximation algorithm.

Chapter 4

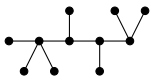
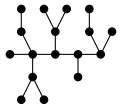
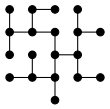
Empirical Setup

In this chapter, we describe the setup for our experiment. We detail what types of trees we use as input. We explain how we check a solution is valid. We give a more in depth overview of the algorithms being tested. We describe the hardware that is used for the experiment.

4.1 Trees Types

The trees used to evaluate the algorithms were of the following six types:

Path Tree	A tree whose vertices can be listed in the order $v_1, v_2, ..., v_n$ such that the edges are $v_i, v_i + 1$ where $i = 1, 2, ..., n - 1$.	
Star Tree	A tree that contains exactly one vertex with degree greater than 1 and $(n - 1)$ leaves.	

Star-like Tree	A tree that consists of a root node with any number of paths attached to it and contains exactly one vertex with degree greater than 2.	
Caterpillar Tree	A tree where all vertices are within distance 1 of a central path.	
Lobster Tree	A tree where all vertices are within distance 2 of a central path.	
Uniform Spanning Tree	A spanning tree selected such that all spanning trees on the same vertex set are equally likely.	

Each algorithm was evaluated using generated trees of each tree type on sizes 10, 100, and 1000.

4.2 Checking for Validity of Solutions

Checking if a given graph is 2-edge connected can be done in $O(|V|+|E|)$ time using depth-first search. The algorithm finds bridges in a graph by checking if there is an alternate path in the DFS tree to any ancestor of v from the subtree rooted at v. If no bridges are found, the graph is 2-edge connected. When talking about the tree augmentation problem, we call the edge set in the tree E and the edge set being considered (the

links) L. In this case, the running time for checking if a graph is 2-edge connected is $O(|V| + |E| + |L|)$.

4.3 Algorithms considered

We consider the following seven algorithms in our experiment.

Algorithm	$\boldsymbol{\rho(n)}$	**Time Complexity**
Randomized	N/A	$O(\|V\|^2)$
Exact	1	$O(2^{\|V\|})$
Frederickson [16]	2	$O(\|V\|^2)$
Nagamochi [31]	$1.875 + \epsilon$	$O(\|V\|^{\frac{1}{2}}\|L \cup E\| + \|V\|^2)$
Even, et al [13]	1.5	$O(\|V\|^3)$
Adjiashvili [1]	$1.666 + \epsilon$	$\|V\|^{O(1)}$
Grandoni [19]	1.458	$\|V\|^{O(1)}$

Table 4.2: The algorithms implemented, approximation factor, and time complexity

4.3.1 The Randomized Tree Augmentation Algorithm

In order to test the practicality of each approximation algorithm, a simple randomized algorithm is also tested. The randomized algorithm is given in Algorithm 1. The randomized algorithm first arbitrarily selects edge for all leaf vertices in the tree and adds each edge to the solution set F. Selecting edges from all leaf vertices is justified because in order for a graph to be 2-edge connected, it cannot have any leaves. The next step is to check if the tree is 2-edge connected. If not, at random and uniformly select a vertex $v \in V$ and at random and uniformly add an edge incident to v not in the solution set. Since the randomized algorithm is relatively fast, we can run the algorithm multiple times and choose the smallest solution. For this experiment, we ran the algorithm 100 times on each instance.

Lemma 1. *The random tree augmentation algorithm runs in time* $O(|L| \cdot (|V| + |E|))$.

Proof. Inserting an edge to the solution takes constant time. When an edge is added to the the solution F, it is removed from L so it cannot be selected again. So, the most edges added to the solution possible is $|L|$. Once an edge is selected, the graph is checked to see if it contains bridges, which takes $O(|V| + |E|)$ time. Therefore, the algorithm cannot take more than $O(|L| \cdot (|V| + |E|)$ time. □

Lemma 2. *The random tree augmentation algorithm uses $O(|V| + |L|)$ space.*

Proof. Checking if a graph is 2-edge connected uses $O(|V|)$ space. The only data structure that the random tree augmentation algorithm needs is an auxiliary graph to store the solution. We use an adjacency list data structure. The maximum number of edges to be added to F is $|L|$. So, the space required is $O(|V| + |E|)$. Hence, the total space usage is $O(|V| + |L|)$. □

Algorithm 1 Randomized Tree Augmentation

```
1: procedure RANDOM(T, L)
2:     F ← ∅
3:     for all v ∈ T do
4:         if degree of v < 2 then
5:             Select a random edge e ∈ L incident to v
6:             F ← e
7:     while T is not 2-edge connected do
8:         Select a random edge e ∈ L where e ∉ F
9:         F ← e
10:    return F
```

4.3.2 Exact Integer Program

This is the well-known set cover formulation for the tree augmentation problem. Let $V[T]$ and $E[T]$ denote the vertices and edges of tree T. Let L (*links*) denote the set of edges disjoint from $E[T]$ (the edges in our tree). Denote a link l connecting two nodes u and v by uv. Associate with every link $l = uv \in L$ the unique path $P_{uv} \subseteq E[T]$ in T connecting u and v. A set of links $S \subseteq L$ is a solution for the TAP instance if and only if the union of the corresponding paths covers the edge set of T. Namely, if $\cup_{l \in S} P_l = E[T]$.

For a set $X \subseteq E[T]$ denote by $\text{cov}(X) \subseteq L$ the set of links that cover at least one edge of X. When X is a singleton, we write $\text{cov}(e)$. The exact integer program contains one variable x_l for each link $l \in L$ and asks to solve

$$\begin{aligned} \text{minimize} \quad & \sum_{l \in L} x_l \qquad \text{subject to} \\ & \sum_{l \in \text{cov}(e)} x_l \geq 1 \quad \forall e \in E[T], \\ & x_l \in [0,1] \quad \forall l \in L. \end{aligned} \tag{4.1}$$

4.3.3 Frederickson Algorithm

Frederickson and Ja'Ja' gave the first approximation algorithm for the tree augmentation problem in [16]. They called the problem BRIDGE-CONNECTIVITY AUGMENTATION. The algorithm (Algorithm 2) given is a 2-approximation algorithm that uses a minimum weight arborescence to find an approximate solution. The conceptual idea of the algorithm is as follows. Root the tree at any leaf r and direct all edges in the tree towards r. Next, add the links as bidirected edges to the tree. Assign a weight of 1 to each link edge and 0 to edges in the tree. Find a minimum weight arborescence on the weighted directed edges. The algorithm finishes by combining the edges in the arborescence with the edges in the tree as an undirected graph. An example of this process can be seen in Figure 4.1.

The algorithm has a few nice properties. First, weights of 1 are used in the unweighted case, so the algorithm can easily be altered to supported the weighted version of TAP. Second, the algorithm has a fast running time, with most of the running time being spent on finding the minimum arborescence. We used Edmond's algorithm (Algorithm 3) for finding a weighted arborescence, which runs in $O(|E| \cdot |V|)$ [10]. Edmond's algorithm was selected for simplicity in its implementation. The original paper gives a running time of $O(|V|^2)$, which assumes using an algorithm from Tarjan that runs in $O(|V|^2)$ time on dense graphs [37]. There exists a more complex minimum spanning arborescence

algorithm that runs in $O(|E| + |V| \cdot \log |V|)$ using Fibonacci heaps [17].

Algorithm 2 Frederickson Bridge-connectivity Augmentation

```
1:  procedure FREDERICKSON(T, L)
2:      F ← ∅
3:      r ← v ∈ T such that v is a leaf
4:      for all e ∈ E do
5:          A' ← e such that e is directed toward r
6:      for all e = (u, v) ∈ L ∪ E do
7:          A ← e_1, e_2 such that e_1 = ⟨u, v⟩ and e_2 = ⟨v, u⟩
8:      for all e = ⟨u, v⟩ ∈ A do
9:          if e ∈ A' and v ≠ r then
10:             cost(e) ← 0
11:         else if e =∈ A' and v = r then
12:             cost(e) ← ∞
13:         else
14:             cost(e) ← 1
15:     A'' ← EDMONDS(D = (V, A))
16:     for all e = ⟨u, v⟩ ∈ A'' do
17:         if cost(e) > 0 then
18:             F ← (u, v)
19:     return F
```

Lemma 3. *The Frederickson bridge-connectivity augmentation algorithm uses $O(|V|^2)$ space.*

Proof. Creating a directed tree uses $O(|V| + |E|)$ space. Creating a structure to store

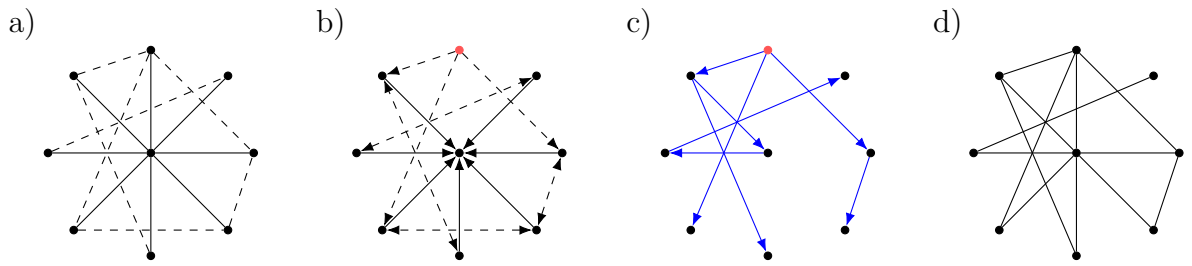


Figure 4.1: a) An instance of TAP with the tree given by solid lines and the links given by dashed lines. b) The instance transformed into a directed graph with the red leaf indicating the root. c) A minimum spanning arborescence from the directed graph in b. d) The recombined solution to TAP from the edges in the original tree and the edges in the arborescence from c.

edge weights using an adjacency list uses $O(|V|+|E'|)$ space. The arboresence is another directed tree that uses $O(|V|+|E|)$ space. Since Edmond's algorithm is recursive and could potentially be called $|V|-1$ times, creating a new structure to store the returned edge weights and arborescence would use a lot of space, relatively. Instead, we contract the arborescence and edge weights and use four auxiliary arrays of size $|V|+1$ during expansion. The contraction removes having to use $O(|V|\cdot|E|)$ space due to the recursion and instead uses $O(|V|^2)$. Hence, the total space usage is $O(|V|^2)$. □

Algorithm 3 Edmond's Minimum Weighted Arborescence

```
1:  procedure EDMONDS(D = (V, E))
2:      for all v ∈ V do
3:          A ← e = ⟨u, v⟩ such that cost(e) = min(cost(e_i = ⟨u_i, v⟩ ∈ E))
4:          π(v) ← u
5:      if A contains a cycle C = (V_c, E_c) then
6:          v_c ← a new vertex representing the cycle
7:          V' ← V\V_c ∪ v_c
8:          if e = ⟨u, v⟩ ∈ E with u ∉ V_c and v ∈ V_c then
9:              E' ← e' where e' = ⟨u, v_c⟩
10:             cost(e') ← cost(⟨u, v⟩)− cost(⟨π(v), v)
11:         else if e = ⟨u, v⟩ ∈ E with u ∈ V_c and v ∉ V_c then
12:             E' ← e' where e' = ⟨v_c, v⟩
13:             cost(e') ← cost(⟨u, v⟩)
14:         else if e = ⟨u, v⟩ ∈ E with u ∉ V_c and v ∉ V_c then
15:             E' ← e' where e' = e
16:             cost(e') ← cost(e)
17:         A' ← EDMONDS(D' = (V', E'))
18:         e_c ← e' = ⟨u, v_c⟩ ∈ A'
19:         e_corr ← ⟨u, v⟩ the corresponding edge of e_c where ⟨u, v⟩ ∈ E and v ∈ V_c
20:         for all e ∈ E_c\e_corr do
21:             mark e
22:         for all e' ∈ A' do
23:             mark corresponding e ∈ E
24:         A ← marked edges
25:     return A
```

4.3.4 Nagamochi Algorithm

The Nagamochi algorithm was the first algorithm to break the 2-factor approximation [31]. However, the algorithm itself is complex and requires checking for a lot of cases and structures inside the graph. The algorithm works by iteratively contracting the graph using subsets of vertices and constructing the link set L during the process. The proof for the running time $O(|V|^{\frac{1}{2}}|E' \cup E| + |V|^2)$ is given in the paper.

Algorithm 4 Nagamochi Tree Cover Algorithm

```
1: procedure NAGAMOCHI(T = (V, E), G = (V, L), ε)
2:     F ← ∅
3:     while T contains more than one vertex do
4:         while Cases 1, 2, 3, or 4 holds do
5:             Execute P1, P2, P3, or P4 respectively
6:             F ← edges retained by the procedure
7:         Choose a minimally leaf-closed subtree T[D(v)]
8:         if condition A3 holds in T[D(v)] then
9:             Compute an edge set F^apx ⊆ L which covers edges in T[D(v)]
10:            F ← F ∪ F^apx
11:            X ← vertices of edges e ∈ E covered by F^apx
12:            T ← T\X and G ← G\X
13:        else
14:            Choose a lowest solo edge g
15:            F_g ← all edges in T*_g
16:            Compute an edge set F^+ ⊆ L which covers F_g with ε > 0
17:            F ← F ∪ F^+
18:            X ← vertices of edges e ∈ E covered by F^+
19:            T ← T\X and G ← G\X
20:    return F
```

4.3.5 Even Algorithm

The Even algorithm has an interesting history. First presented in 2001 [12] as a 1.5-approximation as an extended abstract, the proof was excluded since it was long and complex. Eventually, in 2016 [27] a simple and elegant proof would be published to support the algorithm as a 1.5-approximation. The final evolution of the Even algorithm

(Algorithm 5) has numerous advantages to the Nagamochi algorithm. First, the algorithm has a superior approximation factor to both the Nagamochi and Frederickson algorithms. Second, there is no preprocessing required and fewer cases than in Nagamochi. The Even algorithm boasts a much faster running time mainly restricted by the speed of the blossom algorithm used in implementation. The paper does not include a running time analysis; we provide one for our implementation in Lemma 4.

Lemma 4. *The Even algorithm runs in time* $O(|E| \cdot |V|^2)$.

Proof. The blossom algorithm takes $O(|E| \cdot |V|^2)$ time. Finding a non-deficient semi-closed tree takes $O(|V| \cdot \log |V|)$ time. It is obvious the blossom algorithm dominates the running time. □

Algorithm 5 Even Tree Cover Augmentation

```
1: procedure EVEN(T = (V, E), L))
2:     F ← ∅
3:     M ← maximum matching in L(Lf, Lf)\W
4:     Assign 1 coupon to each unmatched leaf and r
5:     Assign 3/2 coupons to every link in M
6:     Exhaust greedy locking tree contractions
7:     while T\F has more than one node do
8:         Exhaust greedy link contractions and update F and M accordingly
9:         T' ← a non-deficient semi-closed tree of T\F
10:        F' ← an exact cover of T'
11:        Contract T with F'
12:    return F
```

4.3.6 Adjiashvili Algorithm

The Adjiashvili algorithm is an extension of the *cut LP* (4.1 relaxed to an LP), and is called the *bundle LP* [1]. For an integer $\gamma \in \mathbb{Z}_{\geq 1}$, a γ-*bundle* is a union of γ paths in G. These paths need not be disjoint. Denote by $\mathcal{B}_\gamma$ the set of all γ-bundles in G. The bundle LP contains all the constraints from the natural LP and constraints that ensure that each γ-bundle is covered in the fractional solution by links with sufficiently high cost. Formally, we add the following constraint to the natural LP.

$$\begin{aligned} \text{minimize} \quad & \sum_{l \in L} x_l \qquad \text{subject to} \\ & \sum_{l \in \text{cov}(e)} x_l \geq 1 \quad \forall e \in E[T], \\ & \sum_{l \in \text{cov}(e)} c_l \cdot x_l \geq \text{OPT}(B) \quad \forall B \in \mathcal{B}_\gamma, \\ & x_l \geq 0 \quad \forall l \in L. \end{aligned} \tag{4.2}$$

For any $X \subseteq E[G]$, $\text{OPT}(X) \in \mathbb{R}_{\geq 0}$ is the minimum cost of a set of links in L that covers all edges in X. Solving the bundle LP entails calculating the values $\text{OPT}(B)$ for all $B \in \mathcal{B}_\gamma$. This can be done in polynomial time whenever γ is constant and in time $n^{\gamma^{O(1)}}$ in general.

The rounding scheme used for TAP is as follows:

Algorithm 6 Adjiashvili Algorithm

```
1: procedure ADJIASHVILI(T = (V, E), L)
2:     F ← ∅
3:     X ← solution of LP_γ
4:     while there exists an edge e ∈ E covered only by up-links in X do
5:         F ← up-link l covering e that covers the most edges
6:         Contract T with l
7:     while there exists a link l connecting two leaves u, v ∈ V and l ∈ X do
8:         F ← l
9:         Contract T with l
10:    while there exists an uncovered leaf v ∈ T ∪ F do
11:        F ← an arbitrary l ∈ L ∩ X covering v
12:    return F
```

4.3.7 Grandoni Algorithm

The Grandoni algorithm is an improvement on Fiorini et al., which combines the Adjiashvili bundle constraints and a new set of constraints [19]. Grandoni improves the algorithm by introducing a rewiring step. Consider an IP of the form $\min\{c^\intercal x \mid Ax \geq$

$b, x \in \mathbb{Z}^n\}$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$. Then a *Chvátal-Gomory cut* is a constraint of the form

$$\lambda^\intercal Ax \geq \lceil \lambda^\intercal b \rceil$$

where the vector of multipliers $\lambda \in \mathbb{R}^m_{\geq 0}$ is chosen in such a way that $\lambda^\intercal A \in \mathbb{Z}^n$. When λ is restricted to be in $\{0, \frac{1}{2}\}^m$, the constraint is called $\{0, \frac{1}{2}\}$-*Chvátal-Gomory cut*. The necessary constraints can be derived by using the initial set of constraints from the *cut LP* (4.1 relaxed to an LP).

Define S as the set of nodes $S \subseteq V$ such that all edges with a corresponding λ-coefficient of $\frac{1}{2}$ are exactly the edges leaving the node set S. Such edges in the graph are defined as $\delta_G(S) = \{e = \{u, v\} \in E \mid u \in S, v \notin S\}$. Let $\pi(S)$ denote the *multiset* of links $l = (u, v)$ such that the path through the tree from u to v called P_l intersects $\delta_G(S)$, and the multiplicity of l is defined as $\lceil \frac{1}{2} |P_l \cap \delta_G(S)| \rceil$. The $\{0, \frac{1}{2}\}$-*Chvátal-Gomory cut* can then be rewritten as

$$x(\pi(S)) \geq \frac{|\delta_G(S)| + 1}{2}.$$

That is, the sum of the multiplicities of links in our solution covering the edges in our tree with a corresponding λ-coefficient of $\frac{1}{2}$ must be greater than half the number of such edges.

When combined with the *bundle LP* (4.2), the rounding scheme used by Fiorini has improvements over that used by Adjiashvili by reducing the cost of splitting in-links into two up-links. This improvement comes over the rounding scheme for WTAP in Adjiashvili that was not covered here, but Fiorini brings the approximation for WTAP down to $\frac{3}{2} + \epsilon$ which beats the $\frac{5}{3} + \epsilon$ approximation for unweighted TAP and can be used on instances of unweighted TAP.

Grandoni introduces a method of rewiring cross-links to further improve a solution. The method considers the leaves of principle subtrees $T_i = (V_i, E_i)$. A principle subtree is one that includes the root r, u which is a child of r, and all descendants of u. In order to be able to restrict the size of the leaves of all principle subtrees, the analysis consists of only trees with all principle subtrees having $O(1)$ leaves, bounded by some constant. We

call such trees $O(1)$-wide trees or k-wide instances. Such restrictions allows for the use of bundle constraints on the leaves of principle subtrees to find integral solutions for the restricted LP solution, which is shown in [1]. In [19] is a proof how algorithms for k-wide trees can be used to approximate any tree with negligible increase to the approximation factor.

Define two links l_1 and l_2 to be *shadow-minimal* if there exists no shadow s of one link such that the link could be replaced and cover the same path, that is there is no shadow such that $P_{l_1} \cup P_{l_2} = P_s \cup P_{l_2}$. A set of links $L' \subseteq L$ is *shadow-minimal* if all links in L' are pairwise shadow-minimal. Any instance of TAP can be replaced by links that are shadow-minimal. Each principle subtree T_i for a k-wide tree has at most k cross-links with an endpoint inside T_i. There consists of many possible such shadow-minimal link sets, so let $\Lambda_i \subseteq 2^{cov(E_i)}$ be the family of of all shadow-minimal subsets of all cross-links with one endpoint in V_i. This gives us our cross-links, but we also need the links that begin and end in the principle subtree. Let $C(i, R) \subseteq L$ be a minimum cardinality set of links with both endpoints in V_i that satisfies (i) $R \cup C(i, R)$ is shadow-minimal, and (ii) $R \cup C(i, R)$ covers E_i. $C(i, R)$ can be computed efficiently by contracting the edges of T_i covered by R and solving on the residual tree. Since the tree has no more than k leaves, it can be solved efficiently using the *bundle LP*. Additionally, we remove any link from the residual instance that is not shadow-minimal with R and shorten any remaining links in the solution to achieve a solution that is shadow-minimal.

Let $L_i^R = R \cup C(i, R)$, which represents the shadow-minimal cross-links and in-links that give an optimal solution to the subtree T_i. We can solve the entire instance of a k-wide tree by taking the union of all q principle subtrees. That is, $L_1^{R_1} \cup \ldots \cup L_q^{R_q}$. Form new LP constraints as follows. Let $\lambda_i^R \geq 0$ be a variable for each principle subtree T_i and each set $R \subseteq \Lambda_i$, where λ_i^R is interpreted as including the links L_i^R in the solution. Our new constraints, which are combined with the $\{0, \frac{1}{2}\}$-Chvátal-Gomory cut, is as follows:

$$\begin{aligned} x_l &= \sum_{R\in\Lambda_i : l\in L_i^R} \lambda_i^R, && \forall i \in \{1,\dots,q\}, \forall l \in cov(E_i) \\ \sum_{R\in\Lambda_i} \lambda_i^R &= 1, && \forall i \in \{1,\dots,q\}, \\ \lambda_i^R &\geq 0, && \forall i \in \{1,\dots,q\}, \forall R \in \Lambda_i \end{aligned} \tag{4.3}$$

Let (x, λ) be an optimal fractional solution, where x comes from the $\{0, \frac{1}{2}\}$-Chvátal-Gomory cut and λ comes from 4.3. Return the better of two solutions obtained by two different rounding procedures. The first is essentially following the steps in 6 using x. The second is detailed below as a randomized procedure.

Interpret the coefficients of λ_i^R as a probability distribution and sample from it, independently for each T_i. Let L_i be the sampled solution from T_i. Start by combining each local solution $\cup_{i=1}^{q} L_i$, which is a feasible integral solution. Each cross-link l that has its endpoint in T_i and T_j is contained in each of L_i and L_j with probability x_l. We improve the solution by rewiring pairs of cross-links into a single one. Consider all vertices in V_i with a cross-link in L_i for all subtrees. Maximizing the number of rewirings can be done by solving a matching problem over the graph consisting of all such vertices with edges being the cross-links between them.

4.4 Hardware and Implementation

Each algorithm was implemented in Python 3.10 using the NetworkX library. Where possible, the functions implemented by the NetworkX library were used for deletion, insertion, and merging of all graphs. Generation of the input graphs was done using the NetworkX functions. An exception is when NetworkX used a random variable to determine the number of vertices added to a generated graph or did not have a generator for a specific graph type. In order to ensure each graph was of the correct size, some generation functions were created manually. The graph types that had manual generation were the caterpillar, lobster, and starlike graphs.

Each algorithm was evaluated on the Thorny Flat High-Performance Computing (HPC) cluster. The CPU used by the cluster could either be an Intel(R) Xeon(R) Gold 6138 Processor or an Intel(R) Xeon(R) Gold 6126 Processor depending on availability at the time. Thorny Flat uses a time sharing scheme on a queue of user submitted tasks. For this reason, time was measured using clock cycles, rather than wall time.

Chapter 5

Results

In this chapter, we discuss the results of our experiments. We split the discussion into the three categories determined by the size of the input tree.

5.1 Size 10

The performance of each algorithm on each tree of size 10 is shown in Figure 5.1. The caterpillar and lobster graphs are very similar, which is to be expected since the lobster graph is just an extension of a caterpillar graph. All algorithms perform similarly on the path graph. The star graph is where the Frederickson performs the worst, coming close to its 2-factor upper bound. We will see this is exacerbated as the size of the graph increases. The starlike graph has the widest distribution, but the way the starlike graph is generated means that some instances are very close to path graphs while other instances are very close to star graphs. The Grandoni algorithm consistently performs better with respect to solution size than the Adjiashivili algorithm, which will be the case for all sizes. Since the Grandoni algorithm uses the bundle constraints of the Adjiashvili algorithm but improves upon it using additional techniques, this result is expected. The Even algorithm appears comparable to the Grandoni algorithm for graphs of size 10. It is not surprising that the randomized algorithm performs very well for small graphs, since the size of the solution set is relatively small. The randomized algorithm takes the best solution of 100 repetitions, so it is statistically likely to find a good solution for small

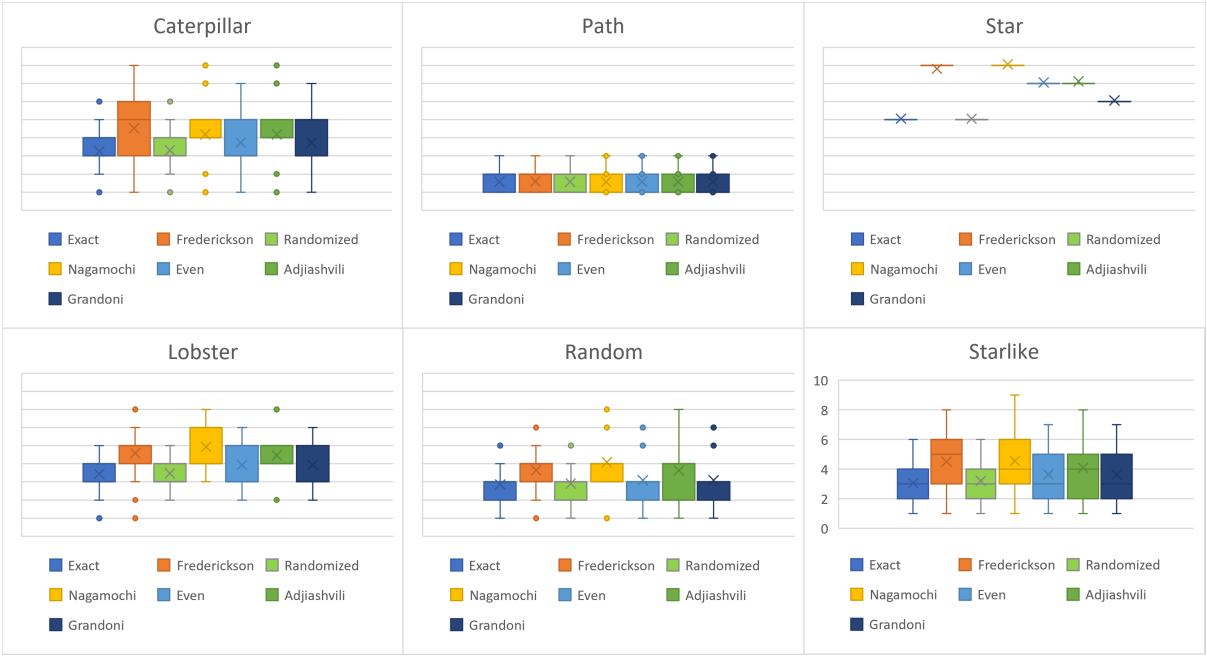


Figure 5.1: Box plots of each algorithm and each graph type of size 10.

graphs.

Comparing the algorithms time and space usage will highlight differences between the algorithms beyond solution quality. Figure 5.2 shows the time usage for each algorithm on graphs of size 10. The main bulk of time for the Frederickson algorithm comes from finding the minimum arborescence using Edmond's algorithm. There will not be much recursion on small graphs, so Edmond's algorithm does not have a significant impact on the running time. The Nagamochi algorithm runs slightly faster than the Even algorithm on small graphs due to less access of the subroutines. Since the Grandoni algorithm employs more techniques, such as the rewiring step, it has a consistently longer running time compared to the Adjiashvili algorithm. The exact algorithm is by far the slowest, and as the size of the graph increases, the exact algorithm did not finish finding a solution in a reasonable amount of time.

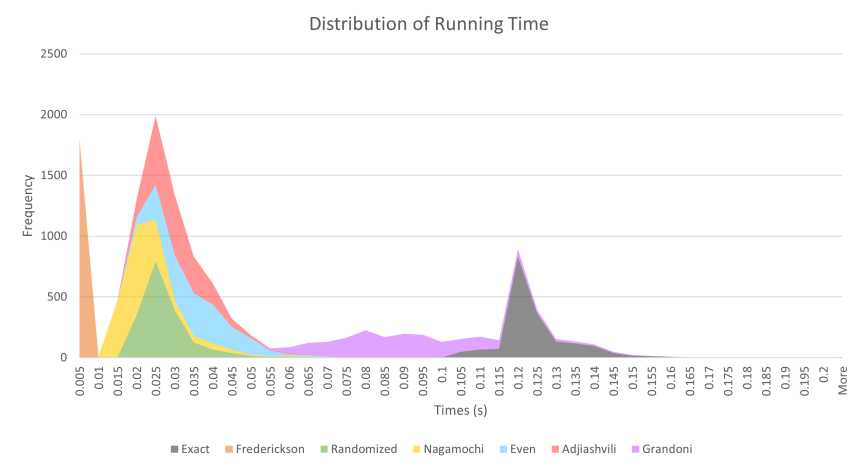


Figure 5.2: The running time of the algorithms on all graphs of size 10.

The peak memory usage for each algorithm on graphs of size 10 is shown in Figure 5.3. The Frederickson algorithm has to recurse up to $(|V| - 1)$ times in the worst case, making it the worst combinatorial algorithm when it comes to space metrics. This is not captured adequately by the memory graphs, since they only show the memory on the heap and recursive calls allocate memory to the stack. In fact, a naive implementation that contracts the graph in each recursion by making a new copy (motivated by the fact that we also need to expand the graph back to get our solution) quickly runs into space issues. Using up too much memory necessitates the use to swap memory to the disk, greatly slowing down the algorithm. A clever use of dictionaries to store the information of the contracted vertices and their edges prevents the space complexity from being $O(|V|^2 \cdot |E|)$ and keeps it as $O(|V|^2)$ instead. The Even and Nagamochi algorithms do not need to use auxiliary data structures, so they are very space efficient as well. The Adjiashvili and Grandoni algorithms require a structure to store the LP solution, using about twice the amount of space as the other approximation algorithms. The randomized algorithm uses an auxiliary structure to store edges, and needs to check for bridges on the selected edges combined with the original tree. Depending on implementation of the NetworkX library, combining the graphs might create a third auxiliary structure. The space usage of the randomized algorithm remains a constant factor of the input graph size. The large search space of the exact solution causes the most memory usage.

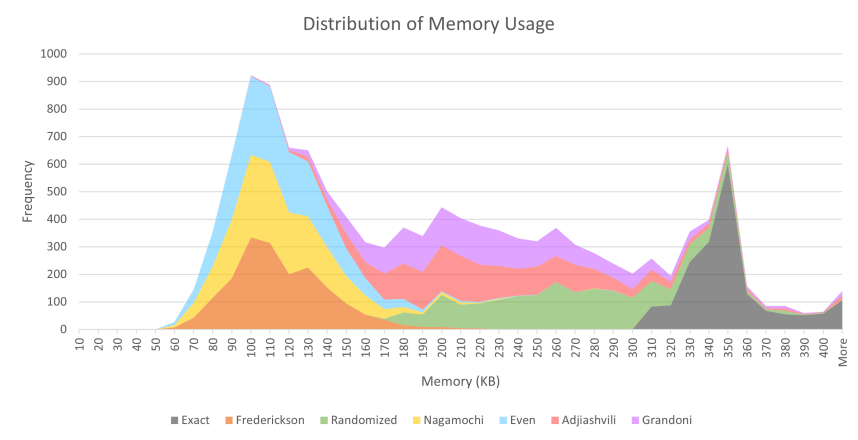


Figure 5.3: The memory usage of the algorithms on all graphs of size 10.

5.2 Size 100

As the graph size increases to 100, the differences between the algorithms become more clear with respect to solution sizes but less clear with respect to time and memory metrics (See Figures 5.4, 5.5, 5.6). The reason there are distinct peaks in the memory usage graph is because generated link sets were of density 0.01, 0.1, and 1. Each greater density of links generated requires more space in general, leading to the distinct spikes. The Nagamochi and Even algorithm are nearly identical in terms of running time and memory usage. The Frederickson algorithm remains the fastest in general. The running time of the the randomized algorithm reaches its worst relative performance at size 100. The speed of the randomized algorithm has not reached a point that it is fast enough where its repetitions do not affect the performance relative to the other algorithms. It is interesting to note that the theoretical bounds predict accurately the relative performance of each algorithm, except for the randomized algorithm. The randomized algorithm performs exceptionally well on smaller graphs and still wins out at size 100. Again, the exact algorithm could not find a solution in a time meaningful enough to include in this comparison.

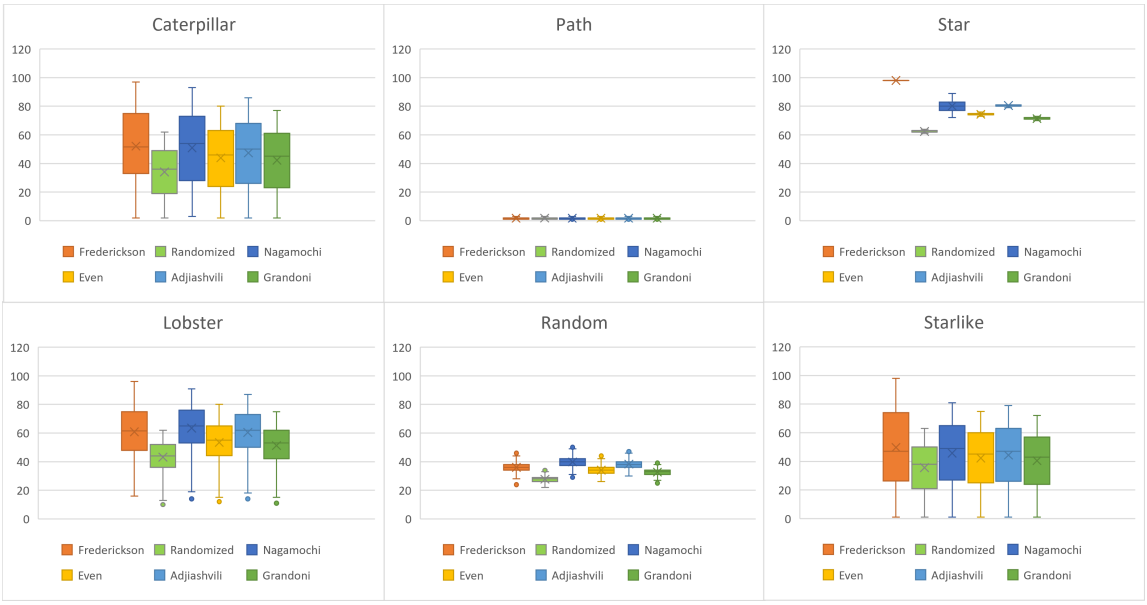


Figure 5.4: Box plots of each algorithm and each graph type of size 100.

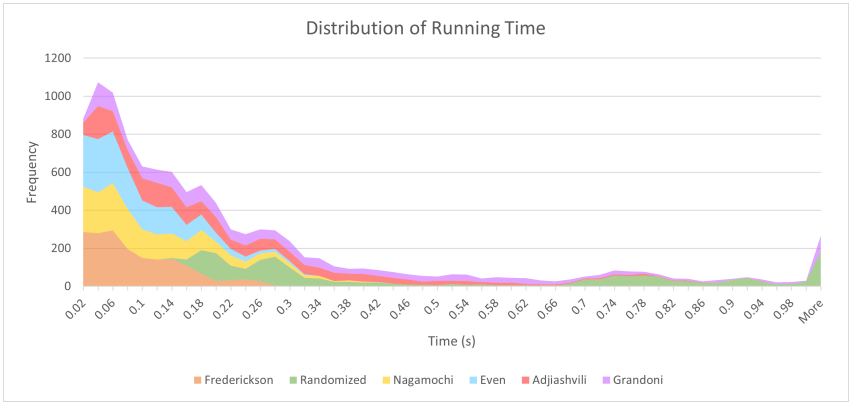


Figure 5.5: The running time of the algorithms on all graphs of size 100.

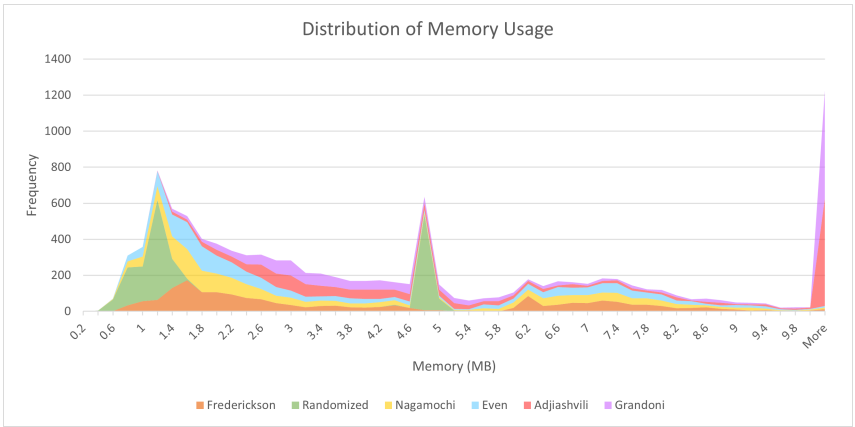


Figure 5.6: The memory usage of the algorithms on all graphs of size 100.

5.3 Size 1000

At size 1000, the randomized algorithm begins to be surpassed by the approximation algorithms that give a guarantee of at most 1.5 (See Figure 5.7). The solution set begins to grow such that the 100 repetitions are not sufficient to find good solutions. The linear programming algorithms start to take much longer than the other approximation algorithms, especially the Grandoni algorithm, since it includes a number of techniques that require searching on the graph (Figure 5.8). Memory is generally not a problem for any algorithm, though the library used for implementation of the linear programming causes the two linear programs to use the most memory (Figure 5.9).

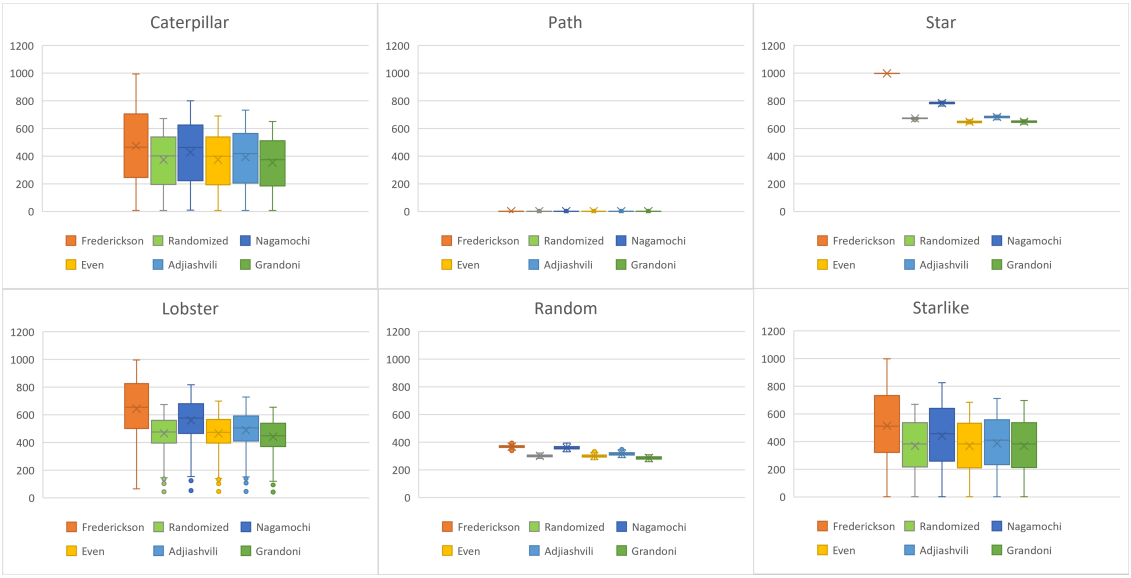


Figure 5.7: Box plots of each algorithm and each graph type of size 1000.

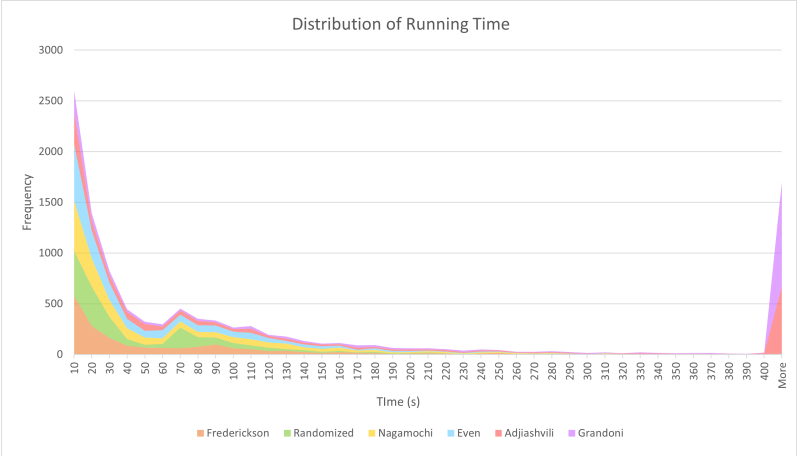


Figure 5.8: The running time of the algorithms on all graphs of size 1000.

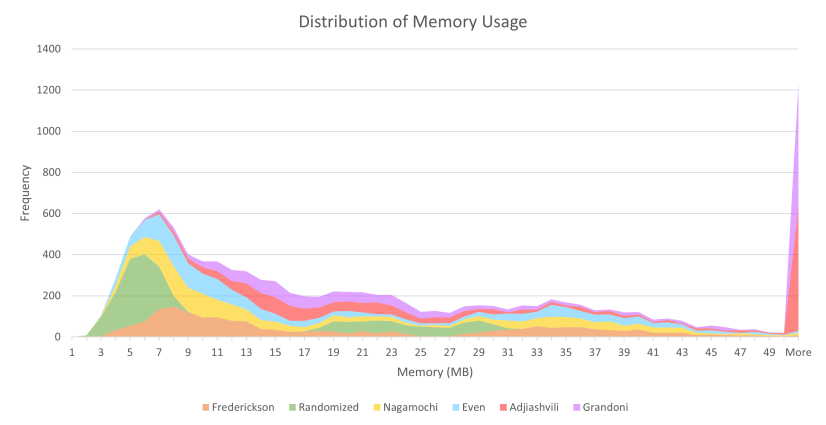


Figure 5.9: The memory usage of the algorithms on all graphs of size 1000.

Chapter 6

Conclusion

In this thesis, we studied approximation algorithms for the tree augmentation problem. We design a simple but novel randomized algorithm for the problem. We implemented several approximation algorithms found in the literature. The randomized algorithm performs as well as the exact algorithm on small graphs and continues to outperform the other algorithms until size 1000, where it remains competitive. The results conclusively show that algorithms with theoretical guarantees are not necessarily superior in practice when applied to TAP. The results also show the relative performances of the approximation algorithms are predicted by their relative theoretical guarantees. For very large graphs, the more sophisticated linear programming methods use more time. As graph sizes grow to be greater than 1000, the time usage of LP based methods begins to grow much faster than the combinatorial methods. The sophisticated but theoretically superior LP based methods are impractical to use on large graphs. The space usage for TAP is primarily caused by the space used for the input graph. The LP based algorithms use polynomial space in the size of the input which is unavoidable due to the techniques used to solve the LP. The combinatorial methods all use linear space in the size of the input since they all use small auxiliary data structures and searching. In conclusion, the exact integer program for TAP is infeasible on anything but small graphs due to time constraints. The LP based algorithms produce the best solutions but require more time and space than the combinatorial approaches. The combinatorial algorithms are both

fast and space efficient, but are the most difficult to implement. A simple randomized algorithm is fast, space efficient, and performs better or as well as on the graphs tested than the more advanced LP and combinatorial algorithms implemented in this thesis.

Future work includes using additional memory metrics, such as including stack memory and memory usage of subroutines for better analyses. We could also use more sophisticated time metrics, such as time usage of subroutines, for improved analyses. The simple randomized algorithm can be augmented to use greedy heuristics to further improve its results. There exist more algorithms that could be implemented for a more exhaustive comparison of the existing literature. Not all algorithms were selected for time purposes in implementation. Finally, the literature refers to [16] as showing weighted TAP to be **APX-hard**. However, the authors do not attempt to prove or use formal methods to show **APX-hardness** in this paper. The reduction used implies **APX-hardness**, but a more formal approach should be done using modern techniques. There are a number of open problems the literature has not addressed. It has not been shown whether unweighted TAP is **APX-hard**. There has been minimal work on producing better exact exponential algorithms. Similarly, an lower bound on the running time for exact algorithms could be found using techniques such as the exponential time hypothesis (ETH) and the strong exponential time hypothesis (SETH). Most work related to TAP has been strictly on finding better approximations. As a result, there remain many open problems and potential research using other techniques concerning TAP.

Bibliography

[1] David Adjiashvili. Beating approximation factor two for weighted tree augmentation with bounded costs. *ACM Trans. Algorithms*, 15(2), December 2018.

[2] Guo-Ray Cai and Yu-Geng Sun. The minimum augmentation of any graph to a k-edge-connected graph. *Networks*, 19(1):151–172, 1989.

[3] 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.

[4] Keren Censor-Hillel and Michal Dory. Fast distributed approximation for TAP and 2-edge-connectivity. *Distributed Comput.*, 33(2):145–168, 2020.

[5] Joseph Cheriyan and Zhihan Gao. Approximating (Unweighted) Tree Augmentation via Lift-and-Project, Part I: Stemless TAP. *Algorithmica*, 80(2):530–559, 2018.

[6] Joseph Cheriyan and Zhihan Gao. Approximating (Unweighted) Tree Augmentation via Lift-and-Project, Part II. *Algorithmica*, 80(2):608–651, 2018.

[7] Joseph Cheriyan, Tibor Jordán, and R. Ravi. On 2-Coverings and 2-Packings of Laminar Families. In Jaroslav Nesetril, editor, *Algorithms - ESA '99, 7th Annual European Symposium, Prague, Czech Republic, July 16-18, 1999, Proceedings*, volume 1643 of *Lecture Notes in Computer Science*, pages 510–520. Springer, 1999.

[8] Joseph Cheriyan, Howard J. Karloff, Rohit Khandekar, and Jochen Könemann. On the integrality ratio for tree augmentation. *Oper. Res. Lett.*, 36(4):399–401, 2008.

[9] S. Dhanalakshmi, N. Sadagopan, and V. Manogna. Tri-connectivity Augmentation in Trees. *Electron. Notes Discret. Math.*, 53:57–72, 2016.

[10] Jack Edmonds. *Optimum branchings*. National Bureau of Standards, 1967.

[11] Kapali P. Eswaran and R. Endre Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[12] Guy Even, Jon Feldman, Guy Kortsarz, and Zeev Nutov. A 3/2-approximation algorithm for augmenting the edge-connectivity of a graph from 1 to 2 using a subset of a given edge set (extended abstract). *Lecture Notes in Computer Science*, 10 2001.

[13] Guy Even, Jon Feldman, Guy Kortsarz, and Zeev Nutov. A 1.8 approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *ACM Transactions on Algorithms*, 5, 03 2009.

[14] Guy Even, Guy Kortsarz, and Zeev Nutov. A 1.5-approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *Inf. Process. Lett.*, 111(6):296–300, 2011.

[15] Samuel Fiorini, Martin Groß, Jochen Könemann, and Laura Sanità. Approximating weighted tree augmentation via chvátal-gomory cuts. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 817–831. SIAM, 2018.

[16] Greg N. Frederickson and Joseph Ja'Ja'. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

[17] Harold N. Gabow, Zvi Galil, Thomas H. Spencer, and Robert Endre Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Comb.*, 6(2):109–122, 1986.

[18] Michel X. Goemans, Andrew V. Goldberg, Serge A. Plotkin, David B. Shmoys, Éva Tardos, and David P. Williamson. Improved Approximation Algorithms for Network Design Problems. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 223–232. ACM/SIAM, 1994.

[19] Fabrizio Grandoni, Christos Kalaitzis, and Rico Zenklusen. Improved approximation for tree augmentation: saving by rewiring. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 632–645. ACM, 2018.

[20] Jiong Guo and Johannes Uhlmann. Kernelization and complexity results for connectivity augmentation problems. *Networks*, 56(2):131–142, 2010.

[21] Anupam Gupta, Ravishankar Krishnaswamy, and R. Ravi. Online and Stochastic Survivable Network Design. *SIAM J. Comput.*, 41(6):1649–1672, 2012.

[22] Jennifer Iglesias and R. Ravi. Coloring down: 3/2-approximation for special cases of the weighted tree augmentation problem. *Oper. Res. Lett.*, 50(6):693–698, 2022.

[23] Kamal Jain. A Factor 2 Approximation Algorithm for the Generalized Steiner Network Problem. *Comb.*, 21(1):39–60, 2001.

[24] Yoji Kajitani and Shuichi Ueno. The minimum augmentation of a directed tree to a k-edge-connected directed graph. *Networks*, 16(2):181–197, 1986.

[25] Samir Khuller and Ramakrishna Thurimella. Approximation Algorithms for Graph Augmentation. *J. Algorithms*, 14(2):214–225, 1993.

[26] Guy Kortsarz and Zeev Nutov. Approximating minimum cost connectivity problems. In Erik D. Demaine, MohammadTaghi Hajiaghayi, and Dániel Marx, editors, *Parameterized complexity and approximation algorithms, 13.12. - 17.12.2009*, volume 09511 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.

[27] Guy Kortsarz and Zeev Nutov. A Simplified 1.5-Approximation Algorithm for Augmenting Edge-Connectivity of a Graph from 1 to 2. *ACM Trans. Algorithms*, 12(2):23:1–23:20, 2016.

[28] Guy Kortsarz and Zeev Nutov. LP-relaxations for tree augmentation. *Discret. Appl. Math.*, 239:94–105, 2018.

[29] Yael Maduel and Zeev Nutov. Covering a laminar family by leaf to leaf links. *Discret. Appl. Math.*, 158(13):1424–1432, 2010.

[30] Dániel Marx and László A. Végh. Fixed-Parameter Algorithms for Minimum-Cost Edge-Connectivity Augmentation. *ACM Trans. Algorithms*, 11(4):27:1–27:24, 2015.

[31] Hiroshi Nagamochi. An approximation for finding a smallest 2-edge-connected subgraph containing a specified spanning tree. *Discrete Applied Mathematics*, 126(1):83–113, 2003. 5th Annual International Computing and combinatorics Conference.

[32] Joseph (Seffi) Naor, Seeun William Umboh, and David P. Williamson. Tight Bounds for Online Weighted Tree Augmentation. *Algorithmica*, 84(2):304–324, 2022.

[33] N. S. Narayanaswamy and N. Sadagopan. A Novel Data Structure for Biconnectivity, Triconnectivity, and k-Tree Augmentation. In Alex Potanin and Taso Viglas, editors, *Seventeenth Computing: The Australasian Theory Symposium, CATS 2011, Perth, Australia, January 2011*, volume 119 of *CRPIT*, pages 45–54. Australian Computer Society, 2011.

[34] 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.

[35] Zeev Nutov. On the tree augmentation problem. *Algorithmica*, 83(2):553–575, 2021.

[36] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[37] Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[38] 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.

[39] Vera Traub and Rico Zenklusen. Local Search for Weighted Tree Augmentation and Steiner Tree. 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 3253–3272. SIAM, 2022.

[40] Shuichi Ueno, Yoji Kajitani, and Hajime Wada. Minimum augmentation of a tree to a K-edge-connected graph. *Networks*, 18(1):19–25, 1988.

[41] David P. Williamson, Michel X. Goemans, Milena Mihail, and Vijay V. Vazirani. A primal-dual approximation algorithm for generalized steiner network problems. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 708?717, New York, NY, USA, 1993. Association for Computing Machinery.

[42] Pawel Winter. Generalized Steiner Problem in Series-Parallel Networks. *J. Algorithms*, 7(4):549–566, 1986.