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

Luke Hawranick Matthew Williamson Jacob Restanio K. Subramani *

Cody Klingler

Abstract

In this article, 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, namely, the Frederickson algorithm [14], the Nagamochi algorithm [25], the Even algorithm [10], and the Adjiashivili algorithm [1], using a simple randomized algorithm as a benchmark. The performance of each algorithm was evaluated using metrics for space usage, running time, and solution quality. We found that the simple randomized is competitive with the approximation algorithms and that the algorithms rank according to their theoretical guarantees, except for Adjiashvili's algorithm which dealt with high running time. The randomized algorithm is easier to implement and understand, and thus most desireable for practical use. Furthermore, the randomized algorithm runs faster and uses less space than any of the more sophisticated approximation algorithms.

1 Introduction

TAP is a well-studied, fundamental networking problem [20] [17] [26]. Consider the network shown in Figure 1. The network becomes disconnected if any of the connections in the network are removed (Figure 2). A network that can become disconnected by a single point of failure is vulnerable to

*This research was supported in part by the Defense Advanced Research Projects Agency through grant HR001123S0001-FP-004.

incidental failures or attack. It is imperative any network serving a critical role is made single fault tolerant by adding additional connections so that any one failure will not disconnect the network (Figure 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 if 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 problem. 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 that the given graph G 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 4). Under this assumption, the problem is known as the Tree Augmentation Problem [9].

We are able to assume the graph given is a tree without loss of generality. If a graph is given, we can find all the connected components in $O(|V| + |E|)$ time [28]. 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 article is organized as follows: Section 2 formally specifies the problem under consideration. Section 4 contains our motivation and describes the related work in the literature. The empirical setup for our experiments is described in Section 5. The results of our implementation are discussed in Section 6. We conclude in Section 7 by summarizing our results and identifying avenues for future research.

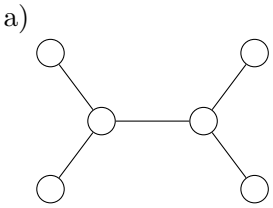


Figure 1: A network where a single edge failure disconnects it

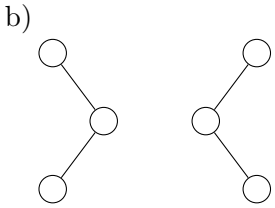


Figure 2: A network being disconnected by the removal of an edge

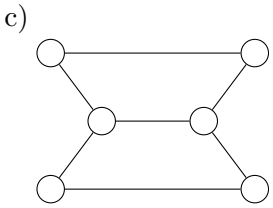


Figure 3: A network that is single fault tolerant

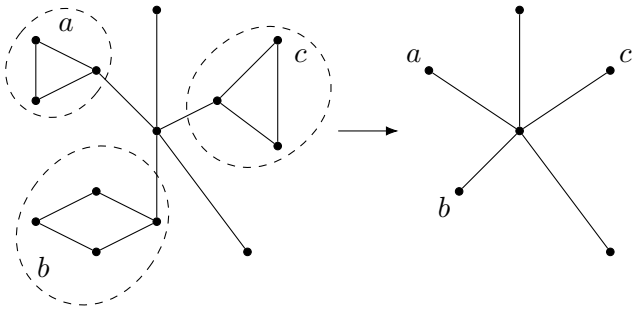


Figure 4: A graph being reduced to a tree by contracting the strongly connected components

2 Statement of Problem

We define the tree augmentation problem as well as the various terms that will be used in the rest of the paper. Recall the definition of the edge-connectivity of a graph.

Definition 1. *A graph $G = (V, E)$ is k-**edge connected** if a subgraph $G' = (V, E \backslash X)$ is connected for all $X \subseteq E$ where $|X| < k$.*

By definition, a tree is 1-edge connected. Adding edges to a graph can only increase its connectivity. The problem, which is the focus of the paper, asks to increase the connectivity of a tree to 2. Notice that after the removal of any edge in a 2-edge connected graph, it remains connected. We formally state the fundamental optimization problem below.

Tree Augmentation Problem
Instance: A tree $T = (V, E)$ and an link set $L \subseteq V \times V$ disjoint from E.
Problem: Find a minimum subset $\tilde{F} \subseteq L$ such that $T = (V, E \cup \tilde{F})$ 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 5 shows an instance of TAP. A simple lower bound on the optimal solution is the ceiling of half the number of leaves in the tree. 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 is also optimal. On any given instance, our link set L may not contain leaf to leaf edges. So, not every instance will have an optimal solution that matches this lower bound.

(a) (b)

Figure 5: An instance of TAP (a) and an optimal solution (b). The dashed lines indicate the link set L, and the red lines indicate the selected edges $\tilde{F}$.

3 Motivation

The key motivation for TAP is to improve the connectivity of a network to make it more fault tolerant [17]. A failure from even a single link may disconnect one or more nodes from the network. As a result, the affected subnetwork becomes inaccessible. Thus, the goal is to design a network such that connectivity is preserved even when a small number of links fail. TAP is a specific instance where we are concerned with maintaining connectivity when a single link fails.

To maintain connectivity, additional links need to be added to the network. However, each additional link has a cost [2]. Therefore, it is desired to minimize the number of additional links or the total cost to augment the network. A general version of TAP is determining the minimum of links to add to a k-edge-connected network such that the resulting network is $(k+1)$-edge-connected [17, 2].

TAP is also equivalent to the problem of finding a minimum sized cover of a laminar set family [5, 21, 24]. Recall that a laminar set family is a collection of sets $\{S_1, S_2, \dots\}$ such that for every i, j, $S_i \cap S_j$ is either $\emptyset$, S_i, or S_j. Laminar families and augmentation problems have applications to the generalized Steiner network problem, where the input consists of a graph and a collection of connectivity requirements, and the goal is to find the subgraph that meets the given requirements with the lowest cost [7]. The Steiner network problem has applications in designing water and electricity supply networks, communication networks, and other large systems where the failure of a single point can affect the whole system [33, 32, 16, 18]. The Steiner network problem is also known as the Survivable Network Design problem, which consists of finding minimum-cost subgraphs such that there are at least n unique paths between every pair of vertices [19]. An equivalent problem is finding minimum-cost subgraphs such that given any cut in a subgraph, there are at least n crossing edges. A survey of the generalized Steiner network problem was conducted in [21].

4 Related Work

The tree augmentation problem was first proposed by Eswaran and Tarjan [9] along with a number of other graph augmentation problems. 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 [14]. It is important to note that TAP is **NP-complete** when the edge set given is not complete.

The first approximation algorithm for WTAP is a 2-factor approximation by Frederickson and Ja'Ja' [14]. 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 assumes the input is a connected tree. Khuller and Thurimella proposed a 2-factor approximation for undirected graphs, even if they are not necessarily connected [20]. The algorithm has a time complexity of $O(|E| + |V| \cdot \log |V|)$ and is simpler than Frederickson and Ja'Ja's algorithm.

Nagamochi proposed an approximation algorithm that achieved an approximation factor better than 2 [25]. 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 [10], though the initial proof was long and complex. Even et al. also devised a much simpler proof that shows a factor 1.8 solution [11] for their algorithm. Even et al. then provided another factor 1.5 approximation proof [12]. It was shown recently that there was in error in [12] but the Even et al algorithm was a 1.5 factor approximation solution in a new proof by Kortsarz and Nutov [22]. 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 [30]. They improve their orignal algorithm using local-search to achieve a $(1.5 + \epsilon)$-factor approximation [31].

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 [5]. 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 $\frac{3}{2}$ [6]. 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 $\frac{28}{15}$ [27]. 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 [23]. 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 [13]. Cheriyan and Gao show another $(1.5 + \epsilon)$-approximation using an SDP relaxation [3, 4]. 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 [17]. 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 links added to the integral solution. Rewiring involves searching for cases where two links in the solution can be reduced to a single link instead.

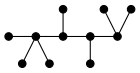
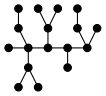
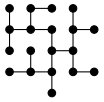
5 Empirical Setup

We describe the setup for our experiment and detail generation of input trees. We subsequently explain how we check that each returned solution is valid. A more in-depth overview of the algorithms being tested are then given, and lastly we describe the hardware that is used for the experiment.

5.1 Experimental Design

The trees used to evaluate the algorithms were of the following six types. Let $T = (V, E)$.

Table 1: Descriptions of Input Classes

Tree Type	**Description**	**Example Graph**
Path Tree	$V = \{v_1, v_2, \ldots, v_n\}$, $E = \{(v_i, v_{i+1}) : i \in [n-1]\}$	
Star Tree	$V = \{v_1, v_2, \ldots, v_n\}$, $E = \{(v_i, v_n) : i \in [n-1]\}$	
Star-like Tree	A union of paths in which each path is adjoined at the same vertex.	
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 uniformly at random from the set of all spanning trees on the same vertex set.	

Our experiment consists of three phases, each corresponding to the density of the link set provided to a problem instance. A link set L can have a density of $d = 0.1, 0.5$, or 0.8. Link sets are generated by iterating over $(V \times V) \backslash E$, selecting an element to add to the link set with probability d, independently of other choices. If $(V, E \cup L)$ is not 2-edge connected by the end of the procedure, then elements of $(V \times V) \backslash (E \cup L)$ are selected uniformly at random until edge-connectivity is augmented.

The performance of each algorithm is tested through an experiment of three repetitions of all pairs of input tree and link set density. The experiment is performed for input trees of size $n = 100$ and $n = 1000$. During each repetition, each algorithm receives the same tree and link set as inputs. Lastly, for each new experiment repetition, new trees and link sets are generated.

5.2 Checking for Validity of Inputs and 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. We use this algorithm to verify 2-edge connectivity of inputs $(V, E \cup L)$, and again in the returned solution $(V, E \cup F)$, where F is the subset of links selected by an algorithm. In each case, the running time for checking if a graph is 2-edge connected is $O(|V| + |E| + |L|)$.

5.3 Algorithms considered

We consider the following five algorithms in our experiment.

Table 2: Algorithms implemented with corresponding approximation factor and time complexity

Algorithm	$\boldsymbol{\rho(n)}$	**Time Complexity**
Randomized (Algorithm 1)	N/A	$O(\|V\|^2)$
Frederickson [14]	2	$O(\|V\|^2)$
Nagamochi [25]	$1.875 + \epsilon$	$O(f(\epsilon) \cdot \|V\|^{\frac{1}{2}} \cdot \|L \cup E\| + \|V\|^2)$
Even, et al [11]	1.5	$O(\|V\|^3)$
Adjiashvili [1]	$1.666 + \epsilon$	$\|V\|^{f(\frac{1}{\epsilon})^{O(1)}}$

5.3.1 The Randomized Tree Augmentation Algorithm

In order to test the practicality of each approximation algorithm, we develop a simple randomized algorithm as a benchmark, given by Algorithm 1. The randomized algorithm first uniformly and at random selects a link for each leaf in the tree and adds it to the solution set F. This is justified, as a graph with leaves has a bridge. The algorithm then checks if the tree is 2-edge connected using the algorithm described in Section 5.2. If not, until $(V, E \cup F)$ is 2-edge connected, a link ℓ is selected uniformly at random from $L \setminus F$ and appended to F. Since the randomized algorithm is very simple and relatively fast, we run the algorithm 100 times and choose the smallest solution.

Algorithm 1 Randomized Tree Augmentation

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

We prove two lemmata about the performance of Algorithm 1, and then concisely detail the algorithm's benefits and drawbacks in Table 3.

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

Proof. Inserting a link to the solution takes constant time. At most $|L|$ edges are added to F. Once a link is added, connectivity is checked at most once with the algorithm in Section 5.2, which takes $O(|V| + |E|)$ time. There are at most $|L| + |E|$ edges in $(V, E \cup F)$ when we search for bridges in every iteration. Therefore, the algorithm runs in $O(|L| \cdot (|V| + |L| + |E|))$ 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 with the algorithm in Section 5.2. 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, requiring $O(|V| + |L|)$ space, as at most $|L|$ edges are added to F. Hence, the total space usage is $O(|V| + |L|)$. □

Table 3: Benefits and Drawbacks of the Randomized Algorithm

Benefits	**Drawbacks**
Simple Implementation	Lacks sophisticated structure
Practically Effective	May not scale well

5.3.2 Frederickson Algorithm

Frederickson and Ja'Ja' gave the first approximation algorithm for the tree augmentation problem in [14]. 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 behind 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 is illustrated in Figure 6, where an instance of TAP is provided, transformed to a directed graph rooted at an leaf selected uniformly and at random, a minimum spanning arborescence is found, and the solution is recombined with the initial tree to produce an approximate solution.

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 $O(|E| \cdot |V|)$ [8]. 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 [29]. There exists a more complex minimum spanning arborescence algorithm that runs in $O(|E| + |V| \cdot \log |V|)$ using Fibonacci heaps [15]. We prove a bound on its space usage, and detail a table of its primary benefits and drawbacks in Table 4.

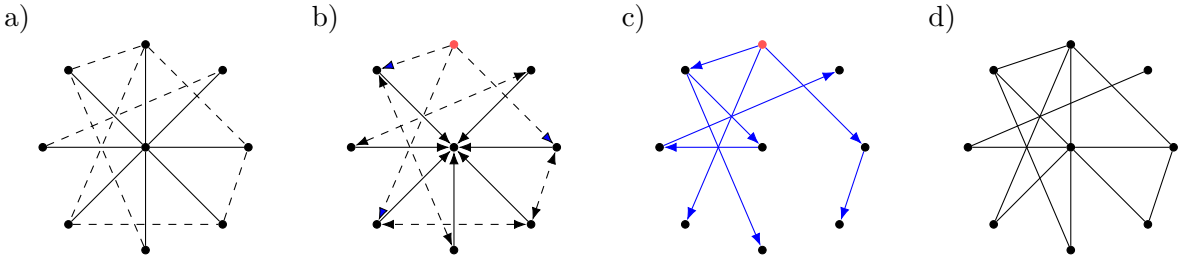


Figure 6: a) An instance of TAP with the tree given by solid lines and the links given by dashed lines. b) The instance transformed into a directed graph with the red leaf indicating the root. c) A minimum spanning arborescence from the directed graph in b. d) The recombined solution to TAP from the edges in the original tree and the edges in the arborescence from c.

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 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

procedure EDMONDS($D = (V, E)$)
  for all $v \in V$ **do**
    $A \leftarrow e = \langle u, v \rangle$ such that cost(e) = min(cost($e_i = \langle u_i, v \rangle \in E$))
    $\pi(v) \leftarrow u$
  if A contains a cycle $C = (V_c, E_c)$ **then**
    $v_c \leftarrow$ a new vertex representing the cycle
    $V' \leftarrow V \backslash V_c \cup v_c$
    if $e = \langle u, v \rangle \in E$ with $u \notin V_c$ and $v \in V_c$ **then**
      $E' \leftarrow e'$ where $e' = \langle u, v_c \rangle$
      cost(e') $\leftarrow$ cost($\langle u, v \rangle$)$-$ cost($\langle \pi(v), v \rangle$)
    else if $e = \langle u, v \rangle \in E$ with $u \in V_c$ and $v \notin V_c$ **then**
      $E' \leftarrow e'$ where $e' = \langle v_c, v \rangle$
      cost(e') $\leftarrow$ cost($\langle u, v \rangle$)
    else if $e = \langle u, v \rangle \in E$ with $u \notin V_c$ and $v \notin V_c$ **then**
      $E' \leftarrow e'$ where $e' = e$
      cost(e') $\leftarrow$ cost(e)
    $A' \leftarrow$ EDMONDS($D' = (V', E')$)
    $e_c \leftarrow e' = \langle u, v_c \rangle \in A'$
    $e_{corr} \leftarrow \langle u, v \rangle$ the corresponding edge of e_c where $\langle u, v \rangle \in E$ and $v \in V_c$
    for all $e \in E_c \backslash e_{corr}$ **do**
      mark e
    for all $e' \in A'$ **do**
      mark corresponding $e \in E$
    $A \leftarrow$ marked edges
  return A

Table 4: Benefits and Drawbacks of Frederickson's Algorithm

Benefits	**Drawbacks**
Simplest approximation algorithm to implement	Edmonds' subroutine recursion causes space usage concern
Fast running time	Worst approximation ratio

5.3.3 Nagamochi Algorithm

The Nagamochi algorithm was the first algorithm to break the 2-factor approximation [25]. 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 $O(f(\epsilon) \cdot |V|^{\frac{1}{2}} \cdot |L \cup E| + |V|^2)$ is given in the paper. We outline the algorithm below in Algorithm 4 and state its primary benefits and drawbacks in Table 5.

Algorithm 4 Nagamochi Tree Cover Algorithm

procedure NAGAMOCHI($T = (V, E), G = (V, L), \epsilon$)
  $F \leftarrow \emptyset$
  while T contains more than one vertex **do**
    while Cases 1, 2, 3, or 4 holds **do**
      Execute P1, P2, P3, or P4 respectively
      $F \leftarrow$ edges retained by the procedure
    Choose a minimally leaf-closed subtree $T[D(v)]$
    if condition A3 holds in $T[D(v)]$ **then**
      Compute an edge set $F^{apx} \subseteq L$ which covers edges in $T[D(v)]$
      $F \leftarrow F \cup F^{apx}$
      $X \leftarrow$ vertices of edges $e \in E$ covered by F^{apx}
      $T \leftarrow T \backslash X$ and $G \leftarrow G \backslash X$
    else
      Choose a lowest solo edge g
      $F_g \leftarrow$ all edges in T_g^*
      Compute an edge set $F^+ \subseteq L$ which covers F_g with $\epsilon > 0$
      $F \leftarrow F \cup F^+$
      $X \leftarrow$ vertices of edges $e \in E$ covered by F^+
      $T \leftarrow T \backslash X$ and $G \leftarrow G \backslash X$
  return F

Table 5: Benefits and Drawbacks of Nagamochi's Algorithm

Benefits	**Drawbacks**
More structured approach, identifying value of some links in specific subgraphs	Most difficult algorithm to implement
Efficient use of space in graph contraction.	Strong dependence on limited reducible cases can have negative effect on choosing best links

5.3.4 Even Algorithm

The Even algorithm has an interesting history. First presented in 2001 [10] as a 1.5-approximation as an extended abstract, the proof was excluded since it was long and complex. Eventually, in 2016 [22] a simple and elegant proof was 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

procedure EVEN($T = (V, E), L$))
  $F \leftarrow \emptyset$
  $M \leftarrow$ maximum matching in $L(Lf, Lf) \backslash W$
  Assign 1 coupon to each unmatched leaf and r
  Assign 3/2 coupons to every link in M
  Exhaust greedy locking tree contractions
  while $T \backslash F$ has more than one node **do**
    Exhaust greedy link contractions and update F and M accordingly
    $T' \leftarrow$ a non-deficient semi-closed tree of $T \backslash F$
    $F' \leftarrow$ an exact cover of T'
    Contract T with F'
  return F

Table 6: Benefits and Drawbacks of Even's Algorithm

Benefits	**Drawbacks**
Most structured, identifying more reducible cases with priorities.	Running time may suffer due to sophisticated case searching and management.
Best approximation ratio.	More difficult implementation.

5.3.5 Adjiashvili Algorithm

The Adjiashvili algorithm is an extension of the *cut 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, a constraint is added to the natural cut LP to obtain the bundle LP below.

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

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 in Algorithm 6 below, and a table of the primary benefits and drawbacks of the algorithm are in Table 7 below.

Algorithm 6 Adjiashvili Algorithm

procedure ADJIASHVILI($T = (V, E), L$)
  $F \leftarrow \emptyset$
  $X \leftarrow$ solution of LP_γ
  while there exists an edge $e \in E$ covered only by up-links in X **do**
    $F \leftarrow$ up-link l covering e that covers the most edges
    Contract T with l
  while there exists a link l connecting two leaves $u, v \in V$ and $l \in X$ **do**
    $F \leftarrow l$
    Contract T with l
  while there exists an uncovered leaf $v \in T \cup F$ **do**
    $F \leftarrow$ an arbitrary $l \in L \cap X$ covering v
  return F

Table 7: Benefits and Drawbacks of Adjiashvili's Algorithm

Benefits	**Drawbacks**
Simpler implementation. The LP does a lot of the work.	Computation of OPT(X) is extremely impractical. See Section 6.1.
Strong approximation ratio	Scalability issues

5.4 Hardware and Implementation

Each algorithm was implemented in Python 3.10 using the NetworkX library, its functions being used for deletion, insertion, and merging in each graph. Each input graph was generated randomly. NetworkX could directly generate paths, stars, and uniform random spanning trees, while we manually implemented the initialization of random instances of caterpillar, lobster, and starlike trees. We use CPLEX solver to obtain a solution to the Adjiashvili Algorithm's bundle LP once it is formulated.

Each algorithm was evaluated on the Regular Memory partition of the Bridges-2 High-Performance Computing (HPC) cluster. A cluster is comprised of nodes of CPUs and memory that are allocated to tasks. The HPC cluster uses AMD EPYC 7742 CPUs with 64 cores capable of running at a base clock speed of 2.25 GHz and a boost clock speed of up to 3.4 GHz. The selected processor is used for the entire execution of the code. Bridges-2 uses a time-sharing scheme in a queue of user-submitted tasks. Once a task is given priority, it executes to completion or until the given maximum allotted time is reached.

6 Results

We now detail the results of our experiments. We split the discussion into two categories determined by the size of the input trees.

6.1 Comments on the Adjiashvili Algorithm

We preface the discussion of results with a deeper analysis of the Adjiashvili Algorithm. Note that the proven approximation ratio is $\frac{5}{3} + \varepsilon$. In formulating the bundle LP, the algorithm requires the consideration of all distinct unions of γ-many paths on T, for $\gamma = \lfloor \frac{168}{\varepsilon^2} \rceil$. For each distinct union B of paths, it is necessary to obtain OPT(B), the minimum number of links in L to cover B. The author provides a subroutine that makes use of a clever observation to reduce the runtime of the subroutine from $O(2^{|L|})$ to $n^{k^{O(1)}}$ where k is the number of leaves in the γ-bundle.

Consider some problem instance where $\varepsilon = 0.1$. The algorithm requires $\gamma = \frac{168}{0.1^2} = 16800$. For any $n < 184$, $\binom{n}{2} < \gamma$, and in formulating the bundle LP, the subroutine has found the optimum solution to the problem instance. If $T = S_{183}$, the running time is incomparable to that of the other

approximation algorithms. Even if $n > 184$, the subroutine must be run multiple times, where the number of leaves in B may continue to grow. If we allow ε to grow to combat the size of k, the strength of the bound is lost. Due to this bottleneck, we were unable to obtain results for the Adjiashvili Algorithm past $n = 25$. In short, the Adjiashvili algorithm is asymptotically good but not practical for real-world applications due to its computational complexity and scalability issues.

6.2 Size 100

We begin a discussion of results on inputs of size 100. Recall that Nagamochi's Algorithm allows a choice of ε. We choose $\varepsilon = 0.1$. In Figure 7, we plot the distribution of each algorithms returned solution sizes, varying the class of the input tree. It is clear that the Even Algorithm can consistently find a very small solution. This can be attributed to its sophisticated techniques for identifying valuable links through maximum matchings and greedy contractions. An interesting trend from the data is that for trees of this size, the randomized algorithm manages to consistently surpass the Frederickson and Nagamochi Algorithms in terms of solution quality. Lastly, we would expect the Nagamochi Algorithm to outperform the Frederickson algorithm, as its upper bound is comparatively lower, however, this seems not to be the case. Nagamochi's relative performance could be attributed to how it prioritizes structural constraints over more optimized link choices.

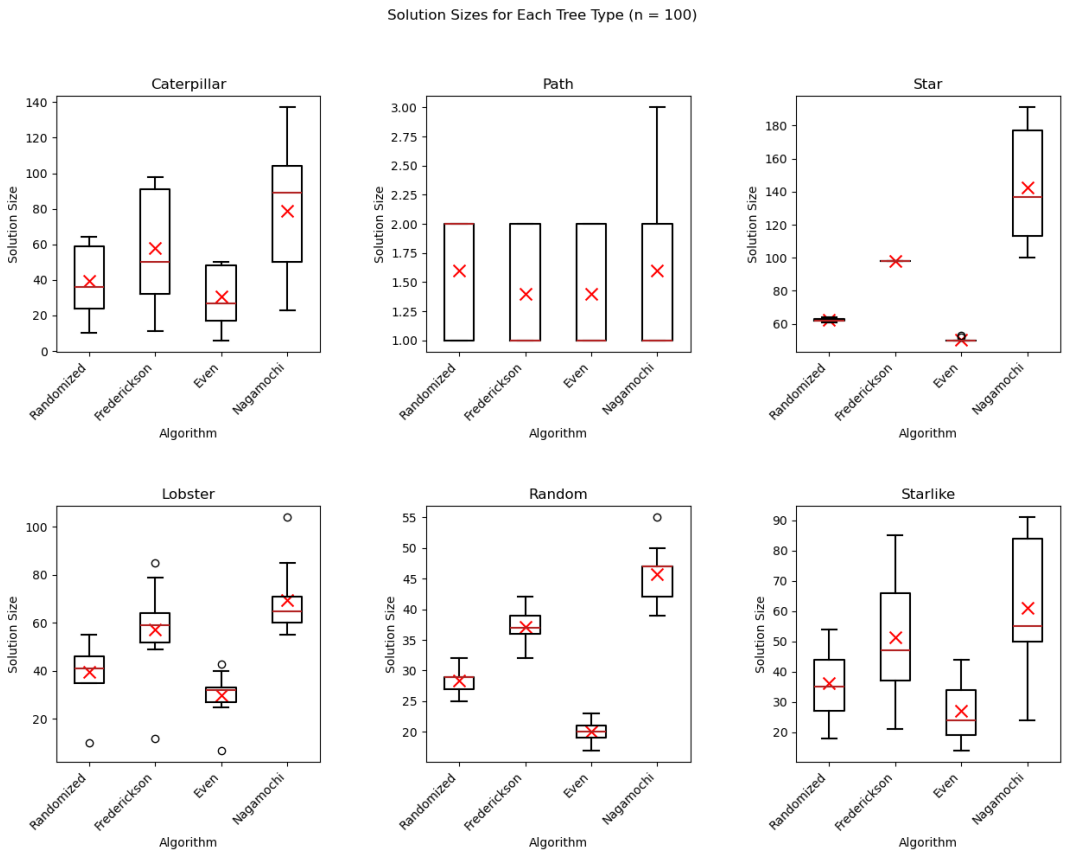


Figure 7: Empirical solution quality, varying input tree classes for $n = 100$.

In Figure 8, we see that the Frederickson Algorithm seems to be the best algorithm in terms of running time, likely due to its straightforwardness, avoiding complex data structures and extensive iteration. The Even Algorithm and Nagamochi seem to have similar empirical running times. It is logical that they perform slightly worse than Frederickson's Algorithm because of how their efficient link selections are due to more complex sub-procedures. Lastly, for this tree size, the randomized algorithm and its 100 trials have a comparatively large running time. This can be explained by the fact that every time the solution F is updated, it is necessary to check whether $T \cup F$ is 2-edge connected. The time that several DFS traversals on a large tree can add up over the 100 trials of the randomized algorithm.

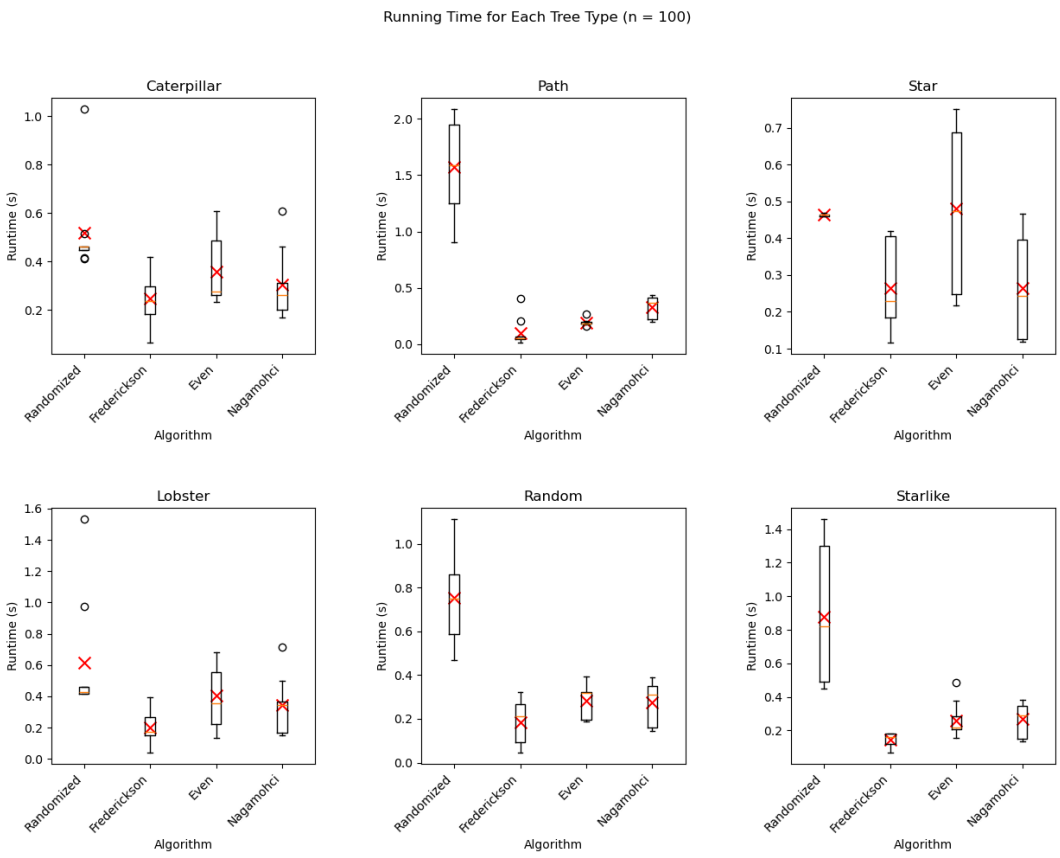


Figure 8: Empirical running times, varying input tree classes for $n = 100$.

From Figure 9, it is obvious and expected that the randomized algorithm champions the memory usage metric due to its minimal data storage and lightweight operations. The memory usage of the Even and Nagamochi Algorithms are comparable, requiring more memory than the randomized algorithm due to the data structures they require. The Frederickson Algorithm depends on Edmonds Minimum Spanning Arborescence Algorithm, which could recurse $(|V| - 1)$-many times to build the arborescence. The call stack can grow significantly for large graphs, which likely explains the Frederickson Algorithms use of memory.

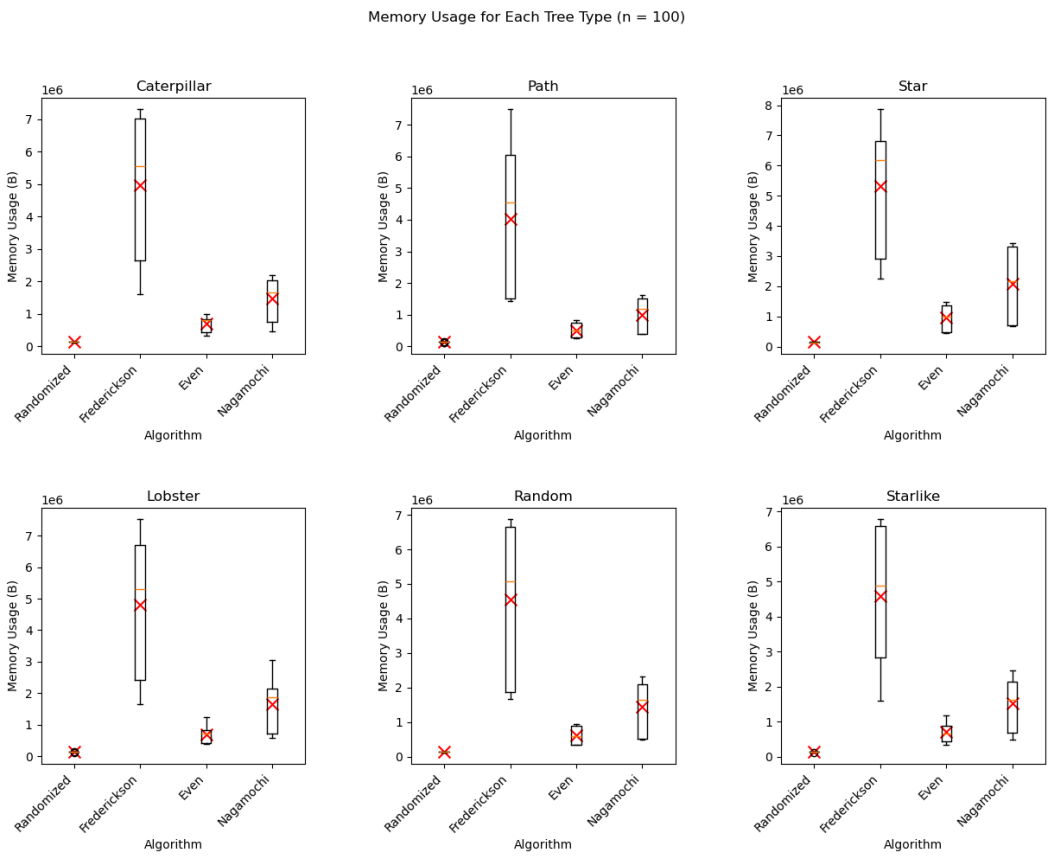


Figure 9: Empirical memory usage, varying input tree classes for $n = 100$.

6.3 Size 1000

In Figure 10, we plot the distribution of each algorithms returned solution sizes, varying the class of the input tree, now on graphs on 1000 vertices. We see that the trend for comparative solution quality remains the same as when $n = 100$ in Figure 7. That is, the Even Algorithm can consistently find a small solution, followed by the randomized algorithm, the Frederickson Algorithm, and the Nagamochi Algorithm.

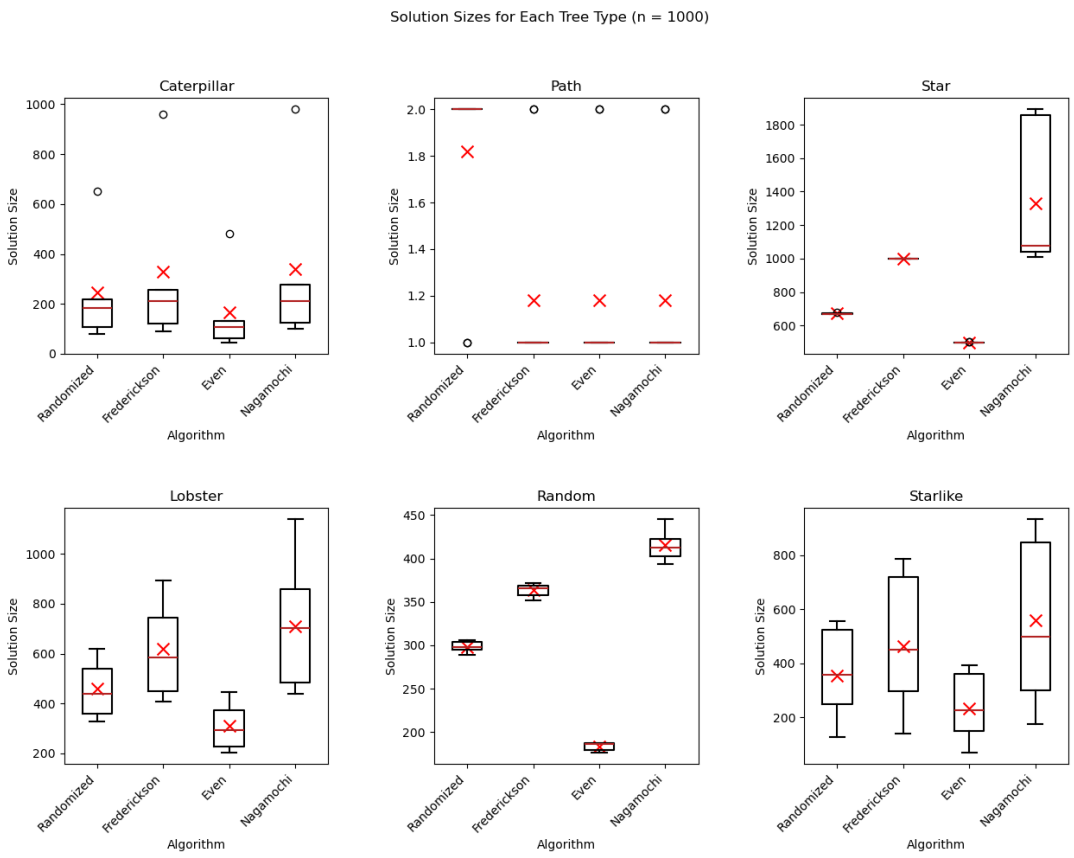


Figure 10: Empirical solution quality, varying input tree classes for $n = 1000$.

Figure 11 shows a different trend than was shown in Figure 8. In most cases, the Nagamochi Algorithm has the largest runtime out of the four. This is due to the fact that the more reducible cases that the algorithm searches for take longer to find than others. Compared to the Even Algorithm, the Nagamochi Algorithm has more reducible cases that contract very small portions of the graph. The Nagamochi Algorithms larger runtime is due to the fact that the algorithm is more inclined to look for smaller reducible cases than larger ones. The randomized algorithm is competitive in terms of runtime for inputs of this size. In particular, the randomized algorithm seems to have a strong runtime whenever all leaves of the tree are close to a central vertex or path, that is, on caterpillar, path, star, and lobster trees. This suggests that randomly selected links can quickly provide a covering of such trees.

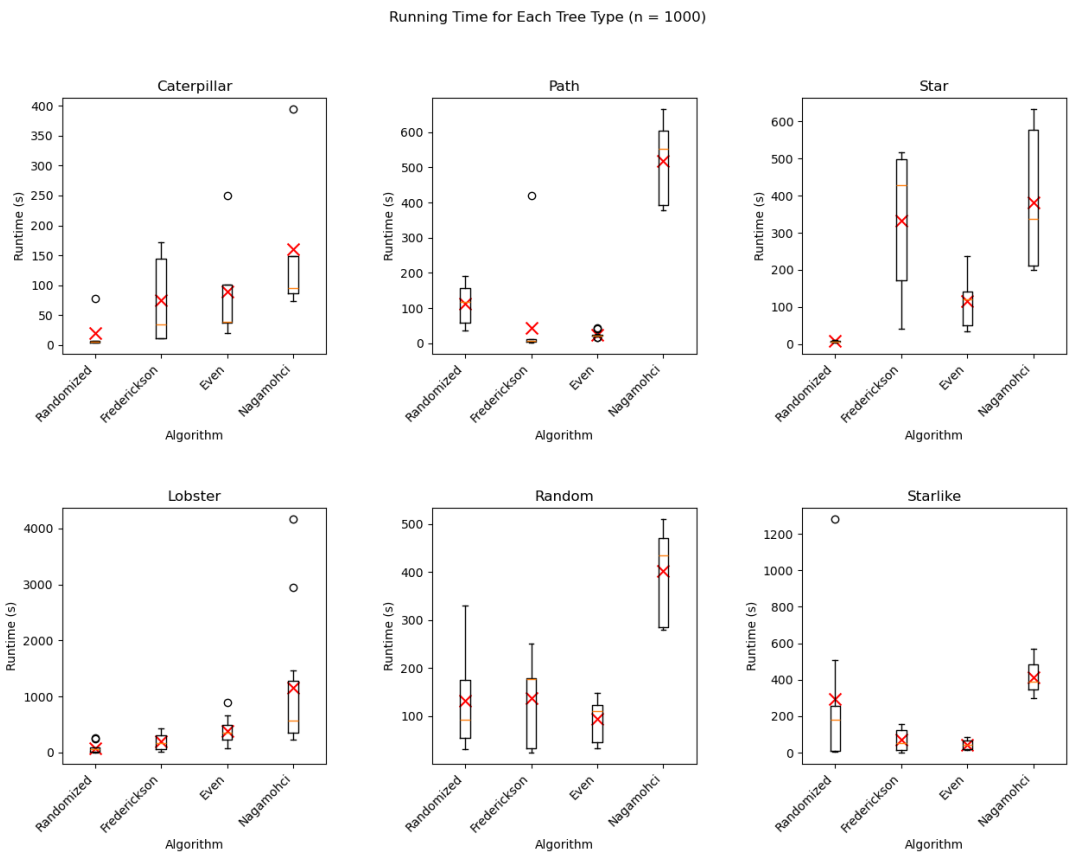


Figure 11: Empirical running times, varying input tree classes for $n = 1000$.

Lastly, the memory comparison metrics in Figure 12 are similar to that of Figure 9. The randomized algorithm exceeds in this regard, while Frederickson uses ample memory due to a large call stack.

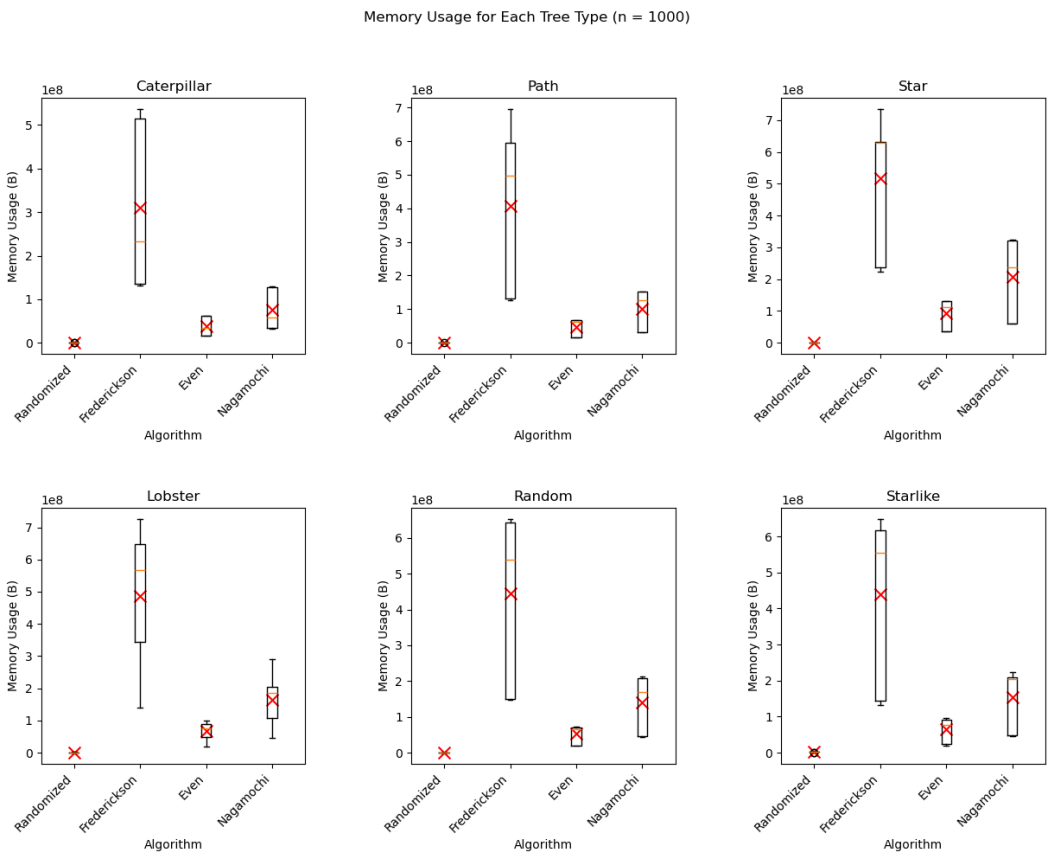


Figure 12: Empirical memory usage, varying input tree classes for $n = 1000$.

7 Conclusion

In this article, we studied approximation algorithms for the tree augmentation problem. We designed a simple but novel randomized algorithm for the problem. We implemented several relevant approximation algorithms found in the literature. The randomized algorithm clearly champions memory usage for all inputs. It consistently returns a smaller average and median solution set size than Frederickson's [14] and Nagamochi's [25] algorithms, and remains competitive in terms of running time. The results conclusively show that algorithms with theoretical guarantees are not necessarily superior in practice when applied to TAP. This especially relates to Adjiashvili's algorithm [1] which guarantees a good approximation ratio, but has a significant running time, rendering it impractical. Otherwise, the results show that the relative performances of the approximation algorithms are predicted by their relative theoretical guarantees. The space usage for TAP is primarily caused by the space used for the input graph. Therefore, space was not a concern for any of the implemented algorithms, with the exception of the LP-based algorithm.

In the future, we aim to expand this study to a larger subset of TAP approximation algorithms to obtain more data and form stronger conclusions about the competitiveness of the randomized

algorithm. The simple randomized algorithm can be augmented to use greedy heuristics to further improve its results. We also aim to repeat the study for WTAP and contrast the findings between the two experiments. Finally, the literature refers to [14] 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 their paper. The reduction used implies **APX-hardness**, but a more formal approach should be done using modern techniques.

References

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

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

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

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

[5] Joseph Cheriyan, Tibor Jordn, 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.

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

[7] Julia Chuzhoy. *Generalized Steiner Network*, pages 349–351. Springer US, Boston, MA, 2008.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] Michel X. Goemans, Andrew V. Goldberg, Serge A. Plotkin, David B. Shmoys, va 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.

[17] 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.

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

[19] Hervé Kerivin and A. Ridha Mahjoub. Design of survivable networks: A survey. *Networks*, 46(1):1–21, 2005.

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

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