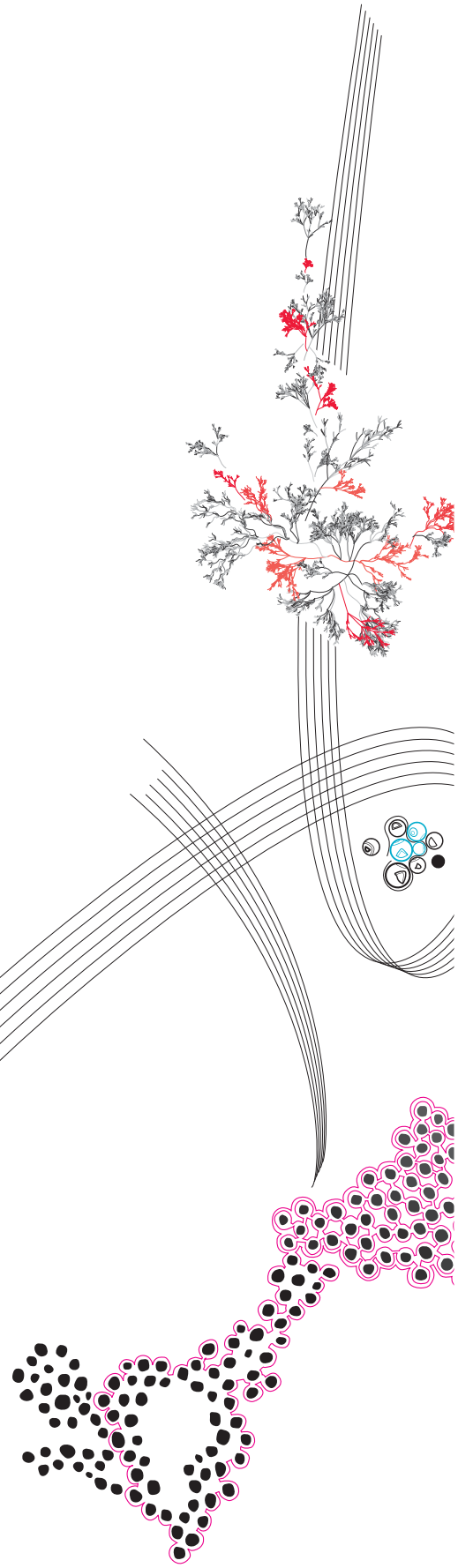


MSc Thesis Education and Communication in Bèta Sciences

A heuristic for finding leafy spanning trees

Jasper Egberink

Supervisors: Pim van 't Hof, Marc Uetz

June 23, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This thesis was written as part of my graduation assignment for the master Education and Communication in Bèta Sciences. This thesis not only marks the end of an interesting research but also the completion of my master's in Enschede. I will always look back with fondness at this time as a student.

I would like to thank my supervisors Pim van 't Hof and Marc Uetz for their help with this research. The many sparring and feedback sessions have helped a lot with the implementation of the ideas and with writing this report. I want to thank both of you for your enthousiasm and for taking the time to supervise this project.

I would also like to thank my friends, family and girlfriend for their support during my study. I have learned a lot the past few years and will always cherish the memories and friendships made during my study.

Jasper Egberink,

June 23, 2023

A heuristic for finding leafy spanning trees

Jasper Egberink

June 23, 2023

Abstract

This research focuses on the maximum leaf spanning tree problem. The objective of this problem is to find a spanning tree containing the maximum amount of vertices with degree 1 (leaves). The goal of this study was to design a heuristic for the MLST problem which is easy to implement and uses Kruskal's algorithm in combination with a weight assignment rule. The resulting LeafyST heuristic assigns edge weights by taking the sum of the degrees of both endpoints of an edge. A spanning tree is then found by running the maximization version of Kruskal's algorithm on the weighted graph.

The performance of the LeafyST heuristic was tested on $G_{n,p}$ random graphs and random geometric graphs containing either 10 nodes or 75 nodes. The amount of leaves in the spanning trees found by the LeafyST heuristic were compared to the optimal solution and the expected value for the number of leaves in any spanning tree. For smaller graphs, all possible spanning trees were calculated and the results of the LeafyST heuristic were compared to the distribution of the leaves in all spanning trees.

The practical performance of the LeafyST heuristic on $G_{n,p}$ random graphs and random geometric graphs is reasonably good. The solutions found by the LeafyST heuristic contain a relatively high amount of leaves when comparing them to the optimum and estimated average. However, there are some theoretical cases where the heuristic performs poorly. This resulted in the conclusion that the LeafyST heuristic is useful in practical cases because it performs reasonably well and is easy to implement. However, the heuristic is not likely to contribute to improving the current best approximation ratio of 2 for the MLST problem.

Keywords: graphs, leaves, spanning trees, Kruskal's algorithm

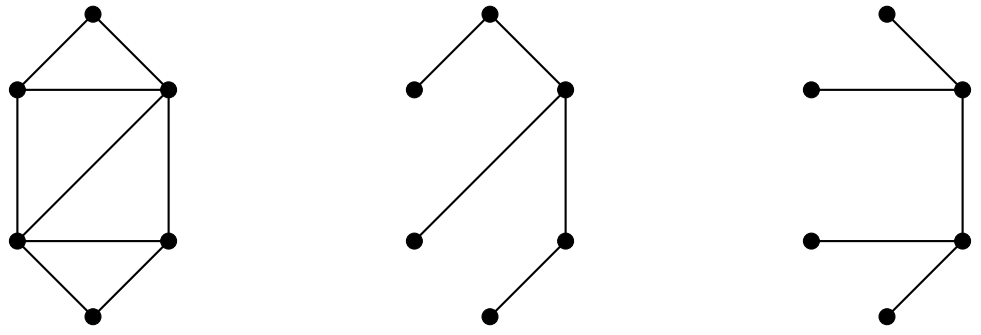
1 Introduction

Many networks found in day-to-day life can be represented in a graph. Graphs are a mathematical representation of these networks using nodes (often called vertices) and edges. In this representation, the edges are subsets of two nodes and represent a connection between these nodes. Consider for example a rail network. Such a network could be represented in a graph by plotting the stations as nodes and the connecting railways as edges.

In the field of graph theory, a lot of research is done to solve certain problems within graphs. One such problem is the problem of finding a minimum spanning tree (MST). A minimum spanning tree connects all the vertices using the minimum number of edges. Another problem in the field of graph theory is the maximum leaf spanning tree problem (MLST). This is also the problem we will focus on in this research. With the MLST problem, the goal is to find a spanning tree for the graph which maximizes the number of leaves. A leaf is defined as a node with degree 1 in the spanning tree, so it is connected to only one other node.

Applications of the MLST problem can be found, for example, in (wireless) communication networks [1]. These networks can have many different forms, they can vary from communicating chips within a computer to communication networks within a country. In these networks, the nodes can represent different things, such as basic sensors to supercomputers. The edges represent the method of communication, this can be through any type of connecting wire or a wireless connection.

In Figure 1 below, an example of a communication network is given. As stated in [1], in communication networks it is important to connect all nodes to each other with either a direct link or via intermediate links, this is equivalent to finding a spanning tree for the network. However, it could be the case that, for example, the intermediate nodes of the spanning tree are more expensive or require more energy than end nodes. In these cases, it is beneficial to maximize the number of end nodes (leaves) in the spanning tree and a solution to the MLST problem can be used to maximize the number of leaves. For the communication network given in Figure 1a, a spanning tree is given in Figure 1b and a solution to the MLST problem is given in Figure 1c. Note that the spanning tree in Figure 1b contains 3 leaves while the MLST in Figure 1c contains 4 leaves.



(A) (B) (C)

Figure 1: Example of a spanning tree and a maximum leaf spanning tree for a communication network

1.1 Research goal

The goal of this research is to investigate if we can use minimum spanning tree algorithms to develop a heuristic to find leafy spanning trees. The heuristic has to be relatively easy to implement and should be able to find good solutions to the MLST problem in two types of random graphs.

1.2 Outline

Chapter 2 of this report will present the related work that is relevant for this study. Next, the LeafyST heuristic will be described and the logic behind it will be explained in chapter 3. Chapter 4 will present the methods used to analyze the performance of the LeafyST heuristic. In chapter 5, the results of the performance analysis will be presented. The discussion of the research and recommendations for future research are given in chapter 6 and the conclusion of this study is presented in chapter 7.

1.3 Preliminaries

In this section, the definitions and notations used will be given. We will start by defining a graph. A *graph* is a pair $G = (V, E)$ where V is the set of *vertices* and E is the set of *edges*. An edge between vertices $u, v \in V$ will be written as (u, v) and if an edge $e = (u, v)$ exists, vertices u and v are the *endpoints* of e and are called *adjacent*. An edge-weigted graph is a graph where each edge has an associated numberical value called the *weight*, edge-weighted graphs are also called weighted graphs in this report. If edge (u, v) exists, u is a *neighbor* of v and vice versa. The set of neighbors of a vertex v in G is denoted as $N_G(v)$. Vertices u and v are *incident* with e and vice versa. In this research all graphs are simple and undirected unless stated otherwise, meaning that edges of type (v, v) do not exist and any edge $e = (u, v) = (v, u)$. The *degree* of $v \in V$ is the number of neighbors of v and is written as $d(v)$. A vertex $v \in V$ with $d(v) = 1$ is called a *leaf*.

If $G = (V, E)$ and $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ then H is called a *subgraph* of G. If $V' = V$ then H is called a *spanning* subgraph of G. Let $S \subseteq V$, then $G[S]$ is the subgraph *induced* by S, whose vertex set is S and whose edge set consists of all edges for which both endpoints are in S. For any vertex set $S \subseteq V$ we denote $\overline{S} = V \setminus S$.

A *walk* W on G is a sequence of vertices $\{v_1, v_2, ...\}$ and edges $\{e_1, e_2, ...\}$ of G where an edge e_i is incident with vertices v_i and v_{i+1}. A walk is *closed* if the first and last vertices of the sequence are equal, otherwise, the walk is *open*. A walk W_k with vertex sequence $\{v_1, v_2, ...v_k\}$ has *end vertices* v_1 and v_k. Its *length* is the number of edges and is equal to k. A walk with end vertices u, v is called a (u, v)-walk. A *trail* is a walk with distinct edges and a *path* is a trail with distinct vertices. A *cycle* is a trail where the end vertices are the same vertex. A graph G is *connected* if for every pair of vertices u, v, there exists a (u, v)-path. A *component* of G is a maximal connected subgraph of G. A *forest* is a graph without cycles and a *tree* is a connected graph without cycles. A tree $T = (V', E')$ is a *spanning tree* of $G = (V, E)$ if $V = V'$. $D \subseteq V$ is a *dominating set* of G if all vertices of G are either in D or adjacent to a vertex in D.

For some mathematical problems, it is very hard to find and/or verify an optimal solution. While an algorithm might be able to find an optimal solution to a problem, it is also important to take into account the running time needed to find this solution. For some problems, finding the optimum is complex and requires a lot of running time. The complexity of a problem is distinguished using complexity classes. A brief description of the terminology for complex problems used in this study will be given below.

A decision problem is a set of instances I that can be partitioned into "Yes" and "No" instances ,denoted by I_y and I_n respectively, such that $I = I_y \cup I_n$ and $I_y \cap I_n = \emptyset$. A problem is solved in polynomial time if the running time can be bounded by a polynomial function of the problem size. The set NP is the set of decision problems for which all "Yes" instances have a proof of being a "Yes" instance that can be verified in polynomial time. The set co-NP is the set of problems for which all "No" instances have a proof of being a "No" instance that can be verified in polynomial time. The set P is the set of problems for which all "Yes" and "No" instances can be solved in polynomial time. The set of NP-hard problems is the set of problems that is at least as hard as all other problems in the set NP. The set of NP-complete problems are problems in the set NP that can be classified as NP-hard. An approximation algorithm is an algorithm that finds a solution to a problem in polynomial time with a provable guarantee on the ratio between the solution and the optimal solution. The guarantee for this ratio is called the approximation factor and is denoted as α.

In this study, we will focus on the MLST problem. The definition of this problem is as follows:

Given a connected undirected graph $G = (V, E)$, find a spanning tree T of G with the maximum number of leaves.

The corresponding decision problem is the following:

Given a connected undirected graph $G = (V, E)$ and an integer k. Does G have a spanning tree T with at least k leaves?

2 Related work

From an optimization viewpoint, the MLST problem is equivalent to the minimum connected dominating set problem (MCDS) for a connected graph G. The MCDS problem is defined as follows: *Given a graph $G = (V, E)$ we want to find the smallest possible subset D of V, such that D induces a connected subgraph and is a dominating set of G.* The vertices in $\overline{D}$ are by definition leaves, because they are adjacent to at least one vertex in D and can thus be included in a spanning tree while having a degree equal to 1.

Observation 1 *A graph $G = (V, E)$, with $|V| = n$, has a spanning tree with k leaves if and only if it has a connected dominating set of size $n - k$.*

From the observation above, we can deduce that if the MCDS problem is solved, the amount of leaves is by definition maximized and the MLST problem is solved too, so the MCDS and MLST problems are equivalent from an optimization viewpoint. For this reason, algorithms designed for any of these problems are taken into account in this study.

The MLST problem is proven to be NP-hard in [5], meaning that if this problem can be solved in polynomial time, any problem in the set NP can also be solved in polynomial time. Since no polynomial time exact algorithm has been found to any NP-hard problem, it is highly unlikely that the MLST problem can be solved in polynomial time. However, other strategies can be used to find reasonably good solutions to NP-hard problems. These other methods include approximation algorithms or heuristics. With an approximation algorithm, a solution is found in polynomial time and this solution can be proven to have a certain quality. For example, a 2-approximation algorithm for the MLST problem finds a solution where the number of leaves is never below half the amount of an optimal solution. Another strategy to find solutions to an NP-hard problem is using a heuristic. Heuristics often allow easier implementation than regular and approximation algorithms and they often find a solution more quickly. However, with a heuristic, there is no guarantee to the quality of the solution.

The MLST problem is not only proven to be NP-hard, but also MAX SNP-hard [4]. This means that there exists no polynomial time approximation scheme (PTAS) for this problem unless $P = NP$. A PTAS finds a solution in polynomial time with an approximation factor of $1 + \epsilon$ for any fixed $\epsilon > 0$.

In [10], an approximation algorithm for the MLST problem was given. This algorithm has a worst case factor of 3, meaning that a solution can be found in polynomial time and in the worst case, the optimum contains three times as many leaves as the solution found by the approximation algorithm. The approximation algorithm of [10] finds leafy spanning trees by creating leafy subtrees of the graph and connecting them. These leafy subtrees are found by adding edges with the highest number of adjacent edges without creating cycles. The approximation ratio found in [10] was improved in [14] to a worst case factor of 2. The approximation algorithm presented in [14] finds a leafy spanning tree by building up a forest F. It uses 4 rules to determine whether a vertex is expanded. Expanding a vertex v adds v to F and adds edges (v, w) for all $w \in N_G(v) \setminus V(F)$. In short, these 4 rules take into account the amount of neighbors a vertex v has. If v has more than 1 neighbor in $\overline{V(F)}$, it is expanded. If v has only one neighbor w, which has 2 or more neighbors in $\overline{V(F)}$, v and w are expanded. Lastly, if $\overline{V(F)}$ contains a vertex v with at least 3 neighbors in $\overline{V(F)}$, v is expanded. If, after this phase, F is not yet spanning, vertices v with $N_G(v) \setminus V(F) \neq \emptyset$ are expanded until F is spanning. After this, the components of F are connected using arbitrary edges between the components.

In [9], Guha and Kuller present an approximation algorithm for the MCDS problem, which also makes it an approximation algorithm for the MLST problem. The algorithm

initializes by coloring all vertices white. It grows a tree T, starting from the vertex with the maximum degree and then 'scans' vertices. Scanning a vertex v colors it black and adds it to T, it also adds edges (v, w) to T for all $w \in N_G(v) \setminus V(T)$. If vertex v is scanned, the white vertices in $N_G(v)$ are colored gray. In order to determine which vertices will be scanned, the yield of vertices is calculated. The yield is the number of neighboring white vertices. The algorithm calculates the yield for all pairs of vertices u, v where u is gray and v is white and the option with the highest yield is chosen. For a pair u, v, the yield of scanning u is calculated and the yield of scanning both u and v is calculated and divided by 2. The option with the highest yield is the yield for the pair u, v. The algorithm runs until all vertices are either black or gray. This algorithm finds a connected dominating set with size at most $2(1 + H(\delta)) \cdot |OPT_{DS}|$. Where OPT_{DS} is the optimal dominating set of the input graph, H is the harmonic function and δ is the maximum degree.

Guha and Kuller also introduce a second algorithm in their paper, which runs in two phases. For this algorithm, the same coloring rule is used as the algorithm described above. The algorithm initializes by coloring all vertices white. It then colors the vertex which results in the maximum reduction of 'pieces' black. Pieces are defined as white vertices or black connected components. The first phase ends when there are no more white vertices. In the second phase, the black components are connected using a Steiner tree. The final solution found by the algorithm is the set of black vertices that form the connected dominating set. This algorithm achieves an approximation ratio of $\ln \delta + 3$.

The best known approximation algorithm for MCDS is given in [13] and has an approximation factor of $2 + \ln \delta$ where δ is the maximum degree of the input graph. This algorithm uses a potential function to determine the maximum number of vertices which can be dominated. The vertex yielding the highest number of leaves is then added and the function is updated each time a vertex is added to the dominating set. The algorithm stops when a connected dominating set is formed.

It is important to note that from an approximation viewpoint, the MLST problem and MCDS problem are not equivalent. This means that an approximation factor for the MLST problem will not necessarily equal that of the MCDS problem [14]. Take for example Figure 1 in the introduction. The optimum for the MLST problem is 4 and for the MCDS problem the optimum is 2. If an approximation algorithm for these problems finds 3 leaves, that means the dominating set consists of 3 nodes. The approximation factor for the MLST problem would be $\frac{4}{3}$ in this case, while the approximation factor for the MCDS problem is $\frac{3}{2}$. This means that from an approximation viewpoint, the MLST problem and MCDS problem are not equivalent.

In the section above, the most important existing approximation algorithms for the MLST and MCDS problems were presented. While these algorithms come with a performance guarantee, the goal of this study is to design a heuristic based on well-known algorithms for finding minimum spanning trees. The heuristic must also be easy to implement and practically fast. The input graphs used in this study are without edge-weights, which allows us to steer a minimum weight spanning tree algorithm in the right direction using edge-weights. The method of assigning these edge-weights is inspired by the methods used in the approximation algorithms that were described in the section above.

3 The LeafyST heuristic

The LeafyST heuristic was designed to find a solution of reasonable quality to the MLST problem using Kruskal's algorithm. Kruskal's algorithm is an algorithm for finding a minimum weight spanning tree. For a graph $G = (V, E, w)$, with $w : V \to \mathbb{N}$, a minimum weight spanning tree T is found by Kruskal's algorithm using the following method:

The algorithm initializes with $T = (V, \emptyset)$. It then sorts the edges by increasing weight and processes all of the edges in this order. If the processed edge does not create a cycle in T, it is added to T. Otherwise, it is discarded. After the last edge is processed, T is guaranteed to be a minimum weight spanning tree of G[7]. Kruskal's algorithm has a running time of $O(m \log n)$ where $m = |E|$ and $n = |V|$ [2].

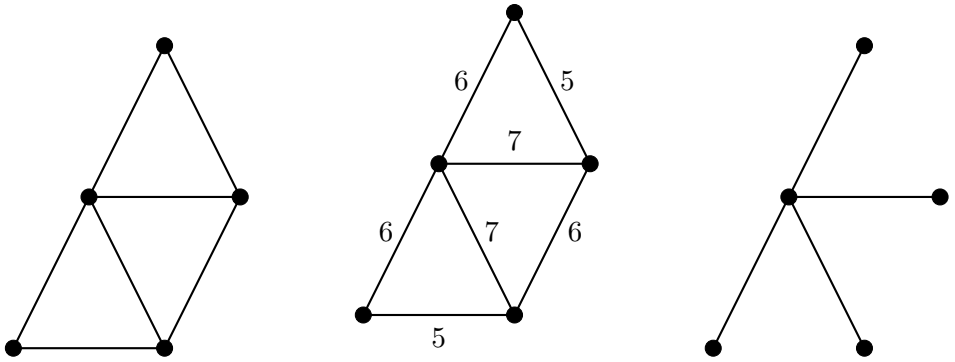
In the LeafyST heuristic, Kruskal's algorithm is slightly adopted such that it finds a maximum weight spanning tree. This algorithm uses the same method described above, but the edges are ordered and processed by decreasing instead of increasing weight.

3.1 Description

The LeafyST heuristic uses a combination of Kruskal's algorithm for maximum weight spanning trees with a weight assigning rule. The heuristic can be described as follows:

Input: Graph $G = (V, E)$
Output: A spanning tree T for G
initialize: all weights $w(e) = 0$
for *every edge* $e = (u, v)$ **do**
| assign weight $w(e)$ equal to the sum of the degrees of the endpoints $d(u) + d(v)$
end
calculate T by running Kruskal's algorithm on G
return T

Figure 2 below shows an example of the functioning of the LeafyST heuristic, where 2a shows the input graph, 2b shows the weights assigned by the LeafyST heuristic and 2c shows the spanning tree found by the LeafyST heuristic.



(A) (B) (C)

FIGURE 2: EXAMPLE OF AN UNWEIGHTED INPUT GRAPH, THE WEIGHTS ASSIGNED AND THE SOLUTION FOUND BY THE LEAFYST HEURISTIC

3.2 Background

In order to find leafy spanning trees, the LeafyST heursitc uses Kruskal's algorithm in combination with a weight assignment method. Since Kruskal's algorithm can solve the maximum weight spanning tree problem in polynomial time, this algorithm could also be useful to quickly find spanning trees with many leaves.

In order to use Kruskal's algorithm, which works on edge weighted graphs, a method to assign weights to the edges had to be designed. The weight assignment method that is used in the LeafyST heuristic is based on ideas used in the approximation algorithms presented in the previous chapter. The approximation algorithms in [9], [10] and [14] all prioritize vertices with many neighbors when building up the tree. The first approximation algorithm for the MLST problem uses degrees of vertices to determine if a vertex is included in the subtree [10]. The approximation algorithm presented in [9] uses a 'scanning' method on pairs of vertices to determine whether the vertices would be included in the dominating set or not. In [14], the approximation algorithm determines the amount of neighbors for (pairs of) vertices in all 4 rules to find the yield. The weight assignment rule for the LeafyST heuristic is based on these methods, because they have been proven to work well for designing approximation algorithms for the MLST (and MCDS) problem.

For the LeafyST heuristic, the scanning rule was simplified to reduce computation time and to make the implementation easier. Vertices with many neighbors that are potential leaves likely have a higher degree than vertices with few neighbors that are potential leaves. This means that, in most cases, vertices with a higher degree should be prioritized when building up the tree. In order to do this, the decision was made to use the sum of the degrees of the endpoints to determine the weight of an edge. Note that this method does not distinguish vertices which are potential leaves from the other vertices, while the methods used in [9] and [14] do.

The time complexity of the heuristic is equal to the time complexity of Kruskal's algorithm. Determining the degrees of the vertices has a time complexity of $O(m)$ and assigning weights to the edges also has a time complexity of $O(m)$ so the complexity of $O(m \log n)$ of Kruskal's algorithm is not affected by the weight assignment rule.

4 Performance analysis

The practical performance of this heuristic has been tested using two types of random graphs: Erdős–Rényi graphs and random geometric graphs. In this section, the two types of graphs and their properties will be given first. After this, the setup for the computational analysis of the LeafyST heuristic will be given and explained.

4.1 Types of graphs

4.1.1 Erdős–Rényi graphs

Erdős–Rényi graphs were introduced by P. Erdős and A. Rényi in 1959 [3]. In this paper, Erdős and Rényi introduce the idea of a random graph with n nodes and N edges. This results in $\binom{\binom{n}{2}}{N}$ possible graphs from which one is randomly chosen. The model that will be used to generate these types of graphs is the model introduced by E.N. Gilbert in [6]. In this model, a $G_{n,p}$ random graph is generated where n is defined as the number of nodes and for each possible edge, p is the probability that the edge will be included in the graph. Both models provide a method to generate a random graph. Since the model of Gilbert was used in the implementation, these graphs will be referred to as $G_{n,p}$ random graphs in this report.

4.1.2 Random geometric graphs

Random geometric graphs are generated by randomly distributing a defined number of nodes onto the unit cube with d dimensions. If the Euclidean distance between any two nodes u, v is smaller than a predefined radius r, these nodes are connected by an edge. In this research, all random geometric graphs that are generated have $d = 2$, so all nodes are randomly distributed on the $[0, 1)^2$ space. The radius of the random geometric graphs is altered to change the expected density of the graphs.

A random geometric graph can be used to model a wireless ad-hoc network, which is a representation for a network of wireless devices [11]. As stated in the introduction of this report, the study of finding leafy spanning trees can be applied in different communication networks. Thus, finding leafy spanning trees in random geometric graphs can be an interesting application of the LeafyST heuristic.

4.1.3 Implementation in Python

Both types of graphs are used to run experiments to test the performance of the LeafyST heuristic. In order to do this, the graphs were generated in Python using NetworkX version 2.8.8. The $G_{n,p}$ random graphs are generated in NetworkX by defining the number of nodes of the graph and the probability that an edge will be included. NetworkX runs the code defined in [12], where it creates an empty graph with the defined amount of nodes and each possible edge is included with probability p.

The random geometric graphs are generated using the same version of NetworkX in Python. The generator first assigns the nodes to a position by taking a random number in $[0, 1)$ for each defined dimension (in the case of this study 2). After the nodes are assigned to a position, the *geometric_edges* function of NetworkX is used to assign edges in the graph if the Euclidean distance between nodes is smaller than the predefined radius.

It is important to note that the graphs generated for this study are all connected graphs. Each graph that is generated for the experiments is tested on connectivity using

the *is_connected* function of NetworkX. If the graph is not connected, a new graph will be generated, until a connected graph is found.

The computational analysis of the heuristic is done on both types of graphs. For the analysis, 50 graphs of both types were generated with an increasing value for the expected density. This is done by increasing the values for p and r after every 10 graphs. The initial value for p and r is 0.2 and this is increased after 10 graphs with a step of 0.1. So the values of p and r are increasing from 0.2 to 0.6.

4.2 Setup for computational analysis of the LeafyST heuristic

To analyse the performance of the LeafyST heuristic, different experiments were conducted. Most of these experiments were done with graphs consisting of 75 nodes. Graphs with 75 nodes were generated to have the highest number of nodes while not requiring too much computational power for the analysis. The different tests that were done in the analysis will be explained below.

4.2.1 Calculating the optimum

The performance of the LeafyST heuristic is analysed by calculating the maximum amount of leaves for a spanning tree of a given graph using a mixed integer program (MIP). The number of leaves found by the LeafyST heuristic is then compared to this optimum to analyse its performance.

For the MIP, graph $G = (V, E)$ is converted to a biconnected digraph $D = (V, A)$, meaning that each edge (u, v) in G is converted to two oppositely directed arcs (u, v) and (v, u) in D. In D the variable $x_{u,v} \in \{0, 1\}$ is used to denote whether the arc (u, v) is included in the solution or not. Variable $y_v \in \{0, \ldots, |V - 1|\}$ is used to define an ordering of the nodes $v \in V$. Furthermore, variable $z_v \in \{0, 1\}$ is used to indicate whether node v is a leaf or not. The MIP finds an arborescence for D which maximizes the amount of leaves, where an *arborescence* is defined as a directed subgraph of D in which a root vertex r is connected to any other vertex v with exactly one directed path. This root vertex can be any vertex of the graph, so for an easier implementation, vertex 0 is always chosen as the root vertex. The MIP used to find the optimum for the amount of leaves is defined as follows:

$$\text{maximize} \sum_{v \in V} z_v$$

subject to

$$\sum_{v \in V} x_{r,v} = 0 \tag{1}$$

$$\sum_{v \in V} x_{u,v} = 1 \qquad u \in V \setminus \{r\} \tag{2}$$

$$y_v \leq y_u - 1 + |V|(1 - x_{u,v}) \qquad u, v \in V \tag{3}$$

$$y_r = 0 \tag{4}$$

$$z_r \leq 2 - x_{u,r} - x_{v,r} \qquad u, v \in V \setminus \{r\}, u \neq v \tag{5}$$

$$z_v \leq 1 - x_{u,v} \qquad u \in V, v \in V \setminus \{r\} \tag{6}$$

Constraint (1) ensures that no outgoing arc for the root node is used, this way, the arborescence is directed towards r.

Constraint (2) ensures that for all nodes except r, exactly 1 outgoing arc is used, meaning each node is included in the arborescence.

Constraints (3) and (4) ensure that the arborescence is acyclic by assigning smaller numbers in the ordering to the nodes closer to the root node. If there would be a cycle in the arborescence, for each arc u, v in the cycle, $x_{u,v} = 1$ and the constraint ensures that $y_v \leq y_u - 1$, so $y_v < y_u$. However, this cannot be true for all values of y in the cycle. Thus, constraint 3 ensures that no cycles can be included in the final solution.

By a combination of constraints (2), (3) and (4), r is ensured to be included in the arborescence. This is because every vertex except r must have exactly 1 active outgoing arc and there are no cycles.

Constraint (5) ensures that $z_r = 0$ if two different nodes u and v are connected to r with an incoming arc for r. Since the arborescence is an in-going arborescence for r, r is a leaf if and only if it has 1 incoming arc.

Finally, constraint (6) ensures that $z_v = 1$ if and only if it has no incoming arc. Since (2) ensures that each node other than r has one outgoing arc, v is a leaf if there is no incoming arc.

By maximizing $\sum_{v \in V} z_v$ the number of leaves in a maximum leaf spanning tree is given as the output of the MIP.

4.2.2 Estimating the expected value of the number of leaves

The number of leaves found by the LeafyST heuristic is compared to the maximum possible number of leaves and the expected number of leaves in any random spanning tree. This expected number of leaves is estimated using the following method.

First, a random integer weight value is assigned to the edges, the weight values are randomly chosen from the interval $[1 : 10^{15}]$. A large interval is used to have practically unique weights. The value 10^{15} is chosen, because this was the largest possible value for which there were no implementation issues. Kruskal's algorithm is used to calculate a minimum spanning tree for the weighted graph. Since the weights are picked from such a large interval, they are almost always expected to be unique implying that the minimum spanning tree is also unique [8]. Thus, by assigning the weights randomly, the resulting spanning tree can be seen as a random spanning tree for the input graph. The expected value for the number of leaves is estimated by taking the average number of leaves from 100 random spanning trees. A T-test was done with a significance level of 5%. The conclusion of the test was that this method of estimating the average number of leaves was correct for around 90% of the graphs tested.

The T-test was done to compare this estimation strategy with the actual average number of leaves of all possible spanning trees. The T-test was done on $G_{n,p}$ random graphs with 10 nodes. By validating the method on graphs with 10 nodes, the estimation strategy can be used to estimate the average number of leaves for graphs with 75 nodes. Since the distribution for the number of leaves is not necessarily a normal distribution, the sample size of the test should be sufficiently large in order for the test to be reliable. It is stated in [15] that "The T-test and least-squares linear regression do not require any assumption of Normal distribution in sufficiently large samples. Previous simulations studies show that "sufficiently large" is often under 100". This means that the T-test can be used to validate the estimation, since 100 spanning trees are generated with the method described above.

The table with the results of the test can be found in the appendix.

4.2.3 Calculating all possible spanning trees

The performance of the LeafyST heuristic is analysed by comparing the amount of leaves found by the heuristic to the optimal solution and estimated average number of leaves for any spanning tree. However, it might be the case that many possible spanning trees contain the maximum number of leaves. Let's say that 30% of all possible spanning trees contain the maximum number of leaves and the LeafyST also finds the optimum. It would be more impressive if the LeafyST would find the optimum if only 1% of all possible spanning trees would contain the maximum number of leaves. To give more insight into the distribution of the number of leaves of all possible spanning trees, these spanning trees are calculated.

All possible spanning trees for the graph are calculated using the SpanningTreeIterator function of NetworkX [12]. The amount of leaves for each spanning tree is then listed. This list is then processed into a frequency table or histogram to represent the amount of leaves in all possible spanning trees. Ideally, all possible spanning trees for graphs with 75 nodes are calculated, since the other steps of the analysis are also done on these graphs. However, this step requires a lot of computation power, since especially for more dense graphs, the amount of possible spanning trees is very high. This step of the analysis is therefore done on graphs with 10 nodes instead, since this turned out to be the highest number of nodes for which the computation time is still manageable.

5 Computational Results

In this section, the results of the performance analysis will be presented. The results are divided into two parts. First, the results for random geometric and $G_{n,p}$ random graphs with 75 nodes will be given. Secondly, the results of the random geometric and $G_{n,p}$ random graphs with 10 nodes will be given. It is important to note that the graphs with 10 nodes give more insight into the quality of the LeafyST heuristic as explained in 4.2.3. However, the results of the graphs with 75 nodes are more relevant for real life applications of the MLST problem, since many applications of this problem require networks with more than 10 nodes.

The most important results of the experiments were stored in tables, these can be found in the appendix. The data in these tables were analyzed using Python and Microsoft Excel.

5.1 Graphs with 75 nodes

For graphs with 75 nodes, the following data is stored in the tables:

- Estimated average number of leaves of all possible spanning trees
- Number of leaves found by the LeafyST heuristic
- Optimal solution to the MLST problem found by the MIP
- Relative difference between the optimum and the solution found by the LeafyST heuristic

From these tables, the main results that were found in the analyses are shown in the following figures. These figures show the number of leaves in the spanning tree found by the LeafyST heuristic for each graph. Note that the values of p and r are increasing with a step of length 0.1 every 10 graphs, so the effect of this increase can also be seen in the figures.

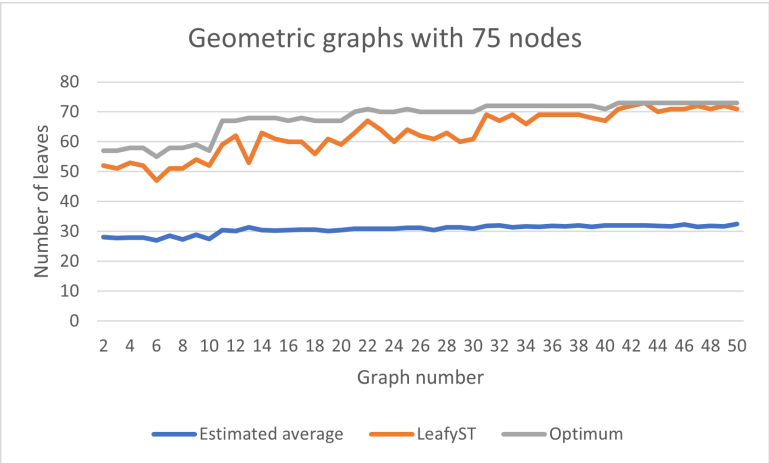


Figure 3: Number of leaves in the spanning trees for random geometric graphs with 75 nodes

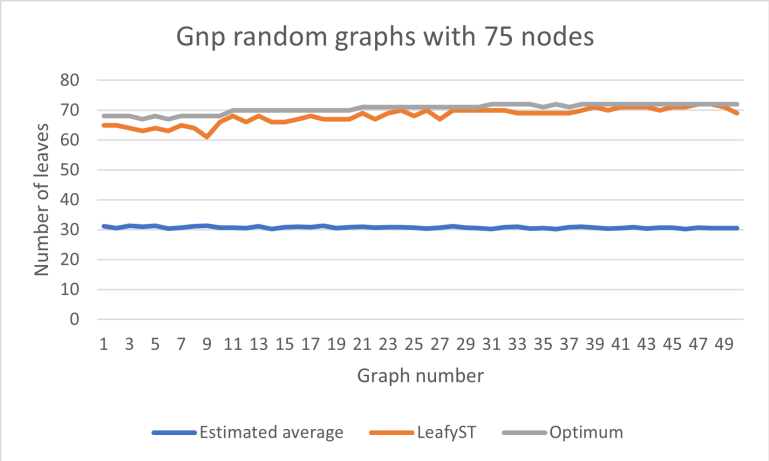


FIGURE 4: NUMBER OF LEAVES IN THE SPANNING TREES FOR $G_{n,p}$ RANDOM GRAPHS WITH 75 NODES

Observation 2 *The performance of the LeafyST heuristic is better for denser graphs.*

When comparing these figures, a notable result is that the performance of the LeafyST heuristic is better for $G_{n,p}$ random graphs than for random geometric graphs. To show this more clearly, a comparison between the performances was made. This was done by calculating the average relative difference between the optimum and the solution found by the LeafyST heuristic for the different values of p and r.

p or r	Geo	GNP
0,2	10%	6%
0,3	12%	4%
0,4	11%	3%
0,5	5%	3%
0,6	2%	2%

FIGURE 5: COMPARISON OF THE RELATIVE DIFFERENCE

This table indeed shows that the LeafyST heuristic performs better for $G_{n,p}$ random graphs, especially when p and r are between 0.2 and 0.4, that is, when the graphs are relatively sparse.

5.2 Graphs with 10 nodes

For graphs with 10 nodes, the experiments that were conducted give a more detailed insight in the performance of the LeafyST heuristic. The size of these graphs allowed us to use the SpanningTreeIterator function that calculates all possible spanning trees for a graph G.

To illustrate the distribution of the number of leaves of all spanning trees, the amount of leaves in these spanning trees is presented in a histogram. This histogram is also used to analyze how the LeafyST performs with respect to this distribution. Figure 5 shows the distribution of the number of leaves in each possible spanning tree for a random geometric graph with 10 nodes and $r = 0,6$.

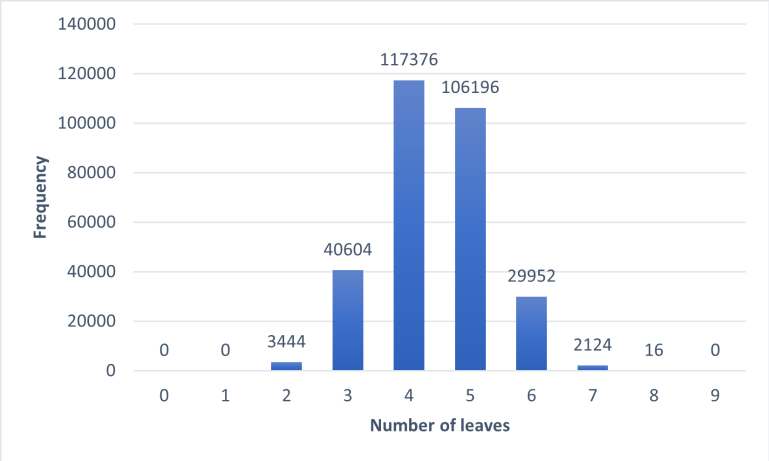


FIGURE 6: HISTOGRAM FOR THE NUMBER OF LEAVES IN SPANNING TREES FOR A GEOMETRIC GRAPH WITH 10 NODES

For this graph, there are almost 300.000 possible spanning trees. The optimal solution for this graph is a spanning tree with 8 leaves and there are 16 possible spanning trees containing 8 leaves. The solution found by the LeafyST heuristic contains 7 leaves, so it did not find the optimum for this graph but still performed relatively well. The histogram also shows that the distribution of the amount of leaves tends to be a normal distribution skewed to the right.

The results for the whole experiment can be found in the tables in the appendix. These tables contain the following data for the 50 graphs of the experiment:

- Optimal solution to the MLST problem
- Average number of leaves of all possible spanning trees
- Number of leaves found by LeafyST heuristic
- Percentage of possible spanning trees with a smaller or equal amount of leaves compared to the solution found by LeafyST
- Total number of spanning trees

The data in these tables were analyzed and the most interesting results are shown in the two graphs below. These graphs show a comparison between the optimal solution, the solution found by the LeafyST heuristic and the mean number of leaves of all possible spanning trees.

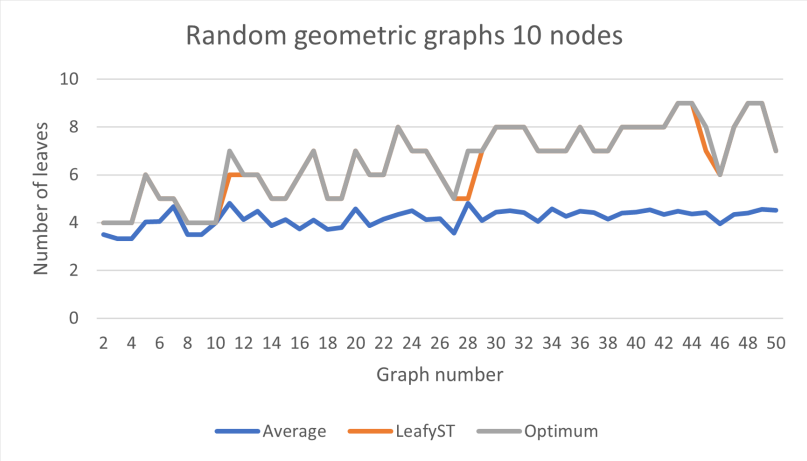


Figure 7: Number of leaves in spanning trees for random geometric graphs with 10 nodes

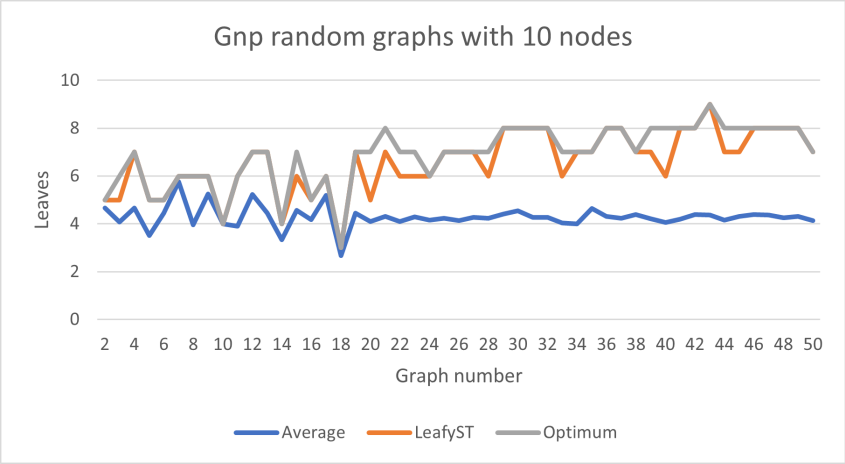


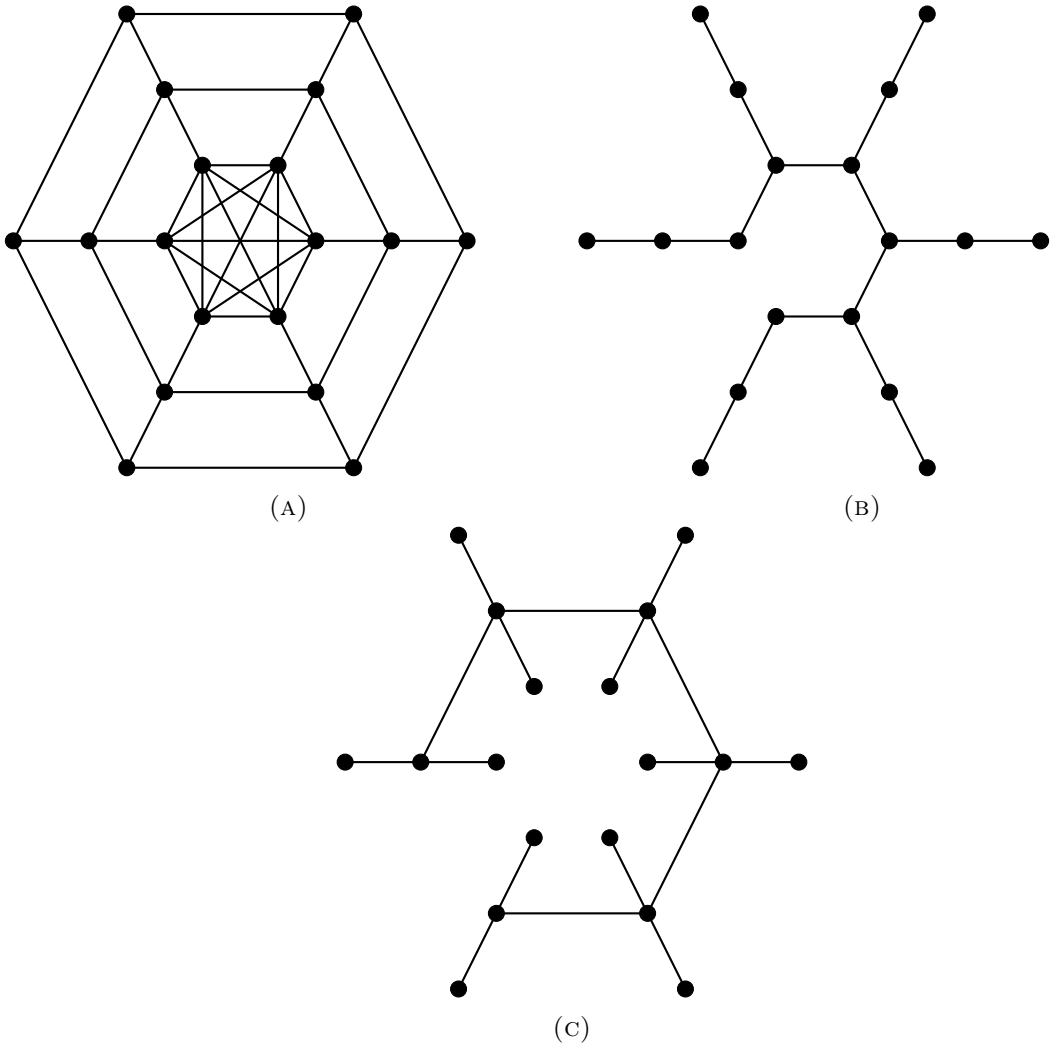
FIGURE 8: NUMBER OF LEAVES IN SPANNING TREES FOR $G_{n,p}$ RANDOM GRAPHS WITH 10 NODES

These figures show that the LeafyST heuristic often finds the optimal solution for graphs with 10 nodes. In the experiment with random geometric graphs, the LeafyST heuristic found the optimal solution for 94% of the graphs. With $G_{n,p}$ random graphs, the heuristic found the optimum for 76% of the graphs.

6 Discussion and Recommendations

6.1 Discussion

The LeafyST heuristic, designed to find spanning trees with many leaves, finds reasonably good solutions on $G_{n,p}$ random graphs and random geometric graphs. However, there are also cases where the quality of the solution found by the LeafyST heuristic is quite far from the optimum. In order to find out what the largest gap to this optimum is, we tried finding a lower bound for the performance. The following graph is an instance where the LeafyST heuristic is off the optimum by a factor of 2. The solution found by the LeafyST heuristic contains half as many leaves as an optimal solution.



(A) (B) (C)

FIGURE 9: EXAMPLE OF A GRAPH, A SOLUTION FOUND BY THE LEAFYST HEURISTIC AND AN OPTIMAL SOLUTION.

Another interesting case is with graphs where the assigned weights are all equal, for example complete graphs. In these graphs, the heuristic could theoretically return each possible spanning tree. This also means that in the case of a complete graph, the heuristic may return a Hamiltonian path containing only 2 leaves. However, in a complete graph it is possible to find spanning trees containing $V - 1$ leaves. This would mean that the performance factor of the heuristic is $\frac{|V|-1}{2}$. This example shows that it is impossible to

prove a constant performance factor for the LeafyST heuristic.

In order to test the actual performance of the LeafyST heuristic on complete graphs, the heuristic was tested on these graphs and always found the optimum. We suspect that this is because of the labeling of edges and vertices done in the implementation. So if we have a complete graph $G = (V, E)$ with $V = \{v_0, v_1, \ldots, v_n\}$ and $E = \{e_0, e_1, \ldots, e_m\}$, we think that the vertices and edges are processed by the heuristic in order of their label, which would explain why the heuristic finds the optimum for complete graphs.

While the practical performance of the heuristic is reasonably good, a well-designed tie-breaking rule could improve upon this performance. This tie-breaking rule could use a method for keeping track of vertices that are potential leaves and steering Kruskal's algorithm to include these vertices in the spanning tree. However, one of the main advantages of the LeafyST heuristic is the easy implementation and the addition of a tie-breaking rule could affect this.

6.2 Recommendations for future research

In this research, the LeafyST heuristic was designed to find solutions of reasonable quality for the MLST problem with a time complexity equal to that of Kruskal's algorithm. This did result in a heuristic with reasonably good practical performance that allowed easy implementation, but we did not prove that this heuristic is an approximation algorithm. We do know the approximation algorithm with the best known approximation ratio, which is an approximation ratio of 2 [14]. Finding a better approximation algorithm for this problem, or proving that such an algorithm does not exist remains an interesting open problem.

In this research, the MLST problem is addressed for unweighted graphs. However, it could also be interesting to investigate if the heuristic can be modified, such that it would find solutions to the weighted maximum leaf spanning tree problem. The goal for this problem is to find a spanning tree where the sum of the weights of the leaves is maximized.

The idea of steering Kruskal's algorithm in the right direction using a weight assignment rule could also be applied to other graph theoretical problems. If, for example, we want to find a spanning tree for an unweighted graph, but the maximum degree of the vertices in this tree should be low, a weight assignment method might steer Kruskal's algorithm in the right direction.

7 Conclusion

The goal of this study was to design a heuristic for finding leafy spanning trees using a well-known minimum spanning tree algorithm. The heuristic should also be practically fast and should be easy to implement. The resulting LeafyST heuristic matches these requirements and as seen in the results, it performs well on $G_{n,p}$ random graphs and random geometric graphs. The heuristic also has a time complexity equal to that of Kruskal's algorithm. From this we can conclude that the LeafyST heuristic is a simple and efficient method to find relatively good solutions for the MLST problem.

References

[1] Bonsma, P. *Sparse cuts, matching-cuts and leafy trees in graphs.* PhD thesis, University of Twente, Netherlands, 2006.

[2] Cormen, T., Stein, C., Rivest, R., and Leiserson, C. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[3] Erdös, P., and Rényi, A. On random graphs. *Publicationes Mathematicae Debrecen 6* (1959), 290–297.

[4] Galbiati, G., Maffioli, F., and Morzenti, A. A short note on the approximability of the maximum leaves spanning tree problem. *Information Processing Letters 52*, 1 (1994), 45–49.

[5] Garey, M., and Johnson, D. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., USA, 1990.

[6] Gilbert, E. N. Random Graphs. *The Annals of Mathematical Statistics 30*, 4 (1959), 1141 – 1144.

[7] Greenberg, H. Greedy algorithms for minimum spanning tree. *University of Colorado at Denver* (1998).

[8] Grimaldi, R. *Discrete and Combinatorial Mathematics; An Applied Introduction.* Addison-Wesley Longman Publishing Co., Inc., USA, 1985.

[9] Guha, S., and Khuller, S. Approximation algorithms for connected dominating sets. *Algorithmica 20* (1996), 374–387.

[10] Lu, H., and Ravi, R. Approximating maximum leaf spanning trees in almost linear time. *Journal of Algorithms 29*, 1 (1998), 132–141.

[11] Nekovee, M. Worm epidemics in wireless ad hoc networks. *New Journal of Physics 9*, 6 (jun 2007), 189.

[12] NetworkX. *NetworkX references*, June 2023. https://networkx.org/documentation/stable/reference/index.html.

[13] Ruan, L., Du, H., Jia, X., Wu, W., Li, Y., and Ko, K. A greedy approximation for minimum connected dominating sets. *Theoretical Computer Science 329*, 1 (2004), 325–330.

[14] Solis-Oba, R., Bonsma, P., and Lowski, S. A 2-approximation algorithm for finding a spanning tree with maximum number of leaves. *Algorithmica* (2017), 374 – 388.

[15] Thomas, L., Diehr, P., Emerson, S., and Chen, L. The importance of the normality assumption in large public health data sets. *Annual Review of Public Health 23*, 1 (2002), 151–169. PMID: 11910059.

A Table geometric graphs with $n = 10$

Graph number	Average	LeafyST	Optimum	Nr of ST	Percentile
1	2,50	3	3	8	100,00%
2	3,50	4	4	8	100,00%
3	3,33	4	4	9	100,00%
4	3,33	4	4	3	100,00%
5	4,03	6	6	864	100,00%
6	4,05	5	5	21	100,00%
7	4,67	5	5	3	100,00%
8	3,50	4	4	8	100,00%
9	3,50	4	4	24	100,00%
10	4,00	4	4	1	100,00%
11	4,82	6	7	272	99,26%
12	4,13	6	6	72	100,00%
13	4,47	6	6	185	100,00%
14	3,88	5	5	168	100,00%
15	4,13	5	5	8	100,00%
16	3,73	6	6	1575	100,00%
17	4,10	7	7	744	100,00%
18	3,71	5	5	24	100,00%
19	3,79	5	5	24	100,00%
20	4,58	7	7	297	100,00%
21	3,87	6	6	375	100,00%
22	4,15	6	6	1107	100,00%
23	4,34	8	8	14595	100,00%
24	4,50	7	7	83632	100,00%
25	4,13	7	7	5376	100,00%
26	4,16	6	6	165	100,00%
27	3,56	5	5	180	100,00%
28	4,81	5	7	672	80,36%
29	4,09	7	7	4025	100,00%
30	4,43	8	8	103464	100,00%
31	4,50	8	8	66776	100,00%
32	4,41	8	8	192538	100,00%
33	4,05	7	7	30996	100,00%
34	4,58	7	7	384	100,00%
35	4,27	7	7	10815	100,00%
36	4,48	8	8	104256	100,00%
37	4,41	7	7	21370	100,00%
38	4,14	7	7	7680	100,00%
39	4,40	8	8	37520	100,00%
40	4,44	8	8	757593	100,00%
41	4,53	8	8	329728	100,00%
42	4,34	8	8	379848	100,00%
43	4,48	9	9	1774080	100,00%
44	4,36	9	9	447804	100,00%
45	4,42	7	8	299712	99,99%
46	3,95	6	6	3936	100,00%
47	4,33	8	8	139320	100,00%
48	4,39	9	9	3502080	100,00%
49	4,55	9	9	211852	100,00%
50	4,51	7	7	12592	100,00%

B Table $G_{n,p}$ random graphs with $n = 10$

Graph number	Average	LeafyST	Optimum	NumberOfST	Percentile
1	4,00	5	5	15	100,0%
2	4,67	5	5	3	100,0%
3	4,07	5	6	881	98,5%
4	4,66	7	7	510	100,0%
5	3,51	5	5	237	100,0%
6	4,44	5	5	100	100,0%
7	5,75	6	6	8	100,0%
8	3,95	6	6	457	100,0%
9	5,25	6	6	8	100,0%
10	4,00	4	4	8	100,0%
11	3,91	6	6	2183	100,0%
12	5,24	7	7	209	100,0%
13	4,46	7	7	3197	100,0%
14	3,33	4	4	6	100,0%
15	4,57	6	7	484	99,2%
16	4,17	5	5	65	100,0%
17	5,20	6	6	55	100,0%
18	2,67	3	3	3	100,0%
19	4,46	7	7	9562	100,0%
20	4,10	5	7	1250	96,4%
21	4,30	7	8	19349	100,0%
22	4,10	6	7	4558	100,0%
23	4,30	6	7	6582	99,8%
24	4,16	6	6	911	100,0%
25	4,24	7	7	3468	100,0%
26	4,13	7	7	6656	100,0%
27	4,28	7	7	4232	100,0%
28	4,23	6	7	3987	99,6%
29	4,41	8	8	355080	100,0%
30	4,55	8	8	52435	100,0%
31	4,28	8	8	741050	100,0%
32	4,28	8	8	56233	100,0%
33	4,03	6	7	31624	99,9%
34	4,01	7	7	30838	100,0%
35	4,65	7	7	6716	100,0%
36	4,30	8	8	356362	100,0%
37	4,23	8	8	592449	100,0%
38	4,38	7	7	11364	100,0%
39	4,21	7	8	43139	100,0%
40	4,06	6	8	52440	99,8%
41	4,20	8	8	274019	100,0%
42	4,38	8	8	2000259	100,0%
43	4,37	9	9	710424	100,0%
44	4,16	7	8	409919	100,0%
45	4,31	7	8	429572	100,0%
46	4,38	8	8	21488	100,0%
47	4,36	8	8	3044346	100,0%
48	4,25	8	8	2426089	100,0%
49	4,31	8	8	791484	100,0%
50	4,14	7	7	75203	100,0%

C Table geometric graphs with $n = 75$

Graph number	Estimated average	LeafyST	Optimum	Gap to optimum
1	27,88	53	59	10,2%
2	28,1	52	57	8,8%
3	27,85	51	57	10,5%
4	27,96	53	58	8,6%
5	27,88	52	58	10,3%
6	26,99	47	55	14,5%
7	28,61	51	58	12,1%
8	27,38	51	58	12,1%
9	28,82	54	59	8,5%
10	27,43	52	57	8,8%
11	30,45	59	67	11,9%
12	30,15	62	67	7,5%
13	31,32	53	68	22,1%
14	30,44	63	68	7,4%
15	30,33	61	68	10,3%
16	30,51	60	67	10,4%
17	30,58	60	68	11,8%
18	30,57	56	67	16,4%
19	30,15	61	67	9,0%
20	30,49	59	67	11,9%
21	30,91	63	70	10,0%
22	30,91	67	71	5,6%
23	30,93	64	70	8,6%
24	30,97	60	70	14,3%
25	31,15	64	71	9,9%
26	31,23	62	70	11,4%
27	30,45	61	70	12,9%
28	31,41	63	70	10,0%
29	31,3	60	70	14,3%
30	30,94	61	70	12,9%
31	31,89	69	72	4,2%
32	31,99	67	72	6,9%
33	31,39	69	72	4,2%
34	31,68	66	72	8,3%
35	31,53	69	72	4,2%
36	31,91	69	72	4,2%
37	31,73	69	72	4,2%
38	31,96	69	72	4,2%
39	31,49	68	72	5,6%
40	31,94	67	71	5,6%
41	31,97	71	73	2,7%
42	31,94	72	73	1,4%
43	31,94	73	73	0,0%
44	31,91	70	73	4,1%
45	31,68	71	73	2,7%
46	32,35	71	73	2,7%
47	31,46	72	73	1,4%
48	31,87	71	73	2,7%
49	31,76	72	73	1,4%
50	32,4	71	73	2,7%

D Table $G_{n,p}$ random graphs with $n = 10$

Graph nr	Estimated average	LeafyST	Optimum	Gap to optimum
1	31,17	65	68	4%
2	30,59	65	68	4%
3	31,38	64	68	6%
4	30,94	63	67	6%
5	31,35	64	68	6%
6	30,34	63	67	6%
7	30,68	65	68	4%
8	31,15	64	68	6%
9	31,39	61	68	10%
10	30,76	66	68	3%
11	30,76	68	70	3%
12	30,59	66	70	6%
13	31,22	68	70	3%
14	30,18	66	70	6%
15	30,89	66	70	6%
16	30,98	67	70	4%
17	30,82	68	70	3%
18	31,27	67	70	4%
19	30,55	67	70	4%
20	30,86	67	70	4%
21	30,96	69	71	3%
22	30,66	67	71	6%
23	30,86	69	71	3%
24	30,92	70	71	1%
25	30,76	68	71	4%
26	30,46	70	71	1%
27	30,77	67	71	6%
28	31,16	70	71	1%
29	30,7	70	71	1%
30	30,61	70	71	1%
31	30,18	70	72	3%
32	30,81	70	72	3%
33	31,09	69	72	4%
34	30,42	69	72	4%
35	30,49	69	71	3%
36	30,28	69	72	4%
37	30,89	69	71	3%
38	31,02	70	72	3%
39	30,65	71	72	1%
40	30,46	70	72	3%
41	30,59	71	72	1%
42	30,87	71	72	1%
43	30,34	71	72	1%
44	30,68	70	72	3%
45	30,63	71	72	1%
46	30,21	71	72	1%
47	30,64	72	72	0%
48	30,55	72	72	0%
49	30,61	71	72	1%
50	30,48	69	72	4%

E Table T-test

Actual Average	STDef	Estimate	Difference	|t |	P(T99>=t)
2,500	0,500	2,54	0,040	0,8	0,425626
3,500	0,500	3,52	0,020	0,4	0,690018
3,333	0,667	3,28	-0,053	0,8	0,425626
3,333	0,471	3,36	0,027	0,565685	0,572888
4,025	0,768	4,2	0,175	2,272993	0,025188
4,048	0,575	4,1	0,052	0,910366	0,36484
4,667	0,471	4,73	0,063	1,343503	0,182181
3,500	0,500	3,52	0,020	0,4	0,690018
3,500	0,500	3,47	-0,030	0,6	0,549877
4,000	0,000	4	0,000	#DIV/0!	#DIV/0!
4,820	0,786	4,8	-0,020	0,252483	0,801192
4,125	0,897	4,11	-0,015	0,167306	0,86747
4,470	0,666	4,44	-0,030	0,454429	0,650515
3,881	0,762	3,76	-0,121	1,586725	0,115764
4,125	0,599	4,07	-0,055	0,917463	0,36113
3,730	0,859	3,77	0,040	0,471077	0,638622
4,103	0,918	4,06	-0,043	0,473781	0,636699
3,708	0,676	3,69	-0,018	0,271316	0,786713
3,792	0,763	3,8	0,008	0,109272	0,913208
4,576	0,830	4,53	-0,046	0,551374	0,58262
3,869	0,834	3,89	0,021	0,247914	0,804714
4,154	0,884	4,17	0,016	0,175711	0,86088
4,344	0,962	4,49	0,146	1,520709	0,13152
4,500	0,926	4,72	0,220	2,37771	0,019341
4,128	0,934	4,16	0,032	0,342934	0,732375
4,158	0,874	4,31	0,152	1,74493	0,0841
3,556	0,685	3,59	0,034	0,502886	0,616161
4,813	0,837	4,85	0,037	0,448223	0,654972
4,090	0,958	4,25	0,160	1,665364	0,099002
4,432	0,947	4,46	0,028	0,292738	0,770335
4,496	0,954	4,67	0,174	1,826422	0,0708
4,411	0,962	4,62	0,209	2,173153	0,032154
4,051	0,895	4,25	0,199	2,216856	0,028921
4,583	0,970	4,6	0,017	0,171815	0,863934
4,272	0,939	4,33	0,058	0,617173	0,538537
4,484	0,951	4,61	0,126	1,326345	0,187777
4,415	0,886	4,67	0,255	2,881651	0,004851
4,140	0,940	4,1	-0,040	0,421151	0,674557
4,405	0,978	4,52	0,115	1,177811	0,241695
4,435	0,933	4,67	0,235	2,515657	0,01349
4,528	0,889	4,59	0,062	0,69217	0,490451
4,339	0,954	4,43	0,091	0,951493	0,343672
4,477	0,961	4,75	0,273	2,842379	0,005439
4,358	0,966	4,52	0,162	1,680987	0,095918
4,417	0,909	4,58	0,163	1,791045	0,076342
3,949	0,829	3,98	0,031	0,368763	0,713092
4,333	0,952	4,48	0,147	1,54292	0,126041
4,394	0,951	4,38	-0,014	0,143333	0,886318
4,553	0,987	4,8	0,247	2,50557	0,013856
4,515	0,856	4,64	0,125	1,461818	0,146958