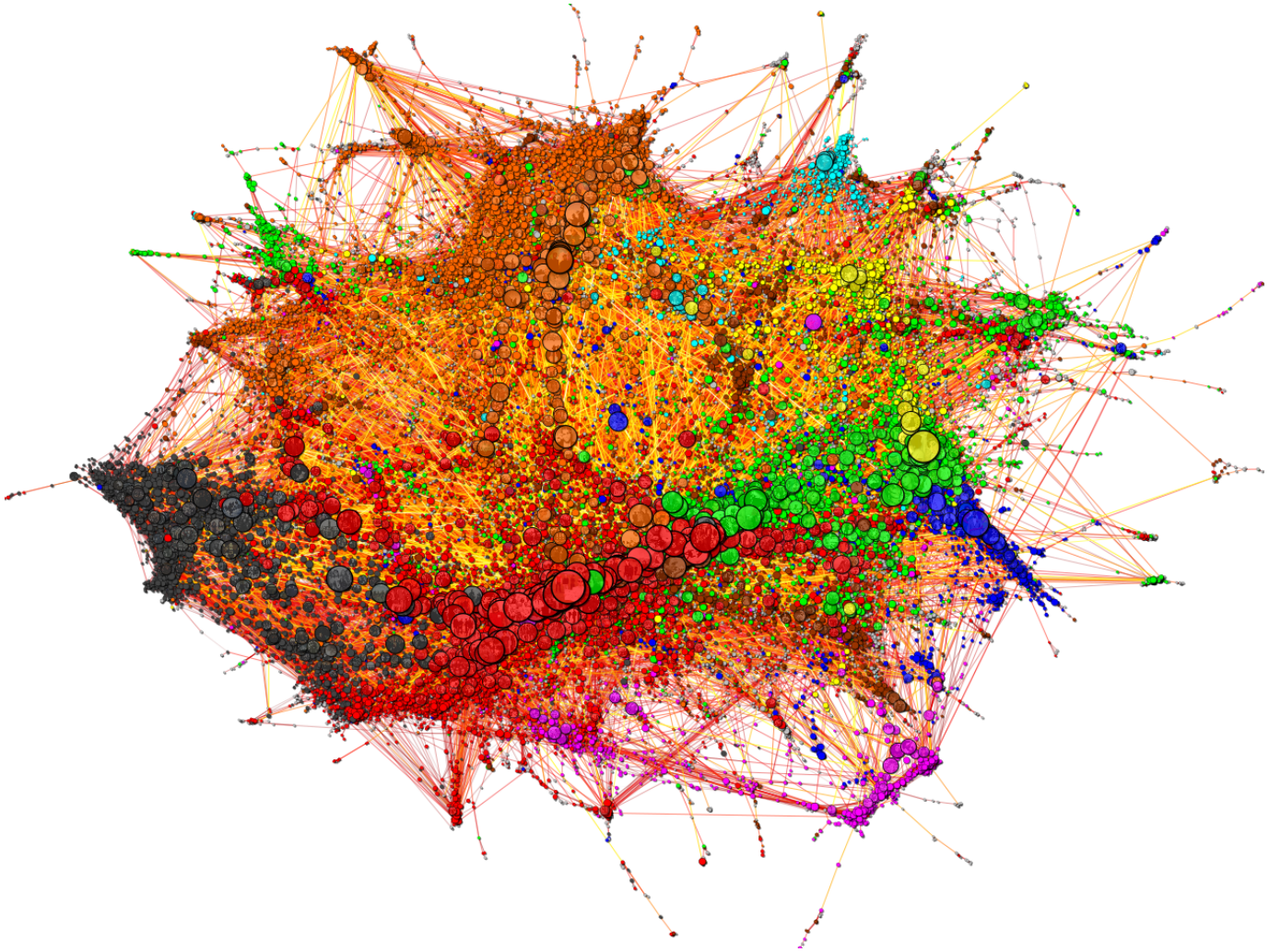
EBC2121 Allocations and Algorithms
Academic year 2020/2021

Approximating the Steiner Tree Problem in Graphs

April 1st, 2021 Meet Shah -i6196781

Maastricht University Max van den Broek - i6161647



Contents

1 Introduction

The world is a fascinating place. It manifests itself in such vivid and mysterious ways that trying to understand it has become a timeless human conquest. In the process of doing so, we have come up with several formulations and models of the world and its objects. A very simple, yet complex visualization of the world is in the form of graph theory. Graphs can be seen everywhere, from the arrangement of planets in the solar system to connections on a social media platform. This implies that techniques which can manipulate and solve graphs become increasingly more useful, and also more potent. Today, graph theory is used to solve some of the most difficult and prevalent problems of life. Many challenges are formulated in terms of graphs, and different algorithms are used to solve those challenges. Implicitly, this reveals two important aspects of graph theory:

1. Being able to represent a particular situation as a graph.
2. Developing algorithms which **efficiently** work upon graphs to solve the problem represented by those graphs.

But what does efficiently mean in this context? Does it mean fast, does it mean to optimality, or does it mean both fast and to optimality? Unfortunately, we often find ourselves in a trade-off between the two. Solving a problem to optimality often takes very long, and solving a problem very fast often gives solutions far from optimality. The key here, is to find a compromise. That is what, in effect, approximation algorithms are. For problems which cannot be solved to optimality quickly, we use approximation algorithms to get a relatively good solution speedily. Such approximation algorithms are vital to a large range of problems, called **NP-Hard** problems. So far, nobody has found a polynomial time algorithm which solves these problems to optimality. Hence, to work with them, it is a must to use approximation algorithms.

One such NP-hard problem is the Steiner tree problem, which we will be analyzing in this report. The Steiner tree problem is a very interesting problem, and has a variety of uses, especially in telecommunication, routing, heat districts and various other industries. In fact, the problem has so many different variants which model a wide range of situations.

The aim of this report is to explain an approximation algorithm for the Steiner tree problem, and expand upon its running time and approximation guarantee.

The next section of the report elaborates upon the properties of the Steiner tree problem. After that, some preliminary concepts are introduced which are used frequently throughout the text. The fourth section of this paper is concerned with explaining the algorithm implemented, with a focus on the construction of an auxiliary graph. Sections 5 and 6 together are used to show the approximation guarantee of the algorithm presented in section 4. After that, our personal implementation of the algorithm is touched upon, and the running time of the algorithm is analyzed. We then discuss the results on a set of instances provided. Then, possible improvements in our algorithm are discussed. Finally the report is concluded.

2 The Steiner Tree Problem

Definition: Given an undirected graph $G = (V, E)$, with non-negative edge costs $c : E \to \mathbb{R}_+$ and a set of terminals $T \subseteq V$, a *Steiner tree* is a tree that connect all the terminals of T. *Minimum cost Steiner tree problem* is the problem where the aim is to minimise the costs of edges included in the Steiner tree.

Decision Variant: Given an undirected graph $G = (V, E)$, with non-negative edge costs $c : E \to \mathbb{R}_+$ and a set of terminals $T \subseteq V$ and a $k \in \mathbb{N}$, does there exist a Steiner tree S such that the cost of S is smaller than k?

The vertices that are not terminals are called Steiner vertices. Intuitively, including vertices that are not terminals could increase the distance and one could opt to exclude these vertices, however, it is possible that via a detour the inclusion of the Steiner vertices can decrease the total cost. If the set of terminals has the same cardinality as the set of vertices than the problem corresponds to the minimum spanning tree. Moreover, if the cardinality of the set of terminals is two, then the problem is the shortest path problem. Hence, the Steiner tree problem is often regarded as a mixture of the shortest path problem and the minimum spanning tree problem. Although polynomial time algorithms exist which solve both of those problems to optimality, no such algorithm exists for the Steiner tree problem, unless $P = NP$.

There are many variants of the problem such as the Steiner Tree problem in directed graphs and directed acyclic graphs. Different forms have also been used since there are different applications for different forms. For instance, the rectilinear Steiner tree is focused on the distance of two points as the difference in their x- and y-coordinates. One application of this version of the Steiner tree is VLSI routing (which looks at horizontal and vertical wires that connect terminals in networks). The VLSI routing problem focuses locating a set of wires that connect nets, which are the terminals. The nets can be reached via electrically equivalent ports. [1] designed an exact algorithm to solve this problem. Other applications of the Steiner tree problem are in the design of telecommunication networks and circuit layout. Different applications require different variants of the Steiner tree problem.

To provide a clearer understanding of the problem, the next section provides a set of feasible and infeasible solutions for a given graph.

2.1 Feasible and Infeasible solutions

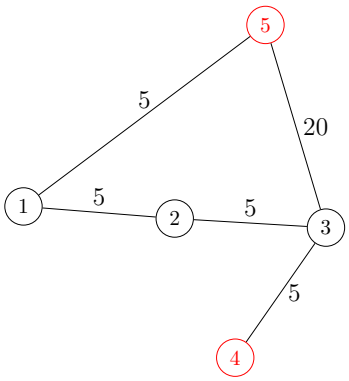


Figure 1: Graph on which to make a Steiner tree

Fig. 1 presents an example of an undirected graph with five vertices of which two are terminals (the ones in red). The aim of the problem is to find a tree such that the terminals are connected at minimum cost. Note that the intermediate vertices can be used to get from a terminal to another terminal.

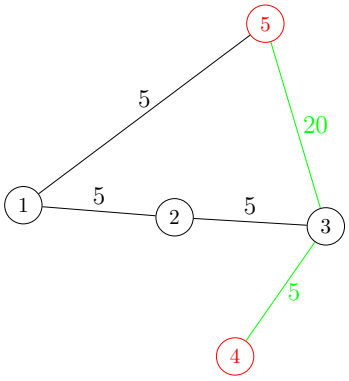


Figure 2: Feasible Solution 1

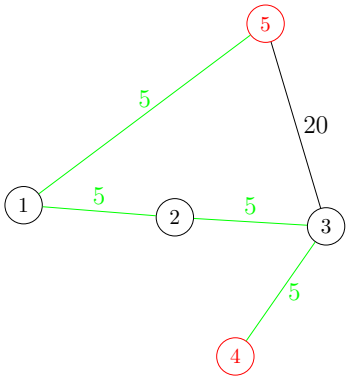


Figure 3: Feasible Solution 2

Fig. 2 and Fig. 3 show feasible solutions to the Steiner tree problem. Note that the cost of the Steiner tree is the sum of all edge costs that are contained in the tree (the path is coloured green). The cost of

the first tree is 25 and contains only two edges. However, a better score can be achieved by taking the path that goes from 5-1-2-3-4, which gives a cost of 20.

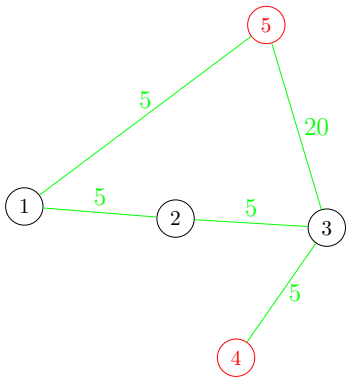


Figure 4: Infeasible Solution 1

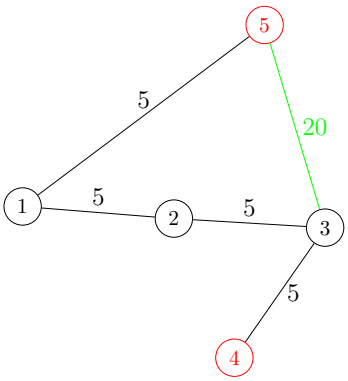


Figure 5: Infeasible Solution 2

Fig. 4 and Fig. 5 show infeasible solutions to the problem. In the first one the terminals are connected but this solution is not a tree since it contains a cycle. The second solution does not contain a cycle, but it also does not contain the bottom terminal (numbered as 4). Note that in this case the number of terminals equals two, hence, we can find an optimal solution in polynomial time by implementing a shortest path algorithm. However, this is a special case! In general, finding an optimal solution for the Steiner tree problem is not straightforward.

2.2 Hardness

The optimisation variant of the Steiner tree problem is NP-hard and the decision problem of the Steiner tree problem is NP-complete, that is, the problem is in the set NP and is NP-hard. It is in NP because, given a cost $k \in \mathbb{N}$, it can be verified in polynomial time whether a given solution is a yes instance by checking that the solution is:

1. Acyclic
2. Connected
3. Contains all terminals T, and only uses edges of G.
4. Has a cost lower than k.

The first three steps can be done in polynomial time by using breadth first searches starting from an arbitrary vertex, checking if all adjacent vertices have not been visited before and checking if all vertices can be visited. The set of visited vertices should also include all terminals T. Calculating and comparing the cost with k can also be done in polynomial time of the input. Hence, the decision variant is in NP. In fact, the optimisation problem is APX-complete [2], which implies that, unless $P = NP$, for an arbitrary ϵ greater than zero, no $(1+\epsilon)$ polynomial time approximation schemes exist for the problem.

2.3 State of Algorithms

Even though the Steiner tree problem cannot be solved to optimality in polynomial time, there are some algorithms which solve it to optimality in non-polynomial time. Certain variants of the problem are also much easier to solve than an arbitrary instance of the problem. One exact algorithm for the Euclidean Steiner tree is the Geosteiner algorithm, which is an exponential time algorithm. Note that due to the exponential run time the algorithm becomes infeasible for large instances. For instance, the algorithm took less than 2 minutes for 50 terminals but took 8 minutes for 100 terminals [3]. Another problem of exact algorithms is that when the dimension of the Euclidean space is larger than 2, general instances with more than 20 terminals have not been solved to optimality [4]. Most often, algorithms which solve the problem to optimality limit a particular dimension of the problem - like the number of vertices, number of terminals or the highest edge weights.

When the input size becomes large having a polynomial time algorithm is preferred. Since the problem is NP-hard, this calls for the use of approximation algorithms. [5] give an overview of variants and what kind of approximations exist. One approach is by [6] who looks at three edges that have one common endpoint and uses them to create a cycle and discard the most expensive edges to form a 11/6-approximation algorithm. Another approach by [7] uses a LP-relaxation based on an iterative randomised rounding technique where the LP-relaxation uses the concept of directed components (two vertices that are connected in an induced subgraph), which are sampled based on their value in the LP relaxation. These are consequently added and the LP is updated. Their algorithm is roughly a 1.39-approximation algorithm and they have an integrality gap of 1.55.

3 Preliminaries

The Steiner problem, or the Steiner tree problem from now refers to the minimum cost Steiner tree problem. Let $G = (V, E)$ be the undirected graph on which the Steiner problem is to be approximated. Each $e \in E$ has an associated edge cost c_e. If e is $\{u, v\}$ this cost can also be represented as c_{uv} or $c(\{u, v\})$ and the smallest cost (shortest distance) between two vertices u and v in G be c^*_{uv} or $c^*(\{u, v\})$

Let c(G') represent the sum of the cost of edges in a graph G'.

Let $T \subseteq V$ be the set of terminals.

A very important definition is that of a Voronoi region of a terminal t:

For each vertex $k \in T$, there exists a Voronoi region $N(k)$ such that for $v \in V$:

Definition: *Voronoi region* $N(k)$: Set of vertices of G which are closer to terminal k than any other terminal in G.

$$v \in N(k) \implies c^*_{vk} \leq c^*_{vt} \quad \forall t \in T$$

If this property holds for multiple terminals for a given vertex, it is in the Voronoi region of one of the terminals.

Hence, $G = \bigcup_{t \in T} N(t)$ and $N(k) \cap N(t) = \emptyset$ if $k \neq t$ for $k, t \in T$

Let $H = (T, E')$ be a graph such that E' contains an edge for each terminal pair, with the cost being the smallest cost between those terminals in G. H can be constructed by first making a metric completion of G (finding all pairs shortest paths) and then taking the subgraph induced by the Terminals. The terms metric completion and metric closure are used interchangeably in this report.

4 Algorithm

It is widely known that a simple asymptotic 2-approximation algorithm exists for the Steiner tree problem, which finds the shortest path between all terminals and makes a subgraph of G on T using those costs, then forms a minimum spanning tree on this, expands it, and then forms a minimum spanning tree on the expanded graph. The most time consuming step of this algorithm is finding the shortest path from every terminal to every other terminal. This generally takes O($|T||V|^2$) using Dijkstra's algorithm for all the terminals. For large instances, this becomes very time consuming.

To overcome this, we implement the approach developed by [8], which uses the Voronoi regions of terminals to efficiently get an auxiliary graph, which can be used to construct a minimum spanning tree

for the subgraph induced by the terminals on the metric closure.

Algorithm 1: Approximation Algorithm for the Minimum Steiner Tree Problem

Input: $G = (V, E)$ with edge costs $c : E \longrightarrow \mathbb{R}_+$ and set of Terminals $T \subseteq V$;
Output: $c(F)$, the cost of a tree which spans all the terminals;

1. Compute the Auxiliary Graph $G^* = (T, E^*)$ with edge costs $c_1(e)$;
2. Construct a minimum spanning tree G_1 of G^* ;
3. Expand each edge of G_1 to the corresponding path in G;
4. Construct a minimum spanning tree G_1' on the graph obtained in Step 3 ;
5. Remove non-terminal vertices of degree 1 from G_1' to get the final graph F ;

Return: $c(F)$, the sum of the distances in F;

While steps 2-5 are relatively straightforward, the following subsection explains how the auxiliary graph G^* is constructed.

4.1 Explanation of Step 1

The auxiliary graph constructed in Step 1 has a very particular structure. For each $t \in T$ we construct the Voronoi Region $N(t)$.

Let there be a function $s : V \longrightarrow T$, such that for a vertex u, $s(u) = t \implies u \in N(t)$. Basically, the function returns the terminal whose Voronoi region u is in.

For each $v \in V$, compute the smallest cost to go from v to s(v): $c^*(v, s(v))$.

Exactly how this is achieved is explained in Section 7.

Then use the following algorithm:

Algorithm 2: Auxiliary Graph Edge Making Algorithm

Input: $G = (V, E)$ with edge costs $c : E \longrightarrow \mathbb{R}_+$ and set of Terminals $T \subseteq V$
Voronoi Regions $N(t)$ $\forall t \in T$ and $c^*\{u, s(u)\}$ $\forall u \in V$;
Output: E^* with costs $c_1(e^*)$ for each $e^* \in E^*$;
$E^* = \emptyset$;
for $e = \{u, v\} \in E$ **do**
  Get $s, t \in T$ such that $s = s(u)$ and $t = s(t)$;
  Compute $p(s, t) = c^*_{us} + c_{uv} + c^*_{vt}$;
  if $\{s, t\} \notin E^*$ **then**
    $E^* = E^* \cup \{s, t\}$ with $c_1(\{s, t\}) = p(s, t)$;
  else if $p(s, t) < c_1(\{s, t\})$ **then**
    $c_1(\{s, t\}) = p(s, t)$;
  else
    continue ;
Return: E^* ;

For each $\{s, t\} \in E^*$, we also remember the edge which was used to construct $c_1(\{s, t\})$ This is then used to expand in step 3 of Algorithm 1.

Note that $c^*_{st} \leq c_1(\{s, t\})$ for all $s, t \in T$ as c^* is the optimal smallest cost. Hence, there is no guarantee that the cost c_1 corresponds to the cost c^*.

4.2 Simple Example of Algorithm Working

In Fig. 6, there are three terminal vertices, 1,4 and 5. There are also two non-terminal vertices 2 and 3. First, we find for each of the non-terminal vertices which terminal vertex's Voronoi region they are in.

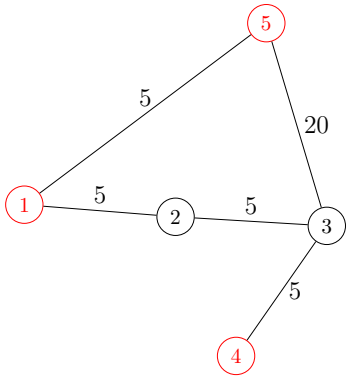


Figure 6: Graph on which to make a Steiner tree

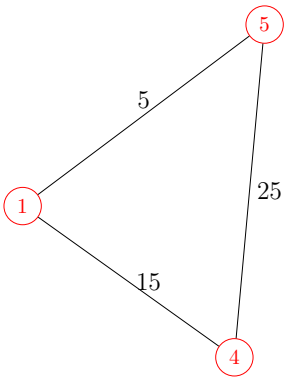


Figure 7: G^*

The shortest path from Vertex 2 to Terminal 1 has cost 5. This is smaller than the costs of its shortest paths to 5(10) and 4(10). Hence, $2 \in N(1)$.

The shortest path from Vertex 3 to Vertex 4 has cost 5. This is smaller than the costs of its shortest paths to 1(10) and 5(15). Hence, $3 \in N(2)$.

We show for two edges how Algorithm 2 works. First consider the edge $\{1, 5\}$. A terminal is always in its own Voronoi region, so this edge corresponds to an edge of $\{1, 5\}$ in E^*. The cost of this edge is $0 + 5 + 0 = 5$.

Now, for edge $\{2, 3\}$, this corresponds to an edge $\{1, 4\}$ in E^*. The cost of this edge is $5+5+5 = 15$. Since at this point no other edge exists between 1 and 3, add this edge in. Similarly, the edge $\{3, 5\}$ will give an edge between $\{4, 5\}$ with cost 25. The graph obtained will be as shown in Fig. 7.

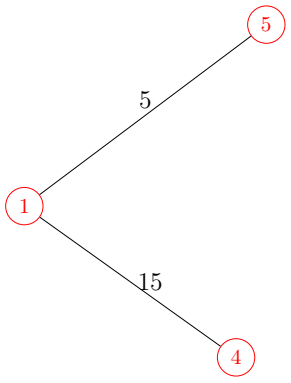


Figure 8: MST(G^*)

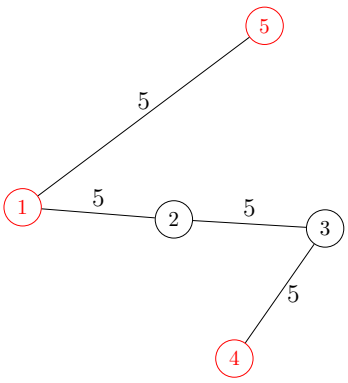


Figure 9: Finally obtained Steiner Tree

Fig. 8 shows the minimum spanning tree of Fig. 7. Upon expansion, this looks like Fig. 9. Note that this is already a tree, does not contain non-terminal vertices of degree 1, and covers all the terminal vertices. Hence, it is a feasible (in this case optimal) Steiner tree for the problem.

5 Equivalence of Auxiliary Graph Minimum Spanning Tree and Metrice Closure Minimum Spanning Tree

To prove the efficacy of Algorithm 1, it is required that the minimum spanning tree obtained after constructing the auxiliary graph according to Algorithm 2 is a minimum spanning tree for the terminal induced subgraph of the metric closure of a graph G, as will be seen in Section 6. They are not trivially equal because the edge costs obtained after using Algorithm 2 do not necessarily correspond to the shortest path between terminals in G. In fact, it may sometimes not even result in a complete graph. An example where the two methods lead to different graphs is Fig. 10. In this figure, the nodes with names starting from V refer to the non-terminal vertices, and those with names starting from T refer to

terminal vertices.

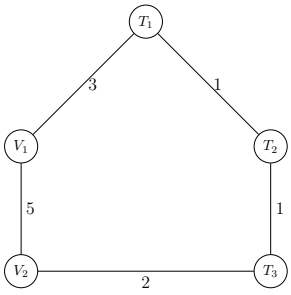


Figure 10: Original Graph

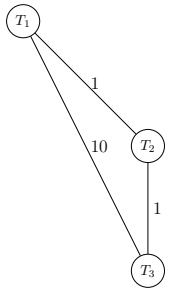


Figure 11: Subgraph induced by terminals

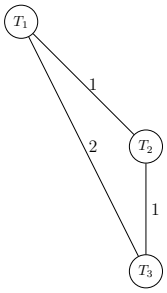


Figure 12: Ideal Induced Subgraph (On Metric Completion)

In Fig. 10 note that $N(T_1) = \{V_1, T_1\}$, $N(T_2) = \{T_2\}$, $N(T_3) = \{V_2, T_3\}$

It is clear that $c^*_{T_1T_2} = 1$, $c^*_{T_2T_3} = 1$ and $c^*_{T_1T_3} = 2$.

However, the graph found by using Voronoi regions to construct between terminal costs is as shown in Fig. 11.

The edge with cost 10 is constructed using $\{V_1, V_2\}$. They belong in separate Voronoi regions, and there are no other edges which would connect T_1 and T_3 which are in their respective Voronoi regions. Clearly, the constructed graph is not the same as the terminal induced subgraph on the metric completion, which is shown in Fig. 12. However, the claim is that the minimum spanning tree of Fig. 11 is a minimum spanning tree of Fig. 12. The proof for this claim is heavily inspired from [8].

In general,

Claim: A minimum spanning tree $G_1 = (T, E_1)$ on $G^* = (T, E^*)$ as defined in Section 4.1 with edge costs $c_1(u, v)$ is a minimum spanning tree on H =(T, E') which is the terminal induced subgraph on the metric completion of a given graph G.

To prove(1): There exists a minimum spanning tree $G_2 = (T, E_2)$ with edge costs c^*_{st} for all $s, t \in T$ of H which is a subgraph of G^*, and its edges have the same cost as the corresponding ones on G^*.

Assume no such minimum spanning tree exists. Let G_2 be a minimum spanning tree of H.

We may assume that we pick G_2 from the set of minimum spanning trees of H such that it minimizes the quantity

$$A = |E_2 \backslash E^*| \text{ with } E^* \text{ defined as in Section 4.1}$$

Among the set of trees with the same minimal A, we can choose the tree which has the lowest $\sum_{e \in E_2} c_1(e)$

For G_2 to be a subgraph of G^*, A has to be equal to zero, and for each edge $\{s, t\} \in E_2$, $c_1(s, t)$ has to be equal to c^*_{st}.

Assume G_2 is not a subgraph of G^*. Then, there are two possibilities:

$$\textbf{1.}\ \exists e = \{s, t\} \in E_2\ s.t\ e \notin E^*$$

One of the edges in G_2 is just not present in E^*, or

$$\textbf{2.}\ \exists e = \{s, t\} \in E_2\ s.t.\ c_1(s, t) > c^*_{st}$$

This basically implies that one of the edges of the minimum spanning tree G_2 that has a different cost associated with it than the same edge in E^*.

Let $v_0, v_1, \ldots, v_k$ be the shortest path from s to t in G, such that $v_0 = s$ and $v_k = t$

For every vertex v_i, let $s(v_i)$ denote the terminal such that $v_i \in N(s(v_i))$

Now, there exists an edge (v_i, v_{i+1}) in G, otherwise the path in G_2 would not be feasible.

Then either, $s(v_i) = s(v_{i+1})$ or an edge f exists between $s(v_i)$ and $s(v_{i+1})$ in E^* if they are not equal, by construction.

For edge f = $\{s(v_i), s(v_{i+1})\}$:

$c^*_f \leq c_1(f) \leq c^*(s(v_i), v_i) + c(v_i, v_{i+1}) + c(v_{i+1}, s(v_{i+1}))$

By construction of the Voronoi regions (such that $c^*(v, s(v)) \leq c^*(v, k)$ for all $k \in T$)

$c^*(s(v_i), v_i) + c(v_i, v_{i+1}) + c(v_{i+1}, s(v_{i+1})) \leq c^*(s, v_i) + c(v_i, v_{i+1}) + c^*(v_{i+1}, t) = c^*_{st}$

Because that is exactly the distance of the path defined.

If the 2nd condition applies, $c_f \leq c_1(f) \leq c^*_{st} < c_1(s, t)$

Assume $\{s, t\}$ is removed from G_2.

The resulting graph is in two components. There must be some g such that $s(v_g)$ and $s(v_{g+1})$ are in different components. This is true because if they are all in the same component, it would mean s and t are in the same component (as $s(v_0) = s$ and $s(v_k) =$t) which is a contradiction.

Consider a graph G'_2 which is $(T, E_2 \backslash \{s, t\} \cup \{s(v_g), s(v_{g+1})\}$

This connects the two unconnected components, and hence is a spanning tree. Moreover, from the inequality above, this tree has a cost lower than or equal to that of G_2, and hence it is also a minimum spanning tree. Under the first condition, it uses one more edge from E^*. This argument can be repeated until A = 0, which means that a minimum spanning tree of H exists which only uses edges from E^*. Moreover, it is a contradiction as we assumed that G_2 was the minimum spanning tree that minimizes $|E_2 \backslash E^*|$.

Under the second condition, the edge of G_2 which had a higher cost in E^* can be replaced with one which has a lower cost in E^*. The same number of edges in of E^* are used, so A does not change. This means that $\sum_{e \in E_2} c_1(e)$ was not minimal given A. This is also a contradiction.

This implies that there is some minimum spanning tree G_2 of H which is a subgraph of G^* and the edges of G_2 have the same cost in H and G^*. □

To Prove(2): Every minimum spanning tree of G^* is a minimum spanning tree of H.

Let G_1 be the minimum spanning tree of G^*

Note that $c^*(G_1) \leq c_1(G_1) \leq c_1(G_2) = c^*(G_2) \leq c^*(G_1)$

This follows because the optimal edge costs are always better than or equal to c_1. Because G_1 is a minimum spanning tree on G^*, its cost must be less than or equal to the costs of the edges of G_2. However, c^* and c_1 coincide for G_2 from the earlier proof, and it is a minimum spanning tree for H. This brings the equality back around. This implies that the cost of a minimum spanning tree on G^* and H coincide. Moreover, every edge in G^* must be in H, and hence a spanning tree of L is a spanning tree of H. These together imply that G_1 is a minimum spanning tree of H. □

6 Approximation Guarantee

The Steiner tree problem is NP-Hard, which implies that no polynomial-time algorithm solves it to optimality. Hence, it is important to find algorithms which can approximate and give a feasible solution in polynomial time. In order to gauge the effectiveness of such algorithms, one way is to calculate the theoretical approximation guarantee for the algorithm.

[9] designed an algorithm which has a $2(1 - \frac{1}{l})$ approximation guarantee for the Steiner Tree problem, where l is the number of leaves (vertices with degree 1) in the optimal Steiner tree. The algorithm implemented is a modification of that algorithm to make it more time efficient, and hence has the same approximation guarantee. Because l is generally unknown, we show a $2(1 - \frac{1}{|T|})$ approximation guarantee, in a proof adapted from [10].

Claim: The algorithm from Section 4 results is a $2(1-\frac{1}{|T|})$-approximation algorithm for the Steiner Tree problem.

First, note that assuming a given problem graph has metric costs ($c_{uv} \leq c_{uw} + c_{wv}$) is not a problem, as one can create the metric completion of a graph by having an edge between each pair of vertices by setting the cost of the edge to be equal to the cost of the shortest path between the vertices. Any Steiner tree solution to the metric completion of the graph can be made into a solution for the original problem by replacing the edges of the resulting graph with the corresponding paths, and then forming a tree again. This will always have a cost of at most that of the metric completion Steiner tree.

Consider the optimal Steiner tree S of graph G with cost c(S) = OPT. S contains all vertices in T, and perhaps some other vertices. Consider the graph obtained by doubling all the edges in S, every vertex now has an even degree, and since S was a tree all the vertices are connected. This implies that the graph obtained is Eulerian, and hence a Euler tour can be formed on the graph. Let such a Euler tour be denoted as ET.

We have that $c(ET) = 2OPT$, as ET is made from doubling the edges which resulted in OPT. Now, make a Hamiltonian cycle C on the vertices of T using ET. In effect, such a cycle can be made by traversing the Euler tour and skipping ("shortcutting") already visited vertices and non-terminal vertices. The cost of this cycle is less than the cost of ET, due to the metric property of the costs. Hence,

$$c(C) \leq c(ET) = 2OPT$$

Removing the highest cost edge from C results in a spanning tree ST on the set of vertices T. The cost of the highest cost edge is at least the average cost of C.

$$c(ST) \leq c(C) - \frac{1}{T}c(C) \leq 2\left(1 - \frac{1}{|T|}\right)OPT$$

Now, in Section 5, we proved that step 2 of Algorithm 1 resulted in a minimum spanning tree MST of the subgraph induced by the terminals on the metric completion of the problem. Every step after this can only reduce the cost of the solution because expanding and forming a minimum spanning tree will not have a higher cost than step 2. Now, the cost of this minimum spanning tree is clearly less than or equal to c(ST), by definition of a minimum spanning tree on the terminal set. Let ALG be the cost of the solution formed by the Algorithm 1. Then we have that:

$$ALG \leq c(MST) \leq c(ST) \leq 2\left(1 - \frac{1}{|T|}\right)c(C) \leq 2\left(1 - \frac{1}{|T|}\right)OPT$$

Therefore, the algorithm is a $2(1-\frac{1}{|T|})$-approximation algorithm. □

6.1 Tightness

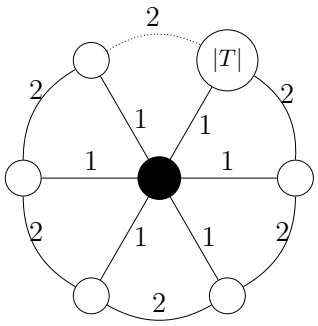


Figure 13: Original Graph(Not all edges are shown)

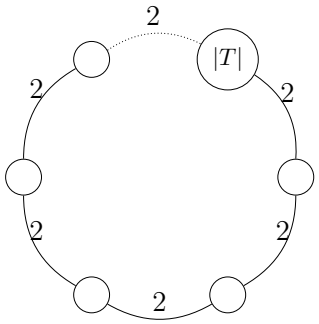


Figure 14: G^*

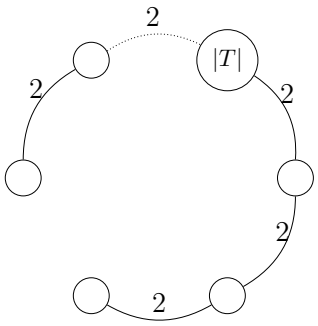


Figure 15: A expansion of $MST(G^*)$

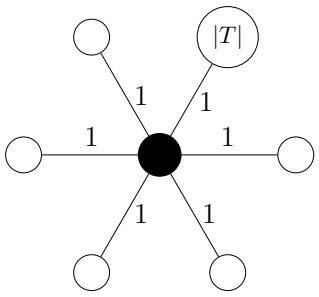


Figure 16: Optimal Steiner tree

Consider the graph in Fig. 13. It contains $|T|$ terminal vertices, each connected to the other through an edge of cost 2. There is one non-terminal vertice, which has a cost of 1 to every single terminal. It can be seen that the optimal Steiner tree of Fig. 16 has a cost of $|T|$.

Now, the non-terminal vertice will randomly be in one of the terminal vertice t's Voronoi region. Now when implementing Algorithm 2, the edges in the final graph G^* need not correspond to the path through the non-terminal node, as there also exists an edge $\{t, t'\}$ $\forall t' \in T$ with same cost as the edge going through the Voronoi region. Hence, we can assume that in some instance of this problem, no edge in G^* corresponds to a path that goes through the central non-terminal node. Because no protection has been implemented against this occurrence in the algorithm, it may happen. Then, the resulting graph is shown in Fig. 15. This graph has a cost $2(|T| - 1)$, as there exists an edge of cost 2 between $|T|$-1 nodes.

Comparing the two values we get that the algorithm resulted in a solution which was $2(1 - \frac{1}{|T|})$ times the optimal. This implies that the analysis of the approximation guarantee is tight. Section 8 shows that this is a worst case scenario, and that in practice it actually provides far better results. This particular graph structure causes that because in the optimal Steiner tree, every terminal node is a leaf.

7 Implementation and Running Time Analysis

A very integral part of any approximation algorithm is its time complexity. Given an approximation guarantee, one would like to achieve it in the most efficient manner possible. The time complexity is dependent on both the data structures used and the different processes conducted in the algorithm. The simple metric closure algorithm described in [9] takes $O(|T||V|^2)$, and its most time consuming step is solving $|T|$ shortest path problems. All improvements of that algorithm try to reduce the time complexity of that step. This is also the fundamental notion of Algorithm 1. In the following subsections, we break down the time complexity of our implementation of the algorithm. Our data structures are made so that adding, removing and getting edges happens in constant time ($O(1)$).

7.1 Constructing Voronoi Regions

To construct Voronoi regions, we first add a new vertex t_0 to the graph, with edges $\{t_0, t\}$ with cost 0 for each $t \in T$. Because one edge is added for each terminal, this takes time O($|T|$). After this, Dijkstra's algorithm is performed with source t_0. Note, first all the terminals will get permanently marked, and the shortest path to any vertex will necessarily go through a terminal. This implies that whenever Dijkstra's algorithm permanently marks a vertex v, that vertex belongs in the Voronoi region of the terminal t on the path to it, because the shortest distance from any terminal to that vertex is smaller than c^*vt. Hence, once the vertex is permanently marked, we can get the terminal whose Voronoi region it is in, and we can get the shortest path from that terminal to the vertex. We implement data structures such that s(u) can be retrieved in constant time.

Now, the best possible implementation of Dijkstra's algorithm, using a fibonacci heap which allows extraction of the minimum in constant time, would allow us to perform this in O($|E| + |V|log|V|$) time complexity [11]. However, this has a very large number of constant time operations which reduce its efficiency. We use a priority queue, which allows us to extract the minimum distance in O($log|V|$) time.

Algorithm 3: Dijkstra's algorithm updated to find Voronoi Regions.

Input: Graph $G = (V \cup t_0, E' = E \bigcup_t \in T\{t_0, t\})$
$c(t_0) = 0, K = \emptyset, R = V \cup t_0 \; \forall v \in V : c(v) = \infty \; \forall t \in T : s(t) = t;$
while $R \neq \emptyset$ **do**
  1. $v = argmin\{c(v) : v \in R\};$
  2. $K = K \cup v$;
  3. $R = R \backslash v$;
  4. $\forall u : \{v, u\} \in E'$:
  if $c(u) > c(v) + c_{vu}$ **then**
    $c(u) = c(v) + c_{vu}$;
    5. Put u in priority queue with cost c(u);
    6. Update predecessor of u to v, and set s(u) = s(v);
Return: Predecessors, Voronoi Regions and c(v) for each $v \in V$.

Note that the while loop has$|V|+1$ iterations, and in each iteration we extract minimum (O($log|E|$) = O($log|V|$)) and do two constant time additions and removals from sets.

Now, the maximum possible times step 4 occurs is twice the number of edges in the graph. In each step, we do a constant time comparision, two constant time setting operations, and we put a vertex into a priority queue. The maximum size of the priority queue, hence, is E, and thus this step can take at most O($log|E|$) which is equivalent to O($log|V|$) as $|E| < |V|^2$.

This makes the time complexity of this step O($|E|log|V| + |V|log|V|$)= O($|E|log|V|$) for connected non-tree graphs. This is the beauty of the algorithm: one must only do one shortest path calculation, which was the bottleneck in previously made algorithms.

7.2 Constructing a Minimum Spanning Tree

Algorithm 2 loops over all edges of the graph and does constant time operations there. Thus its time complexity is O($|E|$).

We construct minimum spanning trees using Kruskal's algorithm with a naive implementation of the Disjoint Set. The algorithm is described in Algorithm 4.

In each iteration of this algorithm, a minimum cost edge is removed from a priority queue, which takes at most O($log|E|$). The find and union operations also take at worst O($log|V|$) (although in our implementation it is less efficient than this). However, generally it takes far less time because the size of sets is rarely V. Hence, the running time of this algorithm is at worst O($|E|log|E|$) = O($|E|log|V|$) as $E < |V|^2$. Now note that the use of this algorithm only occurs on graphs which are smaller in size than the original graph G, and hence it generally runs a lot faster.

Algorithm 4: Kruskal's Algorithm

Input: Set of Edges of a Graph $G = (V, E)$;
Output: Minimum Spanning Tree MST of G. ;
MST $= \emptyset$;
1. Put each edge into a priority queue. ;
while $|MST| \neq |V| - 1$ **do**
  2. Remove the minimum cost edge $\{u, v\}$ from the priority queue.;
  if *findset(u) != findset(v)* **then**
    MST = MST $\cup \{u, v\}$;
    $set(u) \cup set(v)$;
Return: MST ;

7.3 Miscellaneous

Step 3 of Algorithm 1 simply involves looping through the edges of the tree obtained in step 2 ($|T| - 1$ iterations) and updating it to be all the edges along that path. This takes less time than O($|E|log|V|$). Step 4 constructs a minimum spanning tree on this, and which also takes less time than step 1 because the number of edges are lower. Step 5 simply involves looping through all the vertices and checking their degree, and then removing one edge. This can be done in O($|V|$) in the worst case scenario. Hence,

Conclusion: The time complexity of Algorithm 1 through our implementation is O($|E|log|V|$).

[8] states that the algorithm's run time is O($|V|log|V| + |E|$), which is better than our implementation for very dense graphs. However, our implementation is still significantly faster than calculating $|T|$ shortest paths to make a subgraph induced by terminals on the metric closure. It is, of course, polynomial in the input size.

8 Results

To show the validity of the algorithm, it is vital to test it on various different instances of the Steiner tree problem. Because we develop an approximation algorithm, solving instances with a large number of vertices and edges quickly is a must. We implement the algorithm on 199 instances given at [12]. These instances have a very large number of vertices and edges, and vary in their structure, and hence give a good idea about the performance of the approximation algorithm. A distinction is made with respect to even instances and the odd numbered instances, as the odd instances were initially publicly available during the PACE 2018 challenge. In all 199 instances, we get a feasible Steiner tree using our algorithm. [12] provide a list of best known lower bounds to each problem, and we assume those as the optimal value in our analysis.

Note the following definition: For an instance I of the steiner tree problem, let ALG(I) denote the resulting cost of the algorithm, and OPT(I) denote the optimal (when it is known, or the best known lower bound otherwise) cost of the instance.

$$\text{Approximation Ratio} = \frac{ALG(I)}{OPT(I)}$$

The following table provides the average, minimum and maximum approximation ratio for the set of instances.

Instance	All	Odd-Numbered	Even-Numbered
Average Approximation Ratio	1.200	1.215	1.186
Maximum Approximation Ratio	1.916	1.704	1.916
Minimum Approximation Ratio	1.001	1.001	1.001

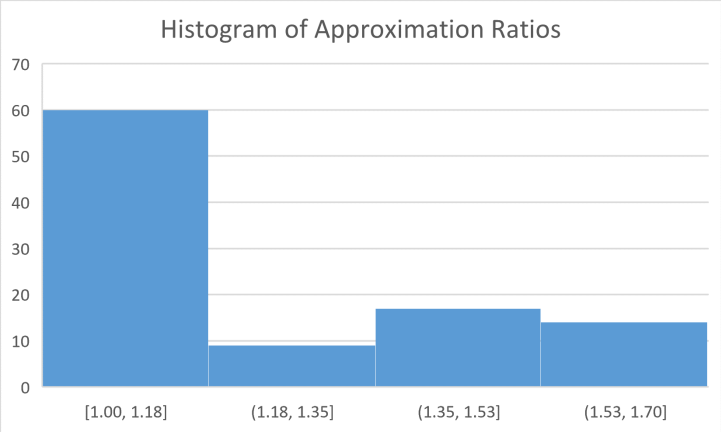


Figure 17: Histogram of Approximation Ratios for odd-numbered instances

Therefore, although the maximum approximation does reach near 2 for some instances, on average the algorithm performs quite well, with an average approximation ratio of 1.2. For some instances, the algorithm provides solutions very close to the optimal solution. Fig. 17 shows that for the odd numbered instances, a preponderance of the approximation ratios falls in the interval [1.00,1.18]. This shows the fallacy with using only approximation guarantees to analyze the performance of algorithms. Because those are based on worst case scenarios, sometimes only very specific instances get to that bound. On average, the algorithm gives decent approximations in polynomial time. Moreover, even for very large instances, such as those with 235686 Nodes, the algorithm gives an approximation several times faster than an implementation of [9] using Dijkstra's algorithm $|T|$ times to get the metric closure induced subgraph. On our machines, for instance 197 (which has the said 235686 nodes), our algorithm takes less than a minute, while our implementation of [9] fails to provide results even after 4 hours using similar data structures.

However, that does not imply that the algorithm is perfect. It is still in our interest to enhance the approximation guarantee to know that a solution is always within a certain bound of the optimal. Moreover, there are ways to improve the solution such that the average approximation ratio also decreases.

9 Possible Improvements and Limitations

9.1 Limitations

The deterministic Algorithm 1 is an efficient way to get a feasible solution to the Steiner tree problem, but like mentioned, this algorithm can only guarantee an asymptotic 2-approximation. Thus for an arbitrary graph, it may provide a an approximation which is double the optimal value. For many purposes, such a bound is too high and hence the algorithm cannot be used in its current state, even if it often provides results better than a 2-approximation. However, there are several ways in which it can be improved, both in terms of its solution quality and running time.

9.2 Adding a reduction process

There are some nodes in a graph which can never be a part of the final Steiner Tree solution, such as non-terminal vertices of degree 1 or 0, and immediately removing them from the graph reduces the number of vertices which must be analyzed in the graph. There are also other reductions possible, such as contracting terminal vertices of degree 1. One can also contract vertices with degree 2. Repeating

these steps until no change occurs in the graph anymore can be very helpful. For some instances, this procedure nearly halves the number of vertices to be analyzed.

There are also more sophisticated reductions possible, as mentioned in [13]. Overall, these reductions can make the time complexity of the algorithm far better.

9.3 Adding a local search procedure

As our algorithm returns a feasible solution to the Steiner tree problem, one may use local search algorithms on the obtained solution to get even better solutions. Some local search techniques are defined in [14]. The applicability of the local search may also result in a better approximation bound for a given solution because sometimes if a local search move cannot be performed, it provides information about the value of the solution.

9.4 Implementing a different algorithm

Like mentioned, through Linear Programming based techniques an approximation guarantee of 1.39 has been achieved [7]. Implementing such an algorithm may provide better results on average. However, there is often a trade-off between the approximation guarantee and the running time of an algorithm, and some algorithms may be infeasible to run on huge instances. What algorithm to choose will depend on personal preferences and need. If a better solution is required, even algorithms which solve the problem to optimality are available, though they take a very long time. If a 'good' solution is required quickly, an algorithm like ours is preferred.

9.5 Implementing more efficient data structures

In our implementation of Dijkstra's algorithm, using a fibonacci heap can have a better time complexity. Similarly, by changing the union and find operations of the Disjoint set used in Kruskal's algorithm, it can be made faster. A union of small improvements can make the overall algorithm faster.

10 Conclusion

The Steiner tree problem in graphs is a difficult to solve, yet important problem in graph theory. In this report, we saw an approximation algorithm for the Steiner tree problem which efficiently brings about a $2\left(1-\frac{1}{|T|}\right)$-approximation for the problem. This algorithm builds upon a traditional algorithm for solving the Steiner tree problem, by optimising its slowest step. This algorithm solves 199 instances from [12] with an average approximation ratio of 1.2, and it runs in $O(\,|E|log|V|\,)$. Nevertheless, even the given algorithm can be improved drastically. There are several ways to improve the solution quality and time complexity of the algorithm, and those have also been discussed.

Approximation algorithms are vital to solving NP-Hard problems such as the Steiner tree problem. Future work can focus upon building even better approximation algorithms for the Steiner tree problem, although it is impossible to get arbitrarily close to a 1 approximation. More work which solves particular types or special instances of the Steiner tree problem can eventually lead to a set of algorithms which can be used depending on the problem at hand. Further, research can also focus on examining the type of Steiner tree problems which occur in real life, and see if they have a special set of characteristics which might make them easier to solve.

There is no bound to the potential of graph theory and approximation algorithms. Making new algorithms and improving upon existing ones is sure to be fruitful to humankind as a whole.

References

[1] Martin Zachariasen and Andre Rohe. 2003. *Rectilinear group Steiner trees and applications in VLSI design* Mathematical Programming 94, 407-433.

[2] Miroslav Chlebik and Janka Chlebikova. 2008. *The Steiner Tree problem on graphs: Inapproximability results.* Theoretical Computer Science 406, 207-214.

[3] Pawel Winter and Martin Zachariasen. 1998. *Euclidean Steiner Minimum trees: An improved exact algorithm.* Networks 30-3, 149-166.

[4] Marcia Fampa, Jon Lee, and Nelson Maculan. 2015.*An overview of exact algorithms for the Euclidean Steiner tree problem in n-space.* International Transactions in Operational Research 23-5, 861-874.

[5] Cedric Bentz, Marie-Christine Costa, Alain Hertz. 2020. *On the edge capacitated Steiner Tree problem.* Discrete Optimization 38.

[6] Alexander Zelikovsky. 1993. *An 11/6-approximation algorithm for the network Steiner problem.* Algorithmica 9, 463-470.

[7] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoβ, Laura Sanita. 2010. *An improved LP-based Approximation for Steiner Tree.* Proceedings of the Annual ACM Symposium on Theory of Computing.

[8] Kurt Mehlhorn. 1988. *A Faster Approximation Algorithm for the Steiner Problem in Graphs.* Information Processing Letters 27, 125-128.

[9] L. Kou, G. Markowsky, and L. Berman. 1981*A Fast Algorithm for Steiner Trees.* Acta Informatica 15, 141-145.

[10] Vijay V. Vazirani. 2001. *Approximation algorithms.* Springer-Verlag, Berlin, Heidelberg.

[11] Michael Fredman and Robert Tarjan. 1984. *Fibonacci Heaps and Their uses in Improved Network Optimization Algorithms.* 25th Annual Symposium on Foundations of Computer Science.

[12] The Program Committee of the Third Parameterized Algorithms and Computational Experiments Challenge. 2018. *Steiner-Tree-PACE-2018-instances.* Github.

[13] Daniel Rehfeldt. 2015. *A Generic Approach to Solving the Steiner Tree Problem and Variants* Technical University Berlin.

[14] Eduardo Uchoa and Renato Fonseca F. Werneck. 2010. *Fast Local Search for Steiner Trees in Graphs* Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments.