Stefan Golja, BSc

On a local search algorithm for the Steiner forest problem: theoretical analysis, implemenation and computational results

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Mathematics

submitted to

Graz University of Technology

Supervisor

Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Eranda Dragoti-Çela

Institute of Discrete Mathematics

Graz, May 2019

Affidavit

Eine Reise, tausend Meilen lang, mit einem ersten Schritt fing sie an.
Laotse

Abstract

The Steiner forest problem is a well-studied NP-hard problem in combinatorial optimization, for which a primal-dual constant factor approximation was shown by Agrawal, Klein & Ravi [AKR95] and Goemans & Williamson [GW95] in the years between 1991 and 1995.

In 2014, Gupta & Kumar [GK14] presented a constant factor approximation algorithm that is purely of combinatorial nature. At that time, this was the first combinatorial algorithm for which a constant approximation factor could be shown.

In this master thesis, we want to deal with another constant factor approximation for the Steiner forest problem: Gross et al. [G17] introduced a local search algorithm (LSA) and proved it to be a polynomial time constant factor approximation for the named problem.

In the first part of this thesis, we introduce the Steiner forest problem (SFP) itself and review some results on SFP. Further we provide some general preliminaries on local search algorithms. Then we introduce the local moves that define LSA including small examples and comments for an easy understanding. After that, a theoretical analysis of the algorithm is provided considering the approximation factor and the time complexity. We first show that the presented algorithm has indeed a constant approximation factor. We distinguish two cases: (a) the case where the local optimum is a tree and (b) the case where the local optimum is a forest. In Case (a) we use a potential function to bound the total length of the locally optimal solution. In the general Case (b), where the local optimum is a forest and not necessarily a tree, we transform the solution in such a way that the results of Case (a) can be applied. The second part of the analysis focuses on the time-complexity of the algorithm. We depict which obstacles have to be handled in order to make the algorithm run in polynomial time without violating the approximation guarantee. There are quite a number of complex transformations and intermediate results needed in this analysis; they have been illustrated by means of small examples and comprehensive visualizations. We have also added a number of comments to explain non-trivial details which have been handled as such in the original paper. The first part closes with a short description of two other methods to solve the Steiner forest problem: a greedy algorithm called Gluttonous and a formulation of the problem as an integer program (IP), that can be handled by some standard solver.

The second part of this thesis discusses in detail the implementation of the local search algorithm. We describe also the functions and classes used in the C++ code. In order to evaluate the performance of LSA we have compared it to two other approaches to solve SFP known in the literature: the Gluttonous algorithm and the exact solution of the IP. We provide some details on the implementation of these alternative methods.

In the last part of this master thesis, we compare the performance of the local search algorithm and the two other algorithms mentioned above on two classes of test instances. The comparison addresses both the running time and the quality of the obtained solutions. The first class of test instances contains small randomly generated test instances. The second class of test instances is obtained by modifying benchmark instances for the Steiner tree problem known in the literature.

Table of contents

Part I.
Theory

1. Statement of the problem and preliminaries

1.1. Basic definitions and notations

1.1.1. Graph theory

Definition 1.1. *(Concepts of graph theory)*
Let $G = (V, E)$ be an undirected graph with non-negative edge lengths $d_e \in \mathbb{R}_{\geq 0}$ for every $e \in E$. Let $n := |V|$ be the number of vertices of the graph. For $W \subseteq V$, let $G[W] := (W, E[W])$ be the vertex induced subgraph and for $F \subseteq E$, let $G[F] := (V[F], F)$ be the edge-induced subgraph that consists of all edges in F and the vertex set $V[F]$ that arises from these edges. A forest is a set of edges $F \subseteq E$ such that $G[F]$ is acyclic. If the forest consists of exactly one connected component, it is called a tree.

Let $u, v \in V$ be two vertices in the graph G. We denote the length of the shortest path between u and v in (G, d) by $dist_d(u, v)$, which is called also the shortest path distance between u and v in (G, d).

Let $\mathcal{D} = \{\{s_i, t_i\} \in V \times V \mid i = 1, \ldots, k\}$ be a set of vertex pairs in the graph G. For technical reasons we number the pairs according to non-decreasing shortest path distances (where ties are broken arbitrarily). Hence, $\mathcal{D} = \{\{s_1, t_1\}, \ldots, \{s_k, t_k\}\}$ and $i < j$ implies that $dist_d(s_i, t_i) \leq dist_d(s_j, t_j)$.

For a subset $S \subseteq E$, we define the total length $d(S)$ to be the sum of the lengths of edges in S, i.e. $d(S) := \sum_{e \in S} d_e$. For a subgraph $H = (V_H, E_H)$ of $G = (V, E)$, we define the length $d(H)$ of H as the length of E_H, i.e. $\sum_{e \in E_H} d_e$, if there is no ambiguity about the graph G.

1.1.2. Combinatorial optimization problems

Definition 1.2. *(Combinatorial optimization problem)*
An instance I of a combinatorial optimization problem can be specified as a pair $(\mathcal{F}_I, c)$ where $\mathcal{F} := \mathcal{F}_I$ is the set of feasible solutions to the instance I and $c : \mathcal{F} \to \mathbb{R}_{\geq 0}$ is a cost function. Let $g \in \{max, min\}$ be either the maximum or the minimum function. The goal is to find some feasible solution $F^* \in \mathcal{F}$, such that

$$c(F^*) = g\{\ c(F) \mid F \in \mathcal{F}\ \} \tag{1}$$

A feasible solution F^* that fulfils Equality (1) is called an optimal solution. Let $\mathcal{I}$ be the set of instances as described above. Then a combinatorial optimization problem can be seen as a quadruple $P = (\mathcal{I}, \mathcal{F}_I, c, g)$. If $g = max$, then we call P a (combinatorial) maximization problem, else a (combinatorial) minimization problem. The function c is often called objective function or objective.

1.1.3. Algorithms

Definition 1.3. *(Exact algorithm)*
Let $P = (\mathcal{I}, \mathcal{F}, c, g)$ be a combinatorial optimization problem. An algorithm ALG is called an exact algorithm that solves P, if for every input $I \in \mathcal{I}$, ALG outputs an optimal (and hence feasible) solution of I.

Definition 1.4. *(α-approximation algorithm for combinatorial minimization problems)*
Let $P = (\mathcal{I}, \mathcal{F}, c, min)$ be a combinatorial minimization problem. An algorithm ALG is called an α-approximation algorithm or α-approximation for the problem P, if there is a constant $\alpha \in \mathcal{R}_{\geq 1}$, such that for every input $I \in \mathcal{I}$, ALG outputs a feasible solution $A(I)$ such that

$$c(A(I)) \leq \alpha \cdot c(OPT(I))$$

holds, where $OPT(I)$ is an optimal solution of the instance I.
If in addition to that, ALG runs in polynomial time with respect to the input size, then we call ALG a polynomial time α-approximation algorithm or polynomial time α-approximation for the problem P.

Definition 1.5. *(Neighbourhood, improving solution)*
Let $P = (\mathcal{I}, \mathcal{F}, c, min)$ be a combinatorial minimization problem and let $I \in \mathcal{I}$ be an instance with $\mathcal{F} := \mathcal{F}_I$ being the set of feasible solutions for I. A neighbourhood function for the instance I is a function

$$\begin{aligned} \mathcal{N} = \mathcal{N}_I : \mathcal{F} &\to 2^{\mathcal{F}} \\ F &\mapsto \mathcal{N}(F) \subseteq \mathcal{F} \end{aligned}$$

$\mathcal{N}(F)$ is called the neighbourhood of the solution F. Let F be a feasible solution. A feasible solution F' is called an improving solution with respect to F (and c), if $c(F') < c(F)$.

Definition 1.6. *(Local search algorithm for a combinatorial minimization problem)*
Let $P = (\mathcal{I}, \mathcal{F}, c, min)$ be a combinatorial minimization problem and let $I \in \mathcal{I}$ be an instance with $\mathcal{F} := \mathcal{F}_I$ being the set of feasible solutions for I. Moreover, let $\mathcal{N} := \mathcal{N}_I$ be a neighbourhood function for I. The following generic procedure is called a local search algorithm for the problem P and the neighbourhood function $\mathcal{N}$ for I with respect to the cost function c:

Algorithm 1 Local search for minimization problems, generic version

```
Require: An instance $I$ of a combinatorial minimization problem $P = (\mathcal{I}, \mathcal{F}, c, min)$, a
    feasible solution $F \in \mathcal{F} = \mathcal{F}_I$ and a neighbourhood function $\mathcal{N} = \mathcal{N}_I$.
Ensure: A feasible solution $A \in \mathcal{F}$ for the instance $I$.
1: Start with $A := F$.
2: while $\exists\ A' \in \mathcal{N}(A)$ such that $c(A') < c(A)$ do
3:     Set $A := A'$
4: end while
5: Output $A$
```

A feasible solution F is a local optimum (or local optimal) for the instance I with respect to the neighbourhood function $\mathcal{N}$, if $c(F) \leq c(F')$ holds for all $F' \in \mathcal{N}(F)$. The replacement of a feasible solution $F \in \mathcal{F}$ by some other feasible solution $F' \in \mathcal{N} \subseteq \mathcal{F}$ is called a local move. If $c(F') < c(F)$ holds, then the local move is called improving (with respect to the cost function c). This is often called performing an (improving) local move (with respect to the cost function c). A feasible solution F is called a near-optimal solution, if there exists an optimal solution F^* and a constant $K \geq 1$, such that $c(F) \leq K \cdot c(F^*)$.

Comments:

- Note that a local search algorithm can be an exact algorithm or an α-approximation algorithm or none of both.

- Crucial for every local search algorithm is the definition of the neighbourhood function: If a feasible solution has a neighbourhood that can be not checked in polynomial time for an improving solution, then the local search algorithm will not be a polynomial time algorithm in general. On the other hand, if the neighbourhoods of feasible solutions tend to be small, they can be searched faster than large neighbourhoods, but the quality of local optimal solutions can be poor.

- In general, the starting solution for a local search algorithm can be generated by some simple approximation algorithm or can be arbitrarily chosen from the set of all feasible solutions.

1.2. The Steiner forest problem

Definition 1.7. *(The Steiner forest problem (SFP))*
The input of the Steiner forest problem consists of an undirected, connected graph $G = (V, E)$, a mapping $d : E \to \mathbb{R}_{\geq 0}$ and a set of demands $\mathcal{D} = \{\{s_i, t_i\} \in V \times V \mid s_i \neq t_i,\ i = 1, \dots, k\}$. The vertices s_i, t_i of each demand-pair $\{s_i, t_i\}$, $1 \leq i \leq k$ are called terminals. Vertices that are not contained in any demand-pair are called non-terminals or Steiner vertices. A feasible solution is a subset of edges $F \subseteq E$ such that for each demand-pair $\{s_i, t_i\}$, $1 \leq i \leq k$, s_i and t_i belong to the same connected component of $G_F = (V, F)$. The goal is to find a feasible solution F^* which minimizes the cost $c(F^*) = d(F^*) = \sum_{e \in F^*} d(e)$. Since $d(e) \geq 0$ $\forall e \in E$, we can restrict the feasible solutions F to be such that (V, F) is a forest. We refer by SFP to the Steiner forest problem.

A graph $G = (V, E)$ together with lengths d and demand-pairs $\mathcal{D}$ forms a Steiner forest instance $I = (G, d, \mathcal{D})$, with $n_t := |\mathcal{D}|$ being the number of demand-pairs.

Comments:
Note that we can define SFP also on an unconnected graph G provided that for each demand-pair $\{s_i, t_i\}$ both terminals s_i and t_i belong to the same connected component of G. Then we would solve an instance of the SFP in each connected component of G and obtain the solution of the whole problem as the union of the solutions over all connected components.

Definition 1.8. *(Width, m_{ind}, potential ϕ)*
Let $I = (G = (V, E), d, \mathcal{D})$ be an instance of the SFP and let $E' \subseteq E$ be a connected set of edges. Recall that for two vertices $s, t \in V$ in G, the symbol $dist_d(s, t)$ defines the shortest path distance of s and t in G with respect to the function d. We define:

$$w(E') := max\{dist_d(s, t) \mid \{s, t\} \in \mathcal{D}, \{s, t\} \subseteq V[E']\}, \text{ for any } E' \subseteq E$$

Moreover, let $F \subseteq E$ be a forest in G with connected components $F_1, \dots, F_l \subseteq F$. Then,

$$w(F) := \sum_{i=1}^{l} w(F_i).$$

$w(E')$ is the maximum distance in the original graph G of any demand-pair connected in E'. By the chosen enumeration of the pairs (see Section 1.1), this is the distance of the pair $\{s_i, t_i\}$ with the largest index i among all pairs in $V[E']$. $w(F)$ is also called the width of the forest F, which is the sum of the widths of its connected components.

With $m_{ind}(E') := max\{i \mid \{s_i, t_i\} \subseteq V[E']\}$, we get $w(E') = dist_d(s_{m_{ind}(E')}, t_{m_{ind}(E')})$.

By Definition 1.1, the total length of the forest F is defined by $d(F) := \sum_{e \in F} d(e)$ and the potential of the subgraph F is defined by

$$\phi(F) := d(F) + w(F)$$

Lemma 1.9. *(Bounds for ϕ)*
With the settings as above, we have $d(F) \leq \phi(F) \leq 2d(F)$.

Proof. Clear by the definition of $w(F)$, since $0 \leq w(F) \leq d(F)$. □

Notations 1.10.
Consider an algorithm ALG that takes as input an instance I of the SFP and outputs a feasible solution (the type of the output is a forest), we often denote this forest by $A := A_I$. We denote by $F := F_I$ an optimal solution to the underlying instance I[1]*.*

Example 1.11. *(Introductory example of the SFP)*

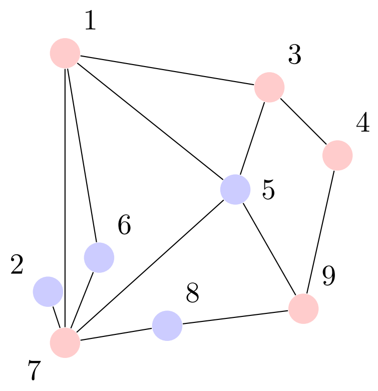


Figure 1: Instance of SFP, terminals marked in red.

Consider the graph $G = (V, E)$ on the vertex set $V = \{1, \ldots, 9\}$ given in the Euclidean plane together with the Euclidean distance d as the length-function depicted in Figure 1. We set the demands as $\mathcal{D} = \{(3,4), (3,9), (1,7)\}$ which are sorted according to non-decreasing shortest path distances. The coordinates of the vertices are given as follows:

1:	(1,9)	4:	(9,6)	7:	(1,0.5)
2:	(0.5,2)	5:	(6,5)	8:	(4,1)
3:	(7,8)	6:	(2,3)	9:	(8,1.5)

One feasible solution for this instance is the forest F_1 on the vertex set V consisting of the edges $\{1,5\}, \{3,4\}, \{3,5\}, \{5,7\}, \{5,9\}$. It can be checked that this is indeed a subgraph of G. Observe that this forest F_1 is actually a tree. The cost of F_1 is equal to the length of the edges, the length of each edge is the Euclidean distance between its endpoints. Hence, the cost of forest F_1 is ≈ 23.1.

Another feasible solution is the forest F_2 on the vertex set V consisting of the edges $\{1,7\}$, $\{3,4\}, \{4,9\}$. This forest consists of two separate trees. The cost of forest F_2 is ≈ 15.9, which is less than the cost of forest F_1.

[1]Since there will always be only one single instance at which we are looking at, we do not need to indicate to which instance the obtained solution or the optimal solution belongs to.

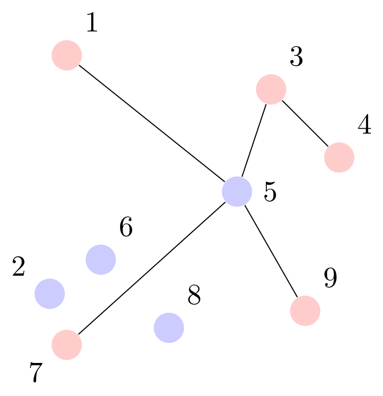


Figure 2: Feasible solution F_1

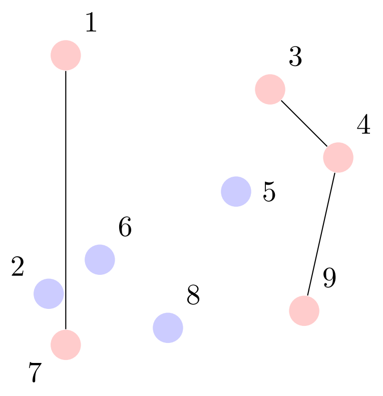


Figure 3: Feasible solution F_2

1.3. SFP as a generalization of other simpler problems

Let's see how the Steiner forest problem relates to other fundamental problems known in combinatorial optimization.

Reachability problem
Input: A directed, not necessarily connected graph $G = (V, E)$ and vertices $u, v \in V$.
Output: A path in G connecting u and v, if such a path exists.
It is easy to see that *depth first search* or *breadth first search* can be used to obtain the desired path, if such one exists. Both algorithms are known to run in polynomial time. Sometimes the output should be only *yes* or *no*, as the answer to the question whether a $u - v$ path in G exists or not.

Shortest path problem
Input: A directed, not necessarily connected graph $G = (V, E)$, edge lengths $d : E \to \mathbb{R}_{\geq 0}$ and vertices $u, v \in V$.
Output: A shortest $u - v$ path[2] in G if a $u - v$ path exists.
This problem is a standard problem in combinatorial optimization and known as *shortest path problem*. Standard algorithms that solves the problem are *Prim's algorithm*, *Dijkstra's algorithm* or *Floyd-Warshall's algorithm*. The last solves even the *all-pairs shortest path problem*, i.e. it determines a shortest path between any two vertices in V. The complexity of *Floyd-Warshall* is cubic in the number of vertices of the graph, hence the *shortest path problem* can be solved in polynomial time.

Steiner tree problem (STP)
Input: A directed (connected) graph $G = (V, E)$, edge lengths $d : E \to \mathbb{R}_{\geq 0}$ and a vertex set $S \subseteq V$.
Output: A tree $T \subseteq E$ of minimal total length that connects all vertices of S.
The *Steiner tree problem* is a generalization of the *shortest path problem*: Given a set of vertices $S \subseteq V$, we ask for a tree of minimum total length that connects all vertices of S. These vertices are called terminals, the vertices in $V \setminus S$ are called non-terminals or Steiner vertices. The STP is NP-hard, see [San03], but simple polynomial time 2-approximations are known. In the case where G is the complete graph and the function d is a metric, the following approach works:

[2] A shortest $u - v$ path in G is a path in G that has the shortest length among all $u - v$ paths, where the length of a path is the sum of the lengths of the contained edges.

Ignore all non-terminals and find a minimum spanning tree on the weighted graph that has vertex set S and edge lengths defined by d. Details can be found in e.g. [Tre11], also have a look on Theorem 3.31.

Steiner forest problem (SFP)
Input: A directed (connected) graph $G = (V, E)$, edge lengths $d : E \to \mathbb{R}_{\geq 0}$ and a set of demand-pairs $\mathcal{D} = \{\{s_i, t_i\} \in V \times V \mid i = 1, \ldots, k\}$.
Output: A forest $A \subseteq E$ of minimal total length such that each demand pair $\{s_i, t_i\}$ lies within exactly one connected component of A.
Knowing the *Steiner tree problem*, it is easy to see that the *Steiner forest problem* from Definition 1.7 is a generalization: Instead of one set $S \subseteq V$, there are many sets $S_i := \{s_i, t_i\} \subseteq V$ gathered as the demand set $\mathcal{D}$ that should be contained in some subgraph $H \subseteq G$, such that each set S_i is contained within exactly one connected component of H. This implies that also the *Steiner forest problem* is NP-hard. Also for this problem, polynomial time 2-approximations are known, but they are not as simple as for the STP. Agrawal, Klein and Ravi [AKR95] showed that there is a polynomial time 2-approximation for the SFP that is based on primal-dual methods and hence not “purely” of combinatorial nature. In 2014, Gupta and Kumar [GK14] presented a polynomial time approximation algorithm for the SFP that is totally based on combinatorial methods and showed that it is a constant factor approximation. Up to that time, all polynomial time SFP algorithms, for which a constant approximation factor was known, where based on linear programming relaxations. Currently no polynomial time approximation algorithm with a better approximation factor than 2 is known for the SFP.

The SFP itself can be generalized in many other ways like the *online-SFP* or the *network connectivity leasing problem* and can be seen as a classical network design problem, see e.g. [AAB04] for more details related to generalizations of the SFP.

In practical, the SFP has applications in the design of road-networks, communication-networks, integrated circuit, etc.

About the naming of the problem
The Steiner forest problem is named after the Swiss mathematician *Jacob Steiner* and belongs to *Karp's 21 NP-complete problems*, for which Richard Karp showed in 1972 that they are all NP-complete. The complexity proofs are done based on reductions from the *boolean satisfiability problem* which is NP-complete due to the *Cook-Levin Theorem.*
Definition 1.7 does not explicitly explain why the term *forest* is appropriate. Remember that the lengths are non-negative. Assume the optimal solution (which is in general a subgraph of G) is not a forest i.e. not cycle free. One can easily remove an edge from every cycle and this does not hurt the connectivity property. If every removed edge has length zero, one gets an optimal solution that is cycle free and hence a forest. If one of the removed edges has positive length, then the obtained feasible solution is cycle free and has lower total length than the optimal solution which contradicts the optimality. Hence, there is always an optimal solution that is a forest.

2. The local search algorithm from Gross et al. for the SFP

The ideas and notations of the Sections 2 to 4 are based on the paper *A Local-Search Algorithm for Steiner Forest* of Matrin Groß, Anupam Gupta, Amit Kumar, Jannik Matuschke, Daniel R. Schmidt, Melanie Schmidt and José Verschae [G17].

2.1. The local search algorithm in a nutshell

As in the generic version of a local search algorithm, also this local search algorithm starts with a feasible solution and performs in each step one of the local moves described below, i.e. it checks every solution in the neighbourhood of the current feasible solution and then take one feasible solution which is best among all of them until no such solution exists any more.

We consider the following local moves (which can be seen as a description of the neighbourhood):

- **edge-set swap:** Add an edge to a tree in the forest and remove one or more edges from the created cycle such that the solution remains feasible.
- **path-set swap:** Add a shortest path between two vertices of a tree in the forest and remove edges from the created cycle such that the solution remains feasible.
- **connecting move:** Connect some trees of the current solution-forest by adding edges between them.

In addition to the local moves, which are considered in each step during the algorithm, there is one final post-processing move at the end of the algorithm:

- **clean up:** Delete all inessential edges, i.e. all edges that do not alter the feasibility of the solution.

A detailed description and illustrating examples of the local moves and the post-processing moves can be seen in Section 2.2.

The algorithm performs a local search with respect to the potential ϕ, and not with respect to the total length of the current solution. Therefore, also the connection move makes sense, it does not decrease the total length of the solution but it possibly decreases the potential ϕ. Consider the SFP instance in the Euclidean plane represented in Figure 4: The forest shown in (a) consists of two trees each of them connecting one demand-pair. For both trees, we see that the width is exactly the length of the tree itself. Since the two trees have equal length L, the potential of the left forest is $4L$. The forest shown in (b), results by connecting the two trees from (a) by the edge $\{1, 2\}$ and hence consists of only one tree. Note that the potential of this tree is L. Hence, the potential of the tree in (b) is given by $3L + d(\{1, 2\})$, which is smaller than $4L$ as long as $d(\{1, 2\}) < L$.

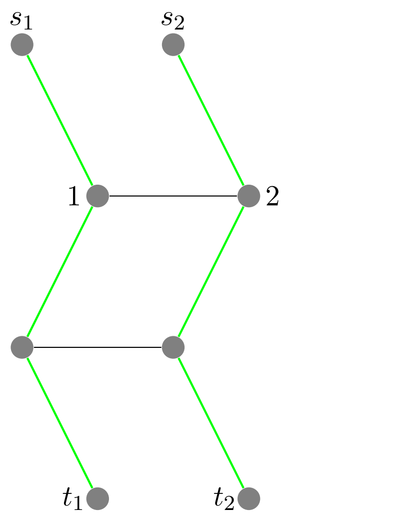
In [G17] can be shown, that there are examples, where performing the moves described above with respect to the total length of the solution gives a local optima with cost $\Omega(log(n))\cdot OPT$, where OPT denotes the cost of an optimal solution. We will show, that performing local search with respect to the potential ϕ yields a constant approximation factor algorithm.

Definition 2.1. *(X-optimal)*
A feasible solution A is called X-optimal with respect to a certain kind of a local move X, if no moves of the kind X are improving.

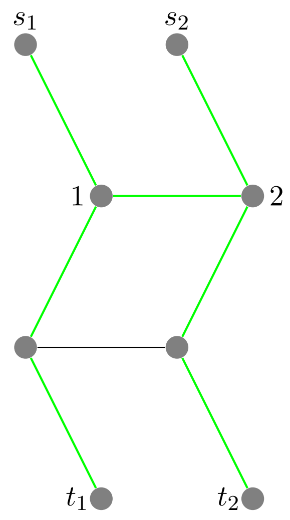
2.2. Local moves and post-processing

2.2.1. Swaps

Swaps are local moves in which we start with a feasible, cycle-free solution A. In other words, A is a feasible forest. Add some edges to create a single cycle and remove one (or more) edges



(a) A feasible solution (in green) for the SFP instance with the depicted graph, the demand-pairs $\{s_1, t_1\}$, $\{s_2, t_2\}$ and Euclidean distances.



(b) A feasible solution A' obtained from the solution in (a) by applying a connecting move and adding the edge $\{1, 2\}$. Notice, that $\phi(A') < \phi(A)$.

Figure 4: A connection move that decreases the potential of the forest. Note that the move does not make any improvement with respect to the total length of the forest.

of that cycle in order to get a solution A' that is on the one hand cycle-free and on the other hand feasible. We distinguish between three types of swaps: *edge-edge swaps*, *edge-set swaps* and *path-set swaps*.

Edge-edge swaps

The most basic move is the so called *edge-edge swap* which adds an edge e that has both end vertices in a tree T of the current solution A. This creates a cycle $\mathcal{C}(\text{e})$ in T. We remove an edge f from the cycle $\mathcal{C}(\text{e})$ to obtain a feasible, cycle-free solution A'.
We denote this move by *edge-edge swap(e,f)*.

Comments

- Since we choose $f \in \mathcal{C}(e)$, we can also choose $f = e$ which implies $A' = A$. This means that the solution A itself is part of its own neighbourhood but it is obvious that we do not obtain an improvement by considering such a move.
- An *edge-edge swap* does not change the number of connected components of the solution, since we only add one edge to one single component and remove one edge from the created cycle.
- Since the number of connected components does not change, the *width*-part in the potential also does not change. This means that $\phi(A) - \phi(A') = d(A) - d(A')$. Hence, removing an edge of highest length in the cycle $\mathcal{C}(e)$ leads to the best possible outcome with the added edge e.
- The distances that are used for the *width*'s of each connected component are the shortest path distances in the original graph G and not the shortest path distances in the graph $G[A]$ induced by the current solution A.
- It will be shown that there are only polynomially many *edge-edge swaps* when analysing the *edge-set swap* in the next paragraph.

Example 2.2. *(Edge-edge swap)*

Consider the following instance I with the graph given below, assume the lengths to be Euclidean distances[3] and three source-sink pairs $\{s_i, t_i\}$ for $1 \le i \le 3$.

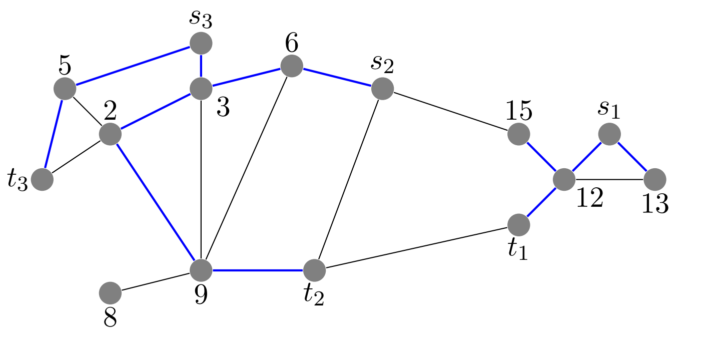


Figure 5: Instance I with current solution A in blue.

The shortest path distances between the demand-pairs in the underlying graph G are given as follows:

$dist_d(s_1, t_1) = 2.83$	Shortest path: $s_1 \to 12 \to t_1$
$dist_d(s_2, t_2) = 4.27$	Shortest path: $s_2 \to t_2$
$dist_d(s_3, t_3) = 5.04$	Shortest path: $s_3 \to 3 \to 2 \to t_3$

Hence, the *width* of the left component of A in Figure 5 is equal to $dist_d(s_3, t_3) = 5.04$ and the *width* of the right component of A is equal to $dist_d(s_1, t_1) = 2.83$. Notice that $d(A) = 24.35$ and $\phi(A) = 24.35 + 5.04 + 2.83 = 32.21$.

Let's assume we add the edge $e = \{t_3, 2\}$ in the left tree of the current solution A. This creates a unique cycle $\mathcal{C}(e) = t_3 - 5 - s_3 - 3 - 2 - t_3$. We can remove any edge of this cycle, let's choose the edge $f = \{5, s_3\}$. We obtain a cycle-free, feasible solution A' depicted in Figure 7.

The new solution has a lower total length and also the potential decreases by the same amount. We obtain $d(A') = 22.99$ and $\phi(A') = 22.99 + 5.04 + 2.83 = 30.85$.

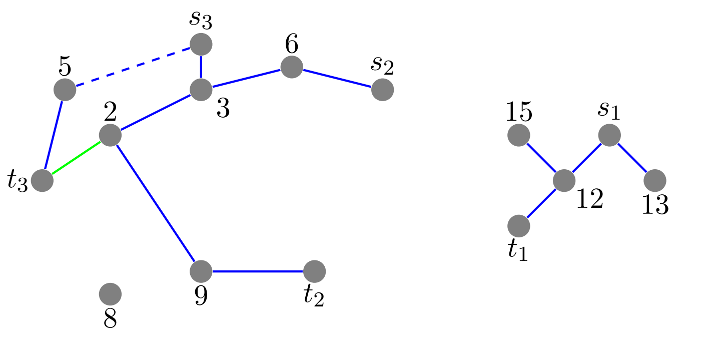


Figure 6: Current solution A in blue, new edge in green and dashed edge get removed.

[3]We do not list up all distances, since it is not necessary for an illustrating example

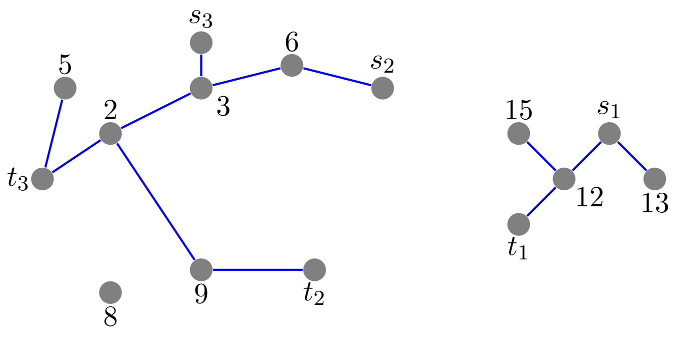


Figure 7: The new solution A'.

Edge-set swaps

The *edge-set swap* is a generalization of the *edge-edge swap* as follows: Add an edge e into a single tree T of the current solution A. This results in a cycle $\mathcal{C}$(e) in T. Then remove a subset S of edges from this cycle $\mathcal{C}$(e) so as to obtain a feasible, cycle-free solution A'.
We denote this move by *edge-set-swap(e,S)*.

In general, the subset S that can be removed from $\mathcal{C}$(e) is not necessarily unique. If we fix some edge $f \in \mathcal{C}(e)$, then there is a unique inclusion-maximal set $R(e, f) \subseteq \mathcal{C}(e)$ of edges that can be removed from $\mathcal{C}$(e) together with f without destroying the feasibility of the solution. Hence, $R(e, f)$ contains f and all edges on $\mathcal{C}$(e) that can be removed in $A \cup \{e\} \setminus \{f\}$ without destroying feasibility. Notice that we can remove any subset $S \subseteq R(e, f)$ and obtain a feasible, cycle-free solution. Clearly, if the local search is applied with respect to the total length of the solution, we would always remove the whole set $R(e, f)$ since the length of any edge is non-negative. However, since the local search is applied with respect to the potential ϕ, removing only a subset S may lead to a better solution than removing $R(e, f)$.

Let's assume $R(e, f) = \{e_1, \dots, e_l\}$ where the edges are in the order of their appearance in $\mathcal{C}$(e). We only consider swaps where S consists of consecutive edges in this order. This means that $S = \{e_i, \dots, e_j\}$ for some $1 \le i < j \le l$. If we would consider all subsets $S \subseteq R(e, f)$, this may lead to a "too" big neighbourhood[4].

Since T is a tree on at most n vertices, there are $\mathcal{O}(n^2)$ choices for the edge e and $\mathcal{O}(n)$ choices for the edge f (the cycle $\mathcal{C}$(e) can consist of at most n vertices). Therefore, also $1 \le i < j \le l \le n$ and hence there are $\mathcal{O}(n^2)$ consecutive sets $S \subseteq R(e, f)$.
Summing up, we get the following number of choices:

1:	Select an edge e that should be added to a tree T of A	$\mathcal{O}(n^2)$
2:	Select an edge $f \in \mathcal{C}(e)$	$\mathcal{O}(n)$
3:	Select a connected subset $S \subseteq R(e, f)$	$\mathcal{O}(n^2)$
	Total number of choices	$\mathcal{O}(n^5)$

This shows that the number of possible *edge-set swaps* to be applied to a feasible solution is polynomial. Thus this type of edge-set swap results in a polynomial-sized neighbourhood.

[4]Indeed, consider an instance with a complete graph on the vertices $1, \dots, n$ with one single demand-pair $\{1, n\}$. A feasible solution would be the set of edges $\{\{1, 2\}, \dots, \{n-1, n\}\}$. Adding the edge $e = \{n, 1\}$ gives a cycle $\mathcal{C}$ of n edges. Let's fix $f \in \{\{1, 2\}, \dots, \{n-1, n\}\}$ to be removed from $\mathcal{C}$. It's easy to see that $R(e, f) = \{\{1, 2\}, \dots, \{n-1, n\}\}$ of size $n-1$. Hence, there would be 2^{n-1} possibilities to choose a subset S.

Comments

- Since we choose $S \subseteq \mathcal{C}(e)$, we can also choose $S = \{e\}$ which implies $A' = A$.
- *edge-edge swap* is a special case of *edge-set swap* with $S = \{f\}$.Thus *edge-set swap*-optimality implies *edge-edge swap*-optimality.
- This also implies that there are only polynomially many *edge-edge swaps*.
- An *edge-set swap* may increase the number of connected components of the solution.
- It is not straightforward to determine the set $R(e, f)$ for given edges e and f.

Example 2.3. *(Edge-set swap)*

Consider the instance from Figure 5 and assume that the edge $e = \{s_2, t_2\}$ is added to the current solution A that is also shown in Figure 5. This creates a unique cycle $\mathcal{C}(e) = s_2 - t_2 - 9 - 2 - 3 - 6 - s_2$. Assume we fix the edge $f = \{2, 9\}$ that should be removed from the cycle.

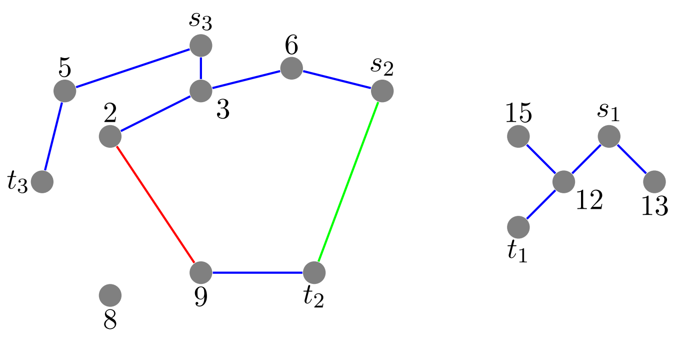


Figure 8: Current solution A with new edge e in green and selected edge f in red.

We can now determine the set of edges $R(e, f)$ that can be removed (together with f) in order to get a new feasible, cycle-free solution A'. It's easy to see that
$R(e, f) = \{\{t_2, 9\}, f, \{2, 3\}, \{3, 6\}, \{6, s_2\}\}$, since removing any subset $S \subseteq R(e, f)$ does not violate the feasibility of the solution.

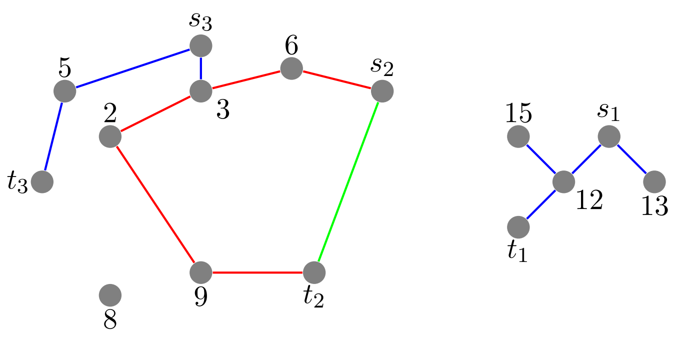


Figure 9: New edge e in green and the set $R(e, f)$ in red.

In general, the set $R(e, f)$ can be determined as follows: For edges e and f contained within one single tree T, compute shortest paths in $A \setminus \{f\} \cup \{e\}$ between any demand-pair that is also contained in T. This results in a set of paths $\mathcal{P} = \{P_1, \dots, P_r\}$. The set $R(e, f)$ contains all edges of $\mathcal{C}(e)$, that are not part of any path $P_i \in \mathcal{P}$, i.e.

$$R(e, f) = \mathcal{C}(e) \setminus \bigcup_{i=1}^{r} E(P_i)$$

The number of components in the solution A' obtained after applying an *edge-set swap(e,S)* depends on the choice of $S \subseteq R(e, f)$. Clearly also the potential $\phi(A')$ depends on the choice of S. For example, by deleting $S = \{f, \{2, 3\}, \{9, t_2\}\}$, we do not change the number of connected components, but by deleting $S = R(e, f)$ we increase the number by one.

By doing all calculations, we get the following values:

$$\begin{aligned}
d(A) &= 24.35 & \phi(A) &= 24.35 + 5.04 + 2.83 &&= 32.21\\
d(A') &= 20.28 & \phi(A') &= 20.28 + 5.04 + 2.83 &&= 28.14\\
d(A'') &= 16.15 & \phi(A'') &= 16.15 + 5.04 + 4.27 + 2.83 &&= 28.29
\end{aligned}$$

We can see that both new solutions have a smaller total length and also a smaller potential than the starting solution. Observe that $d(A'') < d(A')$ while $\phi(A'') > \phi(A')$. In this case, deleting less edges to obtain A' is a better choice since the number of connected components does not increase and hence the width of the solution-forest remains unchanged.

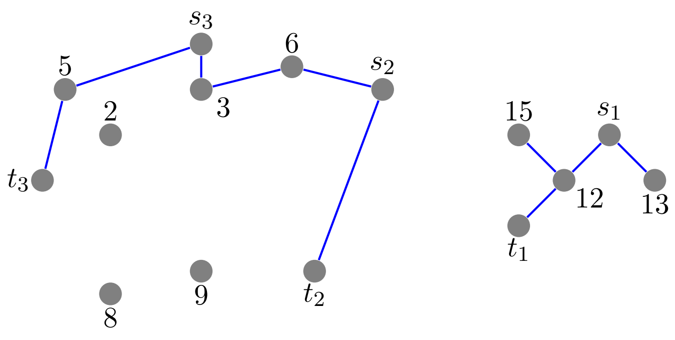


Figure 10: New feasible solution A' if $S = \{f, \{2, 3\}, \{9, t_2\}\}$.

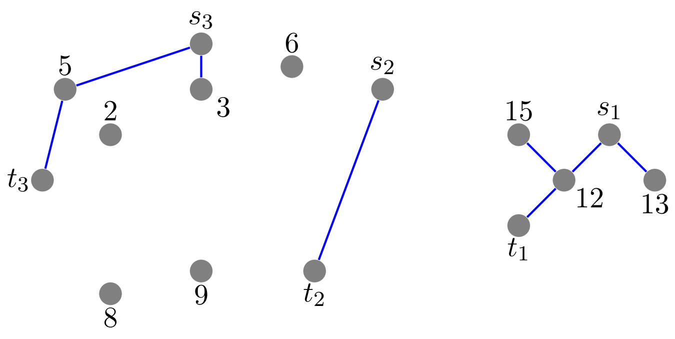


Figure 11: New feasible solution A'' if $S = R(e, f)$.

Path-set swap

Also *edge-set swap* can be generalized: We pick two vertices u, v lying in some tree T of the current feasible, cycle-free solution A and determine a shortest path $\mathcal{P}$ between them in the graph G', obtained as follows from the original graph G. Let $T, c_1, \ldots, c_k$ be the connected components of A. Each c_i is shrunk to a single node $V_{c_i}, 1 \leq i \leq k$ and $V(G') = V(T) \cup \{V_{c_i} : 1 \leq i \leq k\}$. The edge set of G' is obtained from the edge set of G by removing all edges from T as well as edges for which at least one endpoint has been shrunk. For each $\{s, t\} \in E(G)$ with $s \in V(C_i)$ and/or $t \in V(C_j)$ for some $1 \leq i, j \leq k$ with $i \neq j$, add an edge $\{V_{c_i}, V_{c_j}\}$, $\{s, V_{c_j}\}$ or $\{V_{c_i}, t\}$ in G', respectively.

Adding the corresponding edges of P to A creates a cycle in A that may connect two or more components of A. We imagine the path $\mathcal{P}$ as an "virtual" edge $\{u, v\}$, which we denote by $\{u, v\}_{\mathcal{V}}$, that is added to T and then remove a subset of consecutive edges from some $R(\{u, v\}_{\mathcal{V}}, f) \subseteq \mathcal{C}(\{u, v\}_{\mathcal{V}}) \subseteq E(T)$. Such a move can be seen as an *edge-set swap* with the "virtual" edge $\{u, v\}_{\mathcal{V}}$. We denote this move by *path-set swap(u,v,S)*.

Similar as in the case of the edge-set swap the cardinality of the neighbourhood generated by *path-set swap* is of size $\mathcal{O}(n^5)$:

1:	Select two vertices u and v in a tree T of A	$\mathcal{O}(n^2)$
2:	Select an edge $f \in \mathcal{C}(\{u, v\})$	$\mathcal{O}(n)$
3:	Select a connected subset $S \subseteq R(\{u, v\}, f)$	$\mathcal{O}(n^2)$
	Total number of choices	$\mathcal{O}(n^5)$

Comments:

- The current solution A may already contain the edge $\{u, v\}$, but due to shrinking, the shortest $u - v$ path in G' may be even shorter than this edge. In this case, the *path-set swap* move would add this shortest $u - v$ path to A and result in a cycle $\mathcal{C}(\{u, v\}_{\mathcal{V}})$ which consists of the (real) edge $\{u, v\}$ and the virtual edge $\{u, v\}_{\mathcal{V}}$. Then the only possible choice for a set of edges to be deleted is the original edge $\{u, v\}$.
- If the shortest $u - v$ path is the edge $\{u, v\}$, then *path-set swap(u,v,S)* is equivalent to *edge-set swap($\{u, v\}$,S)*.
- The created cycle $\mathcal{C}(\{u, v\}_{\mathcal{V}})$ may contain edges from different components, nevertheless, only edges from the tree T can be removed.
- A *path-set swap* may increase or decrease the number of connected components.

Example 2.4. *(Path-set swap)*

Also for this local move, recall the example from before with a current solution A:

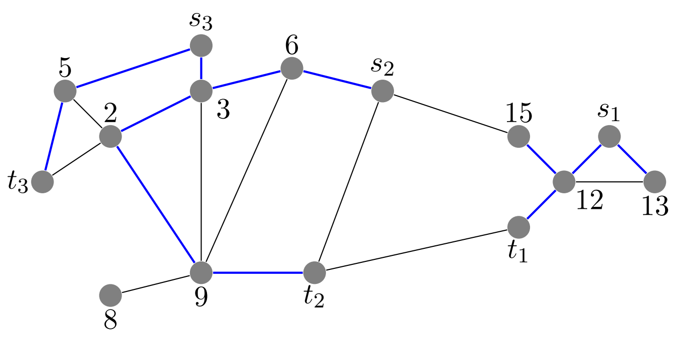


Figure 12: Instance I with current solution A in blue.

Let's choose $u = s_2$ and $v = t_2$ (of course we can also choose non-terminals!), which are part of the left component. The graph, in which we search for a shortest $u - v$ path looks as follows:

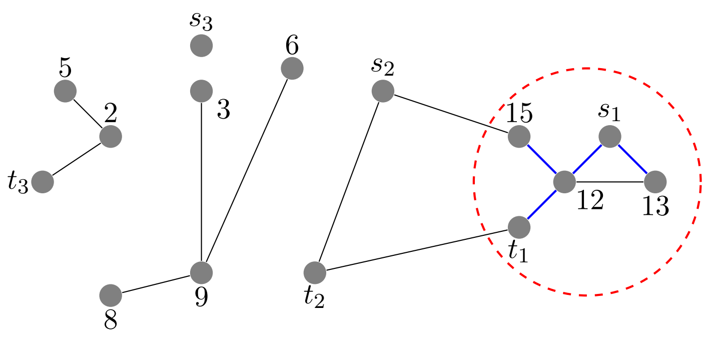


Figure 13: Graph in which we search for a shortest $u - v$ path.

There are two possible paths from u to v in this graph:
The path $s_2 \to t_2$ which is the direct edge of length ≈ 4.27 and the path $s_2 \to 15 \to 12 \to t_1 \to t_2$ (where the edges $\{15, 12\}$ and $\{12, t_1\}$ have length zero) of length ≈ 7.77. Hence, the shortest $u - v$ path is given by the direct edge $\{s_2, t_2\}$. Adding $\{s_2, t_2\}$ to the right connected component in A results in the same situation as in the *edge-set swap* discussed in Example 2.3 and represented in Figure 8.

Now consider an alternative feasible solution $\bar{A}$ depicted in Figure 14. Choose again $u = s_2$ and $v = t_2$. The graph G' in which we search for a shortest $u - v$ path is depicted in Figure 15.

In this case, there is a unique shortest $u - v$ path in G', namely $s_2 \to 15 \to 12 \to t_1 \to t_2$ (where the edges $\{15, 12\}$ and $\{12, t_1\}$ have length zero) of length ≈ 7.77.
We add the path as a virtual edge $\{u, v\}_\mathcal{V}$ to the current solution which results in the graph shown in Figure 16 with the unique cycle $\mathcal{C}(\{u, v\}_\mathcal{V})$ that consists of the edge $\{u, v\}$ and the virtual edge $\{u, v\}_\mathcal{V}$.

We are now allowed to remove some edges from the cycle $\mathcal{C}(\{u, v\}_\mathcal{V})$ that are part of the original connected component T. There is only one such edge, namely the $\{u, v\} = \{s_2, t_2\}$.

From now on, we proceed exactly in the same way as we do in the *edge-set swap*. Thus $R(\{u,v\}_{\mathcal{V}}, \{u,v\}) = \{\{u,v\}\}$ and hence the only possible edge set to remove is $S = \{\{u,v\}\}$. This leads to the following new solution $\bar{A}'$, where instead of the virtual edge $\{u,v\}_{\mathcal{V}}$ the shortest path represented by this edge is added. Note that the number of connected components in $\bar{A}'$ has decreased by one as compared to the number of connected components of $\bar{A}$.

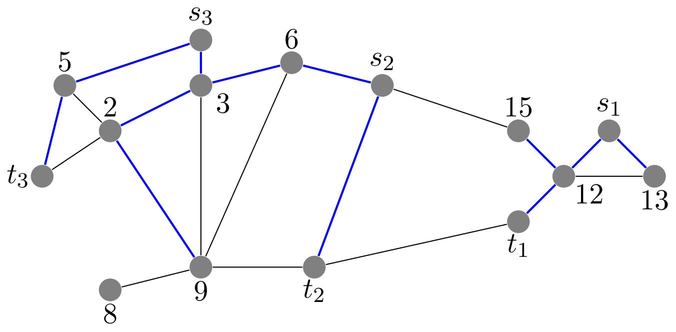


Figure 14: Instance I with a feasible solution $\bar{A}$ in blue.

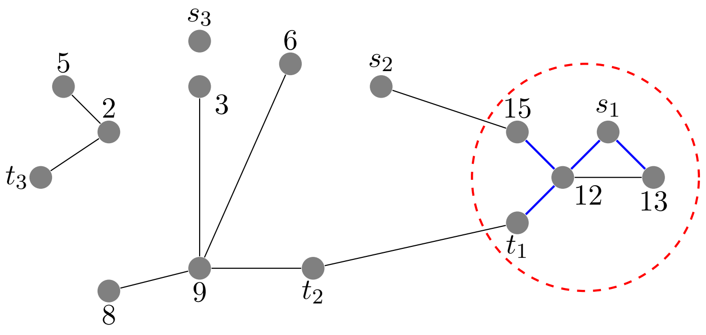


Figure 15: Graph G' in which we search for a shortest $u-v$ path.

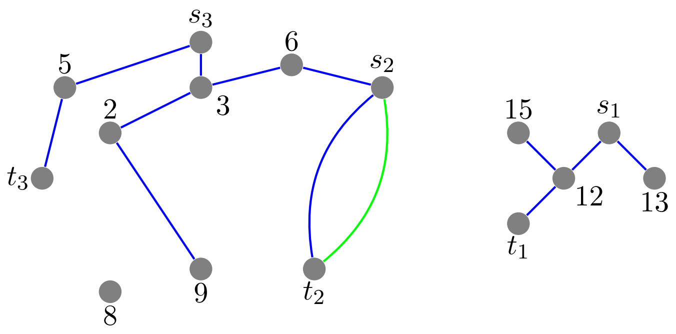


Figure 16: Adding the virtual edge $\{s_2, t_2\}$ in green produces a cycle in T.

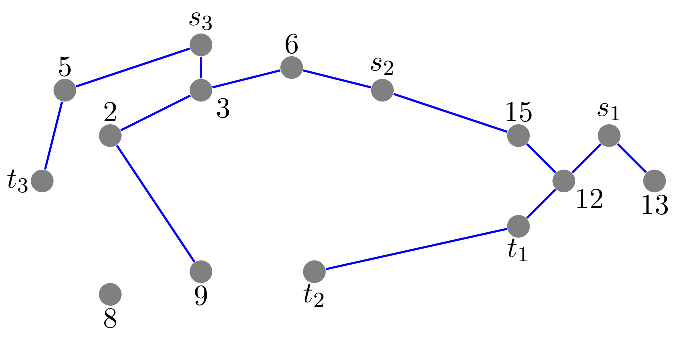


Figure 17: New solution $\tilde{A}'$ obtained by a *path-set swap*.

The following equalities hold:

$$\begin{aligned} d(\tilde{A}) &= 26.12 & \phi(\tilde{A}) &= 26.12 + 5.04 + 2.83 &&= 33.98 \\ d(\tilde{A}') &= 29.61 & \phi(\tilde{A}') &= 29.61 + 5.04 &&= 34.65 \end{aligned}$$

In this case, the *path-set swap* applied to $\bar{A}$ does not lead to a solution $\bar{A}'$ with a lower potential than $\bar{A}$.

2.2.2. Connecting moves

Connecting moves add a set of edges that connect some of the connected components of the current feasible solution A and result in a new feasible solution A' with a smaller number of connected components. Recall that the goal is to decrease the potential ϕ and not the total length of the solution. Since we are dealing with non-negative lengths of the edges, it is clear that $d(A') \geq d(A)$, however a *connecting* move may reduce the potential ϕ.

Formally, let G_A^{all} be the (multi)graph that results from the graph G after contracting all connected components of A in G, deleting self-loops and keeping parallel edges. A *connecting* move consists of picking a tree T in G_A^{all} and adding the corresponding edges to A.
We denote this move by *conn(T)*.

In contrast to the number of possible *swap*-moves, the number of possible *connecting*-moves can be very large and grows exponentially with n. Thus is due to the fact that the number of trees on n labelled vertices is n^{n-2} due to the well known Cayley's formula [St07].
During the analysis of the algorithm, we will show how to modify the *connecting* move in order to obtain a polynomial sized neighbourhood.

Example 2.5. *(Connecting move)*

Let's consider the same SFP instance as in the previous Examples 2.2, 2.3 and 2.4 with a feasible solution A depicted in Figure 18.

The corresponding graph G_A^{all} is shown in Figure 19.

Choose a tree T in the graph G_A^{all}, for example $T = \{\{T_3, 3\}, \{3, 6\}, \{6, T_2\}\}$, which is marked in green in Figure 20.

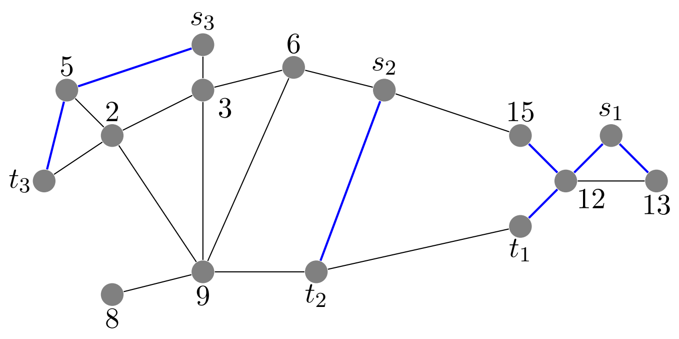


Figure 18: Instance I with current solution A in blue.

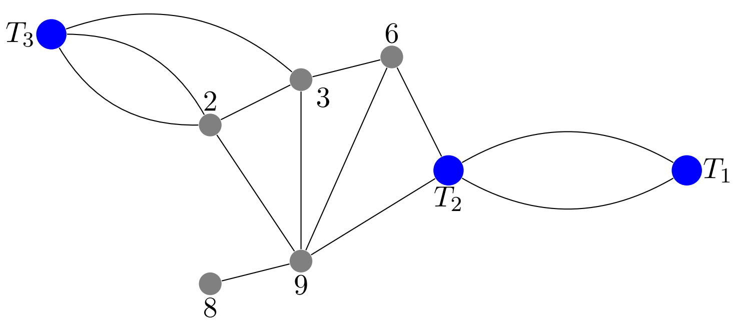


Figure 19: Corresponding graph G_A^{all}.

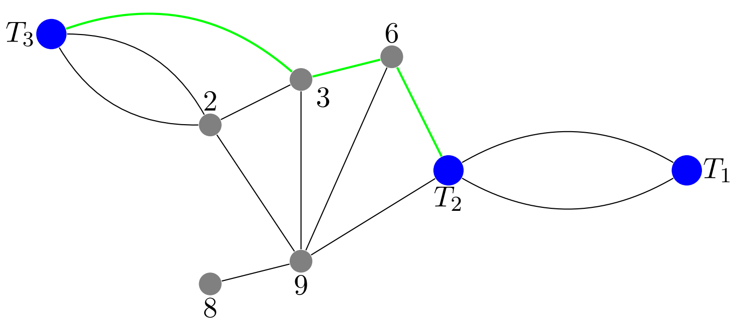


Figure 20: Selected tree T in G_A^{all} in green.

We now add the edges of the original graph corresponding to the edges of T to the current solution A:

$\{T_3, 3\}$	corresponds to	$\{s_3, 3\}$
$\{3, 6\}$	corresponds to	$\{3, 6\}$
$\{6, T_2\}$	corresponds to	$\{6, s_2\}$

The resulting solution A' is represented in Figure 21. Notice that the following equalities hold:

$$\begin{aligned} d(A) &= 15.15 & \phi(A) &= 15.15 + 5.04 + 4.27 + 2.83 &&= 27.29 \\ d(A') &= 20.28 & \phi(A') &= 20.28 + 5.04 + 2.83 &&= 28.15 \end{aligned}$$

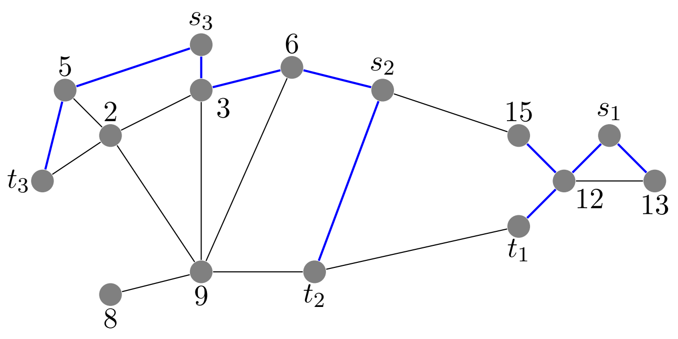


Figure 21: Instance I with obtained solution A' in blue.

We would not apply this connection move, since $\phi(A') > \phi(A)$. However, we see that the total length would increase by about 34 percent while the potential would increase only by about 3 percent.

2.2.3. Clean-up move

This move does not define a neighbourhood of some current feasible solution, it is just a post-processing move at the end to improve the quality of the obtained solution with respect to the total length d.

This move removes the unique inclusion - maximal edge set $S \subseteq A$ such that $A \setminus S$ is a feasible solution. In other words, we erase all unnecessary edges. This might increase the potential $\phi(A)$, but clearly does not increase $d(A)$.

The unique inclusion - maximal edge set $S \subseteq A$ mentioned above can be found as follows: The solution A holds a forest, hence there exists a unique path $\mathcal{P}_i$ between every demand-pair $\{s_i, t_i\}$. Therefore it suffices to take all these paths together and remove from A all edges that are not part of any path, i.e.

$$S = A \setminus \bigcup_{i=1}^{n_t} E(\mathcal{P}_i)$$

Comments

- Since a solution A is a forest, there is a unique path connecting the two terminals in each pair of terminals and hence this path is also the shortest path in $G[A]$. Hence, these paths can be computed in polynomial time by n_t shortest path computations or by doing an all-pair shortest path computation in $G[A]$. Both can be done in polynomial time.
- The *clean-up* move may increase the number of connected components of the solution.
- In contrast to the local moves, the goal of the *clean-up* move is to decrease the total length of the solution and not the potential ϕ. Since the lengths d are non-negative, the solution resulting after applying a *clean-up* move cannot have a larger total length than the previous solution.

Example 2.6. *(Clean-up move)*

Consider again the instance of the previous examples, e.g. Example 2.2 and assume that after applying all local moves we end up with the following solution A:

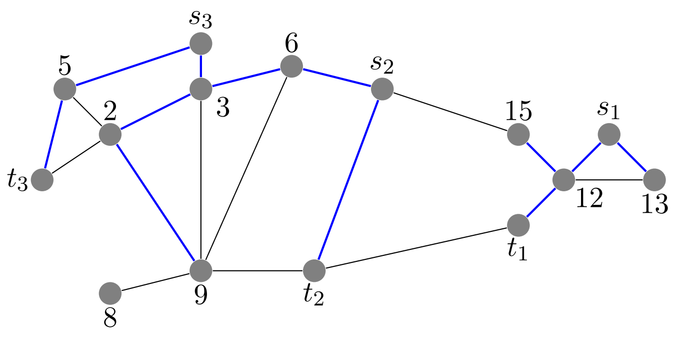


Figure 22: Instance I with a final solution A in blue.

Determine the shortest paths between the terminals for each demand-pair. They are represented in Figure 23 in green.

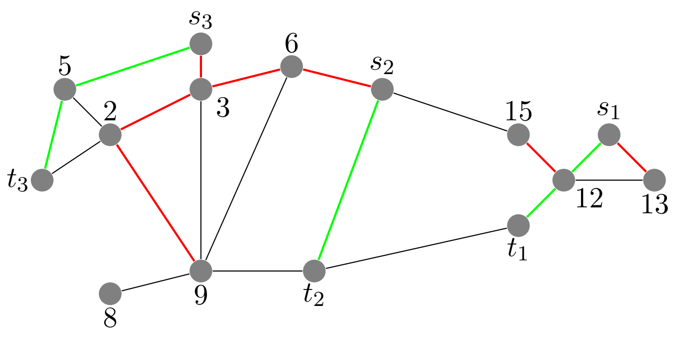


Figure 23: Shortest paths in green, the "rest" of the solution A in red.

The *clean-up* move deletes all red edges from the solution A to obtain the final solution A_f:

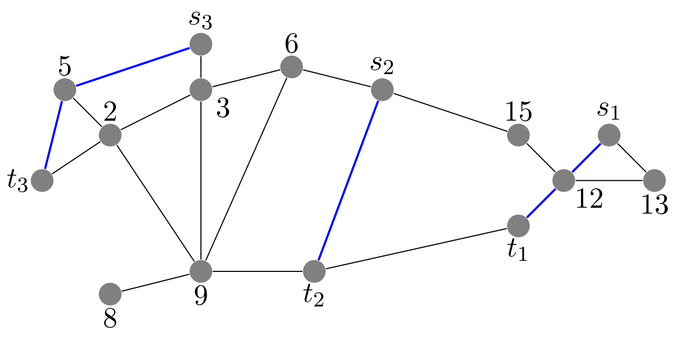


Figure 24: The graph with the solution A_f obtained after the *clean-up* move.

Notice that the following equalities hold:

$$\begin{aligned} d(A) &= 26.12 \qquad & \phi(A) &= 26.12 + 5.04 + 2.83 & &= 33.98 \\ d(A_f) &= 12.32 \qquad & \phi(A_f) &= 12.32 + 5.04 + 4.27 + 2.83 & &= 24.46 \end{aligned}$$

2.3. A generic local search algorithm

We give a formal description of the local search algorithm that is based on the discussed ideas from the previous section:

Algorithm 2 Local search, first approach

```
Require: An instance $I = (G, d, \mathcal{D})$ of the Steiner forest problem and a feasible solution $A$
    for $I$.
Ensure: A solution $A_f$ for the instance $I$.
    Let the neighbourhood $\mathcal{N}(A)$ be the set of all solutions that can be obtained by executing
    an edge-edge swap, edge-set swap, path-set swap or a connecting move on $A$.
    while there exists $A' \in \mathcal{N}(A)$ with $\phi(A') < \phi(A)$ do
        Set $A := A'$.
    end while
    Output the solution $A_f$ that results by applying the clean-up move to $A$.
```

Let's summarize what we already checked and what is still open:

- The number of possible *edge-edge swaps*, *edge-set swaps* and *path-set swaps* is polynomial, each swap can be carried out in polynomial time and it can be checked in polynomial time whether such a move improves the solution.
- The *clean-up* move can be done in polynomial time.
- The number of possible *connecting* moves is in general not polynomial, hence checking whether an improving *connecting* move exist can not be done in polynomial time (it is NP-hard in general).
- Due to the previous remark, the while loop and also the whole algorithm cannot be implemented in polynomial time.
- Up to this point, we have not discussed the quality of the solution produced by this algorithm.

3. Analysis of the local search algorithm: solution quality

In the following sections, we want to establish a way to proof the main theorem in this work:

Theorem 3.1.
There is a local search algorithm for the Steiner forest problem with a constant locality gap. It can be implemented to run in polynomial time.

It's not surprising, that we will consider the algorithm described above as a candidate for the claimed algorithm. During the proof, we have to modify the algorithm slightly, but the main ideas stay the same.

The content of the following sections as well as the ideas and proofs are mainly based on the work of Gross et al. [G17]. The goal is, to give an overview about the ideas and techniques that are used for proving Theorem 3.1. Technical details and lengthy or technical proves are shortened or in some cases not given here at all. They can be seen in Gross et al. [G17] in full length.

Consider a version of Algorithm 2 where only *edge-edge swaps* and *edge-set swaps* are allowed. In Section 3.1 we deal with the case where the local optimum A generated by Algorithm 2 is a tree and that the vertex set of the optimal solution F is the same as the vertex set of the local optimum, i.e. $V[A] = V[F]$.

First in Section 3.1.2 we obtain a bound for the total length of the solution A in the case that the function d is used instead of ϕ in the *while*-loop (see the pseudo code in Section 1.1.3). Then in Section 3.1.3 we give a bound for the case where the potential ϕ is used in the *while*-loop just as in the original version of the Algorithm. Section 3.1.8 gives a bound for the total length of the solution-forest.

In Section 3.2 we consider the general case, where the local optimum A is a forest, hence it can consist of more than one connected component. We show that there exists a constant K such that for any optimal solution forest F, there exists another solution forest F' with cost $c(F') \leq K \cdot c(F)$ having the property, that every connected component lies within some connected component of A. Then the results of Section 3.1 are applied to every connected component of A.

The proof uses the important fact that it suffices to deal with the metric Steiner forest problem as discussed in Section 3.2.2. Also the c-approximate connecting moves are introduced as an alternative of the already introduces *connecting*-moves. It will be shown that it is enough to use this type of connecting moves in order to get a polynomial constant - factor approximation algorithm.

3.1. Case I: The local optimum is a tree

What we want, is to bound the total length of a forest that is locally optimal with respect to the defined moves in Section 2.2. At the beginning, we consider a simpler case, where the local search algorithm outputs a tree as a solution.

Let $A, F \subseteq E$ be two feasible Steiner forests with respect to an instance $I = (G, d, \mathcal{D})$, where F is an optimal (or at least near optimal) solution and A is a feasible tree-solution computed by the algorithm.

For our purposes, we can assume that $V[A] = V[F]$[5].
Moreover, throughout this section, we assume that our solution A is a tree. Note that F does not need to be a tree.

3.1.1. Definitions and preliminaries

Notations 3.2.
Let $G = (V, E)$ be an undirected graph. For a set of vertices $W \subseteq V$ and an edge-set $F \subseteq E$, let $\delta_F(W)$ denote the edges of F leaving W, i.e having exactly one end vertex in W. For a forest $A \subseteq E$, we abbreviate $\delta_F(A) := \delta_F(V[A])$ For two disjoint vertex sets $U, W \subseteq V$, define $\delta_F(U : W) := \delta_F(U) \cap \delta_F(W)$ to be the set of edges that cross between U and W. For two forests $F_1, F_2 \subseteq E$ we abbreviate $\delta_F(F_1 : F_2) := \delta_F(V[F_1] : V[F_2])$.

[5]It will turn out, that in the general setting, the problem can be split up into partial problems that fulfil this equality. Hence we can make this assumption without loss of generality

Definition 3.3. *(compatible edges)*
Let $e = \{s, v_1\}, f = \{v_l, t\} \in A$ be two edges. Consider the unique path $P = s \to v_1 \to \cdots \to v_l \to t$ in $G[A]$ that connects e and f. We call s and t the end vertices and $v_1, \ldots, v_l$ the inner vertices of the path P. Let $G'(P, e, f) := (V[P] \setminus \{s, t\}, E[P] \setminus \{e, f\})$ be the graph containing the inner vertices of P and the edges of P except e and f. Moreover, let $G''(A, e, f) := (V[A], A \setminus \{e, f\})$ be the graph formed by the vertices $V[A]$ and the edge set $A \setminus \{e, f\}$. By construction, there exists a unique connected component $T_{e,f}$ of $G''(A, e, f)$ that contains $G'(P, e, f)$. We say that e and f are compatible with respect to F, if there are no F-edges leaving $T_{e,f}$, i.e. $\delta_F(T_{e,f}) = \emptyset$. In this case, we write $e \sim_{cp} f$.

Example 3.4. *(compatible / not compatible edges)*

Consider the two edge sets A and F shown in Figure 25. Note that the vertex sets coincide, i.e. $V[A] = V[F]$.

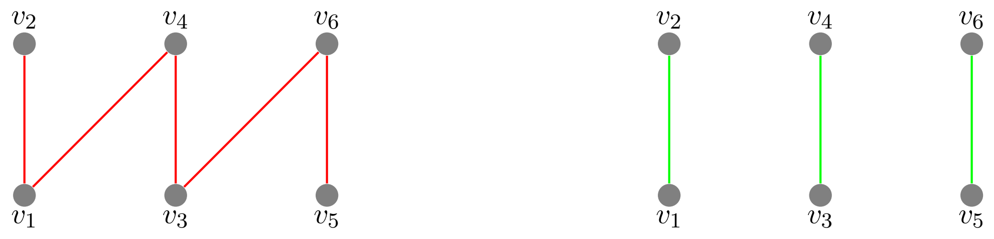


Figure 25: The edge set A on the left and the edge set F on the right.

Suppose that $e = \{v_1, v_4\}$ and $f = \{v_3, v_6\}$. The unique $e - f$ path is given by $P = v_1 \to v_4 \to v_3 \to v_6$. Hence

$$G'(P, e, f) = (\{v_3, v_4\}, \{\{v_3, v_4\}\}) \qquad G''(A, e, f) = (V[A], \{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5, v_6\}\})$$

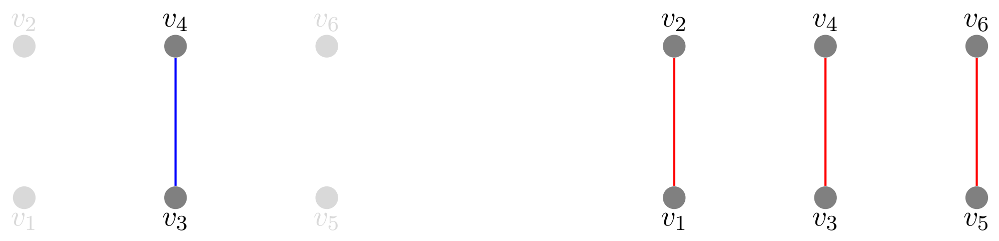


Figure 26: The graph $G'(P, e, f)$ on the left and $G''(A, e, f)$ on the right. Note that the vertices v_1, v_2, v_5 and v_6 are not part of $G'(P, e, f)$ but are depicted pale to make the example more handsome.

It is easy to see, that the $T_{e,f} = (\{v_3, v_4\}, \{\{v_3, v_4\}\})$. There are no F-edges leaving $T_{e,f}$ and hence e and f are compatible.

Now suppose that $e = \{v_1, v_2\}$ and $f = \{v_5, v_6\}$. The unique $e - f$ path is given by $P = v_2 \to v_1 \to v_4 \to v_3 \to v_6 \to v_5$. Hence

$$G'(P, e, f) = (\{v_1, v_3, v_4, v_6\}, \{\{v_1, v_4\}, \{v_3, v_4\}, \{v_3, v_6\}\})$$
$$G''(A, e, f) = (V[A], \{\{v_1, v_4\}, \{v_3, v_4\}, \{v_3, v_6\}\})$$

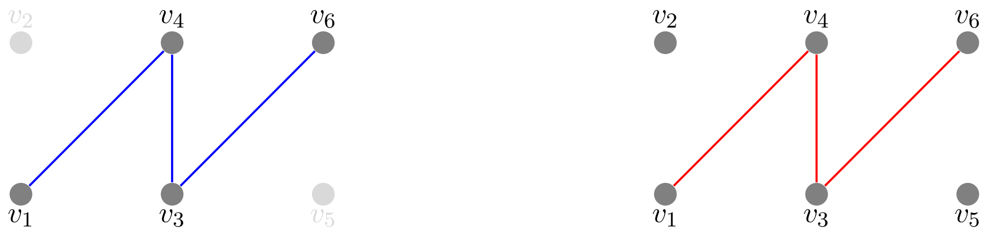


Figure 27: The graph $G'(P, e, f)$ on the left and $G''(A, e, f)$ on the right.

Note that the $T_{e,f} = (\{v_1, v_3, v_4, v_6\}, \{\{v_1, v_4\}, \{v_3, v_4\}, \{v_3, v_6\}\})$. Since there are F-edges leaving $T_{e,f}$, e and f are not compatible.

Lemma 3.5. *(Equivalence relation $\sim_{cp}$)*
Let $e, f \in A$ and $f, g \in A$ be two pairs of compatible edges. Then e and g are also compatible, which means that $\sim_{cp}$ is transitive. Hence, $\sim_{cp}$ is an equivalence relation.

Proof. The proof of transitivity splits up to three different cases that results by different position-constellations of the edges e, f and g in A. Since it uses only the definition of an compatible edge and basic ideas, we omit it. Moreover, the relation is reflexive and symmetric by definition summarizing $\sim_{cp}$ is an equivalence relation. □

Definition 3.6. *(Equivalence classes $\mathfrak{C}$)*
We denote the set of equivalence classes of the equivalence relation $\sim_{cp}$ by $\mathfrak{C}$.
For an equivalence class $S \in \mathfrak{C}$, denote by $l(S) := |S|$ the number of edges in S.

Definition 3.7. *(essential edge, safe edge)*
Let A be a feasible solution of an instance $I = (G, d, \mathcal{D})$ of the SFP. An edge $e \in A$ is called essential, if A is feasible, but $A \setminus \{e\}$ is infeasible. Let T_1, T_2 be the connected components of $A \setminus \{e\}$. Then e is called safe (with respect to F) if $\delta_F(T_1) = \delta_F(T_2) \neq \emptyset$, i.e. if at least one F-edge crosses between T_1 and T_2.

Important!
Remember that we assumed $V[A] = V[F]$. With that in mind, it is easy to see that any essential edge is safe, but the converse is not true in general: Safe edges can be essential or inessential. See the examples in Figures 28 and 29. Therefore, we can classify the edges of A into three distinct classes:

- safe essential edges
- safe inessential edges
- unsafe (and hence inessential) edges

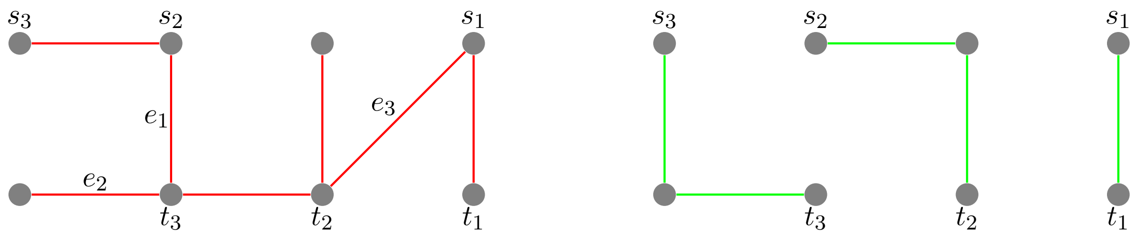
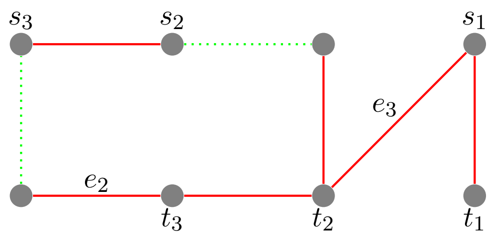
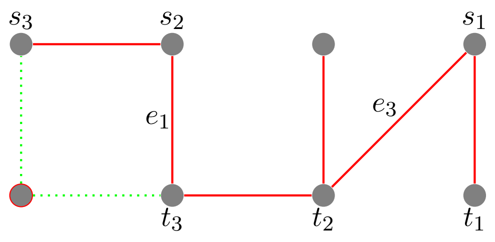


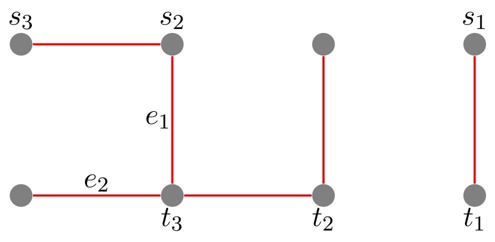
Figure 28: Consider this Euclidean instance with demand-pairs as depicted. On the left a solution A in red and on the right a solution F in green. Note that $V[A] = V[F]$.



Removing the edge e_1 destroyes the feasibility of A, hence e_1 is essential. We can see that there are two F-edges between the two arising components in $A\setminus\{e\}$, and hence the e_1 is safe w.r.t. F.



Removing the edge e_2 does not change the feasibility of A and hence e_2 is inessential. One of the resulting components consist only of the left end-point of e_2, that is connected by two F-edges to the other component. Therefore, e_2 is safe w.r.t. F.



Removing the edge e_3 does not change the feasibility of A and hence e_3 is inessential. Since there are no F-edges between the resulting two components of $A \setminus \{e\}$, e_3 is unsafe w.r.t. F.

Figure 29: The edges e_1, e_2, e_3 have different properties regarding essentialness and safeness

Lemma 3.8.
Let $e, f \in A$ be two compatible edges. Then e is safe w.r.t. F if and only if f is safe w.r.t. F.

Proof. Suppose that the edge $e = \{v_1, v_2\}$ is safe w.r.t. F, $f = \{v_{s-1}, v_s\}$ and let $P = v_1 \to v_2 \to \cdots \to v_{s-1} \to v_s$ be the unique $e-f$ path in A. Denote by $T_e, T_{e,f}$ and T_f the connected components of $A\setminus\{e, f\}$ that contain the vertex v_1, the vertices $v_2, \ldots, v_{s-1}$ and the vertex v_s, respectively. Note that $A\setminus\{e, f\}$ consists of exactly those three components. We need to show that f is safe w.r.t. F, i.e. that there is at least one F-edge that leaves T_f, meaning $\delta_F(T_f) \neq \emptyset$. The compatibility of e and f implies that $\delta_F(T_e : T_{e,f}), \delta_F(T_f : T_{e,f}) \subseteq \delta_F(T_{e,f}) = \emptyset$. Since e is safe w.r.t. F, we have that $\delta_F(T_e) = \delta_F(T_e : T_{e,f}) \cup \delta_F(T_e : T_f) \neq \emptyset$ which implies that $\delta_F(T_e : T_f) \neq \emptyset$. But this means that $\delta_F(T_f) = \delta_F(T_f : T_{e,f}) \cup \delta_F(T_f : T_e) \neq \emptyset$, i.e. that f is safe. □

Lemma 3.9.
Let $e, f \in A$ be two unsafe edges w.r.t. F. Then e and f are compatible.

Proof. Suppose that $e = \{v_1, v_2\}$ and $f = \{v_{s-1}, v_s\}$. Let $P = v_1 \to v_2 \to \cdots \to v_{s-1} \to v_s$ be the unique $e - f$ path in A. Denote by $T_e, T_{e,f}$ and T_f the connected components of $A \setminus \{e, f\}$ that contain the vertex v_1, the vertices $v_2, \ldots, v_{s-1}$ and the vertex v_s, respectively. By assumption, e and f are unsafe w.r.t. F, i.e. $\delta_F(T_e) = \emptyset$ and $\delta_F(T_f) = \emptyset$. We need to show that $\delta_F(T_{e,f}) = \delta_F(T_e : T_{e,f}) \cup \delta_F(T_f : T_{e,f}) = \emptyset$.
Note that $\delta_F(T_e : T_{e,f}) \subseteq \delta_F(T_e)$ and $\delta_F(T_f : T_{e,f}) \subseteq \delta_F(T_f)$, hence $\delta_F(T_{e,f}) = \emptyset$. □

Comment
Note that with the settings as in Lemma 3.8 or Lemma 3.9, equations like

$$\delta_F(T_e) = \delta_F(T_e : T_{e,f}) \cup \delta_F(T_e : T_f) \qquad \delta_F(T_{e,f}) = \delta_F(T_e : T_{e,f}) \cup \delta_F(T_f : T_{e,f})$$

only hold true since $V[A] = V[F]$. This is not true in the general case!

Summarizing we get the following lemma:

Lemma 3.10. *(Summary)*
Let $e, f \in A$ be two compatible edges. Then

1. *e is essential if and only if f is essential.*
2. *e is safe if and only if f is safe.*
3. *If two edges e and f are unsafe, then they are compatible.*
4. *The set $S_u := \{e \in A \mid e \text{ is } unsafe\}$ forms an equivalence class of $\sim_{cp}$.*
5. *If $S \in \mathfrak{C}$ is an equivalence class of $\sim_{cp}$, then either all edges $e \in S$ are essential or none.*
6. *If $S \in \mathfrak{C} \neq S_u$ is an equivalence class of $\sim_{cp}$, then all edges in S are safe.*

Proof. We only have to show (1), the other statements follow from Lemmas 3.8 and 3.9. The second is Lemma 3.8, the third holds true by Lemma 3.9. Points 4 to 6 are direct implications by the former and the fact that $\sim_{cp}$ is an equivalence relation, which was proven in Lemma 3.5.

To show the first point, let $P = v_1 \to v_2 \to \cdots \to v_{s-1} \to v_s$ be the unique $e - f$ path in A where $e = \{v_1, v_2\}$ and $f = \{v_{s-1}, v_s\}$. Denote by $T_e, T_{e,f}$ and T_f the connected components of A that contain the vertex v_1, the vertices $v_2, \ldots, v_{s-1}$ and the vertex v_s, respectively. Suppose that e is essential. We show that there exists a demand-pair s, t with $s \in T_e \cup T_{e,f}$ and $t \in T_f$. This would mean that removing the edge f makes $A \setminus \{f\}$ infeasible and hence the edge f is essential.

Since e is essential, there exists a demand-pair s', t' with $s' \in T_e$ and $t' \in T_{e,f} \cup T_f$. By the compatibility of e and f, we know that there are no F-edges leaving $T_{e,f}$. If $t' \in T_{e,f}$, then the forest F would be infeasible, hence $t' \in T_f$. We set $s = s'$ and $t = t'$ and obtain a pair that yields the essentialness of the edge f. □

Lemma 3.11.
Let $K := \{e_1, \ldots, e_l\} \subseteq A$ be a set of pairwise compatible edges of cardinality at least two. Suppose that one and therefore all edges in K are safe.
Then there exists a path $P \subseteq A$ with $K \subseteq P$.

Idea of the proof. For two edges $e, f \in K$ let $P_{e,f}$ be the unique path in A starting with edge e and ending with edge f. Define $R := \bigcup_{e,f \in K} P_{e,f}$. It is easy to see that $K \subseteq R \subseteq A$ and that R is a tree whose leaves are incident to edges of K. We assume for a contradiction that R is not a path, i.e. that there is some vertex with at degree at least three. This will finally contradict the safeness of an edge of K. □

Lemma 3.12.
Let $S \in \mathfrak{C} \setminus \{S_u\}$ be an equivalence class of safe edges. Let $f \in F$ be an edge.

1. *For any $K \subseteq S$ there is a path $P \subseteq A$ such that $K \subseteq \mathcal{P}$.*
2. *Let $P \subseteq A$ be the unique inclusion minimal path containing S and let $T_0, \ldots, T_l$ be the components of $A \setminus S$ in order they are traversed by P. Then either $f \in \delta_F(T_0 : T_l)$ or there exists $i \in \{0, \ldots, l\}$ such that T_i contains both endpoints of f.*
3. *Let C be the unique cycle in $A \cup \{f\}$. Then $S \subseteq C$ or $S \cap C = \emptyset$.*

Proof. The first follows from Lemma 3.11.
For the second part, denote the edges of S by $e_1, \ldots, e_l$ such that e_i is the edge between T_{i-1} and T_i for all $i \in \{1, \ldots, l\}$. Since all edges e_i, e_j for $i \neq j \in \{1, \ldots, l\}$ are pairwise compatible, there can not be an F-edge crossing between T_i and T_j for any $i \neq j \in \{1, \ldots, l\}$. Also there can not be an F edge crossing between T_0 and T_i for some $i \in \{1, \ldots, l-1\}$, since e_1 and e_{i+1} are compatible. The only possibilities that remain are that either both endpoints of f lie within some component T_i or that $f \in \delta_F(T_0 : T_l)$.
The third point follows now by the second corresponding to the two cases for f. □

Lemma 3.13.
Let $S \in \mathfrak{C}$ be an equivalence class of $\sim_{cp}$ and let $e \in S$. Moreover, let $f \in F$ such that $A \setminus \{e\} \cup \{f\}$ is feasible. Then $A \setminus S' \cup \{f\}$ is feasible for all subsets $S' \subseteq S$ and especially $A \setminus S \cup \{f\}$ is feasible.

Proof. First case: If e is inessential, then all edges in S are inessential by Lemma 3.10 and hence $A \setminus S$ is feasible even without adding f and we are done.

Second case: If e is essential, then by Lemma 3.10 all edges of S are essential and hence safe. We can now apply the second part of Lemma 3.12: Let $T_0, \ldots, T_l$ be the connected components of $A \setminus S$ in the order they are traversed by the path P that is containing S. We get that $\delta_F(T_i) = \emptyset$ for $i \in \{1, \ldots, l-1\}$.

Therefore, every demand-pair $\{s, t\} \in \mathcal{D}$ is either contained in one single T_i or w.l.o.g. $s \in T_0$ and $t \in T_l$. Removing the set S therefore only disconnects the pairs of the second type. We also know that either $f \in \delta_F(T_0 : T_l)$ or both its endpoints lie in the same connected component T_i. If $f \in \delta_F(T_0 : T_l)$, then $A \setminus \{e\} \cup \{f\}$ is feasible since $T_0 \cup T_l$ is a connected component.

If both endpoints of f lie in the same component, then the connected components of $A \setminus \{e\} \cup \{f\}$ are the same as the connected components of $A \setminus \{e\}$ and by our assumption it means that $A \setminus \{e\}$ is feasible. But this is a contradiction to e being essential.
The "especially" part follows trivially with $S' = S$. □

3.1.2. Bounding the total length: first attempt

We want to give a bound for the total length in some special case: As above, we suppose that the solution A obtained by the algorithm is a tree and $V[A] = V[F]$, where F is an optimal or near-optimal solution. Lemma 3.12 helps us to proof the following theorem which then allows us to give an interim statement as a corollary. The proof of the theorem can be seen in [G17].

Theorem 3.14. *(Bipartite graph based charging)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and let A be a feasible tree-solution for I such that $V[A] = V[F]$. Let $\Delta : \mathfrak{C} \to \mathbb{R}$ be a cost

function, that assigns a cost to each equivalence class $S \in \mathfrak{C}$. Suppose that $\Delta(S) \leq d_f$ for all pairs $(S, f) \in \mathfrak{C} \setminus \{S_u\} \times F$ such that the cycle in $A \cup \{f\}$ contains S. Then,

$$\sum_{S \in \mathfrak{C} \setminus \{S_u\}} \Delta(S) \leq \frac{7}{2} \cdot \sum_{f \in F} d_f$$

Corollary 3.15. *(Bounds on d for a special case)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let OPT be an optimal solution (w.r.t. the total length). Let A be a feasible tree solution that does not contain inessential edges and $V[A] = V[OPT]$. If A is edge-set swap-optimal and hence also edge-edge swap-optimal with respect to OPT and d, then

$$d(A) \leq \frac{7}{2} \cdot d(OPT)$$

Proof. Since A contains no inessential edges, there can not be unsafe edges (remember that unsafe implies inessential), hence $S_u = \emptyset$. We set $\Delta(S) := \sum_{e \in S} d_e$ for all $S \in \mathfrak{C}$. Let $f \in OPT$ be an edge that closes a cycle $\mathcal{C}$ in A such that $V[C]$ contains S (cf. Lemma 3.12c). Then, $(A \cup \{f\}) \setminus \{e\}$ is feasible for any edge $e \in S \subseteq \mathcal{C}$, since we remove only one edge of the cycle $\mathcal{C}$, which means that the obtained solution is still a tree. Using Lemma 3.13 we obtain that $(A \cup \{f\}) \setminus S$ is also feasible. Therefore, we can consider the *edge-set swap* that adds the edge $f \in OPT$ and deletes S. By assumption, this move was not improving (with respect to d), since A is *edge-set swap*-optimal with respect to edges from OPT and d. Thus, $\Delta(S) = \sum_{e \in S} d_e \leq d_f$. We can now directly apply Theorem 3.14. In our setting, with $S_u = \emptyset$ and $F = OPT$, this yields

$$\sum_{S \in \mathfrak{C}} \Delta(S) \leq \frac{7}{2} \cdot \sum_{f \in OPT} d_f$$

With the definition of $\Delta(S)$ we get

$$d(A) = \sum_{e \in A} d_e = \sum_{S \in \mathfrak{C}} \sum_{e \in S} d_e = \sum_{S \in \mathfrak{C}} \Delta(S) \leq \frac{7}{2} \cdot \sum_{f \in OPT} d_f = \frac{7}{2} \cdot d(OPT)$$

which proves the claimed result. □

3.1.3. Bounding the potential ϕ

In the previous section, we considered the case, that our solution (of special form) is *edge-set swap*-optimal with respect to the optimal solution OPT and the distance function d. Now we want to consider the *edge-set swaps* with respect to the potential ϕ.

Since *edge-set swaps* may increase the number of connected components, and some of these "new" components may have large widths, *edge-set swaps* that are improving with respect to the lengths may not be improving any more. Handling this situation requires a more careful analysis, but we will still rely on some tools that we already established. As before, consider the case where the feasible solution A is a single tree and $V[A] = V[F]$.

We start with some notation:
Let $S \in \mathfrak{C} \setminus \{S_u\}$ be an equivalence class of safe edges that contain $l(S)$ edges. By the first part of Lemma 3.12, the edges of S lie on a path P. We write $S = \langle e_{S,1}, \ldots, e_{S,l(S)} \rangle$ if the edges lie in this ordering on P. With the edges of the form $e_{S,i} = \{w_{S,i-1}, v_{S,i}\}$, we get the following layout of the path:

$$w_{S,0} \xrightarrow{e_{S,1}} v_{S,1} \to \cdots \to w_{S,i-1} \xrightarrow{e_{S,i}} v_{S,i} \to \cdots \to w_{S,l(S)-1} \xrightarrow{e_{S,l(S)}} v_{S,l(S)}$$

When the context is clear, and that should be the case in this section, we drop some indices and write $e_1, \dots, e_{l(S)}$ and $v_1, \dots, v_{l(S)}, w_0, \dots, w_{l(S)-1}$ instead of the above used names. Hence we describe the path in this way:

$$w_0 \xrightarrow{e_1} v_1 \to \cdots \to w_1 \xrightarrow{e_2} v_2 \to \cdots \to w_{i-1} \xrightarrow{e_i} v_i \to \cdots \to w_{l(S)-1} \xrightarrow{e_{l(S)}} v_{l(S)}$$

Note that $w_i = v_i$ is possible, but always $w_{i-1} \neq v_i$, i.e. the edge e_i is not a self loop.

If we remove the set S, this yields a decomposition of A into $l(S) + 1$ connected components. Let $G_{S,i} = (V_{S,i}, E_{S,i})$ be the connected component that contains w_i and let $G_{S,l(S)} = (V_{S,l(S)}, E_{S,l(S)})$ be the connected component that contains $v_{l(S)}$. These components $G_{S,i}$ are subgraphs of G for all $1 \leq i \leq l(S)$. For convenience, we will identify each component with its edge set $E_{S,i}$.

We think of $E_{S,0}$ and $E_{S,l(S)}$ as the *outer* components of the path P and we see $E_{S,1}, \dots, E_{S,l(S)-1}$ as the *inner* components. It is easy to see that these components form vertex-disjoint subtrees of (the tree) A, as stated below.

Observation 3.16.
Let $S \in \mathfrak{C} \setminus \{S_u\}$ and $i \neq j \in \{1, \dots, l(S)\}$. Then $E_{S,i} \cap E_{S,j} = \emptyset$ and A is the disjoint union of the edge sets $E_{S,0}, \dots, E_{S,l(S)}$ and S.

Definition 3.17. *($\mathfrak{m}(S)$ and $\mathfrak{n}(S)$)*
Let the settings be as above. Let $\mathfrak{m}(S)$ and $\mathfrak{n}(S)$ be the index of the inner components with the largest and second-largest widths, respectively. We use the index m_{ind} from Definition 1.8 to break ties consistently, hence

$$\mathfrak{m}(S) := \operatorname*{arg\,max}_{i \in \{1,\dots,l(S)-1\}} m_{ind}(E_{S,i})$$

$$\mathfrak{n}(S) := \operatorname*{arg\,max}_{\substack{i \in \{1,\dots,l(S)-1\} \\ i \neq \mathfrak{m}(S)}} m_{ind}(E_{S,i})$$

Agreement
Without loss of generality, we assume that the orientation of the path P containing $S \in \mathfrak{C} \setminus \{S_u\}$ is such that $\mathfrak{m}(S) < \mathfrak{n}(S)$.

Definition 3.18. *(In_S and $In_{S'}$)*
Let the settings be as above. We set

$$In_S := \{1, \dots, l(S) - 1\}$$
$$In_{S'} := \{1, \dots, l(S) - 1\} \setminus \{\mathfrak{m}(S), \mathfrak{n}(S)\}$$

i.e. In_S are the indices of the inner components and $In_{S'}$ are the indices of the inner components excluding the vertices which corresponds to the components with the largest and second-largest width, respectively.

Let's have a look on the potential of the solution A which can be written as follows:

$$\begin{aligned}
\phi(A) &= w(A) + d(A) \\
&= w(A) + \sum_{e\in A} d_e \\
&= w(A) + \sum_{e\in S_u} d_e + \sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{e\in S} d_e \\
&= \Big(w(A) + \sum_{e\in S_u} d_e\Big) + \Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{i=1}^{l(S)} d_{e_i}\Big) \qquad (2) \\
&= \Big(w(A) + \sum_{e\in S_u} d_e\Big) + \Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{i=1}^{l(S)} d_{e_i}\Big) \\
&\quad - \Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{i\in In_S} w(E_{S,i})\Big) + \Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{i\in In_S} w(E_{S,i})\Big) \\
&= \underbrace{\Big(w(A) + \sum_{e\in S_u} d_e\Big)}_{\text{Term A}} + \underbrace{\Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i\in In_S} w(E_{S,i})\big)\Big)}_{\text{Term B}} + \underbrace{\Big(\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \sum_{i\in In_S} w(E_{S,i})\Big)}_{\text{Term C}}
\end{aligned}$$

The next goal is to bound the three Terms A, B and C. To do so, we will use some results which we do not prove here, since the proofs are in parts lengthy or of technical nature. All proofs can be seen in full length in Gross et al. [G17].

3.1.4. Bounding Term B

We use the following lemma to give a bound for Term B.

Lemma 3.19.
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and A be a feasible tree-solution for I satisfying $V[A] = V[F]$. Suppose that A is edge-set swap-optimal with respect to F and ϕ. Furthermore, let $S \in \mathfrak{C} \setminus \{S_u\}$ be an equivalence class of safe edges and let $f \in F$ be an edge that closes a cycle in A that contains S. Then

$$d_f \geq \frac{1}{3}\Big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i\in In_S} w(E_{S,i})\Big)$$

Corollary 3.20. *(A bound for Term B)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and A be a feasible tree-solution for I satisfying $V[A] = V[F]$. Suppose that A is edge-set swap-optimal (and hence also edge-edge swap-optimal) with respect to F and ϕ. Then

$$\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \Big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i\in In_S} w(E_{S,i})\Big) \leq 10.5 \cdot d(F)$$

Proof. For an equivalence class $S \in \mathfrak{C} \setminus \{S_u\}$, we set

$$\Delta(S) := \frac{1}{3}\Big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i\in In_S} w(E_{S,i})\Big)$$

Lemma 3.19 ensures that $\Delta(S) \le d_f$ for any pair $(S,f) \in \mathfrak{C}\setminus\{S_u\}\times F$ where f closes a cycle in A that contains S. Note, that the requirements of Theorem 3.14 are fulfilled, which implies that

$$\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \Delta(S) \le \frac{7}{2}\cdot d(F)$$

Plugging in the definition of $\Delta(S)$ yields

$$\sum_{S\in\mathfrak{C}\setminus\{S_u\}} \frac{1}{3}\Big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i\in In_S} w(E_{S,i})\Big) \le \frac{7}{2}\cdot d(F)$$

and multiplying the inequality by 3 gives the result. □

3.1.5. Bounding Term A

We need the following definition for the bound of Term A:

Definition 3.21. *(Removing swap)*
Let $I = (G,d,\mathcal{D})$ be an instance of the SFP with $G=(V,E)$ and A a feasible solution. Let $U \subseteq A$ be a subset of edges such that $A\setminus U$ is feasible. Removing U from A to obtain another feasible solution $A' := A\setminus U$ is called a removing swap with the set U.
Removing swap optimality (with respect to some objective function) means that there is no $U\subseteq A$ such that the resulting feasible solution A' yields a better objective function value.

Remember that (in our standard setting) S_u is the set of unsafe edges of the solution A with respect to an optimal or near optimal solution F. We already know that unsafe edges are inessential, hence they can be removed without destroying the feasibility of the solution. Therefore, a removing swap with the set S_u is always an option.

Lemma 3.22. *(A bound for Term A)*
Let $I=(G,d,\mathcal{D})$ be an instance of the SFP with $G=(V,E)$. Let F be a feasible solution and A be a feasible tree solution for I that is removing swap optimal and $V[A]=V[F]$. It holds that

$$w(A) + \sum_{e\in S_u} d_e \le w(F)$$

Proof. Suppose F consists of $cc(F)$ connected components $F_1,\dots,F_{cc(F)}$. Consider the removing swap that removes all edges in S_u. $A' := A\setminus S_u$ contains $|S_u|+1$ connected components $E_1,\dots,E_{|S_u|+1}$, let's suppose they are numbered such that their width's are non increasing, i.e. $w(E_1)\ge\dots\ge w(E_{|S_u|+1})$.

By definition of S_u, the edges of F do not connect any of those connected components, and with $V[A]=V[F]$ it follows that $cc(F)\ge |S_u|+1$. Furthermore, any component E_i contains a component F_j of the same width, i.e. $w(E_i)=w(F_j)$. Assume that the components $F_1,\dots,F_{cc(F)}$ are numbered such that $w(F_i)=w(E_i)$ for $i=1,\dots,|S_u|+1$.

Let's have a look at the potentials of A and A'. Remember that A is per assumption a tree.

$$\phi(A) = \sum_{e \in A} d_e + w(A) \qquad \phi(A') = \sum_{e \in A} d_e - \sum_{e \in S_u} d_e + \sum_{i=1}^{|S_u|+1} w(E_i)$$

Since A was removing swap optimal, this removing swap was not improving, hence

$$\sum_{e \in A} d_e + w(A) \leq \sum_{e \in A} d_e - \sum_{e \in S_u} d_e + \sum_{i=1}^{|S_u|+1} w(E_i) = \sum_{e \in A} d_e - \sum_{e \in S_u} d_e + \sum_{i=1}^{|S_u|+1} w(F_i)$$

and finally

$$w(A) + \sum_{e \in S_u} d_e \leq \sum_{i=1}^{|S_u|+1} w(F_i) \leq \sum_{i=1}^{cc(F)} w(F_i) = w(F)$$

□

3.1.6. Bounding Term C

To obtain a bound for Term C, we use two results for which we will not give a proof.
As always, the proofs can be seen in Gross et al. [G17]

Lemma 3.23.
Let $S, S' \in \mathfrak{C} \setminus \{S_u\}$ be two different equivalence classes, i.e. $S \neq S'$.
Then exactly one of the two following cases holds:

1. *$S' \subseteq E_{S,0} \cup E_{S,l(S)}$, i.e. S' lies in the outside of the path of S.*
2. *There exists $i \in \{1, \dots, l(S) - 1\}$ with $S' \subseteq E_{S,i}$.*

Definition 3.24. *(The function μ)*
In our standard setting, we set $\mu(S, i) := m_{ind}(E_{S,i})$ for all $S \in \mathcal{C} \setminus \{S_u\}$ and $i \in In_S$.

Lemma 3.25. *(μ is injective)*
Let $S, S' \in \mathfrak{C} \setminus \{S_u\}$, $i \in In_S$ and $i' \in In_{S'}$. Then

$$\mu(S, i) = \mu(S', i') \implies S = S' \text{ and } i = i'$$

i.e. μ is an injective function in S and i.

Comment
Lemma 3.23 is essential to give a compact proof for Lemma 3.25.
We use now Lemma 3.25 to give a bound for Term C.

Lemma 3.26. *(A bound for Term C)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and A be a feasible tree solution for I and denote by $E_{S,i}$ the i-th inner connected component on the path that contains $S \in \mathfrak{C} \setminus \{S_u\}$[6]. Then

[6]Remember that we assumed the orientation of the path such that $\mathfrak{m}(S) < \mathfrak{n}(S)$, i.e. the notation is well defined.

$$\sum_{S \in \mathfrak{C}\setminus\{S_u\}} \sum_{i \in In_S} w(E_{S,i}) \leq w(F)$$

Proof. Suppose F consists of $cc(F)$ connected components $F_1, \dots, F_{cc(F)}$. With Definition 1.8 we can write

$$w(E_{S,i}) = d_G(u_{m_{ind}(E_{S,i})}, \bar{u}_{m_{ind}(E_{S,i})}) = d_G(u_{\mu(S,i)}, \bar{u}_{\mu(S,i)})$$

for all $S \in \mathcal{C} \setminus \{S_u\}$ and $i \in In_S$.
Therefore, it holds that

$$\sum_{S \in \mathfrak{C}\setminus\{S_u\}} \sum_{i \in In_S} w(E_{S,i}) = \sum_{S \in \mathfrak{C}\setminus\{S_u\}} \sum_{i \in In_S} d_G(u_{\mu(S,i)}, \bar{u}_{\mu(S,i)}) \tag{3}$$

Let $\chi(S,i)$ denote the index of the connected component $F_{\chi(S,i)}$ which contains the demand-pair $\{u_{m_{ind}(E_{S,i})}, \bar{u}_{m_{ind}(E_{S,i})}\}$ in F.

Claim: χ is injective with respect to S and i.
Consider $S, S' \in \mathfrak{C}$ and $i \in In_S$, $i' \in In_{S'}$ such that $\chi(S,i) = \chi(S',i')$. By an compatibility argument, we know that $\delta_F(V_{S,i}) = \emptyset$ and $\delta_F(V_{S',i'}) = \emptyset$. Since $F_{\chi(S,i)}$ is connected, it follows that $V[F_{\chi(S,i)}] \subseteq V_{S,i} \cap V_{S',i'}$. This implies that $u_{\mu(S,i)}, \bar{u}_{\mu(S,i)} \in V_{S',i'}$ and also $u_{\mu(S',i')}, \bar{u}_{\mu(S',i')} \in V_{S,i}$. Hence, $m_{ind}(E_{S,i}) = m_{ind}(E_{S',i'})$ or in the context of the function μ this means $\mu(S,i) = \mu(S',i')$. Using Lemma 3.25, we obtain that $S = S'$ *and* $i = i'$ which proves the claim, i.e. the injectivity of the function μ gives the injectivity of the function χ.

Since $d_G(u_{\mu(S,i)}, \bar{u}_{\mu(S,i)}) \leq w(F_{\chi(S,i)})$, we can estimate the right hand side of (3) in the following way:

$$\sum_{S \in \mathfrak{C}\setminus\{S_u\}} \sum_{i \in In_S} d_G(u_{\mu(S,i)}, \bar{u}_{\mu(S,i)}) \leq \sum_{S \in \mathfrak{C}\setminus\{S_u\}} \sum_{i \in In_S} w(F_{\chi(S,i)}) \leq \sum_{i=1}^{cc(F)} w(F_i) = w(F)$$

The last inequality holds since χ is injective and its image is a subset of $\{1, \dots, cc(F)\}$. By summing over all $cc(F)$ connected components of F we may make the sum larger, since all width's are non-negative. $\square$

3.1.7. Wrapping up

Considering all these bounds and tools, we are now able to proof the main result of Section 3.1 and also some nice corollaries:

Theorem 3.27. *(Bound for $\phi(A)$, first version)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and let A be a feasible tree-solution for I such that $V[A] = V[F]$. Furthermore, suppose that A is edge-set swap-optimal with respect to F and ϕ. Then,

$$\phi(A) = d(A) + w(A) \leq w(A) + \sum_{e \in S_u} d_e + 10.5 \cdot d(F) + w(F)$$

In particular, we have

$$d(A) \leq \sum_{e \in S_u} d_e + 10.5 \cdot d(F) + w(F)$$

Proof. Writing $\phi(A)$ in the form as in (2) on Page 30, we obtain immediately the claimed bound by applying Corollary 3.20 and Lemma 3.26 to the Terms B and C:

$$\phi(A) = \Big(w(A) + \sum_{e \in S_u} d_e\Big) + \underbrace{\Big(\sum_{S \in \mathfrak{C} \setminus \{S_u\}} \big(\sum_{i=1}^{l(S)} d_{e_i} - \sum_{i \in In_S} w(E_{S,i})\big)\Big)}_{\leq 10.5 d(F) \text{ by Corollary } 3.20} + \underbrace{\Big(\sum_{S \in \mathfrak{C} \setminus \{S_u\}} \sum_{i \in In_S} w(E_{S,i})\Big)}_{\leq w(F) \text{ by Lemma } 3.26}$$

□

If we additionally apply Lemma 3.22, we get the following reformulation of Theorem 3.27:

Corollary 3.28. *(Bounds for $\phi(A)$, second version)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let F be a feasible solution and let A be a feasible tree-solution for I such that $V[A] = V[F]$. Furthermore, suppose that A is optimal with respect to edge-set swaps and removing swap with respect to F and ϕ. Then,

$$\phi(A) \leq 10.5 \cdot \phi(F)$$

Proof. Starting with the claimed bound of Theorem 3.27 and applying Lemma 3.22 leads almost directly to the claimed bound:

$$\begin{aligned} \phi(A) \leq & \underbrace{w(A) + \sum_{e \in S_u} d_e}_{\leq w(F) \text{ by Lemma } 3.22} + 10.5 \cdot d(F) + w(F) \\ & \leq 10.5 \cdot d(F) + 2 \cdot w(F) \leq 10.5 \cdot \big(d(F) + w(F)\big) \\ & = 10.5 \cdot \phi(F) \end{aligned}$$

□

3.1.8. Bounding the total length: second attempt

Having established a bound for the potential $\phi(A)$ for some feasible tree-solution, we obtain a bound for the total length.

Corollary 3.29. *(A Bound for d)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let A, F be two feasible Steiner forests for I with $V[A] = V[F]$ and A being a tree. Suppose that A is edge-set swap-optimal with respect to F and ϕ. Denote by A' the modified solution, where all inessential edges have been removed from A. Then

$$d(A') \leq 10.5 \cdot d(F) + w(F) \leq 11.5 \cdot d(F)$$

Proof. From Theorem 3.27 we get

$$d(A) \leq \sum_{e \in S_u} d_e + 10.5 \cdot d(F) + w(F)$$

Let S_{is} be the set of safe but inessential edges. We get

$$\begin{aligned}
d(A') &= d(A) - \sum_{e \in S_u} d_e - \sum_{e \in S_{is}} d_e \\
&\leq \sum_{e \in S_u} d_e + 10.5 \cdot d(F) + w(F) - \sum_{e \in S_u} d_e - \sum_{e \in S_{is}} d_e \\
&= 10.5 \cdot d(F) + w(F) - \sum_{e \in S_{is}} d_e \\
&\leq 11.5 \cdot d(F)
\end{aligned}$$

since $w(F) \leq d(F)$ and $\sum_{e \in S_{is}} d_e \geq 0$. $\square$

Comments
Corollary 3.29 gives a bound for the total length of the solution A in the case, that A has the following properties, which are not necessarily true for a general feasible solution:

- A is a tree
- $V[A] = V[F]$
- A is edge-set swap optimal with respect to F and ϕ

We have not yet considered the *path-set swap* and the *connecting*-move. Nevertheless, we will see that we do not need to consider these moves at all to obtain our desired results. While dropping some restrictions of the solution A may increase the bound, considering more local moves may allow a lower bound.

3.2. Case II: The local optimum is a forest

In the general case, both A and F can have multiple connected components. The first idea may be that we apply the results of Section 3.1 to each component individually. The problem is the following: For Theorem 3.27 and also for other results in that section, we assumed implicitly that there are no edges from F which cross between two connected components in A. This was clear since A was a tree and hence consisting of only one connected component. Now, where A can be a forest, this is not necessarily true any more. The main idea is to replace F-edges that cross between two connected components of A by some edges that lie within components of A in such a way that the (modified) solution F' stays feasible and its cost is only a constant multiple of the cost of F. Formally, we want to transform a pair (A, F) into a pair (A, F') such that F' has the desired properties mentioned above. This will then allow us to prove the first part of Theorem 3.1.

3.2.1. Changing the type of instance

In the previous section, we considered the general Steiner forest problem where $G = (V, E)$ was an arbitrary graph and the only restriction to the cost function $d : E \to \mathbb{R}_{\geq 0}$ was to be non-negative. Now, we have to change this setting at some points: We assume that $G = (V, E)$ is the complete graph on the vertex set V and the cost function $d : E = V \times V \to \mathbb{R}_{\geq 0}$ is given by a metric. Hence, an instance I_m in this setting consists of the complete graph $G = (V, E)$, a metric d and a set of demand-pairs $\mathcal{D} \subseteq V \times V$.

Definition 3.30. *(Metric Steiner forest problem)*
The type of the Steiner forest problem described above, where we consider the complete graph and the cost function d is a metric, is called metric Steiner forest problem. We use the abbreviation MSFP for this type of problem.

Comment
If one chooses n points in the Euclidean plane, declare them as vertices and connect each pair of vertices by an edge whose length is the Euclidean distance $d_{\mathcal{E}}$ between the two end-vertices, one get a complete graph G that is embedded in the Euclidean plane. By choosing a set of demand-pairs $\mathcal{D}$, we obtain an instance $I_m = (G, d_{\mathcal{E}}, \mathcal{D})$ of the MSFP. We will call this the *Euclidean version of the Steiner forest Problem.*
The next section shows there is no loss of generality working with the metric version of the Steiner forest problem.

The more important change in our setting is, that we no longer assume that the feasible solution A is a tree. From now on, A can be an arbitrary feasible solution-forest in the graph G. We denote its connected components as $A_1, \dots, A_p \subseteq A$, where the numbering is chosen such that $w(A_1) \leq \cdots \leq w(A_p)$ holds.

As before, we denote by F another feasible solution for the underlying instance.

3.2.2. Metric SFP and the general SFP

The following theorem depicts why we can concentrate on the metric version of the Steiner forest problem without any loss of generality:

Theorem 3.31. *(Metric SFP and general SFP)*
If there exists a polynomial time c-approximation algorithm for the Metric Steiner forest problem, then there is a polynomial time c-approximation algorithm for the (general) Steiner Forest Problem.

Proof. Suppose there is a polynomial time c-approximation algorithm for the Metric Steiner forest problem, let's call it ALG_m. We show that there is a polynomial time c-approximation algorithm solving the (general) Steiner forest problem.

Let $I = (G, d, \mathcal{D})$ be an instance of the (general) Steiner forest problem with $G = (V, E)$ being an connected, undirected graph, $d : E \to \mathbb{R}_{\geq 0}$ being a cost function that is not necessarily a metric, and $\mathcal{D}$ a set of demand-pairs. Let F^* be an optimal solution for I.

For two points $u, v \in V$, let $d_m(u, v)$ be the length of the shortest $u-v$ path in the connected graph G. Note that the function $d_m : V \times V \to \mathbb{R}_{\geq 0}$ defines a metric. Hence, $I_m := (G_m = (V, V \times V), d_m, \mathcal{D})$ is an instance of the metric Steiner forest problem. We can apply ALG_m to I_m and obtain in polynomial time a solution A_m that satisfies:

$$d_m(A_m) \leq c \cdot d_m(F_m^*)$$

where F_m^* is an optimal solution to the metric instance I_m.

For two points $u, v \in V$, that are not connected by an edge, set $d(u, v) := M$ for some sufficiently large constant $M \in \mathbb{R}_{\geq 0}$, for example $M := \sum_{e \in E} d_e + 1$. This should represent the cost infinity for the non-present edge $\{u, v\}$ in G. With this adjustment, we then have $d_m(u, v) \leq d(u, v)$ for every pair of points $u, v \in V$. Then the following inequality holds

$$d_m(F_m^*) \leq d_m(F^*) \leq d(F^*)$$

and hence also

$$d_m(A_m) \leq c \cdot d(F^*)$$

We construct a solution A for the original instance I out of A_m as follows: Replace each edge $e = \{u, v\}$ of A_m by the corresponding shortest $u - v$ path. Note that this construction can be easily done in polynomial time. By construction, it holds that

$$d(A) \leq d_m(A_m)$$

We have an inequality between those two values since some paths may share edges which are then counted only once in $d(A)$ but multiple in $d_m(A_m)$. Putting everything together, we finally obtain

$$d(A) \leq c \cdot d(F^*)$$

which completes the proof. □

3.2.3. Preliminaries

From now on, we are dealing with an instance $I_m = (G, d_{\mathcal{E}}, \mathcal{D})$ of the MSFP. Let's see, what we may assume about the feasible solution F (we always have in mind that F is an optimal solution). Denote the connected components of F by $F_1, \ldots, F_q \subseteq F$. We can assume that there are no inessential edges. Note that every connected component F_i is a tree. We may convert every component F_i into a simple cycle:

- Let $G[F_i]$ be the graph induced by F_i. Double each edge in $G[F_i]$ such that the resulting graph is a multi-graph ("twice around the corner").
- Construct an Euler-Tour P in $G[F_i]$. By construction, every vertex of $G[F_i]$ has even degree and hence, by the Euler- Hierholzer Theorem (see e.g. [EH]), such a tour exists.
- Short-cut P over repeated vertices. This yields a Hamiltonian cycle HC on the vertices $V[F_i]$ in G.
- Short-cut the Hamiltonian cycle HC over non-terminals. This yields a Hamiltonian Cycle HC' on the terminals of $V[F_i]$ in G.
- The edge set of HC' gives the edge set F_i' that induces a simple cycle in G.

Denote by F' the solution obtained by applying the modification shown above to each connected component of F. We have that $V[\mathcal{F}'] \subseteq V[A]$, since the vertices of F' are all terminals, which clearly has to be covered by the feasible solution A.

We can assume that $V[A]$ also consists only of terminals (the triangle inequality allows this) and hence $V[\mathcal{F}'] = V[A]$. For a given set of demand-pairs $\mathcal{D}$, denote by $V_{\mathcal{D}} \subseteq V$ the set of all terminals. From now on, we can assume that $V = V[F] = V[A] = V_{\mathcal{D}}$.

Example 3.32. *(Modifying the forest F)*

Consider the following connected component F_i of the feasible solution F consisting of four terminals $\{s_1, t_1, s_2, t_2\}$ and two non-terminals $\{v_1, v_2\}$ shown in the left picture of Figure 30.

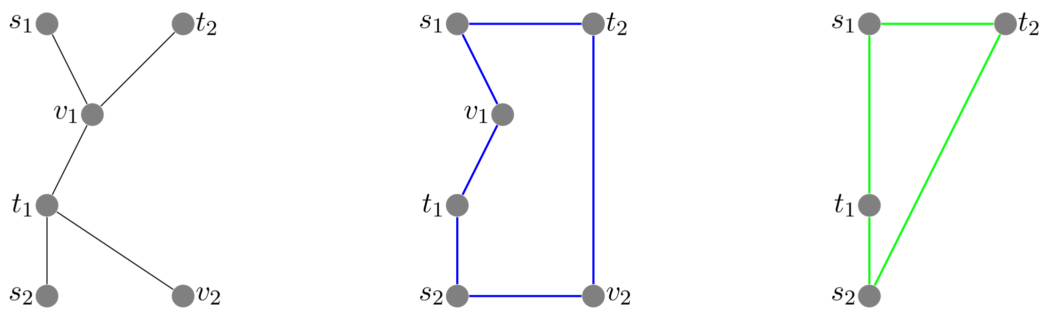


Figure 30: The left picture shows the connected component F_i in its original form, the picture in the middle the constructed Hamiltonian cycle on the vertices of F_i and the right picture the resulting component F_i'.

Finding an Euler-Tour is straight forward:

$$s_1 \to v_1 \to t_1 \to s_2 \to t_1 \to v_2 \to t_1 \to v_1 \to t_2 \to v_1 \to s_1$$

Short-cutting over already visited vertices mean that we delete all vertices which has already appeared on the path. This yields a Hamiltonian path on the vertices of F_i in G. Note, that G is the complete graph, hence all the edges are present in G:

$$s_1 \to v_1 \to t_1 \to s_2 \to v_2 \to t_2 \to s_1$$

Finally, we short-cut also over non-terminal, hence we simply delete the non-terminals from the path and take instead the direct edges. The final simple cycle that passes through all terminals looks as follows:

$$s_1 \to t_1 \to s_2 \to t_2 \to s_1$$

Lemma 3.33.
Let A, F be two feasible solutions for the instance I_m of MSFP and assume that $V[A] = V_{\mathcal{D}}$. Then there exists a solution F' whose connected components $F_1, \ldots, F_q$ are node disjoint simple cycles and which satisfies $V[F'] = V[A] = V_{\mathcal{D}}$. Moreover, it holds that $d(F') \leq 2 \cdot d(F)$ and $\phi(F') \leq 2 \cdot \phi(F)$.

Proof. To obtain F', we apply the modification given on the previous page to each connected component $F_1, \ldots, F_q \subseteq F$. F' consists therefore of simple cycles $F_1', \ldots, F_p'$ where F_i' was obtained from F_i for $1 \leq i \leq q$. By construction, it follows that $V[F'] = V[A] = V_{\mathcal{D}}$. In each component F_i, we start with the edge set F_i, double each edge $e \in F_i$ and short-cut edges whose edge lengths obey a metric, therefore we get $d(F_i') \leq 2 \cdot d(F_i)$ by using triangle inequality and summing over all components yields $d(F') \leq 2 \cdot d(F)$. For the last part, we use that $w(F_i) = w(F_i')$ for $1 \leq i \leq q$, and hence

$$\begin{aligned}\phi(F') = d(F') + w(F') = \sum_{i=1}^{q} d(F_i') + \sum_{i=1}^{q} w(F_i') \leq 2 \cdot \sum_{i=1}^{q} d(F_i) + \sum_{i=1}^{q} w(F_i) \\ \leq 2 \cdot \big(\sum_{i=1}^{q} d(F_i) + \sum_{i=1}^{q} w(F_i) \big) = 2 \cdot \big(d(F) + w(F) \big) = 2 \cdot \phi(F)\end{aligned}$$

□

Definition 3.34. *($\xi(v)$, G_A and $\hat{G}_A$)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let A, F be two solutions for I such that $A_1, \ldots, A_p \subseteq A$ are the components of A and $F_1, \ldots, F_q \subseteq F$ are the components of F where every component F_i is a simple cycle which vertices are only terminals. As mentioned, we can assume $V = V[F] = V[A] = V_{\mathcal{D}}$.

For $v \in V$, we set $\xi(v) := j$ for the unique $j \in \{1, \ldots, p\}$ that satisfies $v \in A_j$, i.e. $\xi(v)$ is the index of the connected component of A that contains the vertex v.

Moreover, set $G_{A,F} := G/\{A_1, \ldots, A_p\} = (V_{A,F}, E_{A,F})$ to be a multi-graph with

$$V_{A,F} := \{1, \ldots, p\} \qquad E_{A,F} := \{e_f \mid f = \{v, w\} \in F : \xi(v) \neq \xi(w)\}$$

We extend the distance function d to $G_{A,F}$ by setting $d_{e_f} := d_f$ for all $e_f \in E_{A,F}$.
Denote by $\hat{G}_{A,F}$ the graph on the vertices $V_{A,F}$ that is the transitive closure of $G_{A,F}$: For all pairs (i, j) with $i \neq j \in V_{A,F}$ it contains an edge e_{ij} whose length $d_{e_{ij}}$ is given by the length of a shortest $i - j$ path in $G_{A,F}$.

Comments

- The graph $G_{A,F}$ results from contracting the connected components of A in F.
  - The vertices of $G_{A,F}$ corresponds to the connected components of A .
  - The edges of $G_{A,F}$ corresponds to edges $f \in F$ that cross between two different components of A.
- Recall the graph G_A^{all} defined in Section 2.2.2 for the *connecting*-moves. Note that $G_{A,F}$ is a subgraph of this graph G_A^{all}.
- Thus, every tree in $G_{A,F}$ induces a *connecting*-move.

3.2.4. Simple cycles and their circuits

Consider an instance of the Metric Steiner forest problem with all the settings as in the previous section. Remember that we can assume the connected components $F_1, \ldots, F_q$ of the solution F being simple cycles.

For each $i \in \{1, \ldots, q\}$, the simple cycle F_i in G induces a circuit C_i in G_A[7]. The edges of the circuit C_i corresponds to those edges of F_i, that cross between two connected components of A. Remember that these edges are those we want to remove such that there are no F-edges crossing between two connected components of A.

Observe that C_i is not necessarily simple (see Figure 31): Whenever F_i visits the connected component A_j of A, the induced circuit C_i revisits the same node $j \in V_A$.

Assume C_i visits exactly s distinct vertices, which corresponds to the s different connected components $A_{\xi_1}, \ldots, A_{\xi_s}$, that are traversed by the simple cycle F_i in G. Name those vertices by the index of the component, from which they arise, i.e. by $\xi_1, \ldots, \xi_s$. We can assume without loss of generality that $\xi_1 > \cdots > \xi_s$.

[7]Note, that the circuit consists of only one single vertex in the case where $V[F_i] \subseteq V[A_j]$ for some $1 \leq i \leq q$ and $1 \leq j \leq p$.

Since the connected components are ordered according to non-decreasing width, we know that $w(A_{\xi_1}) \geq \cdots \geq w(A_{\xi_s})$, hence the ξ_i are ordered according to the widths of the corresponding connected components.

Let n_l be the number of times that C_i visits ξ_l for $l \in \{1, \ldots, s\}$[8].

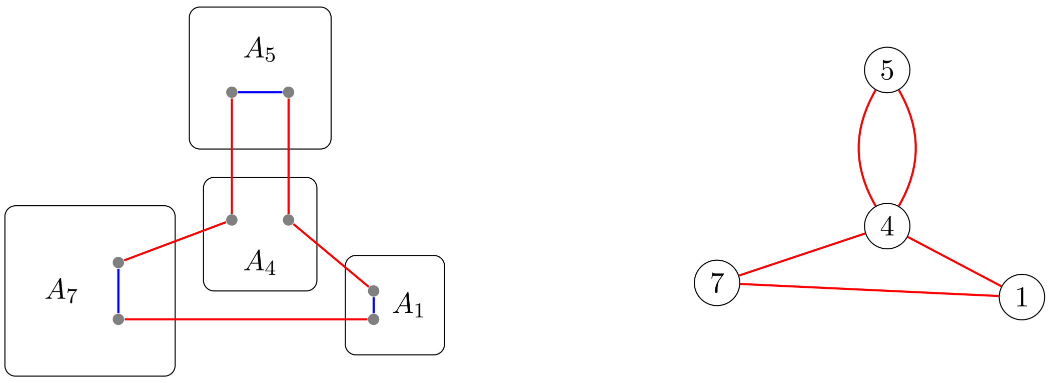


Figure 31: The simple cycle F_i in G depicted on the left induces the circuit C_i in G_A which is shown on the right. The connected components A_1, A_4, A_5, A_7 are numbered in a way such that $w(A_1) \leq w(A_4) \leq w(A_5) \leq w(A_7)$, which is indicated by the size of the rounded rectangles. The edges of F_i in red are those, that cross between two connected components, the blue one are inner-connected-component edges.

Example 3.35.

In Figure 31 we have:
$s = 4$, $\xi_1 = 7, \xi_2 = 5, \xi_3 = 4, \xi_4 = 1$ and $n_1 = 1, n_2 = 1, n_3 = 2, n_4 = 1$.

3.2.5. Connection move-optimality and partitions

The crucial idea for the replacement of C_i is to use the connecting move optimality of A in order to give a lower bound for the total length $d(C_i)$ of the circuit C_i. Note that any subgraph T of C_i that is a tree in G_A can be used for a *connecting*-move.

Consider the simple cycle F_i and the induced circuit C_i from Figure 31. We can partition the simple cycle into subgraphs H_1, H_2, H_3 that corresponds to trees T_1, T_2, T_3 in G_A that partition the circuit as shown in Figure 32 (let's write $H_1 \sim T_1$ if the subgraph H_1 corresponds to the tree T_1 in G_A).

Assuming that A is *connection*-move-optimal, then using any of those trees does not improve the quality of the solution, hence we can give some lower bounds for the $d(T_i)$'s and therefore for $d(C_i)$.

Example 3.36.

For an easier notation set $A = A_1 \cup A_4 \cup A_5 \cup A_7$ and let $A' = A \cup T_1$ be the solution that results by applying $conn(T_1)$ to A.

[8]If we would be very precise, we should have indexed all variables above also with an additional i that indicates to which simple cycle F_i they belong to, e.g. we would have to write $\xi_{i,1}, \ldots, \xi_{i,s}$ instead of $\xi_1, \ldots, \xi_s$. We drop this additional index right from the beginning, since it would only cause more confusion as it helps.

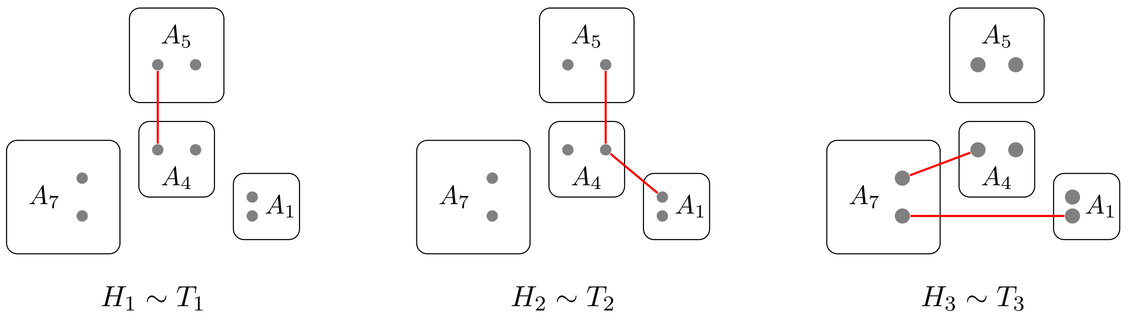


Figure 32: A partitioning of the simple cycle into three sub-sets H_1, H_2, H_3 yields a corresponding partition of the circuit into three trees T_1, T_2, T_3 in G_A. The partition is not unique, see Figure 33.

We get

$$\begin{aligned}\phi(A) &= d(A) + w(A)\\ &= d(A) + w(A_1) + w(A_4) + w(A_5) + w(A_7)\end{aligned}$$

$$\begin{aligned}\phi(A') &= d(A') + w(A')\\ &= d(A) + d(T_1) + w(A_1) + w(A_4) + w(A_5) + w(A_7) - w(A_4)\\ &= \phi(A) + d(T_1) - w(A_4)\end{aligned}$$

Since this move does not improve, we have

$$\phi(A') \geq \phi(A) \iff \phi(A) + d(T_1) - w(A_4) \geq \phi(A) \iff d(T_1) \geq w(A_4)$$

Analogous calculations shows that $d(T_2) \geq w(A_4) + w(A_1)$ and $d(T_3) \geq w(A_4) + w(A_1)$. Taking all these three bounds together, we obtain

$$d(C_i) = d(T_1) + d(T_2) + d(T_3) \geq 3 \cdot w(A_4) + 2 \cdot w(A_1)$$

We may consider one other possible partitioning of the circuit into trees:

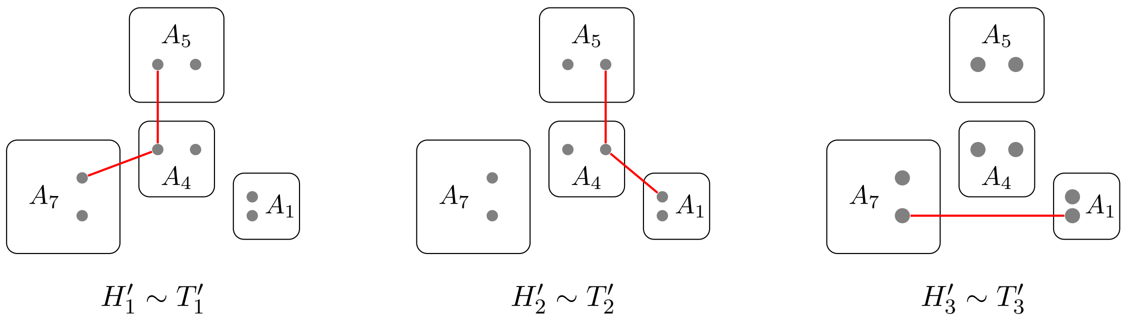


Figure 33: Another partitioning of the simple cycle yields another partition of the circuit into three trees T'_1, T'_2, T'_3.

Doing the same calculation as before and assuming that none of these induced *connection*-moves is improving, we get

$$d(T_1') \geq w(A_5) + w(A_4) \qquad d(T_2') \geq w(A_4) + w(A_1) \qquad d(T_3') \geq w(A_1)$$

and summing over all three trees gives

$$d(C_i) \geq w(A_5) + 2 \cdot w(A_4) + 2 \cdot w(A_1) \geq n_2 \cdot w(A_5) + n_3 \cdot w(A_4) + n_4 \cdot w(A_1) \tag{4}$$

Let's assume that $w(A_5) > w(A_4)$ (by our convention how to enumerate the connected components we always have $w(A_5) \geq w(A_4)$). Then, the second partition provides a better lower bound for $d(C_i)$, since $w(A_5) + 2 \cdot w(A_4) + 2 \cdot w(A_1) > 3 \cdot w(A_4) + 2 \cdot w(A_1)$.

Inequality (4) shows more: This lower bound contains $w(A_{\xi_l})$ at least n_l times for all $l \in \{2, 3, 4\} = \{2, \ldots, s\}$. We will show in Section 3.2.6 that there exists always a partition with this property in some special cases.

Definition 3.37. *(T pays for ξ_l)*
We say a tree T in G_A pays for the vertex ξ_l (once) if it contains ξ_l and at least one other vertex ξ_k with $\xi_k > \xi_l$.

Definition 3.38. *(guarded, minimal guarded)*
Let $C = (e_1, \ldots, e_{|C|})$ be a circuit in G_A that starts at v_1 and visits the nodes $v_1, \ldots, v_{|C|+1} = v_1$ in this order. For this fixed order, we call C guarded, if $v_i < v_1$ for all $i \in \{2, \ldots, |C|\}$. A circuit C is minimally guarded if it is guarded and no sub-circuit $(v_{i_1}, \ldots, v_{i_2})$ with $i_1, i_2 \in \{2, \ldots, |C|\}$, $i_1 < i_2$ and $v_{i_1} = v_{i_2}$ is guarded.

Comments

- In any guarded circuit, the highest component number only appears once (twice, if you start the circuit with this number and end it also there).
- In Figure 31, C_i is guarded, since starting at $v_1 = 7$, in this circuit $7-4-5-4-1(-7)$, the highest number, namely 7, appears only once (twice, if you start with 7 and close the circuit).
- This circuit is also minimally guarded, since it is guarded and the only sub-circuit, namely $4-5-4$ is not guarded. Note, that it is not allowed to change the starting point of the sub-circuit, i.e. in this example we are not allowed to describe the sub-circuit as $5-4-5$.

Definition 3.39. *(c-approximate connection move)*
In the setting of the local search algorithm provided in this work, a c-approximate connection move for some constant $c \geq 1$ is a connecting move $conn(T)$ applied to the current solution A that uses a tree $T \in G_A^{all}$ such that $c \cdot d(T) \leq w(A) - w(A \cup T)$.
A solution A is c-approximate connecting move optimal, if there are no c-approximate connecting moves.

Comment
If a tree $T \in G_A^{all}$ yields a c-approximate connection move, it means that at least the single total length of the tree T is less or equal to the difference of the width's of the solution A and $A \cup T$, respectively. Denote by $A' := A \cup T$ the obtained solution by the connection move with T. Then, with $c \geq 1$ and $d(T) \geq 0$, we get

$$\begin{aligned} c \cdot d(T) \leq w(A) - w(A \cup T) &= w(A) - w(A') \\ \iff \; c \cdot d(T) + w(A') &\leq w(A) \\ \implies \; d(T) + w(A') &\leq w(A) \end{aligned}$$

and further

$$\phi(A') = d(A') + w(A') = d(A) + d(T) + w(A') \leq d(A) + w(A) = \phi(A)$$

which means that a c-approximate connection move is always improving with respect to the potential ϕ or at least yields the same value of ϕ.

For $c' < c$ it's easy to see that a c-approximate connecting move is also a c'-approximate connecting move, but not necessarily vice versa.

If a *connecting*-move, as defined in Section 2.2.2, is improving, then it is at least a 1-approximate connecting move. Since the set of all possible trees in G_A^{all} is not of polynomial size, we can not decide in polynomial time whether there is a 1-approximate connecting move, but we will see later, that deciding in polynomial time whether there exists a c-approximate connecting move for some $c > 1$ is possible.

Lemma 3.40. *(Bounding the total length of a circuit, first Approach)*
Let $C = (e_1, \ldots, e_{|C|})$ be a guarded circuit in G_A that visits the nodes $v_1, \ldots, v_{|C|+1} = v_1$ in this order. Assume that $v_1 = v_{|C|+1} > v_i$ for all $i \in \{2, \ldots, |C|\}$ and that $v_1, \ldots, v_{|C|}$ correspondes to pairwise different vertices $\xi_1 > \cdots > \xi_s$ (it follows that $v_1 = \xi_1$). Furthermore, let n_l be the number of times that C visits node ξ_l for all $l \in \{1, \ldots, s\}$. If A is c-approximate connecting move optimal and there exists a set of trees $\mathfrak{M}$ in G_A, that satisfies

(i) all trees in $\mathfrak{M}$ are edge-disjoint and only contain edges from C

(ii) for all $l \in \{2, \ldots, s\}$ there are at least n_l trees that pay for ξ_l

then the equality

$$\sum_{i=2}^{|C|} w(A_{v_i}) = \sum_{l=2}^{s} n_l \cdot w(A_{\xi_l}) \leq c \cdot \sum_{i=1}^{|C|} d_{e_i} = c \cdot d(C)$$

holds.

Proof. Assumption (i) implies that

$$\sum_{T \in \mathfrak{M}} \sum_{e \in T} d_e \leq \sum_{i=1}^{|C|} d_{e_i}$$

As mentioned in the comments at the end of Section 3.2.3, every tree in $\mathfrak{M}$ is also a tree in G_A^{all} and hence defines a *connecting* move. By our assumption, A is c-approximate connecting optimal, therefore it holds that

$$\sum_{v \in V[T]} w(A_v) - \max_{v \in V[T]} w(A_v) \leq c \cdot \sum_{e \in T} d_e = c \cdot d(T) \tag{5}$$

for every tree $T \in \mathfrak{M}$. We set $low(T) := V[T] \setminus max\{\xi_i \mid \xi_i \in T\}$ and then we have

$$\sum_{i=2}^{|C|} w(A_{v_i}) = \sum_{l=2}^{s} n_l \cdot w(A_{\xi_l}) \overset{(ii)}{\leq} \sum_{l=2}^{s} \sum_{T \in \mathfrak{M}} \mathbb{1}_{low(T)}(\xi_l) w(A_{\xi_l}) = \sum_{T \in \mathfrak{M}} \sum_{l=2}^{s} \mathbb{1}_{low(T)}(\xi_l) w(A_{\xi_l})$$

$$= \sum_{T \in \mathfrak{M}} \sum_{v \in low(T)} w(A_v) = \sum_{T \in \mathfrak{M}} \left(\sum_{v \in V[T]} w(A_v) - \max_{v \in V[T]} w(A_v) \right)$$

$$\overset{(5)}{\leq} \sum_{T \in \mathfrak{M}} c \cdot d(T) = c \cdot d(C)$$

□

3.2.6. The partitioning algorithm

In this section, we show how to partition minimally guarded circuits into a set $\mathfrak{M}$ of trees such that there are at least n_i trees that pay for each ξ_i. In other words, we want to give a recipe for finding a partition of a minimally guarded circuit that fulfils the requirements of the partition needed in Lemma 3.40.

The algorithm on a high level

To be precise, the algorithm computes a sequence of sets $\mathfrak{M}_k$ of trees such that for $k \in \{2, \dots, s\}$, $\mathfrak{M}_k$ contains a partitioning with n_i trees that pay for ξ_i for all $i \in \{1, \dots, k\}$. The output of the algorithm is then $\mathfrak{M} := \mathfrak{M}_s$.

The algorithm maintains a partitioning of C into a set of sub-paths $\mathfrak{P}_k$: These paths are not necessarily simple, but they are all edge-disjoint. The algorithm iteratively splits non-simple sub-paths into simple sub-paths.

In addition to that, the algorithm needs to make sure that the sub-paths can be combined to trees that satisfy the conditions in Lemma 3.40. This is done by building the elements of the set $\mathfrak{M}_k$ not as trees in G_A, but in its transitive closure $\hat{G}_A$ (recall Definition 3.34). Doing it in this way, we ensure that any edge of each tree in $\mathfrak{M}_k$ corresponds to a path in the current partitioning $\mathfrak{P}_k$.

If a tree in $\mathfrak{M}_k$ contains an edge $\{v, w\}$, then the partitioning contains a sub-path $(v, \dots, w)$. We call the edge $\{v, w\}$ transitive, if there exists at least one inner vertex in the sub-path $(v, \dots, w)$. Moreover, we say that a tree $T \in \mathfrak{M}_k$ *claims* a sub-path P of C if one of the edges of T corresponds to P. Each time the algorithm splits a sub-path, it also splits the corresponding edge of a tree in $\mathfrak{M}_k$.

The trees in the final set $\mathfrak{M}_s$ do not contain transitive edges, hence they are subgraphs of G_A. They also leave no sub-path of C unclaimed.

The correspondence between trees and sub-paths, is represented by the following mapping:

Definition 3.41. *(The mapping π_k)*
In the setting as described above, we define

$$\pi_k : \mathfrak{P}_k \to \Big(\bigcup_{T \in \mathfrak{M}_k} T \Big) \cup \{\perp\}$$

that maps a path $P = (v, \dots, w) \in \mathfrak{P}_k$ to an edge $e \in \bigcup_{T \in \mathfrak{M}_k} T$[9] if and only if $e \cap V[P] = \{v, w\}$. If no such edge exists, then it maps the path P to $\perp$, i.e. $\pi(P) = \perp$.

Note that the mapping is indeed well-defined: Consider the path $P = (v, \dots, w)$. If

$$\begin{aligned} \pi_k(P) = e &\iff e \cap V[P] = \{v, w\} \implies e = \{v, w\} \\ \pi_k(P) = f &\iff e \cap V[P] = \{v, w\} \implies f = \{v, w\} \end{aligned}$$

it follows that $e = f$.

The algorithm
The following Partitioning Algorithm itself would deserve a handful of pages, if we want to describe each detail and idea. For our purposes, it suffices to know that such an algorithm exists and how it works on a high level. For the sake of completeness, we provide the pseudo-code of the algorithm and a short example on the following pages.

Observe, that the vertex v in Line 1 of Algorithm 3 is indeed unique: If $v_{i_1}, v_{i_2} \in C$ with $i_1 \neq i_2$ would be two vertices such that $v_{i_1} = v_{i_2} = \xi_2$, then the subcircuit $(v_{i_1}, \dots, v_{i_2})$ certifies that C is not minimally guarded. Note, that the trees in $\mathfrak{M}_k$ are rooted, which is needed for the computation during the run of the algorithm. In the final set $\mathfrak{M}_s$, one can drop the roots of each tree, since they are not needed any more.

[9]Remember that the tree T is represented by its edge set, so we do not need to write $E(T)$.

Algorithm 3 Partitioning algorithm

Require: A minimally guarded circuit $C = (v_1, \dots, v_{|C|+1})$ in G_A with $v_1 = v_{|C|+1}$. Let $\{\xi_1, \dots, \xi_s\}$ be the set of disjoint vertices on C, w.l.o.g. $\xi_1 > \dots > \xi_s$.
Ensure: A set $\mathfrak{M}$ of edge disjoint trees in G_A consisting of edges from C.

```
1:  let $v$ be the unique vertex in $C$ that satisfies $v = \xi_2$
2:  let $T = \{\{v_1, v\}\}$ and let $\mathfrak{M}_2 = \{(T, v_1)\}$
3:  let $\mathfrak{P}_2 = \{(v_1, \dots, v), (v, \dots, v_{|C|})\}$

4:  for $k = \{3, \dots, s\}$ do
5:    let $\mathfrak{M}_k = \mathfrak{M}_{k-1}$ and let $\mathfrak{P}_k = \mathfrak{P}_{k-1}$
6:    let $I = \{j \in \{2, \dots, |C|\} \mid v_j = \xi_k\}$
7:    let $P_j = (v_{P_j}, \dots, w_{P_j})$ be the path in $\mathfrak{P}_{k-1}$ with $j$ as inner node
      or where $w_{P_j} = \xi_k$ for all $j \in I$
8:    for $j \in I$ with $\pi_{k-1}(P_j) = \perp$ do
9:      if $j$ is an inner node of $P_j$ then
10:       let $T = \{\{v_{P_j}, v_j\}\}$ and let $\mathfrak{M}_k = \mathfrak{M}_k \cup \{(T, v_{P_j})\}$
11:       let $\mathfrak{P}_k = \mathfrak{P}_k \setminus \{P_j\} \cup \{(v_{P_j}, \dots, v_j), (v_j, \dots, w_{P_j})\}$
12:     else
13:       let $T = \{\{v_{P_j}, v_j\}\}$ and let $\mathfrak{M}_k = \mathfrak{M}_k \cup \{(T, v_{P_j})\}$
14:     end if
15:   end for
16:   for $(T, r) \in \mathfrak{M}_{k-1}$ do
17:     let $I_T = \{j \in I \mid \pi_{k-1}(P_j) \in T\}$
18:     if $I_T = \emptyset$ then
19:       select $j^* \in I_T$ such that the path from $j^*$ to $r$ in $T$ contains no $j \in I_T \setminus \{j^*\}$
20:       let $T = T \setminus \{\{v_{P_{j^*}}, w_{P_{j^*}}\}\} \cup \{\{v_{P_{j^*}}, v_{j^*}\}, \{v_{j^*}, w_{P_{j^*}}\}\}$
21:       let $\mathfrak{P}_k = \mathfrak{P}_k \setminus \{P_{j^*}\} \cup \{(v_{P_{j^*}}, \dots, v_{j^*}), (v_{j^*}, \dots, w_{P_{j^*}})\}$
22:       for $j \in I_T \setminus \{j^*\}$ do
23:         let $T = T \setminus \{\{v_{P_j}, w_{P_j}\}\} \cup \{\{v_j, w_{P_j}\}\}$
24:         let $T' = \{\{v_{P_j}, v_j\}\}$ and let $\mathfrak{M}_k = \mathfrak{M}_k \cup \{(T', v_{P_j})\}$
25:         let $\mathfrak{P}_k = \mathfrak{P}_k \setminus \{P_j\} \cup \{(v_{P_j}, \dots, v_j), (v_j, \dots, w_{P_j})\}$
26:       end for
27:     end if
28:   end for
29: end for
30: return $\mathfrak{M}_s$ (where the roots of the trees are dropped)
```

Example 3.42.

Consider the following example from Gross et al. [G17].
Let $C = (v_1, \dots, v_{12}) = (7, 1, 2, 1, 4, 1, 2, 5, 1, 3, 2, 7)$ be a minimally guarded circuit in some graph G_A[10] on the vertices $\{\xi_1, \dots, \xi_7\} = \{7, 5, 4, 3, 2, 1\}$. Therefore, $|C| = 11$ and $s = 6$. The circuit C fulfils the requirements for Algorithm 3 and therefore Algorithm 3 can be applied to C. Below, there is a description of the run of the algorithm where the information of all necessary lines of the pseudo-code in the current iteration are present. We denote the trees that will be added to the sets $\mathfrak{M}_k$ by $T_1, T_2, \dots$. Figure 34 depicts the development of the sets $\mathfrak{M}_k$ and $\mathfrak{P}_k$ during the run of the algorithm.

[10] A detailed description of the graph G, the forests A, F and the graph G_A is not necessary in order to apply Algorithm 3 to the circuit C.

Start
Line 1: $\xi_2 = 5, v = v_8$
Line 2: $T_1 = \{\{v_1, v_8\}\}$, $\mathfrak{M}_2 = \{(T_1, v_1)\}$
Line 3: $\mathfrak{P}_2 = \{(v1, \dots, v_8), (v_8, \dots, v_{11})\}$

Iteration where $k = 3$
Line 4: $k = 3$
Line 5: $\mathfrak{M}_3 = \mathfrak{M}_2$, $\mathfrak{P}_3 = \mathfrak{P}_2$, $\xi_3 = 4$
Line 6: $I = \{5\}$
Line 7: $P_5 = (v_1, \dots, v_8)$

Line 8: $\nexists j \in I : \pi_2(P_j) = \perp$

Line 16: Take $(T_1, v_1) \in \mathfrak{M}_2$
Line 17: $I_{T_1} = \{5\}$
Line 18: $I_{T_1} \neq \emptyset$
Line 19: $j^* = 5$
Line 20: $T_1 = T_1 \setminus \{\{v_1, v_8\}\} \cup \{\{v_1, v_5\}, \{v_5, v_8\}\}$
Line 21: $\mathfrak{P}_3 = \mathfrak{P}_3 \setminus \{(v_1, v_8)\} \cup \{(v_1, v_5), (v_5, v_8)\}$
Line 22: $I_{T_1} \setminus \{5\} = \emptyset$

Iteration where $k = 4$
Line 4: $k = 4$
Line 5: $\mathfrak{M}_4 = \mathfrak{M}_3$, $\mathfrak{P}_4 = \mathfrak{P}_3$, $\xi_4 = 3$
Line 6: $I = \{10\}$
Line 7: $P_{10} = (v_8, \dots, v_{11})$

Line 8: $\pi_3(P_{10}) = \perp$
Line 10: $T_2 = \{\{v_8, v_{10}\}\}$ and $\mathfrak{M}_4 = \mathfrak{M}_4 \cup \{(T_2, v_8)\}$
Line 11: $\mathfrak{P}_4 = \mathfrak{P}_4 \setminus \{(v_8, v_{11})\} \cup \{(v_8, v_{10}), (v_{10}, v_{11})\}$

Line 16: $\nexists (T, r) \in \mathfrak{M}_3 : I_T \neq \emptyset$

Iteration where $k = 5$
Line 4: $k = 5$
Line 5: $\mathfrak{M}_5 = \mathfrak{M}_4$, $\mathfrak{P}_5 = \mathfrak{P}_4$, $\xi_5 = 2$
Line 6: $I = \{3, 7, 11\}$
Line 7: $P_3 = (v_1, \dots, v_5)$, $P_7 = (v_5, \dots, v_8)$, $P_{11} = (v_{10}, v_{11})$

Line 8: $\pi_4(P_{11}) = \perp$
Line 13: $T_3 = \{\{v_{10}, v_{11}\}\}$ and $\mathfrak{M}_5 = \mathfrak{M}_5 \cup \{(T_3, v_{10})\}$

Line 16: Take $(T_1, v_1) \in \mathfrak{M}_4$
Line 17: $I_{T_1} = \{3, 7\}$
Line 18: $I_{T_1} \neq \emptyset$
Line 19: $j^* = 3$
Line 20: $T_1 = T_1 \setminus \{\{v_1, v_5\}\} \cup \{\{v_1, v_3\}, \{v_3, v_5\}\}$
Line 21: $\mathfrak{P}_5 = \mathfrak{P}_5 \setminus \{(v_1, v_5)\} \cup \{(v_1, v_3), (v_3, v_5)\}$
Line 22: $I_{T_1} \setminus \{3\} = \{7\} \implies j = 7$
Line 23: $T_1 = T_1 \setminus \{\{v_5, v_8\}\} \cup \{\{v_7, v_8\}\}$
Line 24: $T_4 = \{\{v_5, v_7\}\}$ and $\mathfrak{M}_5 = \mathfrak{M}_5 \cup \{(T_4, v_5)\}$

Line 25: $\mathfrak{P}_5 = \mathfrak{P}_5 \setminus \{(v_5, v_8)\} \cup \{(v_5, v_7), (v_7, v_8)\}$

Line 16: Take $(T_2, v_8) \in \mathfrak{M}_4$
Line 17: $I_{T_1} = \emptyset$

Iteration where $k = 6$
Line 4: $k = 6$
Line 5: $\mathfrak{M}_6 = \mathfrak{M}_5$, $\mathfrak{P}_6 = \mathfrak{P}_5$, $\xi_6 = 1$
Line 6: $I = \{2, 4, 6, 9\}$
Line 7: $P_2 = (v_1, \ldots, v_3)$, $P_4 = (v_3, \ldots, v_5)$, $P_6 = (v_5, \ldots, v_7)$, $P_9 = (v_8, \ldots, v_{10})$,

Line 8: $\nexists j \in I : \pi_2(P_j) = \perp$

Line 16: Take $(T_1, v_1) \in \mathfrak{M}_5$
Line 17: $I_{T_1} = \{2, 4\}$
Line 18: $I_{T_1} \neq \emptyset$
Line 19: $j^* = 2$
Line 20: $T_1 = T_1 \setminus \{\{v_1, v_3\}\} \cup \{\{v_1, v_2\}, \{v_2, v_3\}\}$
Line 21: $\mathfrak{P}_6 = \mathfrak{P}_6 \setminus \{(v_1, v_3)\} \cup \{(v_1, v_2), (v_2, v_3)\}$
Line 22: $I_{T_1} \setminus \{2\} = \{4\} \implies j = 4$
Line 23: $T_1 = T_1 \setminus \{\{v_3, v_5\}\} \cup \{\{v_4, v_5\}\}$
Line 24: $T_5 = \{\{v_3, v_4\}\}$ and $\mathfrak{M}_6 = \mathfrak{M}_6 \cup \{(T_5, v_3)\}$
Line 25: $\mathfrak{P}_6 = \mathfrak{P}_6 \setminus \{(v_3, v_5)\} \cup \{(v_3, v_4), (v_4, v_5)\}$

Line 16: Take $(T_2, v_8) \in \mathfrak{M}_5$
Line 17: $I_{T_2} = \{9\}$
Line 18: $I_{T_2} \neq \emptyset$
Line 19: $j^* = 9$
Line 20: $T_2 = T_2 \setminus \{\{v_8, v_{10}\}\} \cup \{\{v_8, v_9\}, \{v_9, v_{10}\}\}$
Line 21: $\mathfrak{P}_6 = \mathfrak{P}_6 \setminus \{(v_8, v_{10})\} \cup \{(v_8, v_9), (v_9, v_{10})\}$
Line 22: $I_{T_2} \setminus \{9\} = \emptyset$

Line 16: Take $(T_3, v_{10}) \in \mathfrak{M}_5$
Line 17: $I_{T_3} = \emptyset$

Line 16: Take $(T_4, v_5) \in \mathfrak{M}_5$
Line 17: $I_{T_4} = \{6\}$
Line 18: $I_{T_4} \neq \emptyset$
Line 19: $j^* = 6$
Line 20: $T_4 = T_4 \setminus \{\{v_5, v_7\}\} \cup \{\{v_5, v_6\}, \{v_6, v_7\}\}$
Line 21: $\mathfrak{P}_6 = \mathfrak{P}_6 \setminus \{(v_5, v_7)\} \cup \{(v_5, v_6), (v_6, v_7)\}$
Line 22: $I_{T_4} \setminus \{6\} = \emptyset$

Final step
Line 30: Return $\mathfrak{M}_6 = \{T_1, \ldots, T_5\}$

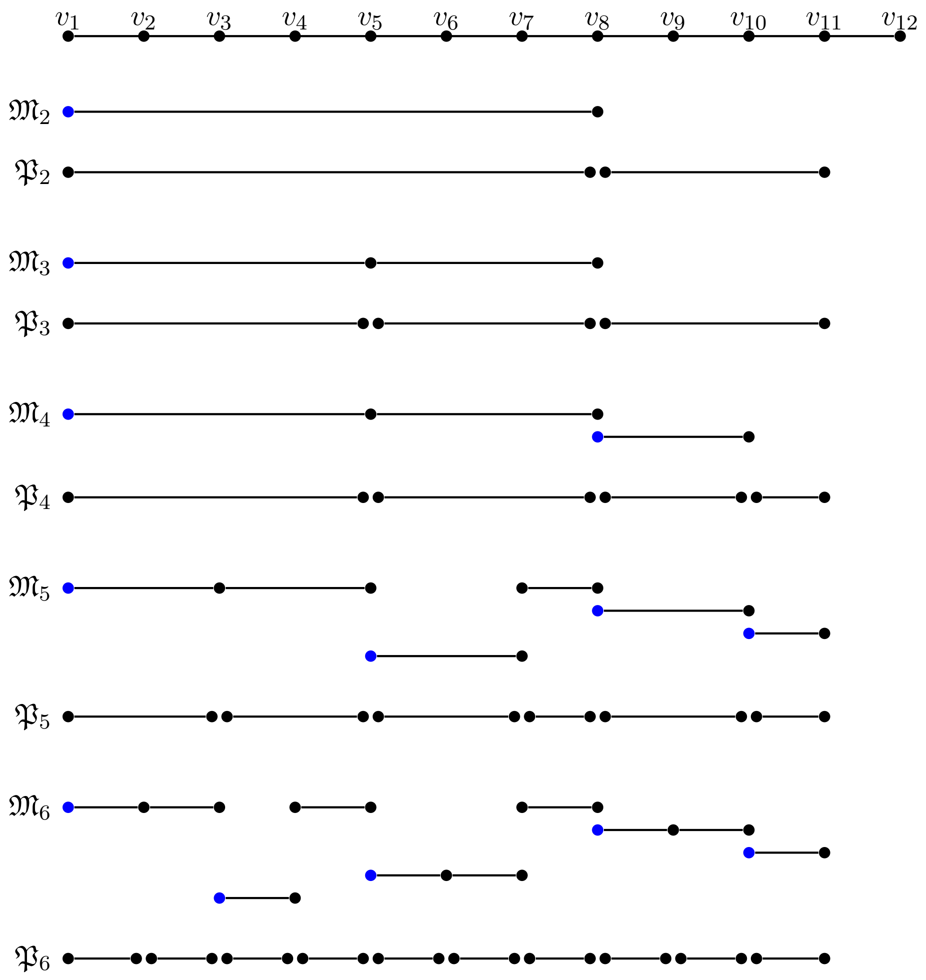


Figure 34: The sets $\mathfrak{M}_k$ and $\mathfrak{P}_k$ during the run of Algorithm 3 for the given circuit C. The roots of the trees in $\mathfrak{M}_k$ are coloured in blue.

The properties of Algorithm 3

Let's list up some properties of the Partitioning algorithm described on the previous pages. We will see then in the next section, how they help us to prove our desired results.

Lemma 3.43. *(Partitioning Algorithm, Part I)*
For all $k = \{2, \ldots, s\}$ it holds that after iteration k there are at least n_i trees in $\mathfrak{M}_k$ that pay for ξ_i for all $i \in \{2, \ldots, k\}$.

Lemma 3.44. *(Partitioning Algorithm, Part II)*
The following statements are true for all $k = \{2, \ldots, s\}$:

1. *The trees in $\mathfrak{M}_k$ are edge disjoint.*
2. *The paths in $\mathfrak{P}_k$ are edge disjoint and it holds that $\bigcup_{P \in \mathfrak{P}_k} P = C \setminus \{\{v_{|C|}, v_{|C|+1}\}\}$.*
3. *If v is an outer node of some $P \in \mathfrak{P}_k$, then $v \in \{\xi_1, \ldots, \xi_k\}$. If v is an inner node, then $v \in \{\xi_{k+1}, \ldots, \xi_s\}$.*
4. *For any $e \in T$, $T \in \mathfrak{M}_k$, $\pi_k^{-1}(e)$ consists of one path from $\mathfrak{P}_k$.*
5. *If $\{v_{j_1}, v_{j_2}\}$ with $j_1 < j_2$ is an edge in T for some (rooted) tree $(T, r) \in \mathfrak{M}_k$, then v_{j_1} is closer to r than v_{j_2}.*

Corollary 3.45. *(Bounding the total length of a circuit)*
Assume that a solution A to an metric SFP instance I_m is c-approximate connecting move optimal. Let $C = (v_1, \dots, v_{|C|+1} = v_1)$ be a circuit in G_A with edges $(e_1, \dots, e_{|C|})$. If C is minimally guarded, then

$$\sum_{i=2}^{|C|} w(A_{v_i}) \leq c \cdot \sum_{i=1}^{|C|} d_{e_i} = c \cdot d(C)$$

Proof. We can apply Algorithm 3 to the minimally guarded circuit C, which gives us a set $\mathfrak{M} = \mathfrak{M}_s$ of edge disjoint trees in G_A that consists of edges from C and also the final partitioning $\mathfrak{P} = \mathfrak{P}_s$ of C into sub-paths[11].

- Lemma 3.44.4 ensures that for all $T \in \mathfrak{M}$ and all edges $e \in T$ there is a unique path $\pi^{-1}(e) \in \mathfrak{P}$.
- Lemma 3.44.3 for $k = s$ means that paths can no longer have inner nodes, hence $\pi^{-1}(e)$ is a single edge and therefore, e also exists in G_A[12].
- Thus, all trees in $\mathfrak{M}$ are trees in G_A.
- By Lemma 3.44.1 and the previous points, the trees of $\mathfrak{M}$ are edge disjoint in G_A.
- Lemma 3.43 ensures that precondition 2 in Lemma 3.40 is fulfilled.

This shows, that we are in the setting as in Lemma 3.40 and fulfil all requirements, hence we can apply the lemma which directly proves the corollary. ☐

Comment
Corollary 3.45 simply put Lemma 3.40 and the results of Algorithm 3 together:

Lemma 3.40 starts with a guarded circuit and the c-approximate optimal solution A. Then it assumes that a special kind of set of trees exists.

The analysis of Algorithm 3 shows, that for minimally guarded cycles (which are guarded by definition), Algorithm 3 provides exactly such a partition as needed in Lemma 3.40. This means, for minimally guarded circuits, the assumptions for Lemma 3.40 are fulfilled automatically and hence the claimed bound for $d(C)$ holds, which is exactly the statement of Corollary 3.45.

3.2.7. Final preliminaries

Definition 3.46. *($F_{\leftrightarrow}$ and $F_{\circlearrowright}$)*
Let $I = (G, d, \mathcal{D})$ be an instance of the SFP with $G = (V, E)$. Let A be a solution to I with connected components $A_1, \dots, A_p$. Remember that for a vertex $v \in V$, the value $\xi(v)$ is the unique index $j \in \{1, \dots, p\}$ such that $v \in A_j$.

For an edge set $F \subseteq E$ in G we denote by $F_{\circlearrowright}$ the set of edges in F that lie within any component of A and by $F_{\leftrightarrow}$ the set of edges in F that connect different components of A, i.e.

$$F_{\circlearrowright} := \{e = \{u, v\} \in F \mid \xi(u) = \xi(v)\}, \qquad F_{\leftrightarrow} := \{e = \{u, v\} \in F \mid \xi(u) \neq \xi(v)\}$$

[11]The algorithm does not explicitly output the set $\mathfrak{P}_s$, but of course this set exists.
[12]Remember that the algorithm in principle built the trees in the transitive closure $\hat{G}_A$ of G_A

If an edge set $F' \subseteq E$ in G satisfies $V[F'] \subseteq V[A_j]$, then we set $\xi(F') := j$, i.e. we extend the mapping ξ to all subsets of edges that lie within one connected component of A. Notice, that in this case $F' = F'_{\circlearrowright}$.

Lemma 3.47. *(Getting rid of $F_{\leftrightarrow}$)*
Let $I_m = (G, d, \mathcal{D})$ be an instance of the MSFP and let A be a solution to I_m with connected components $T_1, \ldots, T_p$. Let $\bar{F}$ be a simple path in G that starts and ends in the same connected component T_{j^*} of A and satisfies $\xi(v) \leq j^*$ for all $v \in V[\bar{F}]$. Assume that $\bar{F} \neq \bar{F}_{\circlearrowright}$. Assume that A is edge-set and path-set swap-optimal with respect to $\bar{F}_{\leftrightarrow}$ and that A is c-approximate connecting move optimal.

Then, there exist a set R of edges on the vertices $V[\bar{F}_{\leftrightarrow}]$ with $(\bar{F}_{\circlearrowright} \cup R)_{\circlearrowright} = (\bar{F}_{\circlearrowright} \cup R)$ that satisfies the properties listed below. Let $F'_1, \ldots, F'_x$ be the connected components of $\bar{F}_{\circlearrowright} \cup R$ in $(V[\bar{F}_{\circlearrowright} \cup R], E[\bar{F}_{\circlearrowright} \cup R])$.

1. *A is edge-set swap-optimal with respect to R.*
2. *It holds that $d(R) \leq d(\bar{F}_{\leftrightarrow})$ and $\sum_{l=2}^{x} w(T_{\xi(F'_l)}) \leq c \cdot d(\bar{F}_{\leftrightarrow})$.*
3. *For all F'_l there exists an index $j \in \{1, \ldots, p\}$ such that $V[F'_l] \subseteq V[A_j]$ and hence $\xi(F'_l) = j$.*
4. *There is only one F'_l with $\xi(F'_l) = j^*$, w.l.o.g say that $\xi(F'_1) = j^*$.*

3.2.8. Taking everything together

Lemma 3.48. *(Transforming the forest)*
Let $I_m = (G, d, \mathcal{D})$ be an instance of the metric Steiner forest problem with the complete graph $G = (V, E)$, a metric $d : E \to \mathbb{R}_{\geq 0}$ and a set $\mathcal{D} \subseteq V \times V$ of demand-pairs. Let $A, F \subseteq E$ be two feasible Steiner Forest solutions for I_m. Suppose that A is edge-edge, edge-set and path-set swap-optimal with respect to E and ϕ, that A is c-approximate connecting move optimal and that A only uses edges between terminals.

Then there exists a feasible solution F' with $d(\mathcal{F}') \leq 2(1 + c) \cdot d(F)$ that satisfies $\mathcal{F}' = \mathcal{F}'_{\circlearrowright}$ such that A is edge-edge and edge-set swap-optimal with respect to F'.

Proof. If $F = F_{\circlearrowright}$ then there is nothing to show and we are done. So suppose $F \neq F_{\circlearrowright}$. Denote the connected components of A by $T_1, \ldots, T_p$ and the connected components of F by $F_1, \ldots, F_q$. Lemma 3.33 shows that by accepting a factor of 2 in the cost, we can assume

- the connected components $F_1, \ldots, F_q$ of F being node disjoint simple cycles and
- $V[A] = V[F] = V_I = V$.

Our goal is to replace each cycle F_i by some edge set $\hat{F}_i$ which satisfies $\hat{F}_i = (\hat{F}_i)_{\circlearrowright}$ while keeping the solution feasible and within a constant factor with respect to the potential ϕ. For an easier writing, let $F = F_i$ be one of the simple cycles.

Let $j^* := \max_{v \in V[F]} \xi(v)$ be the index of the component with the highest width among the components that are visited by F[13]. At least two vertices from the component T_{j^*} have to be on the cycle F. Indeed, assume that there is only one vertex z of F in T_{j^*}. By assumption,

[13]Recall that the components of A have indices such that a lower index means a lower width.

every vertex is a terminal and hence has a mate. Since A is a feasible solution, the mate of z is also within T_{j^*}. The simple cycles of F are disjoint, hence if z does not lie on F, it would not be connected to its mate by F, which contradicts the feasibility of F.

If the two vertices of F in T_{j^*} are adjacent (in F), we delete the edge that connects them and obtain a path that satisfies the assumptions of Lemma 3.47 and hence the lemma guarantees the existence of an edge set R. Otherwise, let v_1, v_2 be the two vertices from T_{j^*} that are not connected by an direct edge in F. This gives a partition of the simple cycle into two paths P_1 and P_2, both with endpoints v_1 and v_2 and both satisfying the assumptions of Lemma 3.47. In this case, the lemma implies the existence of two sets R_l and R_r and we set $R := R_l \cup R_r$. In both cases, we get a set of edges R on the vertices $V[F]$ inducing connected components $F'_1, \ldots, F'_x$ of $F_{\circlearrowright} \cup R$ with the following properties:

(1) A is *edge-set swap*-optimal with respect to R.

(2) For all F'_l there exists an index j such that $V[F'_l] \subseteq V[T_j]$ and hence $\xi(F'_l) = j$. In the case $R = R_l \cup R_r$, notice that the connected components of R_l and R_r are disjoint with the exception of those that contain v_1 and v_2. Thus, no components with vertices from different T_j will get connected.

(3) There is only one F'_l with $\xi(F'_l) = j^*$, assume w.l.o.g. that $\xi(F'_1) = j^*$. In the case $R = R_l \cup R_r$, then all occurrences of vertices from T_{j^*} are connected to v_1 and v_2 in either R_l or R_r and hence they are in the same connected component of R.

(4) It holds that $d(R) \le d(F_{\leftrightarrow})$ and $\sum_{l=2}^{x} w(T_{\xi(F'_l)}) \le c \cdot d(F_{\leftrightarrow})$. In the case that $R = R_l \cup R_r$, we have $d(R) = d(R_l) + d(R_r) \le d(F_{\leftrightarrow})$ and $\sum_{l=2}^{x} w(T_{\xi(F'_l)})$ (which does not include the component that contains v_1 and v_2) can be split up into two parts, such that each part contains exactly the width of those components F'_i that fall into connected components of A that are visited by P_1 or P_2, respectively.

The (partial) solution F' that arises from substituting $F_{\leftrightarrow}$ by R is not necessarily feasible because $F_{\circlearrowright} \cup R$ can consist of multiple connected components. We need to transform R such that all demand-pairs in $F_{\circlearrowright} \cup R$ are connected.

Notice that a demand-pair $u, \bar{u}$ always satisfies $\xi(u) = \xi(\bar{u})$ since A is a feasible solution. Hence, we do not need to connect some connected components with vertices from different T_j, but we may have to connect two connected components F'_{i_1} and F'_{i_2} that are contained within the same connected component T_j of A.

Furthermore, all vertices of F from T_{j^*} are already connected because of Property (3). Fix a $j < j^*$ and consider all connected components F'_l with $\xi(F'_l) = j$. Note that $j < j^*$ implies that the widths of these components are part of $\sum_{l=2}^{x} w(T_{\xi(F'_l)})$

Start with an arbitrary F'_l that contains a terminal $u \in F'_l$ whose mate $\bar{u}$ is in $F'_{l'}$ for some $l' \ne l$ (and $\xi(F'_{l'}) = j$ as supposed above). Connect u to $\bar{u}$ and since $u, \bar{u} \in T_j$, their distance is at most $w(T_j)$. Since $w(T_{\xi(F'_{l'}}) = w(T_j)$, the contribution of $F'_{l'}$ to the sum $\sum_{l=2}^{x} w(T_{\xi(F'_l)})$ is large enough to cover the cost incurred by the connection of the terminals u and $\bar{u}$.

Now, F'_l and $F'_{l'}$ are merged into one component (that lies within T_j). We call this component F'_l. Repeat this process until all terminals in F'_l are connected to their mates while always spending a connection cost that is bounded by the contribution (to the width-sum) of the component that gets merged into F'_l. When F'_l is done, i.e. all terminals are connected to their mates, pick another component and continue in the same fashion.

In the end, the modified set F' is a (partial) feasible solution, and the cost spent for the additional edges is bounded by $\sum_{l=2}^{x} w(T_{\xi(F'_l)}) \le c \cdot d(F_{\leftrightarrow})$. Denote this modification of F' by $\hat{F}$.

The transformation step explained so far can be summarized as follows:

- F is a simple cycle of the feasible solution F.
- We have $F = F_{\circlearrowright} \cup F_{\leftrightarrow}$.
- F' arises from substituting $F_{\leftrightarrow}$ by the set R, i.e. $F' = F_{\circlearrowright} \cup R$.
- $d(R) \le d(F_{\leftrightarrow})$
- $F' = F'_{\circlearrowright}$, but F' is not necessarily a feasible (partial) solution.
- $\hat{F}$ is obtained by adding edges of cost at most $c \cdot d(F_{\leftrightarrow})$ to F'.
- Denote the set of all added edges by E_{add}.
- Therefore $\hat{F} = F' \cup E_{add}$ and $d(E_{add}) \le c \cdot d(F_{\leftrightarrow})$.
- The set $\hat{F}$ is a feasible (partial) solution.

In terms of the total length, this means

$$\begin{aligned} d(\hat{F}) = d(F') + d(E_{add}) &\le d(F') + c \cdot d(F_{\leftrightarrow}) = d(F_{\circlearrowright}) + d(R) + c \cdot d(F_{\leftrightarrow}) \\ &\le d(F_{\circlearrowright}) + d(F_{\leftrightarrow}) + c \cdot d(F_{\leftrightarrow}) = d(F_{\circlearrowright}) + (1+c) \cdot d(F_{\leftrightarrow}) \\ &\le (1+c) \cdot \big(d(F_{\circlearrowright}) + d(F_{\leftrightarrow})\big) = (1+c) \cdot d(F) \end{aligned}$$

We apply this procedure to all components F_i of F with $F_i \neq (F_i)_{\circlearrowright}$ and obtain a solution $\hat{F}$ with $\hat{F} = \hat{F}_{\circlearrowright}$ and $d(\hat{F}) \le (1+c) \cdot d(F)$. Note that A is *edge-edge* and *edge-set swap*-optimal with respect to $\hat{F}$, since A is swap optimal with respect to G and the edges of E_{add} and also the edges in R are from G.

As mentioned at the beginning of the proof, Lemma 3.33 brings in a factor 2 in the cost, hence the final bound for the solution $\hat{F}$ is

$$d(\hat{F}) \le 2 \cdot (1+c) \cdot d(F)$$

and $\hat{F}$ satisfies the desired properties. □

Corollary 3.49. *(Bound for solution A)*
Let $I_m = (G, d, \mathcal{D})$ be an instance of the metric Steiner forest problem with the complete graph $G = (V, E)$, a metric $d : E \to \mathbb{R}_{\ge 0}$ and a set $\mathcal{D} \subseteq V \times V$ of demand-pairs. Let $A, F \subseteq E$ be two feasible Steiner Forest solutions for I_m. Suppose that A is edge-edge, edge-set and path-set swap-optimal with respect to E and ϕ, that A is c-approximate connecting move optimal and that A only uses edges between terminals. Denote by A' the modified solution, where all inessential edges have been removed from A. Then

$$d(A') \le 23 \cdot (1+c) \cdot d(F)$$

Proof. We can apply Lemma 3.48 to the solution forests F and A' and get a solution F' that satisfies $\mathcal{F}' = \mathcal{F}'_{\circlearrowleft}$ and $d(\mathcal{F}') \leq 2 \cdot (1+c) \cdot d(F)$ and such that A' is *edge-edge* and *edge-set swap*-optimal with respect to F'.

This means that there are no F-edges between any components of A', i.e. every connected component of F' lies completely in some connected component A'_j of A'. Since A' does not contain inessential edges, no removing swap is possible which means that A' is removing swap optimal. We can now apply Corollary 3.28 to every connected component A'_j and the part of F' that lies within A'_j. We get

$$d(A') \overset{(Cor.\ 3.28)}{\leq} 11.5 \cdot d(\mathcal{F}') \overset{(Lemma\ 3.48)}{\leq} 23 \cdot (1+c) \cdot d(F)$$

□

The previous results motivate a modified version of Algorithm 2. Note that we can restrict to metric instances since we have shown Theorem 3.31.

Algorithm 4 Local search, second approach

```
Require: An instance $I_m = (G, d, \mathcal{D})$ of the MSFP, a feasible solution $A$ for $I_m$ and a
    constant $c \geq 1$.
Ensure: A solution $A_f$ for the instance $I_m$.
    Let the neighbourhood $\mathcal{N}(A)$ be the set of all solutions that can be obtained by executing
    an edge-edge swap, edge-set swap, path-set swap or a c-approximate connecting move on
    $A$.
    while there exists $A' \in \mathcal{N}(A)$ with $\phi(A') < \phi(A)$ do
        Set $A := A'$.
    end while
    Output the solution $A_f$ that results by applying the clean-up move to $A$.
```

3.2.9. What was shown so far

Corollary 3.49 gives an approximation guarantee for Algorithm 4, which is a modified version of the local search algorithm for the SFP provided in Algorithm 2.

Note that the optimality conditions for solutions by Algorithm 4 are slightly different compared to those in Algorithm 2: In Algorithm 4, we assume that there are no improving c-approximate connecting moves in the solution, whereas in Algorithm 2 we continue with the search for a better solution as long as there are improving *connecting*-moves at all. For $c > 1$, c-approximate connecting move-optimality does not necessarily imply *connecting*-move-optimality, see Definition 3.39 and the comment after the definition.

For c being a constant, the approximation guarantee of Algorithm 4 is $23 \cdot (1+c)$ and hence constant, as shown in Corollary 3.49.

We discussed after the definition of a c-approximate connecting move, that deciding whether there exists a 1-approximate connecting move is not manageable in polynomial time, which results in the same problem we had for our first approach with *connecting*-moves. Of course, we can choose $c = 1$ and obtain a local search algorithm with an approximation guarantee of 46 which is however not running in polynomial time. We will see in the next section, that we can decide whether there exists a 2-approximate connecting move in polynomial time and that this finally helps us to give a polynomial time algorithm that satisfies an approximation guarantee of $69(1+\varepsilon)$ for $\varepsilon > 0$.

Nevertheless, the first statement from our main theorem, namely Theorem 3.1 follows directly from Corollary 3.49 at least for the metric case: There exists a local search algorithm for the MSFP with a constant approximation guarantee.

By applying Theorem 3.31 (Section 3.2.2), we then obtain the same result for the (general) Steiner forest problem and hence the proof of the first part of Theorem 3.1 is now completed. We have now proven, that there exists a local search algorithm for the SFP with a constant approximation guarantee.

4. Analysis of the local search algorithm: time complexity

The previous two sections analysed the algorithm regarding approximation guarantee but did not bother about the running time or convergence. So far, we have seen that there is a local search algorithm that has a constant approximation guarantee, namely Algorithm 2 with a slight changing of the *connecting* moves to c-approximate connecting moves.

In this section, we discuss the time complexity of the algorithm and fix some problematic points. The complexity analysis of Algorithm 2 is simple: The algorithm does not run in polynomial time since in general exponentially many *connecting* moves need to be checked. A modification of Algorithm 2 so as to obtain a local search algorithm which runs in polynomial time and has a constant approximation guarantee would need to address the following crucial aspects.

Aspect 1: Choose a neighbourhood structure so as to be able to optimize over it in polynomial time. In Section 3.2 we have seen, that restricting Algorithm 2 to c-approximate connecting moves suffices to obtain constant approximation guarantee. In the following we show that this move can be handled in polynomial time. To do so, we show that finding an c-approximate connecting move reduces to approximating the k-MST problem and show then that the k-MST problem can be approximated to a constant factor in polynomial time.

Aspect 2: The number of iterations, i.e. the improving local search steps performed by the algorithm should be polynomial in the size of the instance. We show that for our local search algorithm, convergence to a local minimum can be achieved by a standard rounding technique of the edge lengths with a loss of a factor $(1+\varepsilon)$ in the approximation guarantee for any $\varepsilon > 0$.

4.1. k-MST Problems

Let's consider the following minimization problems and their relations:

Definition 4.1. *((rooted) k-MST problem)*
Let $G = (V, E)$ be a graph with a root $r \in V$, a metric $d : V \times V \to \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$.
Task: Find a tree T in G with $r \in V[T]$ and $|V[T]| \geq k$ that minimizes $\sum_{e \in E[T]} d(e)$.
The weighted unrooted k-MST problem is defined in the same way except for the fact that no distinguished root has to be part of the tree.

Comments

- Fischetti et al. [F94] have shown that the unrooted k-MST problem is NP-hard.

- Any algorithm for the rooted k-MST transfers to an algorithm for the unrooted k-MST with the same approximation guarantee: Apply the rooted k-MST algorithm for all possible vertices as roots all possible vertices as the node and return the best solution found. In particular, this holds for optimal algorithms, hence the rooted k-MST problem is also NP-hard.

- Vice versa, algorithms for the unrooted k-MST can be used for the rooted k-MST without any change of the approximation guarantee: Create n vertices with distance zero to the root vertex and search for a tree with $n + k$ vertices. Any such tree T has to include the root r and at least $k - 1$ other vertices. By removing from T the newly created vertices together with their incident edges, we obtain a tree T' containing r and at least $k - 1$ original vertices and having the came cost as T.Thus, any solution for the unrooted k-MST problem is feasible for the rooted k-MST problem and has the same cost [Gar05].

- Thus that the rooted and unrooted version of the k-MST problem are equivalent.

- There are several polynomial time algorithms with constant approximation guarantee for the k-MST problem:
  - Blum, Ravi and Vempala [BRV96] in 1996, approximation factor: 17
  - Garg [Gar96] in 1996, approximation factor: 3
  - Arya and Ramesh [AR98] in 1998, approximation factor: 2.5
  - Arora and Karakostas [AK00] in 2000, approximation factor: $2 + \epsilon$
  - Garg [Gar05] in 2005, approximation factor: 2

Definition 4.2. *(weighted Γ-MST problem)*
Let $G = (V, E)$ be a graph with a metric $d : V \times V \to \mathbb{R}_{\geq 0}$, a function $\gamma : V \to \mathbb{R}_{\geq 0}$ and $\Gamma \in \mathbb{R}_{\geq 0}$. Task: Find a tree T in G with $\sum_{v \in V[T]} \gamma(v) \geq \Gamma$ that minimizes $\sum_{e \in E[T]} d(e)$.

Definition 4.3. *(weighted rooted Γ-MST problem)*
Let $G = (V, E)$ be a graph with a root $r \in V$, a metric $d : V \times V \to \mathbb{R}_{\geq 0}$, a function $\gamma : V \to \mathbb{R}_{\geq 0}$ and $\Gamma \in \mathbb{R}_{\geq 0}$. Task: Find a tree T in G with $r \in V[T]$ and $\sum_{v \in V[T]} \gamma(v) \geq \Gamma$ that minimizes $\sum_{e \in E[T]} d(e)$.

Comments

- It's easy to see that the unweighted version is a special case of the weighted version: Set $\Gamma = k$ and $\gamma(v) = 1$ for all vertices $v \in V$.

- For a better understanding, we call the weighted Γ-MST problem also weighted k-MST problem, i.e. we give Γ the role of k.

- Johnson, Minkoff and Phillips [JMP00] observe the following reduction from the weighted k-MST problem to the unweighted k-MST problem for the case that all $\gamma(v)$ are integers: To create an unweighted instance, start with the vertex set V of the weighted instance, and for each $v \in V$, add $2\gamma(v)n - 1$ vertices with distance zero to v to the graph. This means, there are finally $2\gamma(v)n$ vertices "at" v. We set $k = 2n\Gamma$.Any solution for the unweighted instance can be changed to a solution such that for any vertex $v \in V$ in the weighted instance, either "all" $2\gamma(v)n$ copies of v in the unweighted graph are chosen or none of them. This change does not alter the cost, since picking more vertices at the same location does not increase the cost. Then a solution of the weighted k-MST problem is constructed by picking an original vertex $v \in V$ if and only if all corresponding vertices in the (modified) solution for the unweighted case are selected. This reduction constructs an input for the unweighted k-MST problem that is of pseudo-polynomial size. However, Johnson et al. [JMP00] note that algorithms for the unweighted k-MST problem can typically be adapted to handle the "clouds" of vertices at the same location implicitly without incurring a super-polynomial running time. They specifically state that this is true for the 3-approximation by Garg [Gar96] from 1996 for the rooted k-MST problem.

- Due to a personal communication of the authors of Gross et al. [G17] with Naveen Garg in 2016, the more recent 2-approximation by Garg from 2005 [Gar05] for the k-MST problem can be adapted for the weighted k-MST problem such that the running time is independent of the weights.

4.2. c-approximate connecting move optimality

Let's see now how the weighted rooted k-MST problem helps dealing with the two aspects mentioned in the introduction of Section 4.

Theorem 4.4.
Let I_m be an instance of the MSFP and assume that there exists $\beta > 0$ such that $\forall e \in E : \exists\, l_e \in \mathbb{N} : d(e) = l_e \cdot \beta$. Assume we are given an algorithm Tree-Approx that computes a c-approximation for the weighted rooted k-MST problem. Then we can find an improving connecting move in polynomial time, if such one is existing, or guarantee, that there is no $((1+\varepsilon)\cdot c)$-approximate connecting move at all.

Proof. Let A be a feasible solution forest for the instance I_m. We apply *Tree-Approx* to the graph G_A^{all}, with vertex set $\{1, \dots, p\}$, corresponding to the connected components $A_1, \dots, A_p$ of A. Each connecting move connects some of the p components of A. Let A_i be the connected component with the largest width among all connected components which will get connected. Clearly, i can take $|V[G_A^{all}]| = p$ possible values. Since the width of the components A_i increases with their indices, all vertices from G_A^{all} with indices larger than i can be deleted. Then we set $\gamma(i) := 0$ and $\gamma(j) := w(A_j)$ for $j < i$.

By choosing some of the remaining vertices in G_A^{all}, we can collect prices (the values $\gamma(v)$) between $w_{min} := min\{w(A_i) \mid i \in \{1, \dots, p\}, w(A_i) > 0\}$ and $\sum_{j=1}^{i-1} w(A_j) < p \cdot w(A_p)$.
Now we call *Tree-Approx* for $\Gamma = (1 + \frac{\varepsilon}{2})^l \cdot w_{min}$ for all $l \geq 1$ until

$$\left(1 + \frac{\varepsilon}{2}\right)^l \cdot w_{min} \geq p \cdot w(A_p) \quad \Longleftrightarrow \quad l \geq log_{1+\frac{\varepsilon}{2}}\left(p \cdot \frac{w(A_p)}{w_{min}}\right)$$

Considering all p possible choices for the component A_i as described above, the total number m of *Approx-Tree* calls is bounded by

$$m \leq p \cdot log_{1+\frac{\varepsilon}{2}}\left(p \cdot \frac{w(A_p)}{w_{min}}\right) \leq n \cdot log_{1+\frac{\varepsilon}{2}}\left(n \cdot \Delta\right)$$

where Δ is the largest distance between any terminal and its partner divided by the smallest such distance. The value Δ is polynomial in the input because $w(A_P) \leq n \cdot max\{d_e : e \in E\}$

and $w_{min} \geq \beta$. Therefore, also m is polynomial in the input. If one of the *Tree-Approx* calls returns a solution $T_j, 1 \leq j \leq m$ such that

$$\sum_{e \in E[T_j]} d(e) < \sum_{v \in V[T_j]} \gamma(v)$$

then T_j induces an improving connecting move:

$$d(T_j) = \sum_{e \in E[T_j]} d(e) < \sum_{v \in V[T_j]} \gamma(v) = w(A) - w(A \cup T) \tag{6}$$

and therefore

$$\phi(A \cup T) = \phi(A) + \underbrace{d(T_j) - w(A) + w(A \cup T)}_{< 0 \text{ by } (6)} < \phi(A)$$

Now assume that *Tree-Approx* returns solutions $T_j, 1 \leq j \leq m$ with

$$\sum_{e \in E[T_j]} d(e) \geq \sum_{v \in V[T_j]} \gamma(v) \tag{7}$$

for all calls $1 \leq i \leq l$. By the definition of γ and the *connecting* move, this means that none of the obtained trees induces an improving *connecting* move. We will show that there does not exist a $((1+\varepsilon)\cdot c)$-approximate connecting move in this case. For a contradiction, assume that there exists a $((1+\varepsilon)\cdot c)$-approximate connecting move T^*. Let

$$i^* := arg\ max\{i : A_i \in V[T^*]\} \qquad \text{and} \qquad \gamma_i := \begin{cases} w(A_i) & i < i^* \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Then the definition of the $((1+\varepsilon)\cdot c))$-approximate connection move T^* implies the following inequality:

$$\sum_{e \in E[T^*]} d(e) < \frac{1}{c \cdot (1+\varepsilon)} \sum_{v \in V[T^*]} \gamma(v)$$

Set $\Gamma^* := \sum_{v \in V[T^*]} \gamma(v) \leq p \cdot w(A_p)$. Let l' be the index that satisfies

$$(1+\frac{\varepsilon}{2})^{l'} \cdot w_{min} \leq \Gamma^* < (1+\frac{\varepsilon}{2})^{l'+1} \cdot w_{min}$$

Consider the application of *Tree-Approx* on the instance I_Γ of the weighted Γ-MST with input G_A^{all}, d, $r = i^*$ and γ_i as defined in Equation (8) and $\Gamma = \Gamma' := (1+\frac{\varepsilon}{2})^{l'} \cdot w_{min}$. The above inequalities apply that $\Gamma^* \geq \Gamma'$.

Notice that T^* is a feasible solution for this instance I_Γ since $\sum_{v \in V[T^*]} \gamma(v) = \Gamma^* \geq \Gamma$ satisfies the lower bound. Since any feasible solution gives an upper bound for an optimal solution, we conclude that the cost of an optimal solution T^{OPT} for this instance I_Γ satisfies

$$\sum_{e \in E[T^{OPT}]} d(e) \leq \sum_{e \in E[T^*]} d(e)$$

The algorithm *Tree-Approx* computes a c-approximation, i.e. a solution $\hat{T}$ with $\sum_{v\in V[\hat{T}]}\gamma(v) \geq \Gamma'$ and

$$\begin{aligned}
\sum_{e\in E[\hat{T}]} d(e) &\leq c \cdot \sum_{e\in E[T^{OPT}]} d(e) \leq c \cdot \sum_{e\in E[T^*]} d(e) \\
&\leq c \cdot \frac{1}{c\cdot(1+\varepsilon)} \sum_{v\in V[T^*]} \gamma(v) = \frac{c}{c\cdot(1+\varepsilon)} \cdot \Gamma^* \\
&\leq (1+\frac{\varepsilon}{2}) \cdot \frac{c}{c\cdot(1+\varepsilon)}\Gamma' < \Gamma' \\
&\leq \sum_{v\in V[\hat{T}]} \gamma(v)
\end{aligned}$$

which means that *Tree-Approx* computes an improving *connecting* move for this special setting, which is a contradiction to our assumption corresponding to Equation (7). □

Corollary 4.5.
Consinder an $\varepsilon > 0$ and a polynomial time c-approximation algorithm for the weighted rooted k-MST problem. Then there exists a polynomial time algorithm called Improving-Connecting-Move (ICM) which takes as input an instance of the metric SFP where all edge lengths are an integer multiple of some constant $\beta > 0$ together with a feasible solution A for that instance and outputs an improving connecting move w.r.t. ϕ if a $((1+\varepsilon)\cdot c)$-approximate connecting move exists.

Proof. This follows directly from the statement and the proof of Theorem 4.4. ICM is described in the proof of Theorem 4.4. □

Comment
This result shows that every iteration of Algorithm 4 can be performed in polynomial time at least for $c \geq 2$ and for the case, that all edge lengths are an integer multiple of some constant $\beta > 0$ (cf. Aspect 1 at the introduction of Section 4). What remains to show is that the number of iterations, i.e. the number of improving steps in Algorithm 4 itself is of polynomial size, which is done in the following section.

4.3. Convergence in polynomial time

We apply a standard rounding technique to the edge lengths in order to make the presented local search algorithm running in polynomial time. Therefore, some definitions are necessary.

Definition 4.6. *(d_β and ϕ_β)*
Consider an instance $I_m = (G, d, \mathcal{D})$ of the MSFP with $G = (V, E)$. The metric $d : E \to \mathbb{R}_{\geq 0}$ can be seen as a metric $d : V \times V \to \mathbb{R}_{\geq 0}$ where $d(u, v) := d(e)$ for $e = \{u, v\}$. For $\varepsilon > 0$ and $u, v \in V$, we set

$$\beta := \frac{\varepsilon \cdot max_{\{u,\bar{u}\}\in\mathcal{D}} d(u,\bar{u})}{|E|} \qquad\qquad d_\beta(u,v) := \left\lceil \frac{d(u,v)}{\beta} \right\rceil \cdot \beta$$

Analogous to Definition 1.8, we set

$$w_\beta(E') := max\{dist_{d_\beta}(s,t) \mid \{s,t\} \in \mathcal{D}, \{s,t\} \subset V[E']\}$$

for any connected $E' \subseteq E$ where $dist_{d_\beta}$ is the shortest path distance in G with respect to d_β and for a forest $F \subseteq E$ with connected components $F_1, \ldots, F_l \subseteq F$,

$$w_\beta(F) := \sum_{i=1}^{l} w_\beta(F_i) \qquad\qquad \phi_\beta(F) := d_\beta(F) + w_\beta(F)$$

Lemma 4.7. *(d_β is a metric)*
With the settings as in Definition 4.6, the function d_β defines a metric on $V \times V$.

Proof. Let $u, v, w \in V$. It is easy to see that $d_\beta(u,v) \geq 0$ since $\beta > 0$ and $d(u,v) \geq 0$. Moreover,

$$d_\beta(u,v) = 0 \iff d(u,v) = 0 \iff u = v$$

since d is a metric. This shows the identity of indiscernibles and that d_β is positive definite. The next equation shows the symmetry of d_β. The second equality holds since d is symmetric.

$$d_\beta(u,v) = \left\lceil \frac{d(u,v)}{\beta} \right\rceil \cdot \beta = \left\lceil \frac{d(v,u)}{\beta} \right\rceil \cdot \beta = d_\beta(v,u)$$

Last but not least, the triangle inequality for d_β holds since

$$\begin{aligned} d_\beta(u,w) = \left\lceil \frac{d(u,w)}{\beta} \right\rceil \cdot \beta &\leq \left\lceil \frac{d(u,v) + d(v,w)}{\beta} \right\rceil \cdot \beta = \left\lceil \frac{d(u,v)}{\beta} + \frac{d(v,w)}{\beta} \right\rceil \cdot \beta \\ &\leq \left(\left\lceil \frac{d(u,v)}{\beta} \right\rceil + \left\lceil \frac{d(v,w)}{\beta} \right\rceil \right) \cdot \beta = d_\beta(u,v) + d_\beta(v,w) \end{aligned}$$

where the first inequality holds since the triangle inequality holds for d. □

We can now reformulate Algorithm 4:

Algorithm 5 Local search, polynomial time

```
Require: An instance $I_m = (G, d, \mathcal{D})$ of the MSFP with $G = (V, E)$ being the complete
    graph and $A$ being a solution obtained by connecting each demand-pair by a direct edge
    and deleting an edge from every cycle that appears. This gives a feasible solution for $I_m$.
    Furthermore let $\varepsilon > 0$ be given.
Ensure: A solution $A_f$ to the instance $I$.
    Set $i := 0$ and let $A_0 := A$
    Set $\beta := \frac{\varepsilon \cdot max_{\{u,\bar{u}\} \in \mathcal{D}} d(u,\bar{u})}{|E|}$ and $d_\beta(e) := \left\lceil \frac{d(e)}{\beta} \right\rceil \cdot \beta$
    while $A_i$ admits an improving edge-edge swap, edge-set swap or path-set swap with respect
    to $\phi_\beta$, or the algorithm ICM finds an improving connecting move with respect to $\phi_\beta$ do
        Set $A_{i+1}$ to be the resulting solution after applying the move
        Set $i := i + 1$
    end while
    Output the solution $A_f$ that results by applying the clean-up-move to $A_i$
```

Theorem 4.8.
Assume that the locality gap for swap-optimal and c-approximate connection move optimal solutions is C and let $\varepsilon > 0$. Then Algorithm 5 computes in polynomial time a $(1 + \varepsilon) \cdot C$-approximation for the MSFP, and hence also for the general Steiner forest problem.

Proof. First we concentrate on the runtime: We claim that the algorithm runs in polynomial time. To see this, first note that $d_\beta(e) = \beta \cdot \ell_e$ with $\ell_e \in \mathbb{N}$ for all $e \in E$. Therefore, every improving *swap*-move and every successful run of ICM decreases the potential ϕ_β by at least β, i.e. $\phi_\beta(A_{i+1}) \leq \phi_\beta(A_i) - \beta$. We started with a subgraph of the solution, that connects all demand-pairs by the direct edge[14] and therefore, with n_t being the number of demand-pairs,

$$\phi_\beta(A_0) \leq 2 \cdot \sum_{\{u,\bar{u}\} \in \mathcal{D}} d_\beta(u, \bar{u}) \leq 2 \cdot n_t \cdot \max_{\{u,\bar{u}\} \in \mathcal{D}} d(u, \bar{u}) = \frac{2 \cdot n_t \cdot |E|}{\varepsilon} \cdot \beta$$

since of the trivial setting of β in Algorithm 5. This means that the algorithm terminates after at most $2 \cdot \frac{n_t \cdot |E|}{\varepsilon}$ iterations. Since each iteration can be executed in polynomial time, we get an overall polynomial time algorithm.

Consider the output A_f of the algorithm. It is *path-set swap* optimal (which implies also *edge-edge swap*- and *edge-set swap*-optimality) and c-approximate connecting move optimal with respect to the corresponding potential ϕ_β. The assumption on the locality gap implies that $d_\beta(A_f) \leq C \cdot d_\beta(F_\beta)$, where F_β is the optimal solution of the Steiner Forest instance defined by the metric d_β. Furthermore it holds that $d_\beta(F_\beta) \leq d_\beta(F)$ for any optimal solution F of the original instance defined by the metric d.

[14]Remember that for the MSFP we always assumed that $G = (V, E)$ is the complete graph.

We observe that

$$\begin{aligned} d(A_f) &\le d_\beta(A_f) \le C \cdot d_\beta(F_\beta) \le C \cdot d_\beta(F) = C \cdot \sum_{e \in F} \left\lceil \frac{d(e)}{\beta} \right\rceil \cdot \beta \\ &\le C \cdot \sum_{e \in F} \left(\frac{d(e)}{\beta} + 1 \right) \cdot \beta \le C \cdot \big(d(F) + |F| \cdot \beta \big) \le C \cdot \big(d(F) + |E| \cdot \beta \big) \\ &= C \cdot \big(d(F) + \varepsilon \cdot \max_{\{u,\bar{u}\} \in \mathcal{D}} d(u, \bar{u}) \big) \le C \cdot \big(d(F) + \varepsilon \cdot d(F) \big) = (1+\varepsilon) \cdot C \cdot d(F) \end{aligned}$$

which finally proves the whole statement. □

4.4. Summary

Due to Garg [Gar05] and a personal communication between the authors of [G17] with Garg, there is a polynomial time 2-approximation for the weighted k-MST problem and hence Theorem 4.4 provides a polynomial time algorithm that can guarantee $2(1+\varepsilon)$-approximate connecting move optimality for every $\varepsilon > 0$.

By using this result in the assumptions of Corollary 3.49 we get a locality gap of $23(1 + C(1+\varepsilon)) = (69 + 46\varepsilon)$. Note that we do not know if we can decide 2-approximate connecting move optimality in polynomial time, so we have to consider $2(1+\varepsilon)$-approximate connecting move optimality.

Taking $C = (69+46\varepsilon)$ and ε as above in the conditions of Theorem 4.8, we obtain an algorithm that computes in polynomial time a solution within a factor of $(1+\varepsilon)(69+46\varepsilon) = 69(1+\hat{\varepsilon})$ with $\hat{\varepsilon} = \frac{113}{69}\varepsilon + \frac{46}{69}\varepsilon^2 \le 3\varepsilon$ for $\varepsilon < 1$. So $\hat{\varepsilon}$ is small if ε is small.

By summarizing we obtain the following result:

Theorem 4.9.
For every $\varepsilon > 0$ there is a local search algorithm that computes in polynomial time a solution A to an SFP instance I such that

$$d(A) \le (1+\varepsilon) \cdot 69 \cdot d(F)$$

where F is an optimal solution for I.

Theorem 4.9 clearly implies Theorem 3.1.

5. Two more approaches for SFP

In Part II, we implement a modified version of the local search algorithm (Algorithm 2). In order to compare the behaviour of the implementation with respect to running time and solution quality, we consider two more approaches for the SFP. The implementation of both has been done in the Master's Seminar report of Stefan Golja [Gol18].

5.1. The Gluttonous Algorithm

Consider the approximation algorithm “Gluttonous” for the Steiner forest problem. It's a classical greedy algorithm of purely combinatorial nature. We give a short overview about the preliminaries, the algorithm itself and the analysis of the algorithm below. For details we refer to the paper *Greedy Algorithms for Steiner Forest* of Anupam Gupta and Amit Kumar [GK14], in which the Gluttonous algorithm was presented and analysed. Some more detailed information, implementation details and examples can be found in the Master's Seminar report of Stefan Golja [Gol18].

We will use this implementation provided by [Gol18] to compare the performance of the local search algorithm introduced in this Master thesis with the Gluttonous algorithm, at least for some smaller instances that can be handled by both algorithms in manageable time.

Definition 5.1. *(Metric space)*
Let V be a vertex set and $d : V \times V \to \mathbb{R}_{\geq 0}$ be a metric. We call $\mathcal{M} = (V, d)$ a metric space. Hence, an instance of the metric Steiner forest problem as described in Definition 3.30 can be given by a metric space $\mathcal{M}$ and a set of demand-pairs $\mathcal{D}$.

Definition 5.2. *(Supernodes, clustering, active nodes, inter-supernode edges)*
Let $I = (\mathcal{M}, \mathcal{D})$ be an instance of the MSFP. A supernode S is a subset of terminals. A clustering $\mathcal{C} = \{S_1, \ldots, S_q\}$ is a partition of the terminals into supernodes. By the trivial clustering we mean the clustering where each terminal builds a supernode on its own. Given a clustering, we call a terminal u active if it belongs to a supernode S that does not contain its mate $\bar{u}$. A supernode S is active if it contains some active terminal. An inter-supernode edge is an edge with endpoints belonging to different supernodes.

Definition 5.3. *($d_{\mathcal{M}/\mathcal{C}}$)*
Let $I = (\mathcal{M}, \mathcal{D})$ be an instance of the MSFP and $\mathcal{C} = \{S_1, \ldots, S_q\}$ be a clustering.
Take the complete graph on the vertex set V, for an edge $\{u, v\}$ set the length $d'(u, v) := d(u, v)$ if u and v are both terminals lying in different supernodes in $\mathcal{C}$ or if at least one of them is a non-terminal. If u and v are both terminals and lie in the same supernode, set the length $d'(u, v) := 0$. We call this graph $G_{\mathcal{C}}$ and define the $\mathcal{C}$-punctured distance to be the shortest path distance in $G_{\mathcal{C}}$ with respect to d', which we denote by $d_{\mathcal{M}/\mathcal{C}}(\cdot, \cdot)$. The extension of $d_{\mathcal{M}/\mathcal{C}}(\cdot, \cdot)$ to supernodes S_i and S_j is defined in a natural way by

$$d_{\mathcal{M}/\mathcal{C}}(S_i, S_j) := d_{\mathcal{M}/\mathcal{C}}(u, v) \ for\ any\ u \in S_i,\ v \in S_j \tag{9}$$

Comment
Since the length $d'(e)$ of an edge $e = (u, v)$ in $G_{\mathcal{C}}$ that connects two vertices of the same supernode is zero, it does not matter which specific vertices $u \in S_i$ and $v \in S_j$are chosen for the calculation of $d_{\mathcal{M}/\mathcal{C}}(S_i, S_j)$ in (9).

In the following we include the pseudo-code of the Gluttonous algorithm.

Algorithm 6 The Gluttonous algorithm

```
Require: An instance $I = (\mathcal{M}, \mathcal{D})$ of the metric Steiner forest problem.
Ensure: A solution $A$ to the instance $I$.
1: Let $\mathcal{C}$ being the trivial clustering and $E'$ being the empty set.
2: while there exists active supernodes in $\mathcal{C}$ do
3:     Calculate all $\mathcal{C}$-punctured distances.
4:     Find active supernodes $S_1, S_2$ in $\mathcal{C}$ with minimum $\mathcal{C}$-punctured distance.
       Break ties by choosing the lexicographically smallest pair.
5:     Update the clustering to $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{S_1, S_2\}) \cup \{S_1 \cup S_2\}$.
6:     Add to $E'$ the edges corresponding to the inter-supernode edges on the shortest path
       between $S_1$ and $S_2$ in the graph $G_{\mathcal{C}}$.
7: end while
8: Output a maximal acyclic subgraph $A$ of $E'$.
```

Some comments about Algorithm 6:

- The sum of the lengths of the edges added in Step 6 is equal to $d_{\mathcal{M}/\mathcal{C}}(S_1, S_2)$. This sum is called the merging distance of the corresponding step.
- The algorithm maintains the following invariant: If S is a supernode, then the terminals in S lie in the same connected component of A.
- The algorithm terminates when there are no more active terminals, so each terminal u shares a supernode with its mate $\bar{u}$, hence the final forest A connects all demand-pairs.
- Since the total length of the edges in E' is at most the sum of the merging distances and we output a maximal sub-forest of A, we get that the cost of the obtained solution is at most the sum of all merging distances.

Theorem 5.4. *(Approximation factor for Gluttonous)*
The Gluttonous algorithm is a constant-factor approximation for the metric Steiner forest problem that runs in polynomial time. In particular, Gluttonous is a polynomial time 96-*approximation of the MSFP.*

5.2. SFP as an integer program

With the help of the previous section, we are able to compare the implemented version of Algorithm 2 with another approximation algorithm. Here, we introduce an integer linear programming (ILP) of the SFP which allows us to compare to optimal solutions for instances of small size. The ILP is solved by some standard solver like *Gurobi*.

We use the ILP - formulation IPuf of Schmidt, Zey and Margot [SZM17] as described below. For this, remember some basics from graph theory: A cut-set in G is a subset $S \subseteq V$ of vertices of G. Any cut-set induces a cut $\delta(S) := \{\{i,j\} \in E \mid |\{i,j\} \cap S| = 1\}$. If $S = \{i\}$ is a singleton, we abbreviate $\delta(i) := \delta(\{i\})$. For a directed graph $D = (V, A)$, we distinguish between the outgoing cut $\delta^+(S) := \{(i,j) \in A \mid i \in S \text{ and } j \in V \setminus S\}$ and the incoming cut $\delta^-(S) := \{(i,j) \in A \mid i \in V \setminus S \text{ and } j \in S\}$. Note that we can always transform an undirected graph into an directed graph by replacing each edge $\{i,j\}$ by the two arcs (i,j)

and (j,i) with costs $c((i,j)) = c((j,i)) := c(\{i,j\})$.

For the ILP - formulation IPuf, we start with a slightly generalized form of the SFP compared to that in Definition 1.7: Given an undirected graph $G = (V, E)$ with edge lengths c_e for every edge $e \in E$ and terminal sets $T^1, \dots, T^K \subseteq V$, the task is to find a cycle-free subgraph of G of minimum total length $d(E(G))$ in which the vertices in each terminal set are connected.

A feasible forest $F = (V_F, E_F)$ for $(G, T^1, \dots, T^K)$ is a subgraph of G which is a forest such that for all $1 \leq k \leq K$ and for all $s, t \in T^k$, there exists an $s-t$ path in F. Without loss of generality, we can assume that the terminal sets are pairwise disjoint: if T^k and T^l for some $1 \leq k < l \leq K$ share at least one vertex, then a forest F is feasible for $(G, T^1, \dots, T^K)$ if and only if it is feasible for $(G, T^1, \dots, T^{k-1}, T^k \cup T^l, T^{k+1}, \dots, T^{l-1}, T^{l+1}, \dots T^K)$. We denote the set of all terminal nodes by $\mathcal{I} := T^1 \cup \cdots \cup T^K$ and denote by $\tau(t) := k$ to denote the index of the unique terminal set that contains the terminal $t \in \mathcal{I}$. For each terminal set T^k we select an arbitrary vertex $r^k \in T^k$ as a fixed root vertex and define $\mathcal{R} := \{r^1, \dots, r^k\}$ to be the set of all root vertices.

To formulate the SFP as an integer linear program, we transform the undirected graph G into a directed graph and introduce a binary variable x_{ij} for each edge $\{i,j\}$. Moreover, we define two flow variables f^t_{ij}, f^t_{ji} for each non-root terminal $t \in \mathcal{I} \setminus \mathcal{R}$. Then, a selection of edges induced by the x_{ij} variables forms a feasible Steiner forest, if it allows us to send one unit of flow from the k-th root r^k to any terminal $t \in T^k$ for all $k \in \{1, \dots, K\}$. This leads to the following formulation:

$$\min \sum_{\{i,j\} \in E} c(\{i,j\}) \cdot x_{ij} \tag{IPuf}$$

such that

$$\sum_{\{i,j\} \in \delta^+(i)} f^t_{ij} - \sum_{\{i,j\} \in \delta^-(i)} f^t_{ij} = \begin{cases} 1 & \text{if } i = r^{\tau(t)} \\ -1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases} \qquad \forall\, i \in V \;\; \forall\, t \in \mathcal{I} \setminus \mathcal{R} \tag{1a}$$

$$f^t_{ij} + f^t_{ji} \leq x_{ij} \qquad \forall\, \{i,j\} \in E \;\; \forall\, t \in \mathcal{I} \setminus \mathcal{R} \tag{1b}$$

$$f^t_{ij}, f^t_{ji} \in \{0,1\} \qquad \forall\, \{i,j\} \in E \;\; \forall\, t \in \mathcal{I} \setminus \mathcal{R} \tag{1c}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, \{i,j\} \in E\} \tag{1d}$$

Part II.
Implementation

6. Basic ideas and questions

Hard- and software
The algorithm was implemented with VisualStudio 2017 in the language *C++*.
We use an ASUS R556U with Intel Core i5-6200U processor to execute the algorithm.

Comment
Every time we use the phrase "the Local Search algorithm", we mean the local search algorithm described in Algorithm 5, or later, the modified version Algorithm 7.

6.1. Identifying problematic aspects

Recall the local search algorithm defined at the end of Part I:

Algorithm 5 Local search, polynomial time

```
Require: An instance $I_m = (G, d, \mathcal{D})$ of the MSFP with $G = (V, E)$ being the complete
    graph and $A$ being a solution obtained by connecting each demand-pair by a direct edge
    and deleting an edge from every cycle that appears. This gives a feasible solution for $I_m$.
    Furthermore let $\varepsilon > 0$ be given.
Ensure: A solution $A_f$ to the instance $I$.
    Set $i := 0$ and let $A_0 := A$
    Set $\beta := \frac{\varepsilon \cdot max_{\{u,\bar{u}\} \in \mathcal{D}} d(u,\bar{u})}{|E|}$ and $d_\beta(e) := \left\lceil \frac{d(e)}{\beta} \right\rceil \cdot \beta$
    while $A_i$ admits an improving edge-edge swap, edge-set swap or path-set swap with respect
    to $\phi_\beta$, or the algorithm ICM finds an improving connecting move with respect to $\phi_\beta$ do
        Set $A_{i+1}$ to be the resulting solution after applying the move
        Set $i := i + 1$
    end while
    Output the solution $A_f$ that results by applying the clean-up-move to $A_i$
```

Before implementing the heart of the algorithm, we need to define, which data types and types of classes will be used. A bunch of question arises in this context, especially how

... to specify the input?
... many different types of input should we handle and what structure do they have? Do we need some transformation so that we can apply Algorithm 5?
... does an instance look like? Which information has to be present?
... to store the solution that gets modified in each step?
... to initialize this solution?
... to implement the *swaps*?
... to implement the *Improving-Connecting* move?
... to implement the *clean-up* move?
... to calculate $R(e, f)$ for some edges e and f?
... to handle situations, where trees may get connected or disconnected?
... to deal with the rounding technique for the edge lengths?

In the following we will address this kind of issues.

6.2. A concept for the implementation

Different types of input instances
There are basically two possible ways to get an instance. We can read the necessary information from a *.txt*-file or create a random instance. The following input file formats are supported:

In the general case, the (connected) graph is given by its number of vertices, its edges together with lengths and the demand-pairs. The general input structure looks as follows:

Number of points = vertices n (leads to point names/numbers $1, \ldots, n$)
Number of edges m
Edges (in form of first vertex, second vertex, length)
Number of demand-pairs k
Demand-pairs (in form of k pairs of vertices)

In many applications, the graph is given by a number of vertices in the Euclidean plane and edges between any pair of vertices. This are the so called Euclidean instances. The lengths attached to each edge are the Euclidean distances that can be computed easily by knowing all coordinates of the vertices. In this case, the structure of the input file looks as follows:

Number of points n (leads to point names/numbers $1, \ldots, n$)
Coordinates of all points (in form of pairs of real values)
Number of demand-pairs k
Demand-pairs (in form of k pairs of points)

For testing purposes, we will also consider randomly generated instances. We implement a tool to get a random Euclidean instance with a given number of vertices and demand-pairs as well as limits for x- and y-coordinates.

In order to evaluate the quality of the solutions generated by the implemented algorithm, we will make use of some instances out of the *SteinLib - Library* (see Section 10 for further information), where some special kind of input format is used. We provide the possibility to handle the format of those kind of input data from a *.txt* file.

Transformation of an input instance
The local search algorithm is applied to instances which have some particular properties: a) the underlying graph has to be complete, b) the lengths have to obey a metric and c) the demand-pairs has to be ordered in such a way, such that their shortest path distances in the original graph are non-decreasing. This is achieved by applying the transformation described in Theorem 3.31 and by reordering the set of demand-pairs. Note, that we have to store the shortest paths between any two vertices so as to be able to "re-transform" a solution, i.e. to expand an edge that represents a shortest path.

Storing the instance
For storing the whole instance, an appropriate class will be necessary. The instance is entirely described by the number of points, the distance matrix, the number of demand-pairs and the demand-pairs. In order to handle this situation in a comfortable way, we will implement a simple class for matrices (since there is no data type"matrix" in C++) and also a class for a triple of values. Note that there is no need to change the underlying data type while doing the transformation described above.

A concept for the storage of the solution
The easiest way to store the solution would be to store its adjacency matrix. The problem with that approach is, that we need information about the trees of the forest to perform the local moves. Therefore, it is easier to develop a more sophisticated model in order to do the necessary operations without much effort. The idea is the following: We design a class that represents a tree, it should bare information about the member vertices, its edges and additional information like the total length and the potential. It should be easy to execute operations as removing or adding an edge.

Having established a class for the trees, we go one step further and develop a class for a forest. Basically, it should be a conglomeration of trees. The goal is to handle the following situations without much effort during the algorithm: Query in which tree a given vertex is contained, check whether two vertices are contained within one single tree, add an edge to the forest, connect two trees, delete edges of a single tree and update this tree (it may be the case that the tree splits up in many components) or add a vertex to a tree. In addition to that, we should be able to check if a forest is feasible, i.e. if each demand-pair is contained within one single tree.

An initial solution to start
Each local search algorithm has to start with a feasible solution. In our case, we start with the following solution: Since the underlying graph is the complete graph, we can connect each terminal by the direct edge to its mate. The obtained graph may contain cycles, hence we delete an edge from each cycle (take an edge of highest cost) to obtain a feasible solution.

For technical reasons, we perform the steps above only with the help of the adjacency matrix. After that, we build the initial forest as an instance of the corresponding forest - data type based on the given adjacency matrix: We use a DFS - approach to determine the connected components, then create an instance of the corresponding tree - data type for each component and gather all trees in the forest - instance.

The *edge-edge swap*
The idea how to handle the *edge-edge swaps* is simple: For each tree T, we take any pair of vertices $u, v \in T$ (remember that we deal with the complete graph) and connect it by the direct edge $e = \{u, v\}$, if this edge is not yet present in the current solution-forest. Before adding this edge e, we determine the unique shortest $u - v$ path $\mathcal{P}$ in the current solution-forest. The path $\mathcal{P}$ together with the edge e gives the unique cycle $C(e)$. Knowing this path $\mathcal{P}$, it suffices to consider an edge $f \in C(e)$ of highest cost among all edges of $\mathcal{P}$ and delete this edge f to obtain a new solution-forest. Note that all these steps can be performed without much effort, when the underlying data structure is implemented as described above.

The *edge-set swap*
Since this move is a generalization of the *edge-edge swap*, the basic idea is the same. As before, adding an edge e that is not present in the actual forest to a single tree gives a unique cycle $C(e)$. Now we have to consider all possible edges $f \in C(e)$ which can be deleted. Given the edges e and f, we determine the set $R(e, f)$. We show below how this is carried out. The final process is to divide the set $R(e, f)$ as described in Section 2.2 and check if one of the obtained forests is a feasible solution with smaller potential. Note that by removing a subset $S \subseteq R(e, f)$, we may split up the corresponding tree into many sub-trees, hence the underlying data-type should be able to handle this situation.

Determining the set $R(e, f)$
We use the following approach: After adding the edge e and deleting the edge f, the corresponding component is a tree T' (and hence has no cycles). Remember that we want to know, which edges of $C(e)$ can be deleted together with f without destroying the feasibility of the current solution-forest. Therefore, we determine for each demand-pair a shortest path in this tree T'. It is easy to see, that none of those edges from a shortest path can be part of $R(e, f)$. Hence, we initialize $R(e, f) := \{f\}$ and then add all edges from $C(e)$ that are not part of any shortest path that connects a demand-pair. The elements of $R(e, f)$ are sorted according to the order of their appearance on the cycle $C(e)$.

The *path-set swap*
We proceed in the following way: In each tree $T \subseteq F$ of the current solution-forest, we take any pair of member vertices $u, v \in T$ of the tree and calculate a shortest path $\mathcal{P}$ in the graph described in Section 2.2.1 (each tree $T' \subseteq F$ with $T' \neq T$ get contracted and the edges of T were deleted). In addition to that, we also determine the unique shortest $u-v$ path $\mathcal{P}_C$ in the current solution-forest F. The next task is to add the path $\mathcal{P}$ to the current solution-forest, note that we may need to connect some trees in the forest, hence the forest - data type should provide the corresponding methods. Then, we determine the set $R(e, f) \subseteq \mathcal{P}_C$ (remember that we are not allowed to delete edges from $\mathcal{P}$ any more) and continue as in the *edge-set swap* case.

Disconnecting a single tree of the forest
During the *edge-set swap* or the *path-set swap* it may occur that the deletion of some edges of a tree T disconnects T into components $C_1, \ldots, C_r$. Therefore, we have to re-build the underlying data-structure, i.e. we have to create a new tree for each connected component C_i, for $1 \leq i \leq r$, and delete the "old" tree T from the data-structure.

Connecting trees in the forest
The task to connect some trees in the forest is not trivial: If we add an edge $e = \{u, v\}$ that has its endpoints in different trees T_1, T_2, then we need to connect those two trees. Therefore, we proceed as follows: Take the vertex- and edge-set of tree T_2 and add them to T_1. Add also the edge e to T_1. Update the properties of tree T_1 and delete the tree T_2 from the forest.

About the *Improving-Connecting* move
Theoretically, the best way to determine an improving connecting move is by computing a weighted k-MST approximation as described in Section 4.1, which causes much work for the implementation. The first idea was, to use a given $k - MST$ approximation algorithm as a black-box in our implementation. Unfortunately, we were not successful with this approach. Originally we wanted to use one of the approximation algorithms suggested by Christian Blum and Matthias Ehrgott [EB02], for which also the code of the implementation is provided by the authors. However, the implementation of the algorithms can be only used on a Linux operating system. We tried to fix the problem, but after many hours of work we had to realize that this was a dead end. Since we feared that searching for other algorithms or implementations would end up in the same way, we decided to go on as follows: Remember that the original *connection* move takes some trees of the current forest and connects them. We want to use a local improvement technique to provide a set of trees that can be found in polynomial time and which is of polynomial size. We describe in Section 6.3 how this set can be found. Remember that the problem with the general *connecting* move was, that the set of possible trees, and hence the whole neighbourhood, may be of exponential size.

Post-processing: The *clean-up* move
If we do the *clean-up* move as described in Section 2.2.3, we need to determine shortest paths in the local optimal solution A obtained after the improving steps between the two demand vertices of any demand-pair of the instance. This gives a set of paths $\mathcal{P}$. In order to perform the *clean-up* move, we drop all edges of A that are not part of any shortest path in $\mathcal{P}$.

The modification of the edge lengths
We saw in Section 4.3 that we need to apply a special rounding technique to the edge lengths to achieve a polynomial running time of the Local Search algorithm instead of a pseudo-polynomial running time. For our purposes, we will not implement this rounding technique. It will be very likely that the running time of each single iteration has far more impact on the total running time as the number of iterations itself and also the size of the underlying instance (number of vertices, number of demand-pairs) will be crucial for the total running time.

Useful tools
A tool that we will need quite often is the all-pair shortest path computation due to Floyd and Warshall [KV06]. Hence, an implementation as an own standing function will be helpful. Moreover we will implement the standard Depth-First-Search (DFS) and also a variation of DFS, that detects cycles in a given graph and always deletes an edge of highest cost from the underlying cycle.

6.3. The implementation of connecting moves

In Section 4.1 have seen, that we need to solve a weighted k-MST problem in order to deal with the *Improving-Connecting* move, which itself is a concept to handle the general connecting move described in Section 2.2.2. As noted in Section 4.1, the k-MST problem itself is NP-hard. There is quite a number of heuristics and approximation algorithms for the k-MST problem proposed in the literature, see e.g. [EB02]. However, for the purpose of this thesis we decided to use simpler heuristics to search for an improving connecting move. There heuristics are described in the following Sections 6.3.1 and 6.3.2.

6.3.1. The 2-Conn neighbourhood

Consider a feasible solution F of the current instance consisting of the trees $T_1, \dots, T_s$. Consider the (multi)graph G_F^{all} obtained by shrinking every tree of F to a single point, removing self loops and keeping parallel edges (see also Section 2.2.2). A shortest path between two trees among $T_1, \dots, T_s$ is a path of shortest length that connects the two vertices representing those trees in G_F^{all}. Note that such a path can lead through some other trees of the forest. Adding such a shortest $T_i - T_j$ path that connects two trees $T_i, T_j \subseteq F$ means connecting all trees visited by the shortest $T_i - T_j$ path in G_F^{all}.

The *2-Conn* neighbourhood of F consists now of all solutions F' obtained by connecting two trees among $T_1, \dots, T_s$ as described above. Since the number of trees in the current solution forest is bounded by the number of vertices, there are $\mathcal{O}(n^2)$ many solutions in the *2-Conn* neighbourhood of F and hence the size of this neighbourhood is polynomial in the number of vertices. Note that also all calculations needed to obtain all the solutions can be done in polynomial time. Note also that the *2-Conn* neighbourhood is a subset of the original *connecting* neighbourhood.

6.3.2. The 3-Conn neighbourhood

The *3-Conn* neighbourhood for a solution F is defined analogous to the *2-Conn* neighbourhood: We select three trees of the current solution forest and connect them by shortest paths.Actually this means computing an optimal solution of a Steiner tree problem in G_F^{all} with terminals T_i, T_j, T_k, where T_i, T_j, T_k are the three selected trees. Since there are only three terminals, we solve this problem by enumeration. Indeed, there are three ways to connect three specific trees $T_i, T_j, T_k \subseteq F$. First, find shortest paths $P_{i,j}$ connecting T_i and T_j in G_F^{all} and $P_{j,k}$ connecting T_j and T_k in G_F^{all}. Take the union of these two paths and remove from it all unessential edges. Let the final set of edges obtained in this way be called $E^{(i,k)}$. Notice, that we did not use the shortest path connecting T_i and T_k directly. Compute analogously $E^{(i,j)}$ and $E^{(k,j)}$. The set of edges of minimum total length connecting T_i, T_j, T_k is $arg\ min\{d(E^{(i,k)}), d(E^{(i,j)}), d(E^{(k,j)})\}$. We allow $T_j = T_k$ and hence the *2-Conn* neighbourhood is a subset of the *3-Conn* neighbourhood. The size of this neighbourhood is polynomial in the number of vertices, since we can choose $\mathcal{O}(n^3)$ triples of trees that should get connected. Also here, all other computations can be done in polynomial time. Moreover, the neighbourhood is a subset of the original *connecting* neighbourhood.

Notation:
With *2/3-Conn* neighbourhood we mean the *2-Conn* neighbourhood or the *3-Conn* neighbourhood, but not both at the same time.

Example 6.1. *(2-Conn neighbourhood and 3-Conn neighbourhood)*

Consider the following example of the Euclidean Steiner forest problem with fifteen vertices collected in $V = \{1, \ldots, 15\}$ with coordinates given as follows:

1	=	$(1,1)$	6	=	$(3,11)$	11	=	$(17,10)$
2	=	$(10,0)$	7	=	$(13,5)$	12	=	$(20,7)$
3	=	$(6,8)$	8	=	$(10,10)$	13	=	$(13,15)$
4	=	$(2,14)$	9	=	$(17,3)$	14	=	$(20,14)$
5	=	$(4,14)$	10	=	$(17,11)$	15	=	$(7,5)$

and seven demand-pairs as follows:

Demand 1:	$\{4,5\}$	Demand 5:	$\{10,11\}$
Demand 2:	$\{4,6\}$	Demand 6:	$\{10,7\}$
Demand 3:	$\{3,9\}$	Demand 7:	$\{13,2\}$
Demand 4:	$\{14,10\}$		

Combining these information, we get the following terminals and non-terminals:

Terminals:	$\{2,3,4,5,6,7,9,10,11,13,14\}$
Non-Terminals:	$\{1,8,12,15\}$

A possible solution forest F_0 consists of three trees T_1, T_2 and T_3:

Tree T_1:	Edges $\{4,5\}, \{4,6\}$
Tree T_2:	Edges $\{2,9\}, \{2,3\}, \{3,13\}$
Tree T_3:	Edges $\{7,10\}, \{10,11\}, \{10,14\}$

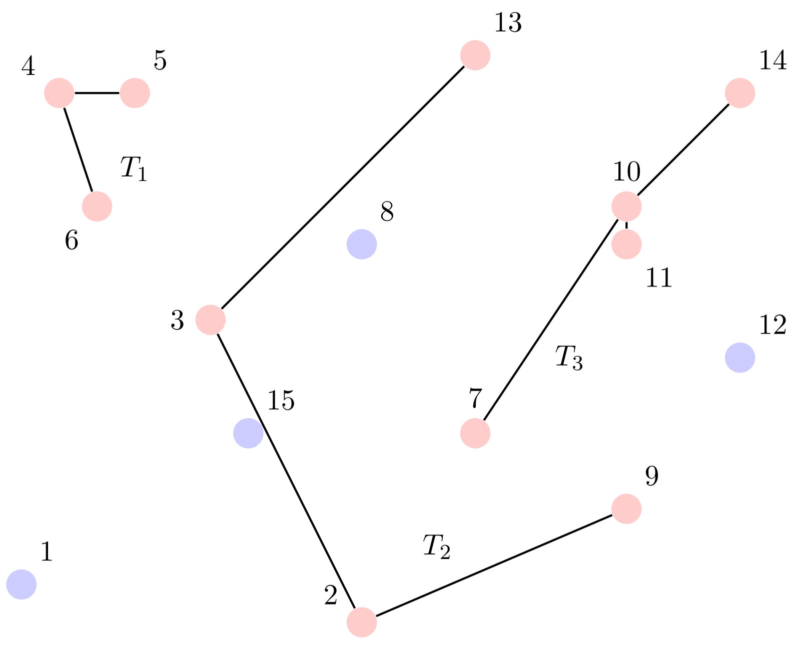


Figure 35: The underlying instance in Example 6.1: terminal in red, non-terminals in blue. The solution forest F_0 pictured by the black edges splits up in trees T_1, T_2 and T_3.

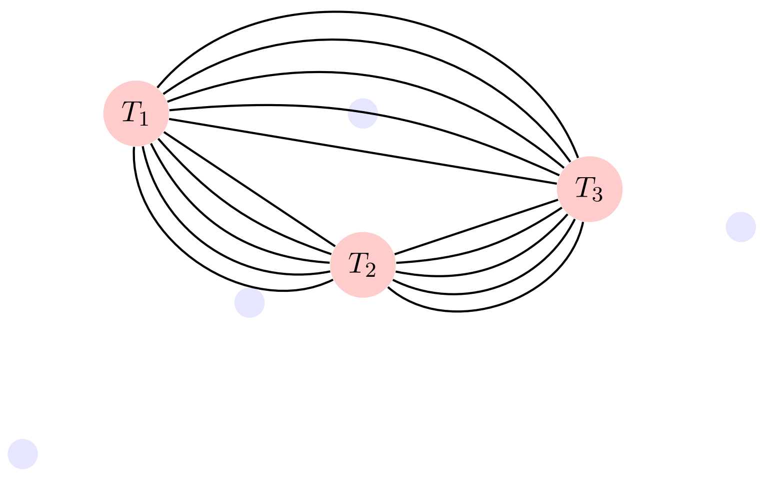


Figure 36: The corresponding multigraph $G_{F_0}^{all}$. There are many edges between the shrunken components T_1, T_2 and T_3, see Table 1. Note that the non-terminal do not play a role, hence we do not draw the edges involving the non-terminals.

We have the following edges between the shrunken connected components T_1, T_2 and T_3 in $G_{F_0}^{all}$. The edges marked in blue are those of shortest length among all edges between two of the three trees.

$T_1 - T_2$		$T_2 - T_3$		$T_1 - T_3$	
edge	length	edge	length	edge	length
$\{4, 2\}$	16.12	$\{2, 7\}$	5.83	$\{4, 7\}$	14.21
$\{4, 3\}$	7.21	$\{2, 10\}$	13.04	$\{4, 10\}$	15.29
$\{4, 9\}$	18.60	$\{2, 11\}$	12.21	$\{4, 11\}$	15.52
$\{4, 13\}$	11.04	$\{2, 14\}$	17.20	$\{4, 14\}$	18
$\{5, 2\}$	15.23	$\{3, 7\}$	7.62	$\{5, 7\}$	12.73
$\{5, 3\}$	6.32	$\{3, 10\}$	11.40	$\{5, 10\}$	13.34
$\{5, 9\}$	17.02	$\{3, 11\}$	11.18	$\{5, 11\}$	13.60
$\{5, 13\}$	9.05	$\{3, 14\}$	15.23	$\{5, 14\}$	16
$\{6, 2\}$	13.04	$\{9, 7\}$	4.47	$\{6, 7\}$	11.66
$\{6, 3\}$	4.24	$\{9, 10\}$	8	$\{6, 10\}$	14
$\{6, 9\}$	16.12	$\{9, 11\}$	7	$\{6, 11\}$	14.04
$\{6, 13\}$	10.77	$\{9, 14\}$	11.40	$\{6, 14\}$	17.26
		$\{13, 7\}$	10		
		$\{13, 10\}$	5.66		
		$\{13, 11\}$	6.40		
		$\{13, 14\}$	7.07		

Table 1: Edges and their lengths in the multi-graph $G^{all}_{F_0}$.

For the *2-Conn* neighbourhood, we have to consider all pairs of trees of the current solution forest and determine the shortest paths between any pair. This results in the *2-Conn* neighbourhood of F_0 which consists of three feasible solutions F_1, F_2, F_3 as follows.

Connecting trees T_1 and T_2 means adding the path $3 \to 6$ that consists of the edge $\{3, 6\}$. This results in replacing T_1 and T_2 by the tree $T := (V(T_1) \cup V(T_2), E(T_1) \cup E(T_2) \cup \{\{3, 6\}\}$ and obtaining the solutio $F_1 := \{T, T_3\}$.

Connecting trees T_2 and T_3 means adding the path $7 \to 9$ that consists of the edge $\{7, 9\}$. This yields the solution $F_2 := \{T_1, T'\}$ where $T' := (V(T_2) \cup V(T_3), E(T_2) \cup E(T_3) \cup \{\{7, 9\}\})$.

Connecting trees T_1 and T_3 means adding the path $6 \to 3 \to 2 \to 9 \to 7$, consisting of the edges $\{3, 6\}, \{2, 3\}, \{2, 9\}, \{7, 9\}$, where the edges $\{2, 3\}, \{2, 9\}$ are part of the current solution forest F and hence have length zero. This means we add the edges $\{3, 6\}$ and $\{7, 9\}$ that forms a tree in $G^{all}_{F_0}$. Note that connecting the two trees T_1 and T_3 finally connects all trees T_1, T_2 and T_3 since the shortest path from tree T_1 to T_3 go through tree T_2 in the graph $G^{all}_{F_0}$. This results in the solution $F_3 := \{T''\}$ where $T'' := (V(T_1) \cup V(T_2) \cup V(T_3), E(T_1) \cup E(T_2) \cup E(T_3) \cup \{\{3, 6\}, \{7, 9\}\})$.

In general in the *3-Conn* neighbourhood, a solution resulting from the minimal length connection of T_1, T_2, T_3 might arise. However, in this example the shortest path among T_1 and T_3 in $G^{all}_{F_0}$ already goes through T_2, and hence this is also the required minimum weight connection of T_1, T_2, T_3 in $G^{all}_{F_0}$. So, in this example, the *2-Conn* neighbourhood of F_0 and the *3-Conn* neighbourhood of F_0 coincide. In general, of course this is not the case and we simply have the following inclusions:

2-Conn neighbourhood $\subseteq$ *3-Conn* neighbourhood $\subseteq$ general *connecting* neighbourhood,

where the last neighbourhood is not necessary of polynomial size, but the first two are.

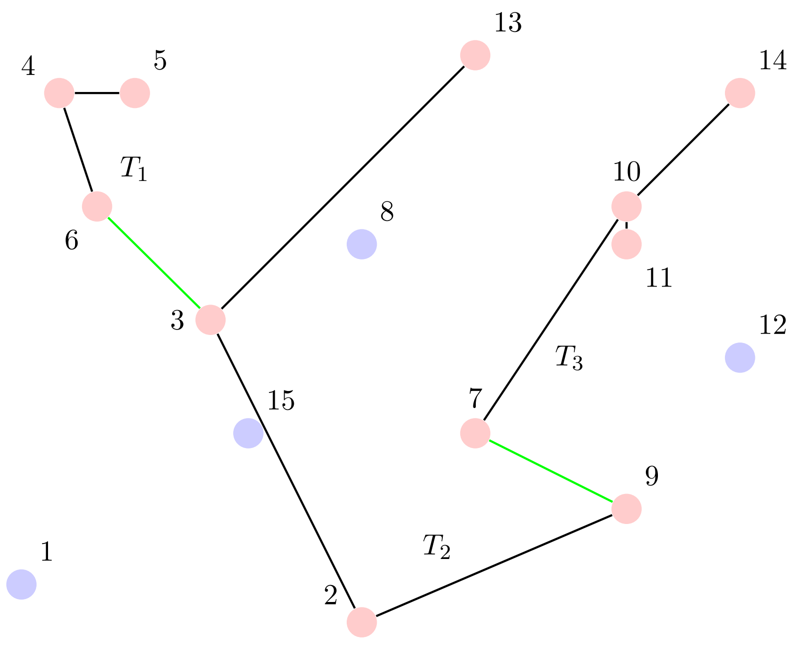


Figure 37: Example for the *2/3-Conn* neighbourhood. Connecting the trees T_1 and T_2 results in adding the edges $\{3, 6\}$ and $\{7, 9\}$ shown in green to the current solution. This finally connects all three trees. Note that all additional moves of the *3-Conn* neighbourhood yields also this outcome.

Using the *2-Conn* neighbourhood or the *3-Conn* neighbourhood
In our implementation of the Local Search algorithm, we will use the *2-Conn* or the *3-Conn* neighbourhood instead of the *improving-connecting* neighbourhood. We will try to determine,if there is a crucial difference in the running time when using either the *2-Conn* neighbourhood or the *3-Conn* neighbourhood. Note that it might be the case, that using the algorithm with the *3-Conn* neighbourhood yields a worse solution than using the algorithm with the *2-Conn* neighbourhood.. This might be illogical on the first glance, since the *2-Conn* neighbourhood is a subset of the *3-Conn* neighbourhood. However, the quality of the finally reached local minimum does not improve with the sizes of the local search neighbourhoods in general.

Note that we cannot give any approximation guarantee for this modified version of the Local Search algorithm. However it is obvious to see that this version of the local search algorithm runs in pseudo-polynomial time for the metric Steiner forest problem.

Terminology:
Let F be a feasible solution for the metric SFP. If there is a feasible solution F' in the *2-Conn* (*3-Conn*) neighbourhood of F with $\phi(F') < \phi(F)$, we say that F admits a *2-Conn* (*3-Conn*) improving move with respect to ϕ.

This finally yields in the following modification of the local search algorithm considered in Part I:

Algorithm 7 Local search with *2-Conn* (*3-Conn*)

```
Require: An instance $I_m = (G, d, \mathcal{D})$ of the metric Steiner forest problem with $G = (V, E)$
    being the complete graph and $A$ being a solution obtained by connecting each demand-
    pair by an direct edge and deleting an edge from every cycle that appears. This gives a
    feasible starting solution for $I_m$.
Ensure: A solution $A_f$ to the instance $I$.
    Set $i := 0$ and let $A_0 := A$
    while $A_i$ admits an improving edge-edge swap, edge-set swap, path-set swap or 2-Conn
    (2-Conn) move with respect to $\phi$ do
        Set $A_{i+1}$ to be the resulting solution after applying the move
        Set $i := i + 1$
    end while
    Output the solution $A_f$ that results by applying the clean-up-move to $A_i$
```

7. Classes and functions

As we saw in the previous section, we need to implement some classes and functions which help then to implement the main part of the algorithm. Here we give a short overview about these classes and functions. Further details about the implementation and more comments can be seen in the code provided as an external appendix.

Note that the whole project consists of three main parts: The Local Search algorithm (Algorithm 7), the Gluttonous algorithm (Algorithm 6) and a function, that computes an optimal solution of the integer program formulation of the Steiner forest problem (IPuf) by using appropriate solver. We use *Gurobi 8.0.1* with an academic license.

A description of the functions and classes that are used only for the Gluttonous and the IP part can be seen in [Gol18], here we describe only the functions and classes that are needed for the Local Search and the general part of the project.

7.1. Classes

class "file_io"
This class is used to read the data from a *.txt* file and store it as a vector of doubles. This class was provided by Prof. Gundolf Haase from Karl Franzens University in the course of his coding lecture and is allowed to be used also in this project.

class "matrix"
This class is only introduced for convenience. It stores a matrix as a vector of *doubles*, i.e. the rows of the matrix are concatenated as a vector. The methods *getEntry* and *changeEntry* simply do the nasty translation from row-column index to the corresponding index of the vector. Methods for getting the dimension of the matrix and getting the whole matrix are implemented.

class "triple"
We implement this class to handle all situations, where three values of the type *double* are needed to be gathered in a comfortable way. It can be used to represent an edge of a graph, with the first two values being the end-vertices and the third one representing the cost of the

edge, for example[15]. There are *getter*-methods for all three values and also a *setter*-method for the third value.

class "tree"
As the name suggests, the class is used to represent a tree in the graph-theoretic sense. The attributes of a tree are an *integer* name, member vertices in form of a vector of *double*, edges in form of a vector of *triple*, its total length (sum of all edge lengths) and its potential, both of type *double*. There are two *constructors*: The first takes only one edge that is then considered as the whole tree, the second has as input a list of member vertices, demand-pairs and the adjacency and distance matrix of the underlying graph. The tree is created out of all those information. The method *vertexInTree* checks if a given vertex is contained within the tree, *addVertexToTree* and *removeVertexFromTree* add / remove a vertex to / from the tree. The last should be used only in a higher logic to prevent invalid operation as deleting a vertex that is still covered by an edge. To add an edge to the tree, we use *addEdgeToTree*. The methods queries first whether the edge is already present in the tree. The method *removeEdgeFromTree* deletes an edge given by two end-vertices. If one of the end-vertices is not covered by some other edge of the tree any more, it uses *removeVertexFromTree* to delete this vertex from the tree. To calculate the total length and the potential of the tree, we can use the methods *calcTotalLength* and *calcPotential*. Note that *calcPotential* first calls *calcTotalLength* and then calculates the potential. The tree also stores information about that demand-pair, which is connected by the tree and has the highest shortest path distance in the original graph among all pairs that are connected by the tree (remember that we need that pair to calculate the potential). We store this pair as a variable named *highestPair* of type *triple*[16] as part of the tree and it can be changed by the method *changeHighestPair*. Typical *getter*-methods are implemented.

class "forest"
An object of type forest has the following attributes: A vector of *trees* named components, a vector of *triple* named demands, a *matrix* called adjacency and two double values *potential* and *totalLength*. The *constructor* is some kind of initialization of the forest. It takes the distance matrix and the demand-pairs as input and creates an initial forest as follows: Since we are dealing always with a complete graph, we add the direct edge connecting two terminals to a virtual graph that is given by its adjacency matrix. This virtual graph may contain cycles, hence we use the method *getCycleFree* to delete one edge from each cycle using also the method *modifiedDFS*. For details of these two functions have a look on Section 7.2. The result after this operation is the adjacency matrix of the initial forest. We apply the classical DFS search (using the method *DFS*) on the adjacency matrix to determine the components of this virtual graph, where each component can be seen as a tree of the forest. With this information, we create an object of type *tree* for each of these trees and add it to the *forest* with the method *addTree*. Let's have a look on further methods: *addInnerTreeEdge* and *deleteInnerTreeEdge* can be used to add / delete an edge that is contained within / should be removed from one single *tree* of the *forest*. Several checks are made before executing the necessary operations. The method *inWhichTree* returns the name of the *tree* where a given vertex is contained (the value zero is returned if the vertex is not part of any *tree*), *getTree* returns the whole *tree* of a given name and *sameTree* returns true if two given vertices are contained within one single *tree*. The method *mergeTrees* takes two *trees* T_1, T_2 as input and output a subgraph H that contains both T_1 and T_2. Note that H is not a tree yet, since it consists of the two connected components T_1 and T_2. The aim is that H become

[15]Of course, for this situation we do not need the first two values being of type double, but with this setting we can also use the class in another context.

[16]We store one terminal, its mate and the cost of the shortest path between them in the original graph as the third value

a tree, therefore one should use the method in the following setting: We want to add an edge e that connects two trees T_1, T_2 in the graph theoretical sense to obtain a new tree $T' := (V[T_1] \cup V[T_2], E[T_1] \cup E[T_2] \cup \{e\})$. Therefore, we first call the method *mergeTrees* with input T_1, T_2. This gives a subgraph H as described above. H is not yet a tree in the sense of graph theory, since the connecting edge e is missing. In the second step, we add e to H that makes H to a tree in the graph theoretic sense. Let's consider another situation: We remove an edge from a *tree*, this may disconnect the tree in the graph theoretical sense. The method *updateTree* does then exactly the same as the *constructor*, namely it queries the remaining components (trees) and builts a new *tree* for each of them, add them to the *forest* and delete the former *tree* from the forest by using *removeTree*. Please check the comments at the beginning of the code of *removeTree* to understand how to use this method. *updatePotential* calculates all potentials (and hence also all total lengths) of the contained *trees* and then sum them up to one value for the whole *forest*. To add a vertex to a given tree, we use *addVertexToTree*, the method *checkFeasibility* determine on the one hand if each member vertex is contained in only one tree and on the other hand if each demand-pair is contained within one single tree. Error-messages are output in the case of an invalid setting. Typical *getter*-methods are implemented for all attributes of the class.

class "sfpLocal"
This class should represent an instance of the Steiner forest problem. It consists of an *integer* numOfPoints for the number of vertices/points, a *matrix* distances to store the distance matrix of the underlying graph, another *integer* numOfDemands for the number of demand-pairs and a *vector* of *triple* called demands to store the demand-pairs. We chose the data type *triple* for a demand in order to store both terminals and also the length of a shortest path in the original graph, which is often needed for some computations. Note that the classname-postfix "Local" is necessary since we also use the class *sfpGlut* that represents basically also an instance of the Steiner forest problem, but with slightly different attributes in order to use all functions of the project *Gluttonous algorithm* without any changes. The class *sfpLocal* provides all *getter*-methods and the method *setDemands*, which changes the whole set of demand-pairs. We will need it in order to do the transformation of an instance as described in Section 6.

Each class that is used as a new data type also contains a method that enables an output on the screen via the standard *cout* command.

7.2. Functions

To have a slender structure of the main program, the code is organized in the following functions. Here is a short overview, details about the transferred parameters can be seen in the comments of the code in the external appendix.

function "getInput"
As the name suggests, this function is used to get the input data in one of the structures described in Section 6. For each way of getting the input information, there are several checks to ensure a valid input which is then returned as an instance of type *sfpLocal*. Note that the instance which we obtain by this function is not necessarily one with a complete graph or demand-pairs obeying the desired ordering. Specific transformation of the instance for local search, Gluttonous or the integer program solver are done afterwards in the corresponding functions for technical reasons.

function "iToS"
Transforms an integer variable into a variable of type string.

function "displaySolution"
This function outputs some information about the underlying instance and details of the obtained solution like calculation time, cost of the obtained solution and the adjacency matrix of the solution on the screen.

function "writeSolutionToFile"
In general, the function does the same as *displaySolution*, but it writes the information about the obtained solution into a *.txt* file.

function "allPairShortestPaths"
Based on the underlying distance matrix, this function calculates shortest paths between all pairs of vertices. We use the *all-pair shortest path* algorithm of Floyd-Warshall [KV06] in this function. The algorithm calculates the cost of the shortest paths and the predecessor matrix containing the information about all shortest paths.

function "getShortestPath"
Determines the vertex sequence of a shortest path between two given vertices using the predecessor matrix obtained by *allPairShortestPaths*.

function "addPathToSolution"
Given the adjacency matrix of a graph and a path connecting two vertices of the graph, this function adds the edges of the path to the graph by setting the corresponding entries of the adjacency matrix.

function "modifiedDFS"
This function implements basically the well known depth first search (DSF) starting at a given vertex. In contrast to the classical DFS, the intention here is to find a cycle if it exists, and then delete an edge from the cycle. The way this function is implemented is only meaningful if it is used within the function *getCycleFree*.

function "getCycleFree"
This function returns a cycle free subgraph of the input graph by deleting one edge of highest cost in each cycle. The approach implements a modified DFS search that terminates if a cycle was found, deletes an edge of highest cost from the cycle and restarts the DFS search. This procedure is repeated until no more cycles are found.

function "calcCostOfSolution"
We use this function to calculate the cost of a given graph, i.e. the sum of all edge lengths. This function does not check whether the given graph is indeed a feasible solution to the SFP instance.

function "transformInstance"
The input of this function can be an arbitrary instance of the Steiner forest problem in form of the data type *sfpLocal*. The output, or to be precise some part of the output, should be of the same data type, but in the form desired by the local search algorithm implemented in the function *doLocalSearch*. Therefore, we do the transformation described in Theorem 3.31 with the help of *allPairShortestPaths* to compute the metric closure of the input graph. In addition to that, we reorder the demand-pairs such that the costs of the shortest paths that connect them in the original graph are non-decreasing. We return the transformed instance and also the predecessor matrix obtained by *allPairShortestPaths*. It is necessary to return also this predecessor matrix in order to re-transform an obtained solution, as described below.

function "retransformSolution"
This function is used to handle the following situation: Consider a graph G_{SP} where an edge $e = \{i, j\}$ represents a shortest $i - j$ path in some underlying graph G, hence G_{SP} is the metric closure of G. We want to determine the graph G_{expand}, where all paths are present in their expanded form. Therefore, we take each edge $e = \{i, j\} \in G_{SP}$, get the information about the shortest $i - j$ path that is represented by e and add the path to G_{expand} using the function *addPathToSolution*. We finally return the adjacency matrix of G_{expand}.

function "getR_e_f"
We use this function to determine the edge-set $R(e, f) \subseteq C(e)$ for some edges e and f as described in the theory part in Section 2.2.1. Note that the description of this set is easy to understand, but it is not trivial how to get the concrete set $R(e, f)$. Details can be seen in the code. We use a *vector* of *triple* to return the edges of $R(e, f)$, although we do not need the third value of the data type *triple*.

function "doLocalSearch"
The function implements the main part of the local search algorithm. It takes an (arbitrary) instance of the Steiner forest problem and call *transformSolution* in order to obtain an instance in the desired format. The function does not explicitly need a feasible solution to start with since this initial solution is obtained by the *constructor* as described in the class *forest*. Then the Algorithm 7 is implemented where one has to specify which substitution for the connecting neighbourhood (*2-Conn* or *3-Conn*) should be used. The algorithm itself returns the adjacency matrix of the obtained solution to the transformed instance. Then *retransformSolution* is applied to obtain the corresponding solution to our original instance.

function "SFPLocalToSFPGlut"
Since we use two different data types for storing an instance of the Steiner forest problem (namely *sfpLocal* and *sfpGlut*), we use this function to transform an instance given in the *sfpLocal*-format to an instance of the data type *sfpGlut*. This function enables the usage of all functions inherited from the project *Gluttonous Algorithm* without any changes. Note that this transformation has nothing to do with the transformation mentioned above in the functions *transformInstance* and *retransformSolution* based on Theorem 3.31.

function "getRandomEuclideanInstance"
Creates a random instance of the Euclidean Steiner forest problem with a given number of points and demand-pairs as well as limits for the coordinates. The function generates the desired number of points and demand-pairs in the following way: The vertices are chosen uniformly at random from the integer-based two dimensional grid in the first quadrant with the corresponding limits in each direction, the demand-pairs are chosen uniformly at random from the set of all possible demand-pairs. All chosen points are pairwise distinct. Also the demand-pairs are chosen pairwise distinct (two pairs can share one single vertex, but not both vertices). There is a trigger which yields that the function ignores the given number of desired demand-pairs that should be chosen and choose this number uniformly at random from the set of all possible numbers of demand-pairs (depends only on the number of vertices). This can be helpful for some statistical experiments. The function returns an instance of type *sfpLocal*. At the end of the function, all necessary information about the generated instance are stored in an *.txt* file in order to be able to use the instance later again. The format of the *.txt* file is compatible with the requirements of the function *getInput*.

function "doSeriesOfRandomInstancesLocalSearch"
With this function, one can create a desired number of instances with a common given num-

ber of vertices and demand-pairs and then try to solve every single instance using the Local Search algorithm by applying *doLocalSearch*. Information about the results is output on screen and saved in *.txt* files. See Section 9.4 for results obtained by using this function.

function "doSeriesOfRandomInstancesGluttonous"
The function does basically the same as *doSeriesOfRandomInstancesLocalSearch*, but as the name suggests, it uses *doGluttonous* instead of *doLocalSearch*, i.e. we try to approximate the solution using the Gluttonous algorithm instead of the Local Search algorithm. See Section 9.5 for results obtained by using this function.

function "doSeriesOfRandomInstancesGurobi"
The function does the same as *doSeriesOfRandomInstancesLocalSearch*, but instead of approximating the solution by the Local Search algorithm, here we try to solve the instances to optimality by using the integer program formulation introduced in Section 5.2 and the *Gurobi-Solver*. See Section 9.6 for results obtained by using this function.

function "doSeriesOfRandomInstancesLocalGluttonousGurobi"
This function is a kind of aggregation of the last three functions: We create a desired number of instances with a common given number of vertices and demand-pairs. For every single instance, we try to achieve three solutions: one from applying *doLocalSearch*, one from *doGluttonous* and another one from *doGurobi*[17]. Information about "the outcome" is provided on the screen and saved in *.txt* files. Note that this function can applied only to instances with a limited number of vertices, edges and demand-pairs; otherwise stack-overflows and prohibitively long running times become more and more probably with increasing size of the instance. For more details and results see Section 10.

function "doSeriesOfRandomInstancesLocalGluttonous"
We use this function to create a series of random instances and then apply Local Search and the Gluttonous algorithm to each instance and compare the outcomes. This means that the function does basically the same as the previous function, but without solving each instance by the *Gurobi*-Solver. Details about results of applying this function can be seen in Section 10.

function "doSeriesOfRandomInstancesLocalSearchComparingConnectingMoves"
The function creates a series of random instances in the same way as in the functions above. Then, it tries to solve each single instance involving the *2-Conn* move on the one hand and then involving the *3-Conn* move on the other hand. A statistical evaluation is shown on screen and saved into a *.txt* file at the end of the execution.

7.3. The main

Since we invested a lot of time to create helpful functions, the main file can be implemented in a very slender and clear way. The following features are implemented:

- Get the input from *getInput* and apply LSA.
- Get the input from *getInput* and apply the Gluttonous algorithm.
- Get the input from *getInput* and apply the *Gurobi-Solver*.
- Get the input from *getInput*, apply LSA and the Gluttonous algorithm.
- Create a series of random instances and apply LSA to each instance.

[17]Note that this function has the name *solvingByGurobi* in [Gol18].

- Create a series of random instances and apply LSA to each instance once with using the *2-Conn* neighbourhood and once with using the *3-Conn* neighbourhood.
- Create a series of random instances and apply the Gluttonous algorithm to each instance.
- Create a series of random instances and apply the *Gurobi-Solver* to each instance.
- Create a series of random instances, apply LSA and the Gluttonous algorithm to each instance.
- Create a series of random instances, apply the Gluttonous algorithm and the *Gurobi-Solver* to each instance.
- Create a series of random instances, apply LSA, the Gluttonous algorithm and also the *Gurobi-Solver*.

Comment
Note that depending on which feature one wants to use, there are different limits for the number of vertices, edges and demand-pairs of the input in order to avoid stack overflows or far too long running times. For more details please check Part III.

Part III.
Applying the algorithm

8. The algorithm applied to a small example

8.1. Some words at the beginning

In Section 6, we have developed Algorithm 7 as a modification of the original Local Search Algorithm (Algorithm 5) proposed by Gross et al. [G17]. Let's consider the application of the algorithm to a concrete instance. Basically, we would have to list up all possible moves in the neighbourhood for the solution in the current iteration and choose the best move. Since the neighbourhood can be huge even in the case of *small* instances, we will not list up all the moves but just track the evaluation of the solution during the run of the algorithm.

Remember, that in Algorithm 7, we have introduced two possible neighbourhoods as a substitute for the original *connecting*-neighbourhood. Our local search algorithm involves exactly one of them. For illustration purposes we will apply the algorithm twice, one variant involving the *2-Conn* neighbourhood and the other variant involving the *3-Conn* neighbourhood. We may obtain different solutions and different forests after some iterations.

In the Sections 8.2 - 8.5 below we will introduce an SFP and apply both variants of the algorithm to that instance starting with a specific starting solution. Then the solutions generated by both variants of the algorithm will be compared.

8.2. The underlying instance

We consider the following instance of the metric Steiner forest problem:
We assume the metric space to be the Euclidean plane, hence the metric d is the Euclidean distance.

The instance consists of the following 20 vertices/points in V:

1	=	$(24,2)$	8	=	$(3,21)$	15	=	$(22,26)$
2	=	$(2,12)$	9	=	$(18,27)$	16	=	$(28,29)$
3	=	$(9,15)$	10	=	$(28,7)$	17	=	$(1,21)$
4	=	$(13,1)$	11	=	$(0,19)$	18	=	$(8,12)$
5	=	$(4,4)$	12	=	$(6,5)$	19	=	$(26,27)$
6	=	$(3,1)$	13	=	$(27,1)$	20	=	$(11,20)$
7	=	$(3,21)$	14	=	$(0,9)$			

and six demand-pairs in the set $\mathcal{D}$ as follows:

Demand 1:	$\{18,4\}$	Demand 4:	$\{20,16\}$
Demand 2:	$\{19,20\}$	Demand 5:	$\{11,20\}$
Demand 3:	$\{1,12\}$	Demand 6:	$\{10,13\}$

Combining these information, we get the following terminals and non-terminals:

Terminals:	$\{1,4,10,11,12,13,16,18,19,20\}$
Non-Terminals:	$\{2,3,5,6,7,8,9,14,15,17\}$

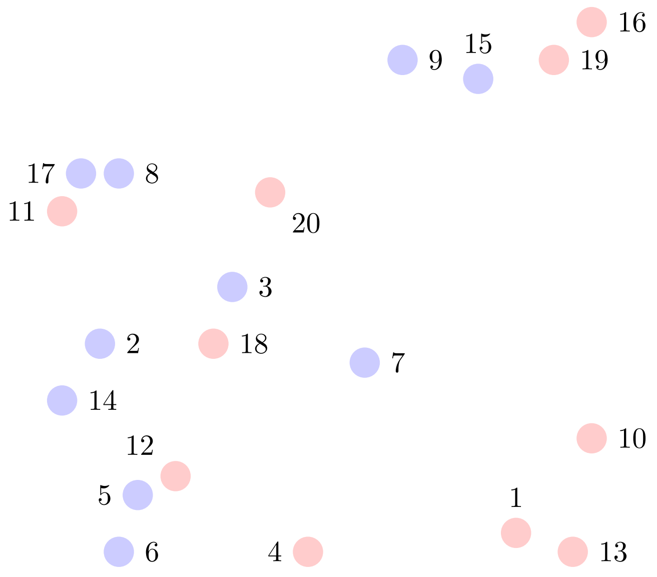


Figure 38: Instance Overview. The vertices are points in the Euclidean plane, terminals colored in red and non-terminals in blue.

8.3. The application of both variants of the algorithm

At the beginning, we start in both cases with a feasible solution obtained as described in Algorithm 7, i.e. we add all the direct edges that connect a demand-pair and remove cycles if necessary.

Initial forest for both variants of the algorithm:

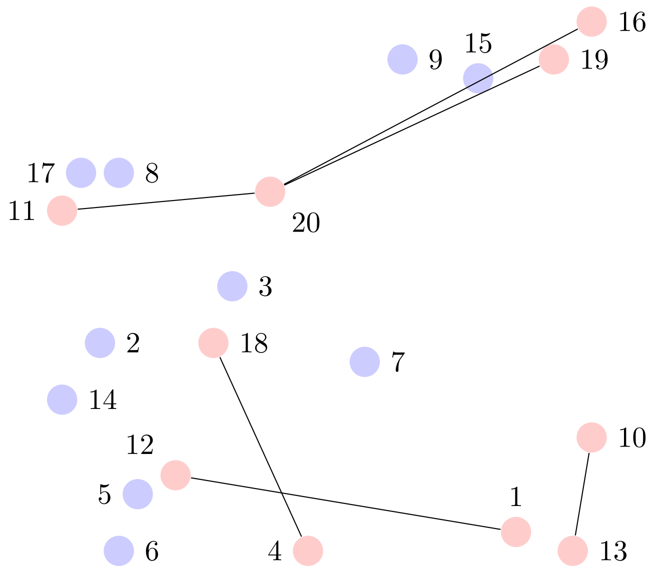


Figure 39: Note that no cycles where created by adding the edges connecting the vertices within each demand-pair, hence no edges were removed.

Forest after iteration 1, using Algorithm 7 with *2-Conn*:

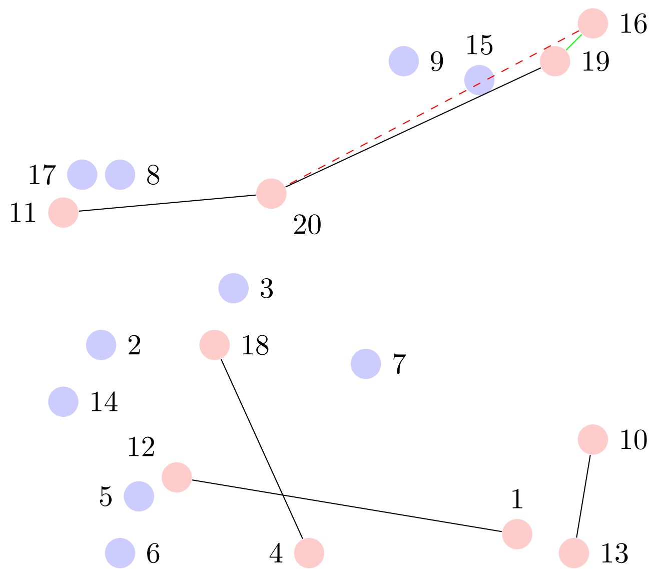


Figure 40: The first chosen move is an *edge-edge swap*: Adding the edge $\{16, 19\}$ creates a cycle $16 - 19 - 20 - 16$ in which the edge with the highest Euclidean distance is $\{16, 20\}$. Hence, we remove this edge and obtain a better feasible forest.

Forest after iteration 1, using Algorithm 7 with *3-Conn*:

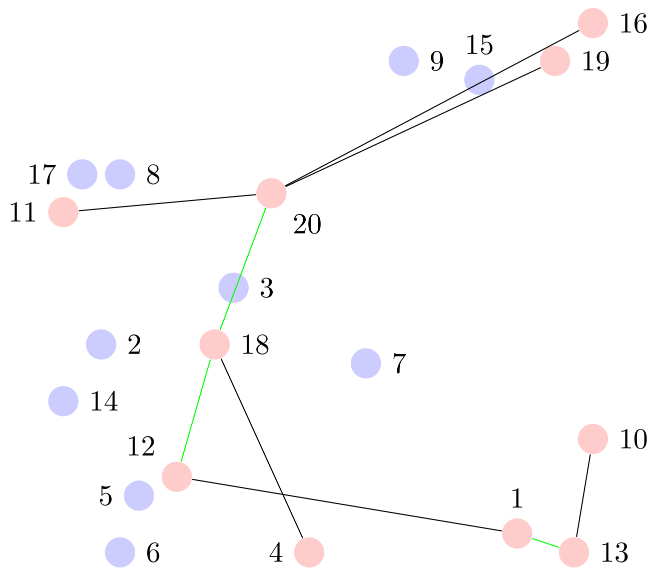


Figure 41: The best move is a *3-Conn* move: It connects the components $C_1 = \{11, 16, 19, 20\}$, $C_2 = \{4, 18\}$ and $C_3 = \{10, 13\}$. The shortest path in G_A^{all} connecting C_1 and C_2 is given by the edge $\{18, 20\}$, the shortest path connecting C_2 and C_3 uses the fourth component $C_4 = \{1, 12\}$ as an intermediate vertex, i.e. is given by $18 \to C_4 \to 13$ and adds finally the edges $\{12, 18\}$ and $\{1, 13\}$. Therefore, the obtained solution consists of one single tree.

Forest after iteration 2, using Algorithm 7 with *2-Conn*:

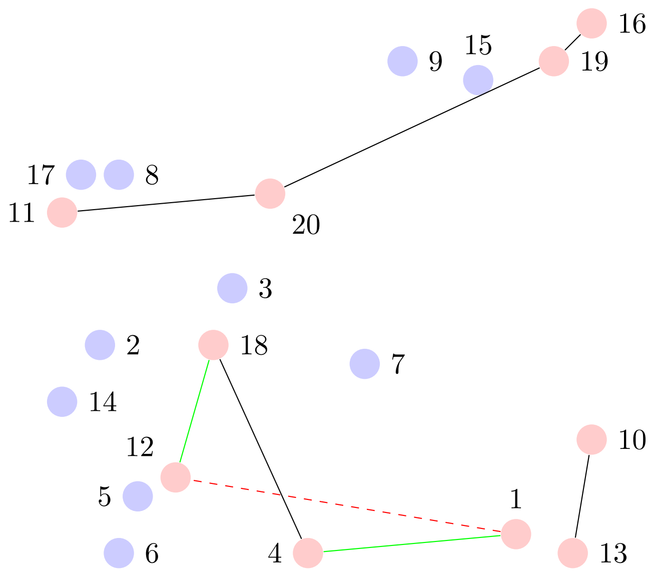


Figure 42: We obtain the next forest by a *path-set swap*: Choosing vertices 1 and 12, adding the edges $\{1, 4\}$ and $\{12, 18\}$ yields a connection of the chosen vertices (shortest path $1 \rightarrow 4 \rightarrow 18 \rightarrow 12$ where the edge $\{4, 18\}$ has length zero). The only possibility to delete edges in the created cycle (due to definition) is to delete $\{1, 12\}$.

Forest after iteration 2, using Algorithm 7 with *3-Conn*:

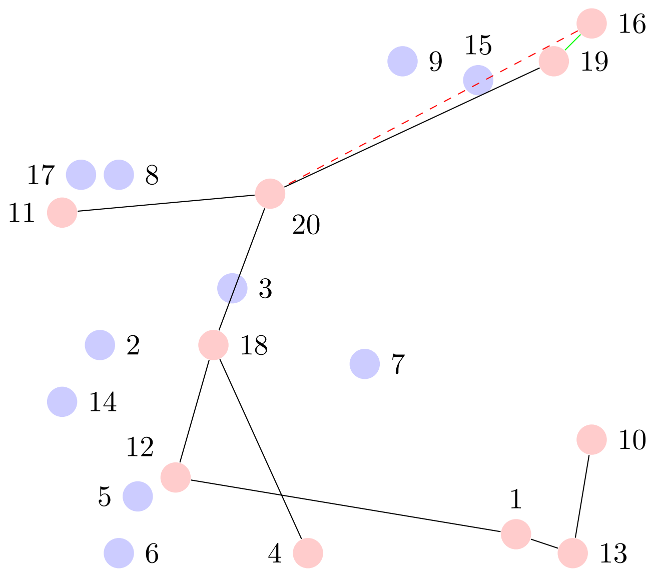


Figure 43: We have seen this move already in Figure 40: Adding the edge $\{16, 19\}$ and removing $\{16, 20\}$ yields a solution with a lower potential and also a lower total length.

Forest after iteration 3, using Algorithm 7 with *2-Conn*:

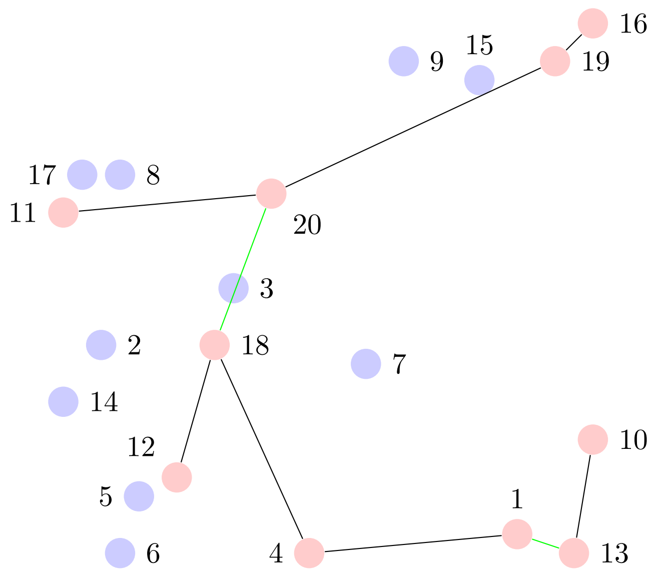


Figure 44: We apply a *2-Conn* move: To connect components $C_1 = \{11, 16, 19, 20\}$ and $C_2 = \{10, 13\}$ we add the edges $\{1, 13\}$ and $\{18, 20\}$ that form a path in G_A^{all}. Hence, all three components get connected.

Forest after iteration 3, using Algorithm 7 with *3-Conn*:

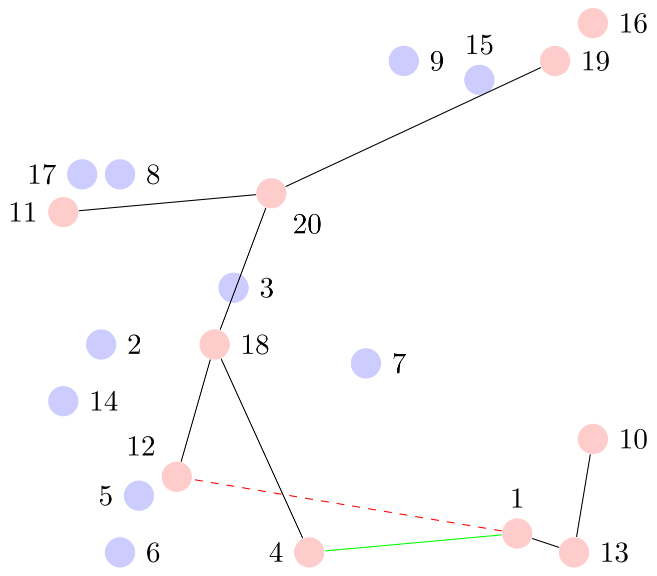


Figure 45: The forest results from an e*dge/edge-swap*: We have to add the edge $\{1, 4\}$ and then remove the edge $\{1, 12\}$. Removing a second edge from the created cycle would make the forest infeasible.

Forest after iteration 4, using Algorithm 7 with *2-Conn*:

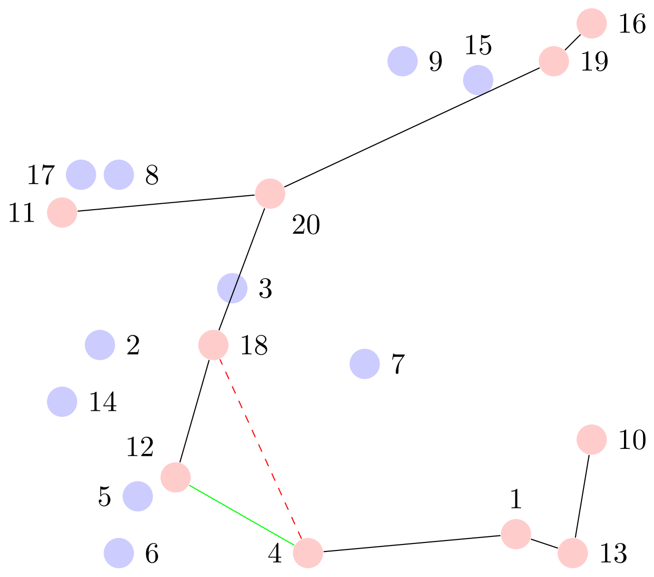


Figure 46: This *edge-edge swap* leads to the best possible forest in the neighbourhood: Adding the edge $\{4, 12\}$ and removing the edge $\{4, 18\}$.

Forest after iteration 4, using Algorithm 7 with *3-Conn*:

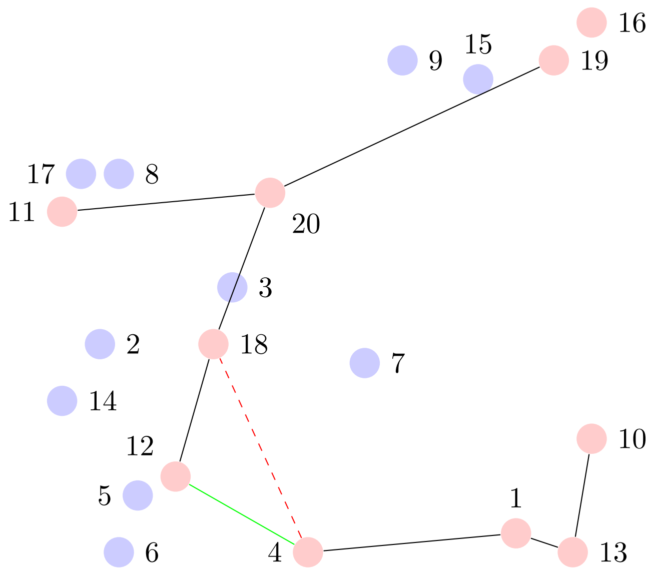


Figure 47: Note that the forest at the beginning of the iteration is the same as in the case, where we use the *2-Conn* neighbourhood. Allowing *3-Conn* moves does not change the best possible move, hence we get the same outcome as above.

Forest after iteration 5, using Algorithm 7 with *2-Conn*:

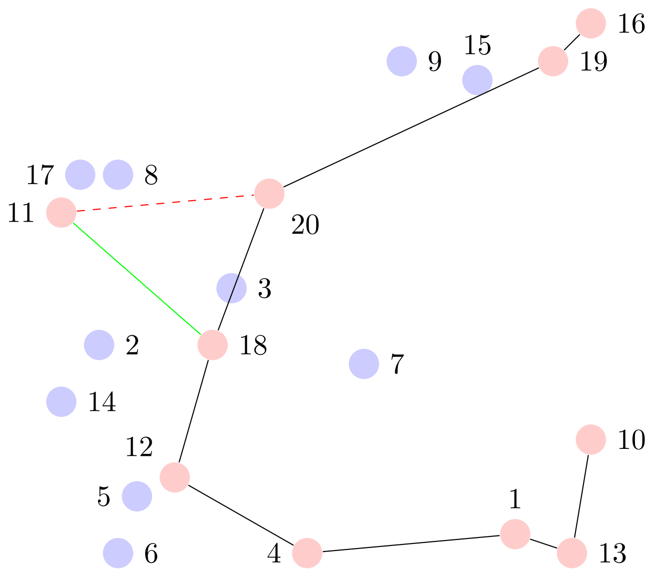


Figure 48: Also the last iteration brings an *edge-edge swap*: Adding the edge $\{11, 18\}$ and removing the edge $\{11, 20\}$.

Forest after iteration 5, using Algorithm 7 with *3-Conn*:

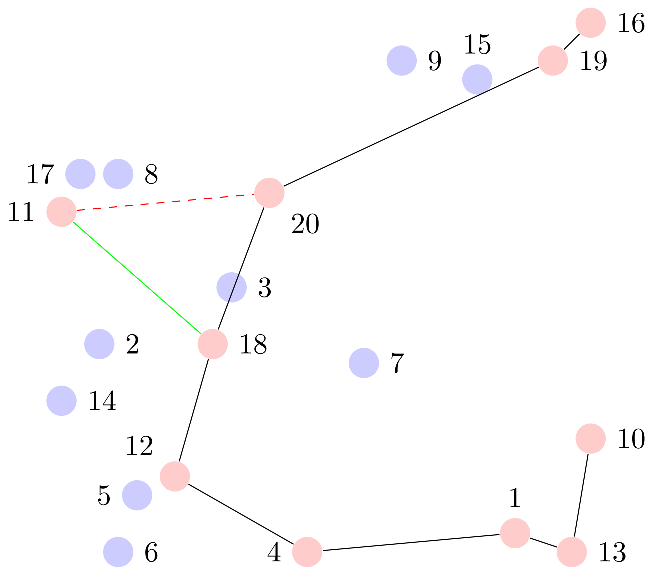


Figure 49: As in iteration 4, both algorithms perform the same move on the same underlying forest.

Forest after *clean-up* in both cases:

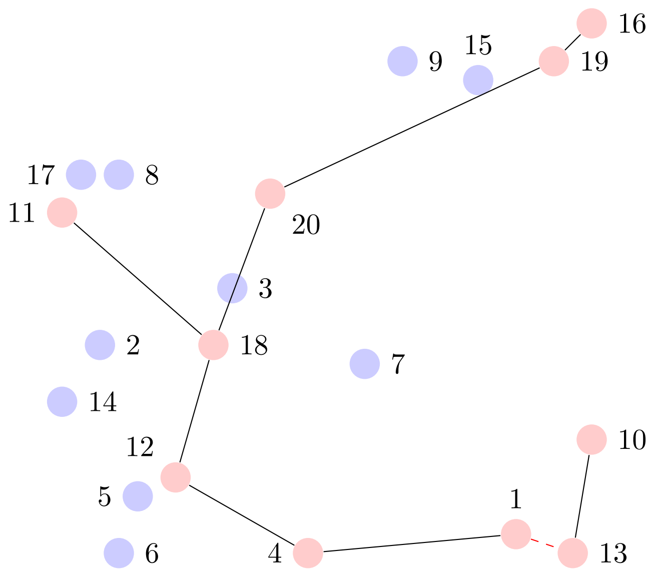


Figure 50: We can remove the edge $\{1, 13\}$ from the forest without hurting the feasibility. Removing one of the remaining edges would lead to an infeasible forest. This forest is the solution obtained by both algorithms.

8.4. Overview

The following tables summarize the course of each variant of Algorithm 7 when applied to the instance introduced in Section 8.2.

Algorithm 7 with *2-Conn* neighbourhood:

Iteration	move	total length	potential	components
Start		83.25	138.9	4
1	E/E	66.84	122.5	4
2	P/S	66.92	110.5	3
3	C-2	78.62	97.86	1
4	E/E	74.60	93.84	1
5	E/E	74.19	93.42	1
Clean-up		71.03		2

Algorithm 7 with *3-Conn* neighbourhood:

Iteration	move	total length	potential	components
Start		83.25	138.9	4
1	C-3	102.2	121.47	1
2	E/E	85.83	105.1	1
3	E/E	78.62	97.86	1
4	E/E	74.60	93.84	1
5	E/E	74.19	93.42	1
Clean-up		71.03		2

8.5. An optimal solution

The following forest gives an optimal solution to the described instance. We obtain the solution by using the function *doGurobi* presented in Section 7.2.

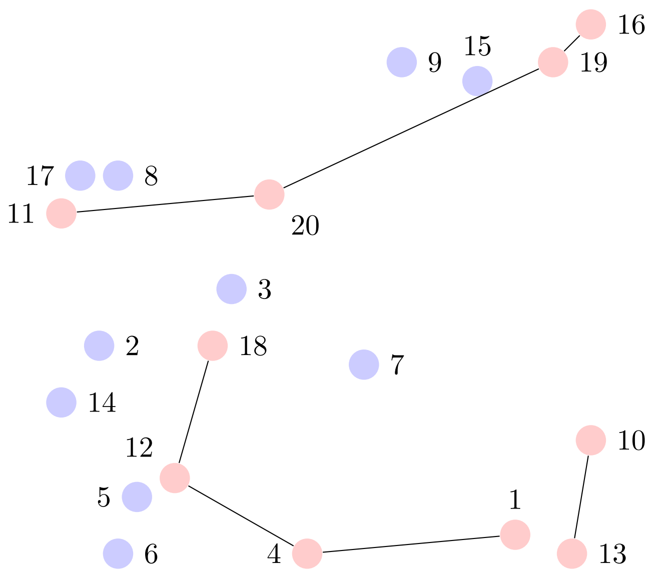


Figure 51: An optimal solution to the instance.

Let's compare the obtained solutions regarding total length and running time:

Solution	total length	running time
Alg. 7 with *2-Conn*	71.03	10.56 s
Alg. 7 with *3-Conn*	71.03	16.66 s
Optimal solution	62.90	1.082 s

Observe that both variants of the algorithm output the same solution that has a total length which is about 13 percent higher than the total length of an optimal solution. For this instance, with only a small number of vertices and demand-pairs, also the running time of the algorithm is much higher than the time needed to solve the integer programming formulation of the problem by an off-the-shelf software.

Note, that the first variant of the algorithm (*2-Conn* neighbourhood) generated a feasible solution with a total length which is smaller than the total length of the final solution. The objective function value of the solution after iteration 1 is much closer to the optimal objective function value than the objective function value of the final obtained solution. This can happen, since the algorithm tries to decrease the potential, but not necessarily the total length of the forest.

Comment
One can improve the quality of the solution obtained by Algorithm 7 as follows: Store the best solution forest B with respect to the original edge lengths d found so far during the run of the algorithm and output either B or A_f, depending on which has the lower total length.

9. Running times of the algorithms

This section is structured as follows: At the beginning, we discuss the generation of the test instances, then we compare the performance of both variants of Algorithm 7 involving neighbourhoods *2-Conn* and *3-Conn*. The goal of this process is to decide which neighbourhood is more beneficial to the algorithm such that we can concentrate on this variant of the algorithm in the following tests.

Having fixed one of the two neighbourhoods for our algorithm, we want to compare the three implemented algorithms Local Search, Gluttonous and the integer program in terms of running time. It should be clear, that we cannot hope to achieve a solution within reasonable time if we choose an instance with a very large number of vertices, edges and demand-pairs, since the implementations have not been optimized regarding efficiency and there are also hardware restrictions.

Moreover we will compare the implemented algorithms also in terms of quality of their output solutions and in the case of small instances we will also address the optimality gaps.

9.1. Randomly generated test instances

On a low level, we will do the following process within our tests: For some fixed number of vertices and demand-pairs, we create a given number of instances with exactly those input parameters in a random way, in order to determine an average running time or an average approximation factor. We refer to solving a sequence of instances all of them sharing a given set of input parameters, i.e. the number of vertices and the number of demand-pairs, as a single test. To obtain meaningful values from a statical point of view, the number of instances in each test should be preferably high, but on the other hand, single tests should not take too much time. We restrict ourselves to single tests with a total running time of no more than two hours.

Convention
If we write a "?" for the value of the running time (or the approximation factor), then we have not finished the test since it took longer than two hours.

It should be clear, that when applying the algorithm on *large* instances, we cannot hope to obtain a solution within reasonable time. First, we should clarify, what the adjective *large* means for an instance of the (metric) Steiner forest problem.

According to Theorem 3.31 we can convert any instance of the general Steiner forest problem into an instance of the metric Steiner forest problem. Thus, it suffices to concentrate on this type of problem. To be precise, we will consider randomly chosen instances of the Euclidean type where the vertices are points in the Euclidean plane, each pair of vertices is connected by a direct edge (the underlying graph is therefore the complete graph) and the distance attached to an edge is the Euclidean distance between its end points. We will choose the coordinates of the vertices within some given bounded rectangle, but this does not restrict the setting, since we can always scale down larger instances such that they fit into this setting (remember that we have implemented the function *getRandomEuclideanInstance*). Hence, the input of an instance consists of two parameters: the *number of vertices* n and the *number of demand-pairs* $|\mathcal{D}|$.

We obtain a random instance of Euclidean type as follows: The vertices are chosen from the integer-grid [0, 100] x [0, 100] uniformly at random such that no two vertices coincide[18]. Among the generated vertices, we choose the given number of demand-pairs uniformly at random such that no two pairs coincide. For this procedure, the function *getRandomEuclideanInstance* described in Section 7.2 is used.

During the tests, we prevent the implementation to output information on the screen in order to have no negative influence on the running times.

9.2. An unexplainable slowdown

In the the Master Seminar [Gol18], the Gluttonous algorithm was tackled from the theoretical and practical point of view. In this context, we did some tests regarding the average running time of the algorithm. Those tests where made in October of 2018 with the same hard- and software as the one used for the test now. With the same implementation of Gluttonous some tests were performed in January 2019. The latter tests were unexpectedly slow as compared to the tests in the past[19]. In general, it was the same code, we have only changed a few lines concerning information output. Since we have always stored the different versions of the implementation, we were able to try the same version of the code as we used for the tests in October 2018, and also with this version, we had still the same problem. The values listed below should give an impression of the observed behaviour. We should admit, that the underlying test instances where not necessarily the same, but since we take the average over a larger set and try it for different input - sizes, the deviation is conspicuous.

Vertices	demand-pairs	Instances	Avg. time 2018	Avg. time 2019
8	10	100	0.004 s	0.089 s
10	15	100	0.006 s	0.162 s
20	10	100	0.056 s	1.194 s
50	10	20	1.056 s	18.52 s
100	5	20	no Test	87.68 s
100	10	20	09.52 s	?

Table 2: Running times of Gluttonous from 2018 compared to those in 2019.

The observed slowdown factor is around 20. Since we tried the same version of the code, the problem could not be within the lines of the code. Also the settings in Visual Studio where the same. The next step was to update Visual Studio, it may could be that with a new version available the old version get slow. We observed the same slowdown factor with the updated version of Visual Studio.

Personal communication with experts in information technology revealed the following potential explanation. In summer 2018, two security problems called *Meltdown* [Lipp18] and *Spectre* [Koch18] where uncovered by a group of students and researchers of the University of Technology Graz. Later in the year, a series of software updates were necessary to fix those security problems. It is well known that some of those updates also causes a restriction of the CPU performance.

[18]Thus it is possible to generate $101 \cdot 101 = 10.201$ different vertices.

[19]To be precise, the suspicious situations came up while using the implementation of the current local search algorithm. From one day to the other, quasi overnight, the running times seemed to have multiplied. Since we had not yet reference running times for this algorithm, we tried the Gluttonous algorithm as well, since we indeed had some reference values for that.

Unfortunately the slowdown described above has an impact on the size of instances which can be solved by Gluttonous in reasonable time.

9.3. 2-Conn versus 3-Conn

Our primary goal is to find out the impact of the neighbourhood used in Algorithm 7 with respect to running time and quality of the solution. Let's make the following experiment: We apply both variants of the algorithm (involving *2-Conn* and *3-Conn*) to randomly generated instances as described in Section 9.1 and compare the running times and the objective function values of the obtained solutions, respectively. We categorize the result of the comparison in "*2-Conn* better than *3-Conn*", "*3-Conn* better than *2-Conn*" and "Tie". For each set of input parameters (number of vertices and number of demand-pairs), we generate ten instances randomly. The following table shows the outcome of the experiment.

$\|V\|$	$\|\mathcal{D}\|$	#inst	$\O t_2$ $[s]$	$\O t_3$ $[s]$	$Win2$	$Win3$	Tie
8	5	10	0.284	0.272	0	0	10
8	10	10	2.636	2.596	0	0	10
10	5	10	1.910	1.840	0	0	10
10	10	10	7.320	7.224	0	0	10
10	25	10	12.28	13.11	0	0	10
15	5	10	3.997	4.491	0	0	10
15	10	10	22.59	22.55	0	0	10
15	25	10	103.1	104.0	0	0	10
20	5	10	6.740	7.810	0	0	10
20	10	10	77.96	83.84	0	0	10
25	5	10	9.025	9.497	0	0	10
25	10	10	94.96	103.1	0	0	10
30	5	10	15.53	18.46	0	0	10
30	10	10	202.5	234.9	0	0	10
40	5	10	36.96	40.47	0	0	10
40	10	10	?	?	0	0	10
50	5	10	60.69	62.96	0	0	10
50	10	10	?	?	0	0	10

Table 3: Comparing the two variants of Algorithm 7.

$|V|$ indicates the number of vertices of the instance and $|\mathcal{D}|$ indicates the number of demand-pairs, where *#inst* shows the number of considered instances. The two values $\O t_2$ and $\O t_3$ stand for the running times in seconds of the algorithms by using the *2-Conn* or the *3-Conn* neighbourhood, respectively. $Win2$ ($Win3$) indicates the number of instances, where the algorithm with the *2-Conn* (*3-Conn*) neighbourhood yields a lower objective function value as the algorithm with the *3-Conn* (*2-Conn*) neighbourhood. If both variants of Algorithm 7 yield the same objective function value when executed on a single instance, then this instance is counted as a Tie.

We can see, that for all tested instances, there is no difference in the objective function value of the solutions obtained by the two variants of the algorithm. There is also no essential difference in the running times. Of course, we have only tried a small number of different instances and also the size of the instances (number of vertices, number of demand-pairs) was chosen to be small, in order to get the results within a reasonable time.

Convention
From now on, we use Algorithm 7 always with the *3-Conn* neighbourhood, since there is no noteworthy time overhead in contrast to the algorithm that uses the *2-Conn* neighbourhood. However, the *3-Conn* neighbourhood is a superset of the *2-Conn* neighbourhood and in general it could yield a better solution.

9.4. Running times of local search

In this section, we want to get an estimation for the size of instances which can be solved by the local search algorithm within the time limit of about two minutes.

Let's have a look on the average time our computer needs to output a solution to an SFP instance by applying Algorithm 7 with the *3-Conn* neighbourhood (function *doLocalSearch)*. The times were obtained as follows: For each set of input parameters, a given number of instances of the Euclidean SFP were generated randomly as described in Subsection 9.1 (function *getRandomEuclideanInstance*). For each set of input parameters, the average running time was computed (function *doSeriesOfRandomInstancesLocalSearch*).

Vertices	Demand-pairs	Instances	Average running time
8	3	50	0.270 s
8	5	50	1.110 s
8	10	50	2.994 s
8	20	50	3.215 s
10	3	50	0.335 s
10	5	50	1.688 s
10	10	50	6.937 s
10	20	50	11.26 s
15	3	50	0.723 s
15	5	50	4.382 s
15	10	50	25.77 s
15	20	50	81.61 s
20	3	50	1.380 s
20	5	50	7.525 s
20	10	50	71.85 s
20	20	50	153.3 s
25	3	50	1.871 s
25	5	50	11.46 s
25	10	20	126.2 s
25	20	20	410.1 s
30	3	50	3.104 s
30	5	50	17.79 s
30	10	20	198.8 s
40	3	50	5.510 s
40	5	50	39.89 s
40	10	20	465.5 s
50	3	50	11.88 s
50	5	50	73.43 s
50	10	20	?

Vertices	Demand-pairs	Instances	Average running time
75	3	50	41.40 s
75	5	20	257.1 s
100	3	20	98.34 s
100	5	20	?

Table 4: Running times of local search (Algorithm 7 with *3-Conn* neighbourhood).

9.5. Running times of Gluttonous

An analogous experiment as in the last section is done with respect to the Gluttonous algorithm (functions *doGluttonous* and *doSeriesOfRandomInstancesGluttonous*). The results are summarized in the following table.

Vertices	Demand-pairs	Instances	Average running time
8	3	100	0.064 s
8	5	100	0.062 s
8	10	100	0.089 s
8	20	100	0.093 s
10	3	100	0.082 s
10	5	100	0.110 s
10	10	100	0.162 s
10	20	100	0.195 s
15	3	100	0.215 s
15	5	100	0.374 s
15	10	100	0.521 s
15	20	100	0.753 s
20	3	100	0.476 s
20	5	100	0.725 s
20	10	100	1.194 s
20	20	100	1.747 s
25	3	50	1.043 s
25	5	50	1.450 s
25	10	20	2.420 s
25	20	20	3.543 s
30	3	50	1.634 s
30	5	50	2.502 s
30	10	20	4.061 s
40	3	50	3.286 s
40	5	50	5.667 s
40	10	20	10.21 s
50	3	50	6.091 s
50	5	50	10.60 s
50	10	20	18.52 s
75	3	50	21.94 s
75	5	20	37.97 s
100	3	20	54.20 s
100	5	20	87.68 s

Table 5: Running times of Gluttonous.

9.6. Finding an optimal solution by solving the IP formulation

In Section 5.2, we discussed the integer programming (IP) formulation of the Steiner forest problem. We consider this IP formulation for randomly generated test instances (see Section 9.1) and solve then to optimality by applying the Gurobi solver. This solution approach is referred to as "exact algorithm". For each set of input parameters we generate a number of test instances and report the average running time in Table 6.

A random number of demand-pairs means, that for each instance the number of demand-pairs was chosen uniformly at random from the set of all possible numbers of demand-pairs. For an instance with n vertices, we can choose between one and $\frac{n \cdot (n-1)}{2}$ distinct demand-pairs.

Vertices	Demand-pairs	Instances	Average running time
8	random	100	0.408 s
10	random	100	0.867 s
15	5	100	1.005 s
15	10	100	5.753 s
20	5	100	2.453 s
20	10	100	43.71 s
20	20	10	328.5 s
22	10	100	85.51 s
24	10	10	252.9 s
25	5	100	8.317 s
30	5	10	21.32 s
30	10	10	?
40	5	10	465.6 s
50	-	-	stack overflow

Table 6: Running times of solving the IP formulation of SFP.

Finally, let us compare the average running times of the three algorithms (local search, Gluttonous, Gurobi solution of the IP formulation). To do so, we apply the algorithms to instances with a variable number of vertices and a constant number of demand-pairs. We choose the number of demand-pairs to be five, this allows us to consider instances with up to hundred vertices. The results are summarized by the following plot.

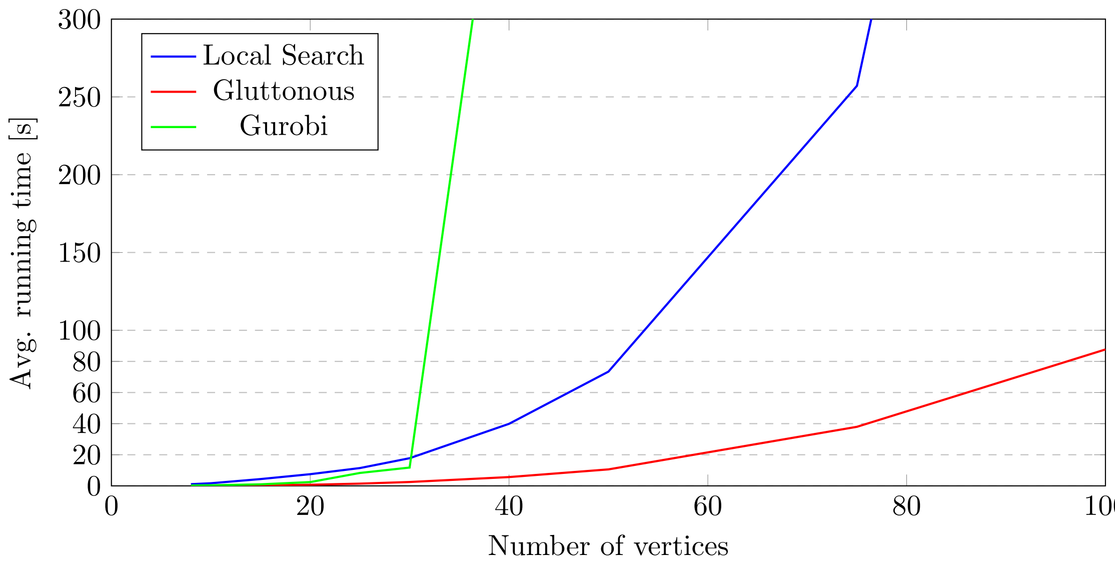


Figure 52: Comparing the running times of local search, Gluttonous and the IP formulation approach.

10. Performance of the algorithms

10.1. Quality of the solution for random instances

In this section, we want to analyze the quality of the solutions obtained by Algorithm 7 and Gluttonous. As discussed in Section 9, we know for which *size* of instances we can hope to find a solution within reasonable time. We focus on two scenarios as follows: The first scenario concentrates on instances which can be solved by all three approaches (Algorithm 7, Gluttonous, Exact Algorithm) within a reasonable time. In this scenario, we can compare[20] the solutions obtained by the Local Search algorithm (LSA) and by Gluttonous to the optimal solution. The second scenario concentrates on instances that can be solved by both the Local Search algorithm and Gluttonous within a reasonable time, but not by the exact algorithm. In this scenario, we just compare the solutions obtained by the two approximation algorithms.

Tables 7 and 8 summarize the results of the experiments in the two scenarios. In the tables we use the following notations.

$\lvert V \rvert$	Number of vertices
$\lvert \mathcal{D} \rvert$	Number of demand-pairs
#inst	Number of instances
$\varnothing t_L$ $[s]$	Average running time of LSA
$\varnothing t_G$ $[s]$	Average running time of Gluttonous
$\varnothing t_O$ $[s]$	Average running time of the exact algorithm
$\varnothing f_L$	Average approximation factor of LSA
$\varnothing f_G$	Average approximation factor of Gluttonous
$L < G$	Number of instances where LSA found a better solution than Gluttonous
$L > G$	Number of instances where Gluttonous found a better solution than LSA
$L = G$	Number of instances where the solutions generated by Gluttonous and LSA have same objective function value
$L = Opt$	Number of instances where LSA found an optimal solution
$G = Opt$	Number of instances where Gluttonous found an optimal solution

[20]In this context, this means that we compare the objective function values of the obtained solutions.

$\lvert V\rvert$	$\lvert\mathcal{D}\rvert$	#inst	Øt_L [s]	Øt_G [s]	Øt_O [s]	Øf_L	Øf_G	$L < G$	$L > G$	$L = G$	$L = Opt$	$G = Opt$
8	3	25	0.240	0.043	0.152	1.018	1.013	0	2	23	20	22
8	5	25	0.948	0.057	0.177	1.003	1.008	1	0	24	20	19
8	10	25	3.233	0.072	0.316	1.006	1.006	0	0	25	24	24
10	3	25	0.273	0.069	0.169	1.019	1.026	1	0	24	20	19
10	5	25	1.508	0.095	0.246	1.073	1.088	1	0	24	18	18
10	10	25	6.810	0.138	0.629	1.392	1.392	0	0	25	21	21
15	3	25	0.611	0.204	0.363	1.033	1.065	3	0	22	14	13
15	5	25	5.316	0.370	0.913	1.215	1.222	2	0	23	14	13
15	10	25	26.74	0.576	5.490	1.253	1.253	0	1	24	12	13
15	15	25	59.09	0.729	12.15	1.308	1.308	1	0	24	13	13
20	3	25	1.119	0.423	0.739	1.055	1.057	1	1	23	13	13
20	5	25	6.622	0.753	2.580	1.572	1.572	4	2	19	8	7
20	10	25	55.36	1.079	36.16	1.502	1.502	0	0	25	3	3
20	15	25	155.0	1.280	119.0	1.356	1.356	0	0	25	5	5
25	3	25	1.862	0.834	1.325	1.138	1.156	3	0	22	15	12
25	5	25	12.83	1.337	3.205	1.240	1.252	2	0	23	12	11
30	3	25	3.565	1.434	2.288	1.196	1.197	1	1	23	13	13
30	5	25	19.61	2.294	38.42	1.308	1.322	4	0	21	4	4
40	3	25	6.229	3.270	6.460	1.048	1.056	2	0	23	9	8
40	5	25	34.90	5.062	39.99	1.977	1.990	5	0	20	5	5

Table 7: Comparing the solutions obtained by the three algorithms (LSA, Gluttonous, exact algorithm) in the first scenario.

We found the following results for the second scenario:

$\|V\|$	$\|\mathcal{D}\|$	#inst	$\varnothing t_L$ [s]	$\varnothing t_G$ [s]	$L < G$	$L > G$	$L = G$
25	10	25	99.11	2.140	0	0	25
25	15	25	308.6	2.668	0	1	24
30	10	20	218.9	3.970	0	0	20
30	15	20	?	?	?	?	?
40	5	20	49.12	6.035	2	0	18
40	8	20	198.4	8.257	2	0	18
40	10	20	544.3	9.751	3	0	17
50	3	20	14.04	6.560	1	0	19
50	5	20	77.59	10.84	3	1	16
50	8	20	364.7	15.65	2	0	18
75	3	20	40.43	22.67	2	0	18
75	5	20	254.1	36.60	3	2	15

Table 8: Outcome of the experiment in the second scenario.

Based on the results of the two experiments we can make the following observations with respect to the behaviour of LSA and Gluttonous:

- For instances with less demand-pairs, both approximation algorithms found an optimal solution for a good portion of the number of considered instances.
- The higher the number of demand-pairs, the lower is the percentage of instances for which the algorithms (separately) found an optimal solution.
- Both approximation algorithms yield an average approximation factor smaller than two in all test instances.
- There is no significant difference in the average approximation factors of the two approximation algorithms.
- For most of the tested instances both algorithms found solutions with the same objective function value.
- The average running time of LSA is a multiple of the running time of Gluttonous in all test instances. There are some particular instances for which the running time of LSA is 50 times higher than the running time of Gluttonous.
- For instances with a small number of demand-pairs (five or smaller) the average running time of the exact algorithm is often smaller than the average running time of LSA.

Finally, let us consider instances with a larger number of vertices. Due to prohibitively long running times we just solve one randomly generated instance per set of input parameters (and not a series of instances with the same input parameters as in the previous experiments). The outcome can be seen in the table below, where *Local* is the objective function value obtained by LSA and *Glut* is the corresponding value obtained by Gluttonous.

Name	$\|V\|$	$\|\mathcal{D}\|$	$Øt_L$ $[s]$	$Øt_G$ $[s]$	*Local*	*Glut*
Test01	75	8	1093	51.71	242.6	242.6
Test02	75	10	3588	78.23	310.8	310.8
Test03	60	10	681	26.94	252.9	252.9
Test04	60	12	890	32.65	274.5	274.4
Test05	60	15	4580	44.20	369.8	369.8
Test06	50	15	4007	31.98	313.1	313.1
Test07	55	12	1889	27.20	199.2	199.2
Test08	65	11	4342	47.09	258.8	258.8
Test09	100	5	356	88.33	186.3	186.3
Test10	100	8	3794	154.2	295.0	295.0
Test11	100	10	?	167.9	?	236.2

Table 9: Comparing LSA and Gluttonous for some single test instances.

For all ten instances, where both algorithms provide a solution within two hours of running time, the objective of the obtained solution is the same. As for running times, also this experiment shows that our implementation of Gluttonous is much faster than that of LSA.

10.2. Performance on SFP instances known in the literature

At *www.steinlib.zib.de/steinlib.php* one can find instances of the Steiner Tree Problem together with the corresponding best objective function value known so far. Since the Steiner Tree Problem is a special case of the Steiner forest problem, we can use them to test our algorithms. To execute our implementation of LSA and Gluttonous on this instances in the given *SteinLib - Format*, we have provided the choice *SteinLib Format Graph* as an input format. Note that one has to modify the original format of the given files slightly, but of course, without modifying the underlying instance. Details can be found in the comments that are made in the code.

In the table below, we summarize the results obtained by applying LSA and Gluttonous to some of the instances mentioned above. *Opt* indicates the optimal value for each instance, *Local* is the value of the solution obtained by LSA. f_L is the quotient *Local/Opt.* The last column shows the running time of the LSA. Note, that according to some comments on the website mentioned above, all instances in Table 10 can be solved in less than a minute. It is not specified which software and hardware was used.

Comment
The Steiner Tree instances are given in the general setting described in Definition 1.7 with the exception that there are no demand-pairs specified, but just a set of terminals. These Steiner Tree instances can be transformed to SFP instances by introducing a demand-pair for each pair of terminals in the original instance. We use the construction described in Theorem 3.31 to obtain an instance in the desired input format to apply LSA and Gluttonous.

Name	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{D}\rvert$	Opt	Local	f_L	t_L [s]
berlin52	52	1326	120	1044	1069	1.024	255.9
brazil58	58	1653	325	13655	?	?	?
p455	100	4950	10	1138	1166	1.025	48.60
p456	100	4950	10	1228	1239	1.009	46.65
p457	100	4950	45	1609	1642	1.021	362.5
p458	100	4950	45	1868	1868	1.000	308.0
p459	100	4950	190	2345	2348	1.001	2229
p460	100	4950	190	2959	3010	1.017	2239
p461	100	4950	1275	4474	?	?	?
lin01	53	80	6	503	503	1.000	3.986
lin04	157	266	15	1239	1267	1.023	284.5
msm1844	90	135	45	188	196	1.043	215.8
msm4224	191	302	55	311	333	1.071	2763

Table 10: LSA applied to some Steiner Tree instances.

Finally, we consider the performance of Gluttonous on those instances. The results are summarized in Table 11, where *Glut* denotes the objective function value of the solution obtained by Gluttonous and f_G indicates the approximation factor of Gluttonous, i.e. $f_G = Glut/Opt$. The last two columns contain the running times of the respective algorithms.

Name	Opt	Local	Glut	f_L	f_G	t_L [s]	t_G [s]
berlin52	1044	1069	1069	1.024	1.024	255.9	22.86
brazil58	13655	?	13682	?	1.002	?	42.98
p455	1138	1166	1166	1.025	1.025	48.60	44.45
p456	1228	1239	1239	1.009	1.009	46.65	44.41
p457	1609	1642	1642	1.021	1.021	362.5	102.4
p458	1868	1868	1868	1.000	1.000	308.0	101.0
p459	2345	2348	2348	1.001	1.001	2229	182.7
p460	2959	3010	3010	1.017	1.017	2239	185.3
p461	4474	?	4491	?	1.004	?	401.5
lin01	503	503	503	1.000	1.000	3.986	5.561
lin04	1239	1267	1267	1.023	1.023	284.5	211.0
msm1844	188	196	196	1.043	1.043	215.8	68.02
msm4224	311	333	333	1.071	1.071	2763	734.0

Table 11: LSA and Gluttonous applied to some Steiner Tree instances.

We see that for all instances which are solved by both algorithms within a reasonable time, the objective function value of the obtained solutions coincide. For some instances, the running times are close to each other, but there are also instances (e.g. p459, p460), where the running time of LSA is about ten times the running time of Gluttonous. Note also, that LSA never outperforms Gluttonous in terms of running time.

11. Conclusion

In this Master Thesis, we analyzed and implemented a local search algorithm (LSA) for the Steiner forest problem originally introduced by Gross et al. [G17]. The moves that define the neighbourhood of the feasible solution are easy to understand, but some of them cause troubles when coming to the time analysis of the algorithm. We discussed in details the approaches proposed in the literature to handle those obstacles and obtain an algorithm, that has on the one hand a polynomial running time and on the other hand a constant approximation factor. The resulting algorithm, which has both properties, is not as easy to implement compared to some other known approximation algorithms for the Steiner forest problem, as for example, the Gluttonous algorithm. The main problem is, that the presented local search algorithm is rather of theoretical nature while the Gluttonous algorithm is more practical.

Nonetheless, at least we implemented a modified version of the local search algorithm. The modification consists in replacing the complex *Improving-connecting* move with some simpler local improvement technique (*2-Conn* and *3-Conn* neighbourhood) and in dropping the rounding technique of the edge lengths in order to achieve a more practical algorithm. Clearly, the theoretical results on the approximation guarantee and time complexity of LSA do not hold for the modified version of the algorithm. Different tests on randomly generated instances of small size show some interesting details of our implementation. First, the algorithm performs very well on instances with a small number of vertices and demand-pairs (less than forty vertices and less than fifteen demand-pairs), and secondly, the approximation factor is in most cases smaller than two. When applying Gluttonous to the same instances, the obtained objective function values are in most cases the same as those obtained by LSA, but the running times of Gluttonous are a good deal better. This behaviour can also be observed for tests where the number of vertices is below one hundred and the number of demand-pairs is below twenty. Summing up, our implementation of the local search algorithm yields almost-always solutions of the same quality as the solutions obtained by Gluttonous, only in very few cases we obtain a better solution. However, the running time of our LSA implementation is in general a multiple of the running time of Gluttonous, hence there is no substantial reason to prefer this modification of the local search algorithm to Gluttonous.

An interesting open question which was not dealt within this work would be to implement the original polynomial time local search algorithm (Algorithm 5) with the proved approximation factor and to compare it with Gluttonous and the exact algorithm in terms of running time and solution quality. It would be also interesting to know whether our modified version of the LSA would be competitive with the original LSA in terms of both running time and solution quality.

References

[AKR95] A.Agrawal, P.Klein, R.Ravi, *When trees collide: an approximation algorithm for the generalized Steiner problem on networks*, SIAM J. Comput., 24(3):440-456, 1995

[AK00] Sanjeev Arora, George Karakostas, *A 2 + ϵ Approximation Algorithm for the k-MST Problem*, Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (David Shmoys, ed.), SODA'00, Society for Industrial and Applied Mathematics, 2000

[AR98] Sunil Arya, H. Ramesh, *A 2.5-factor Approximation Algorithm for the k-MST Problem*, Information Processing Letters 65, no. 3, 117-118, 1998

[AAB04] Baruch Awerbuch, Yossy Azar and Yair Bartal, *On-line generalized Steiner Forest*, Theoretical Computer Science, Volume 324, Issues 2-3, 20. September 2004, Pages 313-324, 2004

[BRV96] Avrim Blum, R. Ravi, Santosh Vempala, *A Constant-factor Approximation Algorithm for the k-MST Problem*, Proceedings of the Twenty-eight Annual ACM Symposium for the Theory of Computing (Gary L. Miller, ed), STOC'96, ACM, pp.442-448, 1996

[EB02] Christian Blum, Matthias Ehrgott, *Local search algorithms for the k-cardinality tree problem*, Science Direct, Volume 128, Issues 2–3, Pages 511-540, 2003.

[EH] I.N. Bronshtein, K.A. Semendyayev, Gerhard Musiol, Heiner Mühlig, *Handbook of Mathematics*, Springer, 6th edition, ISBN: 3662462214, 9783662462218, 2015

[F94] Mattheo Fischetti, Horst W. Hamacher, Kurt Jornsten, Francesco Maffioli, *Weighted k-Cardinality Trees: Complexity and Polyhedral Structure*, Networks 24, no. 1,11-21, 1994

[Gar96] Naveen Garg, *A 3-approximation for the Minimum Tree Spanning K Vertices*, Proceedings of the Thirty-seventh Annual Symposium on Foundations of Computer Science, FOCS'96, IEEE Computer Society, pp. 302-309, 1996

[Gar05] Naveen Garg, *Saving an Epsilon: A 2-approximation for the k-MST problem in Graphs*, Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing (Ronald Fagain and Hal Gabow, eds.), STOC '05, ACM, pp.396-402, 2005

[GW95] Michel X. Goemans and David P. Williamson, *A general approximation technique for constrained forest problems*, SIAM J. Comput., 24(2):296-317, 1995

[Gol18] Stefan Golja, *Steiner forest problem - The Gluttonous Algorithm*, Elaboration for the Master's Seminar, 2018.

[G17] Martin Gross, Anupam Gutpa, Amit Kumar, Jannik Matuschke, Daniel R. Schmidt, Melanie Schmidt, José Verschae, *A Local-Search Algorithm for Steiner Forest*, Cornell University Library, arXiv:1707.02753, 2017

[GK14] A.Gupta, A.Kumar, *Greedy Algorithms for Steiner Forest*, Cornell University Library, arXiv:1412.7693v1, 2014.

[JMP00] David S. Johnson, Maria Minkoff and Steven Phillips, *The Prize Collecting Steiner Tree Problem: Theory and Practice*, Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '00, Society of Industrial and Applied Mathematics, pp. 760-769, 2000

[Koch18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom, *Spectre Attacks: Exploiting Speculative Execution*, 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[KV06] Bernhard Korte, Jens Vygen, *Combinatorial optimization: theory and algorithms*, Springer Verlag, 2006.

[Lipp18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher,Daniel Genkin, Yuval Yarom and Mike Hamburg, *Meltdown: Reading Kernel Memory from User Space*, 27th USENIX Security Symposium (USENIX Security 18), 2018.

[San03] Alessandro Santuari, *Steiner Tree NP-completeness Proof*, Exercise for the Computational Complexity course taken at the University of Trento, 2003.

[SZM17] Daniel R. Schmidt, Bernd Zey and Francois Margot, *MIP Formulations for the Steiner forest problem*, Cornell University Library, arXiv:1709.01124, 2017.

[St07] Angelika Steger, *Diskrete Strukturen, Band 1: Kombinatorik, Graphentheorie, Algebra*, Springer Verlag, 2007.

[Tre11] Luca Trevisan, *Handout 2 for CS261 - Optimization*, Handout 2 for the course "Optimization" on January 6, 2011 at Stanford University, https://people.eecs.berkeley.edu/~luca/cs261/lecture02.pdf, 2011.

List of Figures

List of Tables