Technische
Universität
Wien

Vienna
University of
Technology

Master's Thesis

Metaheuristic Approaches for Designing Survivable Fiber-Optic Networks

written at the Institute for
Computer Graphics and Algorithms
of the Vienna University of Technology

under supervision of
Univ. Prof. Dipl. Dr. techn. Günther Raidl
Dipl.-Ing. Daniel Wagner

by
Thomas Bucsics, Bakk. tech.
Vorgartenstr. 129-143/1/24
1020 Wien

Abstract

In den letzten Jahren sind Glasfaser-basierte Netzwerke für einzelne Haushalte leistbar geworden. Materialien und Konstruktionsarbeiten, um ein flächendeckendes Netzwerk aufzubauen, bringen hohe Kosten mit sich. Daher wäre ein exakter Algorithmus zur Lösung des Problems der Berechnung von optimalen Kabelrouten wünschenswert. Aufgrund der hohen komputationalen Komplexität des Problems können in vielen Fällen nicht innerhalb eines akzeptablen Zeitrahmens berechnet werden. Heuristiken und Meta-Heuristiken stellen in diesem Fall Methoden zur Erstellung approximativer Lösungen dar, die oft als Ausgangspunkt für zeitintensivere Methoden dienen können oder direkt für praktische Zwecke ausreichend sind. Diese Arbeit beschreibt eine formale Betrachtungsweise der mit der Anbindung von Kunden an ein bestehendes Netzwerk verbundenen Probleme, die das beschriebene Problem in die Konstruktion und Augmentierung eines Steinerbaumes aufspaltet. Es wird ein Überblick über den derzeitigen Stand der Forschung auf diesen Gebieten, sowie dem Gebiet der Gesamtlösung eines solchen Problems gegeben, der sowohl exakte als auch approximative, heuristische und meta-heuristische Methoden miteinbezieht. Einige mögliche Konstruktionsheuristiken zur Lösung dieser Probleme, sowie mögliche Optimierungsalgorithmen zur Verbesserung solcher Lösungen werden im Detail dargestellt. Weiters werden Move-Operatoren und Nachbarschaften beschrieben. Meta-heuristische Methoden wie lokale Suche, Simulated Annealing, Variable Neighborhood Descent und Variable Neighborhood Search werden unter deren Nutzung untersucht, sowie einige hybride Methoden, die eine Menge an heuristisch konstruierten Lösungen als Basis nutzen, um ein Problem, das kleiner als das Gesamtproblem ist, zu definieren, das dann von exakten oder anderen, zeitintensiveren Methoden gelöst werden kann. Eine Beispielimplementierung, die auch im Zuge von Experimenten genutzt wurde, sowie die durchgeführten Experimente und verwendeten Probleminstanzen werden beschrieben. Abschliessend werden die Resultate dieser Experimente den Resultaten einander gegenübergestellt und ihre Effektivität anhand von Vergleichen mit exakten Verfahren gemessen.

During the last years fiber-optic based communication networks have become affordable for individual households. Due to the costly nature of the materials and infrastructure that have to be deployed to distribute such a network over a large area, exact algorithms for calculating a provably optimal cable routing are desirable. However, due to the computational complexity of the problem, such algorithms are not always able to determine an optimal solution within acceptable time. Heuristics and metaheuristics can provide techniques to calculate approximate solutions that are often sufficient as a starting point for certain, more time-consuming approaches, or even sufficient for practical purposes. This thesis defines a formal way to view the problems that are posed by connecting a set of customers to an existing infrastructure, and will split up the described problem into the construction and augmentation of a Steiner tree. An overview of some state-of-the-art approaches for the solution of the problem at hand, as well as for the solution of the sub-problems on which our heuristics are based, is given. Several construction heuristics are suggested, as well as local improvement procedures based on different neighborhood structures. Local Search, Simulated Annealing, Variable Neighborhood Descent and Variable Neighborhood Search are furthermore studied in detail. Also, hybrid methods using different heuristics to construct a diverse set of candidate solutions which are then used to define a problem smaller than the original problem to be solved by exact or other, more time-demanding methods, is proposed. An implementation of these methods is described, and conducted experiments are documented. Experimental results of the various described methods will be compared, and their effectiveness will be measured by comparison to existing exact methods.

Acknowledgments

First of all, I would like to thank the two persons who made this master's thesis possible: Daniel Wagner and Günther Raidl at the department for algorithms and data structures. Their continuous support and patience and resources made developing, researching and writing this paper just that much easier. Secondly, I'd like to thank my parents, Anna and Stefan Bucsics for their support. A lot of thanks also go to the people at ANECON, who gave me the possibility of working part-time to finish my studies, and for not wanting me to abandon those studies, even when the demand for our services was high. Last, but not least, I want to thank my significant other, Sabine, for supporting me lovingly during this time, and enduring the annoyances of having a partner who at times was lost in his work.

Contents

1 Introduction and Background

1.1 Designing the Last Mile in Fiber Optics Networks

Fiber-optic based networks have become more and more common during the last few years, and have reached households. The materials and infrastructure that have to be deployed to distribute such a network over a large area (mainly cable laying) are expensive, so algorithms for calculating efficient cable routes are needed. Since the problem this poses is a rather complex one, such algorithms are often not able to determine an optimal solution within an acceptable period of time. For this reason we decided to explore heuristic and meta-heuristic approaches in this field.

1.2 NETQUEST

The *NETQUEST* project was started at the Carinthia University of Applied Sciences in 2005. It is supported by the FHplus program which was founded by the Austrian Research Promotion Agency. It focuses on the development of decision supporting tools for network carriers for simulation and optimization of cable laying routes or network augmentation projects. The project is supported by a consortium consisting of various academic and commercial partners, including the Institute for Computer Graphics and Algorithms of the Vienna University of Technology.[17] This thesis and the corresponding implementation are meant to be part of that project.

1.3 Overview of this Thesis

First, we will introduce some concepts, notations and definitions related to the problem treated in this document. Then we will give an overview of existing approaches for the problems we described, including exact, heuristic and approximative approaches. We will introduce our own solution construction and improvement heuristics, and then present various meta-heuristic and hybrid approaches employing these heuristics. Following this, we will give an overview of an implementation of the described heuristic and meta-heuristic approaches, and show tests methods and results. Finally, we will compare and evaluate the experimental results, and present some thoughts on possible future approaches based upon the results of this thesis.

2 Definitions and Notations

2.1 Definition of Heuristic

There are many hard optimization problems in Computer Science. Usually, one aims to find algorithms to solve them with provably good run times and provably good solution quality. Heuristics, often described as ”rule of thumb” algorithms, give up one of these two goals, in our case the goal related to solution quality. The solutions provided by the algorithms described in this thesis will not have a proved approximation quality, and will, in some cases, not even be able to determine any valid solution for a given problem, even if one exists. They are, however, designed to have a polynomial running time, and give a valid solution for most real-world instances we considered. One of the first uses of heuristic construction methods can be found in [13].

2.2 Definition of Meta-Heuristic

The term "meta-heuristic" was first introduced in [14]. Meta-Heuristics are heuristic methods for solving complex computational problems with the property of employing other heuristic or approximative methods, usually in a fashion such that the meta-heuristic method needs little or no information about the heuristic methods it uses besides of how to access it. This has the advantage of being able to use a variety of heuristics with one and the same meta-heuristic algorithm.

2.3 Basic Graph-Related Definitions and Notations

For the purpose of this thesis we will refer to *networks* simply as *graphs* and only consider connected, undirected graphs.

A graph consisting of a set of *nodes* or *vertices* V and a set of *edges* or E, with each edge e connecting two nodes $(v, w) \in V$, and a *cost function* $c(e)$, denoting the cost of an edge $e \in E$, will be denoted as $G = (V, E, c)$. *Sub-graphs* of such a graph $G = (V, E, c)$ will be denoted as $G_X = (V_X, E_X)$ or $X = (V_X, E_X)$, with X identifying the sub-graph, $V_X \subseteq V$ and $E_X \subseteq E$.

If an edge $e \in E$ connects two nodes $v \in V$ and $w \in V$, e will be called *incident* to v and w, and v and w will be called *adjacent* to each other.

We will use the term *degree* of a node v to denote the number of incident edges a specific node v has within a graph or sub-graph. We will use the terms weight and cost of an edge $e \in E$ interchangeably to denote $c(e)$.

The term *path* will denote a sequence of nodes, such that from each connected node there exists an edge to the next node in the sequence. The term *cycle* denotes a path whose first node is also the last node of the sequence. For the purpose of this thesis, we will usually refer to cycles that do not contain any nodes more than once, except for the starting node. We shall refer to this as a *cycle without repeated vertices*.

2.4 Definition of the OPT-problem

One possibility for a formal view for the basic problem we consider would be to use a weighted, connected, undirected graph $G = (V, E, c)$ with nodes V, a set of edges E representing straight segments of potential cable routes, and a cost function $c(e)$ assigning a cost to each edge $e \in E$ to model potential cable routes. A set of vertices $J \subseteq I \subseteq V$ describes all the nodes already within an existing infrastructure that allow the connection of additional cable routes. A further set of vertices $C \subseteq V$ describes *customer nodes* that have to be connected to the infrastructure. Two disjunct subsets $C_1 \subseteq C$ and $C_2 \subseteq C$ with $C_1 \cap C_2 = \emptyset$ describe customers that require a single connection to the infrastructure (denoted C_1) and customers that require a node-redundant connection to the infrastructure (denoted C_2). Our goal is to satisfy all the connection requirements posed by the nodes C_1 and C_2 using a set of edges from the edge set E of minimum total weight. This problem will be referred to as the *OPT-problem* [3].

2.5 Technical Conditions

The OPT problem can also be seen as a set of constraints, under which a solution subgraph of the problem graph has to be found, with its set of edges having minimal total weight [3].

- *The Junction constraint:* All the customer nodes C have to be connected to the existing infrastructure. New connections can only be attached to junction nodes $J \subseteq V_I$.
- *The Biconnectivity constraint:* Customer nodes contained in C_2 have to be connected to the existing infrastructure in a vertex redundant way, meaning that at least two node-disjoint paths from the infrastructure to each node in C_2 have to be included within the solution.
- *The Non-crossing constraint:* Connections may not cross each other in a geometric sense. This constraint will not be considered by some of the meta-heuristic approaches described in this paper.

2.6 The Steiner Tree Problem

Given an undirected, weighted graph $G = (V, E, c,)$ and a subset $T \subseteq V$ representing so-called *terminal nodes*, a tree on G represented by a sub-graph $G_T = (V_T, E_T)$ that contains all the nodes in T is called a *Steiner tree.* Any non-terminal node $s \notin T$, $s \in V_T$ is called a *Steiner node*. The problem of finding a Steiner tree with minimum weight is one form of the *Steiner tree problem*, and thus NP complete, which means it is computationally hard and not solvable in polynomial time (unless $P = NP$[1]) [12][18].

The Steiner tree problem is considered to be especially relevant to the areas of routing in computer networks, VLSI layout and phylogeny (the study of the evolution of life forms) [24].

For our purposes, the term *terminal node* will usually apply to all nodes $v \in (C \cup J)$.

[1] P (Polynomial) and NP (Non-Polynomial) are complexity classes, P denoting problems that can be solved in polynomial time, and NP is the class of problems that can be solved in polynomial time by non-deterministic Turing-machines. This complexity class was first described in [7], even though the term ”NP” or ”NP-complete” was not used in this publication. The question of whether or not $P = NP$ is unsolved at the time of writing of this thesis.

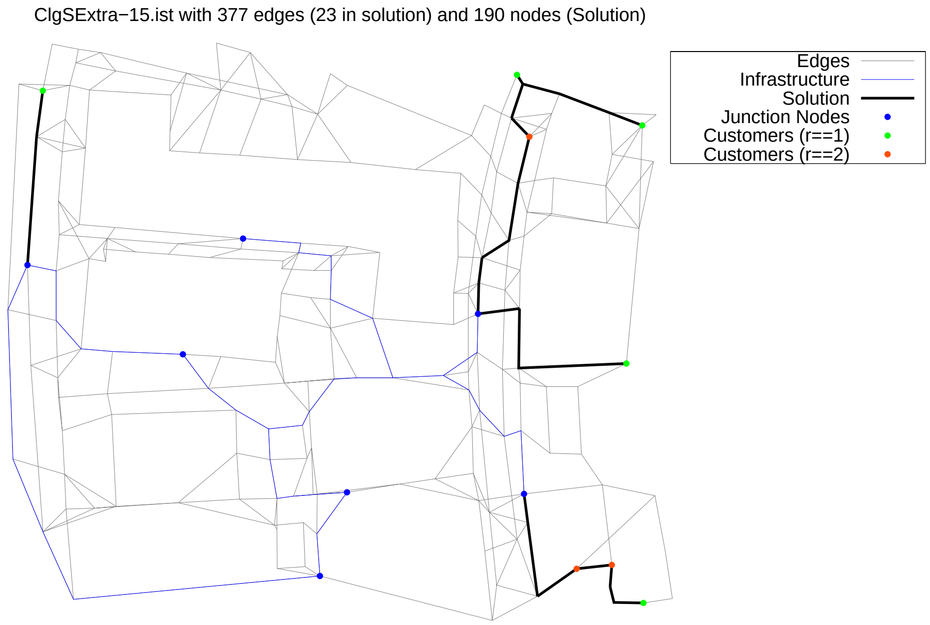

ClgSExtra−15.ist with 377 edges (23 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 1: Example of a Steiner tree on a real-world problem instance *ClgSmall-I1-15*

2.7 Key-Nodes and Key-Paths

A *key-node* $k \in K$ is a non-terminal-node in a Steiner tree T (or an OPT-solution subgraph $G_S = (V_S, E_S)$) for a problem instance graph $G = (V, E)$, if that k has a degree of at least 3 within T (or G_S), with K being the set of all key-nodes within T (or G_S).

A *key-path* is a path that has either a terminal node or a key-node at each end, and whose intermediate nodes (if any exist) are all Steiner nodes with a degree of 2 within T.

2.8 The Vertex Biconnectivity Augmentation Problem

A connected, undirected graph $G = (V, E)$ has a *vertex-connectivity* degree of $C_V(G)$, $C_V(G) \geq 1$ if at least $C_V(G)$ nodes and their incident edges have to be removed from G in order to make G no longer connected. An alternative formulation would be to define G as being vertex c-connected if c nodes and their incident edges can be removed from G without G becoming disconnected after this operation. Proof of the equality of these two formulations can be found in [5].

The process of turning a sub-graph $G_S = (V_{S,} E_S)$ of an undirected, weighted graph $G = (V, E, c)$ into a sub-graph with $C_V(G_S) \geq 2$ by adding a set of edges $AUG \subseteq E$ from G is called *vertex biconnectivity augmentation* (this thesis will from now on use the term *augmentation* to denote this). The problem of finding such a set AUG having minimum total weight in G is called the *vertex biconnectivity augmentation problem*, and is also NP complete [12][1].

For our purposes, biconnectivity needs only to be established between an infrastructure node and a node in C_2, and not on the whole graph.

2.9 Symbols Used in Figures of Graphs

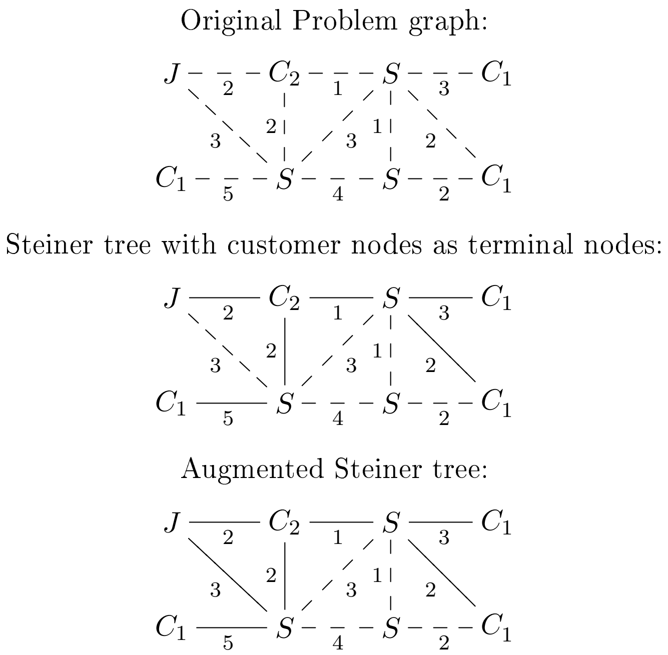
We will use some figures to give examples of problem graphs and the effect of various algorithms on them.

Nodes will be denoted using the following notation:

- J will denote an infrastructure junction node $j \in J$.
- S will denote a possible Steiner node $s \in S$.
- C will denote a terminal node $c \in C$.
- C_1will denote a node $c_1 \in C_1$ requiring a single connection to a junction node.
- C_2 will denote a node $c_2 \in C_2$ requiring a redundant connection to the infrastructure sub-graph.
- A node of the form X^Y will indicate a node of type X with a special mark Y, in order to make an reference to that specific node possible.

Edges will be depicted after the following fashion:

- A connection from one node to another of the form " —— " will denote an edge in the graph or sub-graph referenced in the title of the graph.
- A connection of the form " – – – " in a sub-graph illustration will denote an edge in the original problem graph that is not contained in the sub-graph.
- A connection of the form " ⟶ " in a sub-graph illustration will denote an edge that has been added to the shown sub-graph by the step of the algorithm that is to be illustrated by the illustration. If the arrow points toward a node, and the set of newly included edges seems to generally point away from this node (such as, for example, a junction node), then it is usually to indicate that this edge was included to satisfy the redundancy constraint.
- A connection of the form " ·········· " in a sub-graph illustration will denote an edge that has been removed from the sub-graph by the algorithm or step of an algorithm that is to be illustrated by the graph.
- A number above or below any connection between two nodes (for example " $\underset{x}{\text{——}}$ " or " $- \overset{x}{-} -$ ") will denote x as the cost of that edge within the problem graph.


Original Problem graph:

Steiner tree with customer nodes as terminal nodes:

Augmented Steiner tree:

J: junction node, C_1: customer node requiring a single connection, C_2: customer node requiring a redundant connection, S: possible Steiner node

Figure 2: Example of the construction of a heuristic solution by Steiner tree construction and augmentation

3 Related Work

3.1 Overview

A great deal of work can be found on topics related to the OPT-problem, especially on Steiner trees and connectivity augmentation. In the following, we do not claim to cover all of the related work in this field, but will just highlight a few methods to illustrate what kind of approaches exist for the problems we have described.

3.2 Related Exact and Approximative Approaches

3.2.1 Linear Programming Approaches for the Steiner Tree Problem

Many exact approaches for the Steiner tree problem have been designed. An overview of some fundamental examples can be found in [24]. They are generally based on mixed or integer *Linear Programming*, which is an approach using a transformation of the underlying problem into an optimization problem. Linear programs are usually represented using a matrix A with dimensions $m \times n$, an m-vector b and an n-vector c. The goal is to find a vector x with n elements such that n maximizes an *objective function* $\sum_{i=1}^{n} c_i x_i$ while respecting the m constraints defined by $Ax \leq b$. Many problems in operations research can be formulated this way, and many approaches as well as implementations for solving such problems exist for Linear Programming problems. Similarly, minimization problems can be formulated [8].

Some of the more common formulation types (which we will not describe in detail, since this would require exact formulation of the respective constraints) for the Steiner tree problem summarized in [24] are:

- *Cut-Formulations* minimize the total solution cost while using so-called Steiner cut constraints to guarantee that in any arc-set representing a solution there is a path from a terminal to any other terminal

- *Flow-Formulations* minimize total solution cost under the constraint of a flow of one unit of a commodity from a terminal node to every other terminal node.

- *Tree-Formulations* are tree-based formulations using linear relaxations and minimizing total solution cost.

3.2.2 Cutset-Based Approximation for the Minimum-Cost Vertex-Connectivity Problem

This approach, described in [25], is based on the concept of vertex neighborhoods and cut sets. The vertex neighborhood V_{sub}^n of a sub-graph $G_{sub} = (V_{sub}, E_{sub})$ of a graph $G = (V, E)$ is the set of all nodes $v \in (V \setminus V_{sub}) | (v, w) \in E$ with w being any node $\in V_{sub}$). A cut-set C is a set of nodes from a graph $G = (V, E)$ that, when all nodes within the set, and all incident edges are removed from G, causes G to have less connected components.

The approach described in [25] uses a number of phases equivalent to the desired vertex-connectivity k. In the kth phase, the graph becomes k-connected. It does this by using an augmentation function that, in turn, uses a function $h(S)$ (with $S \subseteq V$) that, in phase p, returns 1 if $S \subseteq V_S$ (with $V_S = (V, E_S)$ being the current solution sub-graph of G) is a cut set of size $p - 1$ and is the smaller of two parts of G when the vertex neighborhood of G_R(with G_R being the sub-graph of G defined by the nodes in S and all incident edges) is removed. The augmentation function returns a set of edges $F \cup E$ which increases the vertex neighborhood for G_R. The sub-graph $G'_S = (V, E_S \cup F)$ becomes the new solution graph G_S for the next phase. [25] shows that S becomes k-connected in phase k.

This approach can be shown to have a guaranteed quality of 3 for the case of $0, 1, 2$-connectivity (which is similar to the problem which is treated in this thesis) and a general quality for k-connectivity of $2H(k)$ with $H(n) = 1 + \frac{1}{2} + ... + +\frac{1}{n}$).

3.2.3 Multi-Commodity Flow Approaches for the OPT-Problem

One such approach is described in [29]. It is based on the concept of sending commodities from a root node representing the infrastructure to each customer node. For customers requiring a redundant connection to the infrastructure, two commodities are sent, which may not share any edges or have any other nodes other than the root node or the target customer node in common.

An Integer Linear Programming formulation for the problem is given, which is formulated using, among others, the same constraints that we described earlier, and uses total solution sub-graph cost as objective function.

Experiments showed that this approach is able to solve smaller or rather sparse problem instances to optimality within reasonable time.

3.3 Related Heuristic Approaches

3.3.1 Shortest Path Based Heuristics for the Steiner Problem

A Shortest Path Heuristic by Takahashi and Matsuyama This approach is a shortest-path heuristic (denoted by the authors with the abbreviation *SPH*) to construct a solution $G_S = (V_S, E_S)$ on a given problem graph *G=(V,E,c)* with a set of terminal nodes $T \subseteq V$. It first calculates a shortest path tree represented by the sub-graph $S_T = (V_T, E_T)$ on the problem graph G, starting from a given root node $r \in T$. It then adds each terminal node $t \in T$ to G_S by its shortest path, by iteratively adding all predecessor nodes of t, as well as the edges in T that connect one predecessor to the next, from the shortest path tree S_T to G_S until r is reached. The order of addition of the nodes is by ascending distance to the root tree [28].

3.3.2 Minimum Spanning Tree Based Heuristics for the Steiner Problem

Shortest-Path Minimum Spanning Tree This heuristic resembles Kruskal's minimum spanning tree algorithm, which is described, among many others, in [8]. It was presented among others in [26].

It first splits the problem graph instance $G = (V, E, c)$ with a set of terminal nodes $T \subseteq V$ into a set of connected components. Initially, there are $|T|$ such components. In each iteration, the shortest path in G for joining two such connected components is found, and the two components and this path are joined into a new, bigger component. This is repeated until all components have been merged into a single component, containing all the nodes in T [26].

Pruned Minimum Spanning Tree This heuristic was presented in [26]. It first constructs a minimum spanning tree $S_T = (V_T, E_T)$ on a given problem graph $G = (V, E, c)$ with a given set of terminal nodes $T \subseteq V$. If all leaves of S_T are terminal nodes, S_T is returned. If not, nodes with a degree of one within S_T are removed, and a new minimum spanning tree S'_T is computed on the graph induced by the nodes of S_T. This procedure is repeated until all leaves of the most recent minimum spanning tree S''_T are terminal nodes [26].

3.4 Related Meta-Heuristic Approaches

Many meta-heuristic approaches for the Steiner tree problem and other related problems have been evaluated, and we will only describe a few which we selected as examples for a much larger set of existing approaches.

3.4.1 Hybrid Local Search for the Steiner Problem in Graphs

One approach using hybrid Local Search is described in [9].

An initial solution is created using the shortest path heuristic algorithm SPH we described earlier.

The local search algorithm itself is based upon this construction heuristic and on the concept of *key-nodes*. The algorithm either tries to introduce or remove key-nodes from the current solution G_S. The set of key-nodes within the solution G_S will be denoted, as defined earlier, by K.

- In the first case, a non-terminal, non-key node $v \in (G_S \setminus T) \setminus K$ is selected from the solution sub-graph. A new set of terminal nodes $T' = \{v\} \cup K \cup T$ is defined, and a new solution G'_S is calculated using these terminal nodes and the construction heuristic described earlier, with a random terminal node $r \in T'$ as root.

- In the second case, a new set T' of terminal nodes is defined, but this time with $T' = (K \setminus \{v\}) \cup T$. As before, the construction heuristic we already described constructs a new solution G'_S with T' as the set of terminal nodes, starting from a random root $r \in T'$. Now nodes of degree one within G'_S are iteratively removed.

In both cases, if G'_S has a lower total edge cost than the current solution G_S, G'_S becomes the new current solution G_S.

The results provided by this algorithm are considered to be competitive with other approaches [9].

3.4.2 Hybrid GRASP with Perturbations for the Steiner Problem in Graphs

Similar to the previously described approach of a hybrid local search, a second, more advanced approach for solving the Steiner problem on a weighted problem Graph $G = (V, E, c)$ with a given set of terminal nodes $T \subseteq V$ was proposed in [26]. This approach is based on the concept of a Greedy Randomized Adaptive Search Procedure (GRASP, first presented in [11]), which is an iterative meta-heuristic approach based on a two-phase algorithm: construction and local search, over multiple iterations, with the best solution that was encountered being returned.

In this case, a multi-start approach is described, in which the construction phase of a GRASP algorithm is replaced by the combination of several construction heuristics with a weight perturbation strategy.

Initially, a valid solution is generated using a randomized greedy algorithm, which is then used as a starting point for a local search. Candidate algorithms are the heuristics SPH, Shortest-Path Minimum Spanning Tree and Pruned Minimum Spanning Tree which were all described earlier in the section about heuristic and approximate approaches.

For local search the key-node based approach we described earlier is used, or a key-path based neighborhood. This neighborhood is found by removal of key-paths, with subsequent reconnection of the two components that the original solution is split into by this action. The created solution becomes the new candidate solution if the new solution is better than the previous candidate solution, or if it is equal but contains more terminal nodes as extremities.

At each iteration, weight perturbations are applied to the problem graph to introduce some noise into the original weights. The weight perturbation strategy incorporates learning mechanisms associated with intensification and diversification strategies that have their origin in the field of tabu search.

After the maximum number of iterations is reached, an improving strategy based on path-relinking is applied to the best solution that was encountered. First, a pool of a number of different elite solutions that was found during the search is retained from the set of generated solutions. Then path-relinking is applied to this pool, thus generating a new generation of candidate solutions. This is repeated until no further progress can be made. Three strategies for path-relinking are described in [26]:

- *Path-relinking by complementary moves* uses symmetric differences of two candidate solutions to determine a move that is to be applied to the initial solution.
- *Path-relinking by weight penalization* applies a weight perturbation on a pair of elite solutions, and then applies the shortest-path-heuristic to the perturbed instance. The local search procedure is then applied to the resulting solution.
- *Adaptive hybrid path-relinking* applies both schemes for combining every solution in the pool with the best solution in the pool, and then applies the path-relinking scheme with the lowest average computation time to all other pairs of solutions in the pool, or path-relinking by weight penalization if computation times are equal.

3.4.3 Ant Colony Algorithms for Steiner Trees

Ant colony optimization algorithms are a class of algorithms that try to mimic the complex cooperative patterns of social interaction present in real ant colonies. Ants can communicate through pheromones, among other methods. This activity leads to an emergent phenomenon known as swarm intelligence. Ant colony optimization algorithms were originally introduced as a meta-heuristic for the traveling salesperson problem which is known to be NP complete and path-based. They are considered to be effective methods for various other combinatorial optimization problems, and to be easy to implement in decentralized environments [27].

An approach to find Steiner trees on a given problem graph $G = (V, E, c)$ with a set of terminal nodes $T \subseteq V$ is presented in [27]: Several ants cooperatively generate a tree, with each separate branch being defined by the paths traveled by each ant. Initially, an ant is placed at each terminal node. The ants move along an edge $e \in E$ in each iteration. e is determined using a stochastic method that is biased to draw the ant towards paths traced out by any ant. Each ant keeps a tabu list of already visited nodes in order to avoid cycles. When ants collide with each other or with the path of any other ant, they are merged, and the two paths taken by the two ants form a sub-tree. As soon as all ants become merged into a single entity, they form a Steiner tree.

On their test instances (which consisted of randomly generated graphs with costs from one to ten, a number of nodes between fifty and one hundred, and up to two hundred edges, all taken from [4]) using empirically fine-tuned parameters with a maximum of two thousand trials, their algorithm was able to reach the global best solution within ten sample runs at least once for twelve of fourteen selected problem instances. This is taken as an indication of the effectiveness of this approach [27].

3.4.4 Memetic Algorithms for the Vertex Biconnectivity Augmentation Problem

Memetic algorithms were introduced in [23] and are based on the concept of evolution, represented by the retaining of a fairly large *population* of different solutions, which is then iteratively modified and possibly improved using *recombination* (the extracting of a *child solution* from two or more *parent solutions*), as they are in *genetic algorithms*. Additionally, a Local Search meta heuristic is used for local improvements.

One approach for a memetic algorithm for solving the vertex biconnectivity problem is described in [21].

This approach is based on the concept of *block-cut graphs*, first described in [1]. A *block* is a maximal sub-graph of a problem instance graph $G = (V, E)$ that is already vertex biconnected. Two blocks have at most one node in common. These nodes shall be called *cut-points* or *cut-nodes*. A *block-cut-tree* is a tree $T = (V_T, E_T)$ representing the relations of blocks and cut-points within G. A *block-node* represents a block excluding its cut-points. "Empty" block-nodes (blocks without nodes that are not cut-points) are discarded and replaced by an edge connecting the two cut-points (this is different from the definition in [1]).

This tree can be augmented, and the resulting graph can be superimposed and back-mapped on the original problem graph. A solution graph $G_{AB} = (V_T, E_{TA})$ for augmentation of the block-cut tree is constructed while respecting a few safe reductions: self-loops in the augmentation edge set, edges connecting the same nodes in T and augmentation edges connecting two cut-nodes adjacent to the same block-node are discarded, and of multiple augmentation edges connecting the same nodes from the block-cut tree only the edge with minimum weight is included. Then G_{AB} is back-mapped onto the original problem graph G.

Also relevant is the concept of *covered block-cut tree edges*. These are defined as a block-cut tree's edge that are within a path represented by an augmentation edge.

A few additional reductions are introduced: augmentation edges containing only edges covered by another augmentation edge are discarded, and augmentation edges presenting the only possibility of connecting two block-cut-nodes of different blocks are fixed by moving them from E_A to E_T. The resulting cycles are transformed into new block-nodes.

Initially, a solution for the initial population is created by starting with an empty edge-set and iteratively adding random, non-redundant edges until all cut-nodes are covered. The random selection is biased towards cheaper edges. Local improvement (in this case being very similar to and having the same effect as purification) is applied.

For recombination, an operator with high heritability was chosen. Edges common in both parents are always adopted, and edges from the parents are added until all cut-nodes are covered.

For mutation, an edge of a candidate solution is removed, and the two resulting components are reconnected using edges different from the removed one. Again, local improvement is applied.

For testing, random instances created with a program called *Zhu's generator*[2] as well

[2] Available at www.ads.tuwien.ac.at/research/NetworkDesign/Augmentation.

as modified graphs from TSPLib (which will be described later) were used. Empirical results show that this approach scales well and is competitive with compared similar approaches used for comparison.

4 Solution Construction and Improvement Heuristics

This section makes heavy use of Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree algorithm (both described [8]). Dijkstra's shortest path algorithm was first presented in [10] and Kruskal's minimum spanning tree algorithm was first presented in [20].

4.1 Solution Construction Overview

We see that the junction constraint can be satisfied by finding a Steiner tree in a graph $G = (V, E, c)$, with the set of junction nodes J and the set of customer nodes C (in other words, all nodes except S) being terminal nodes.

Given a Steiner tree that is represented by the sub-graph $T = (V_T, E_T)$, we will, for each sub-graph in G that describes the nodes of a path from a node $c \in C_2$ to the first junction node $j \in J$ that is encountered, find a set of edges AUG_c that turns this sub-graph into a vertex biconnected sub-graph, provided that such a set exists. Actually, we only need vertex biconnectivity between the first and last nodes of that path, but this is necessarily given if all the nodes of the path are vertex biconnected. The heuristics we will describe later will make use of that fact. We will call this process *augmentation* of that path. If we then take the set of all the edges found in this way $AUG_{edges} = \bigcup_{i \in C_2} AUG_i$ and the set of nodes AUG_{nodes} containing all the nodes incident to an edge contained in AUG_{edges} to construct a sub-graph $G_S = (V_T \cup AUG_{nodes}, E_T \cup AUG_{edges})$, G_S will obviously comply with both the junction and biconnectivity constraint. Since the infrastructure is a connected sub-graph, requiring the nodes in C_2 to be connected to the same junction node in a vertex biconnected manner does not interfere with the generality of this approach.

Algorithm 1 High-level overview of how solutions can be generated using Steiner trees and augmentation

```
solution_graph := determine_steiner_tree($G$, $C$, $j \in J$) #Create a Steiner tree with a source node in the infrastructure#
for all sub-graphs $G_{sub} = (V_{sub}, E_{sub})$ representing paths in steiner_tree from $j$ to a node $c_2 \in C_2$, including $c_2$ and $j$ do
    solution_graph := solution_graph ∪ determine_augmentation_graph($G_{Ssub}$, $G$) #add the nodes needed for augmentation to the solution.#
end for
return solution_graph
```

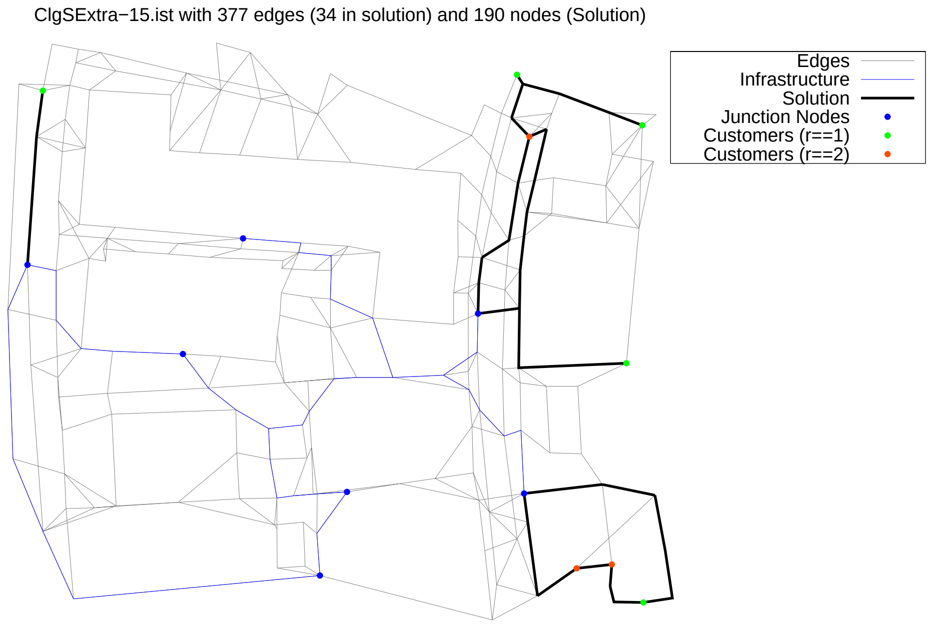

ClgSExtra−15.ist with 377 edges (34 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 3: Example of a Steiner tree where all customer nodes requiring a redundant connection have been augmented *ClgSmall-I1-15*

If we can construct a Steiner tree without using any geometrically crossing edges, and then augment it as described without using any edges that cross either each other or edges used in the initial Steiner tree sub-graph, we have constructed a sub-graph that complies with all three constraints. This is one way of obtaining a valid solution for a given problem instance, and it is the way the construction heuristics we will describe later are designed to work.

It has to be noted that even if we use a Steiner tree using a set of edges with minimum total weight and then augment the connections between the infrastructure and nodes in C_2 using a set of edges with minimum weight, this does not mean we have solved the problem as a whole with a set of edges with minimum total weight. For certain problem instances, this method might not find a solution, even though solutions may exist.

If both phases terminate successfully, however, the result is a valid solution sub-graph G_S on the problem graph G. However, the use of multiple heuristics might lead to redundant edges in G_S where they are not needed. Therefore, a process called *purification* can be applied to the solution sub-graph G_S. Also, a few other heuristic solution improvement methods are available.

4.2 Handling of a Non-Crossing constraint

Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree algorithm can be modified in order to be able to respect the non-crossing constraint. We will consider the following four possibilities for doing this:

1. *Cheapest edge:* If two edges cross, only the cheaper edge is available for inclusion in the solution sub-graph.

2. *Most expensive edge:* If two edges cross, only the more expensive edge is available

for inclusion in the solution sub-graph.

3. *Random edge:* If two edges cross, a random edge of these two is made available for inclusion in the solution sub-graph.

4. *First edge:* When an edge is added to a solution sub-graph, all edges crossed by it are made unavailable for selection. In case all other edges that caused an edge to become unavailable are removed from the solution, that edge becomes available again for inclusion in the solution sub-graph.

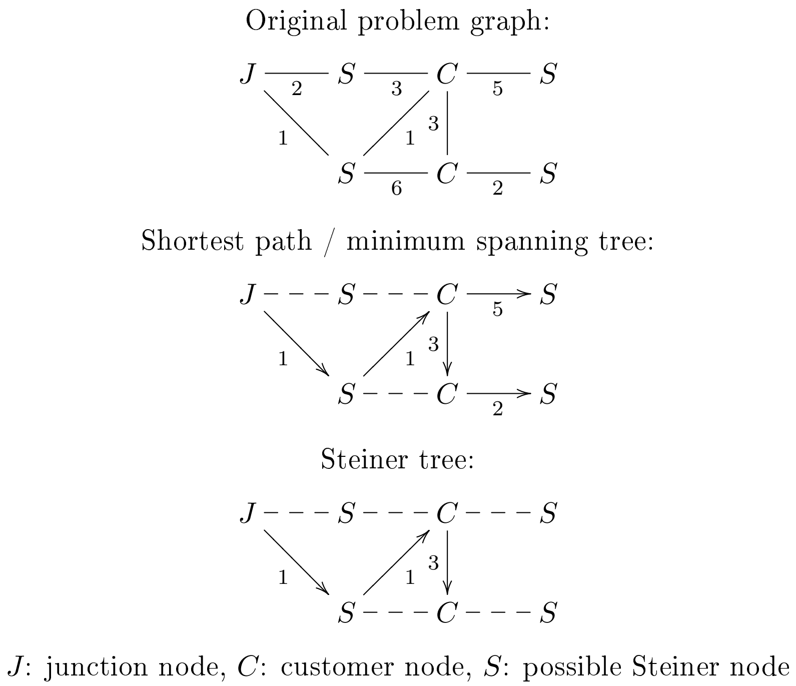
The fourth modification type "first edge" causes the effect of some algorithms we will describe later to become dependent on the order in which certain elements are treated (for example, the order in which nodes requiring a redundant connection to the infrastructure are augmented).

4.3 Heuristics for Steiner Tree Construction

As described earlier, the heuristic construction algorithms used here work by first creating a Steiner tree using the infrastructure nodes defined in V_I and customer nodes defined in C as terminal nodes, and then augmenting the nodes in C_2, using heuristic methods for either purpose.

4.3.1 Single Source Shortest Path Heuristic

This heuristic (which we shall call "single source shortest path" or "SSSP") constructs a tree $T = (V_T, E_T)$ representing the shortest paths from an infrastructure junction node $j \in J$ to all other nodes within the problem graph $G = (V, E)$. During a purging phase for each node $c \in C$ the path in T connecting the source to c is taken, and added to a solution graph G_S, having all the properties of a Steiner tree on the original graph G with all nodes in C as terminal nodes.



Original problem graph:

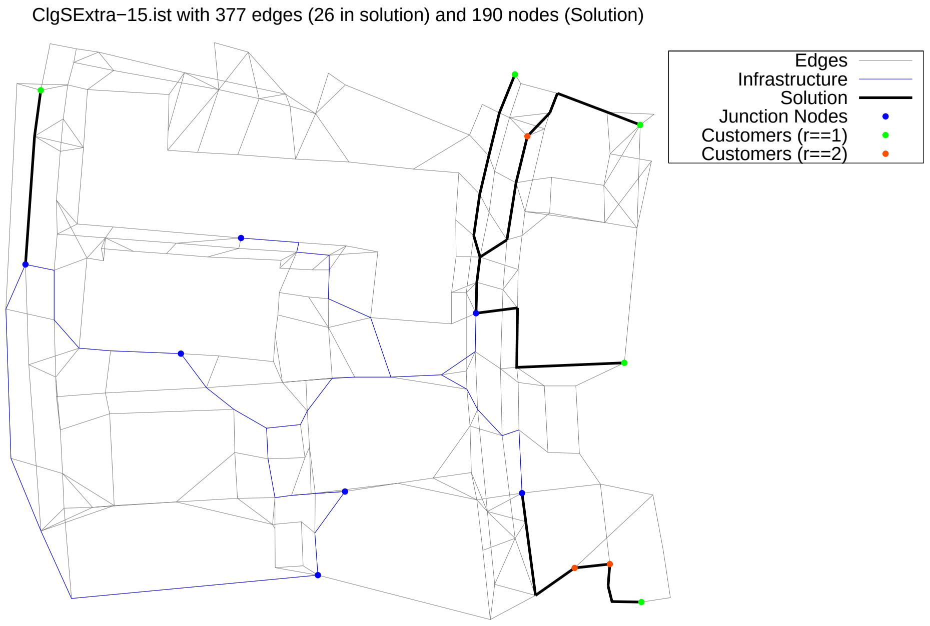
Shortest path / minimum spanning tree:

Steiner tree:

J: junction node, *C*: customer node, *S*: possible Steiner node

Figure 4: Example for the single source shortest path and minimum spanning tree heuristics: The initial problem graph, the shortest path / minimum spanning tree and the resulting Steiner tree

This Steiner tree heuristic relies on the use of Dijkstra's shortest path algorithm, and is very similar to the shortest-path-heuristic approach presented in [28], except for the fact that in our case the terminal nodes are connected to the root in no particular (possibly even random) order, and the root node is always a junction node.



ClgSExtra−15.ist with 377 edges (26 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 5: Example of the effect of the SSSP Steiner tree heuristic on real-world instance *ClgSmall-I1-15*

4.3.2 Minimum Spanning Tree Heuristic

This very greedy Steiner tree heuristic (which we shall refer to as "minimum spanning tree" or "MST") is based on Kruskal's minimum spanning tree algorithm as described in [8]. It works very in a very similar way to the single source shortest path tree heuristic. The major difference is that in this heuristic, the minimum spanning tree is used instead of the shortest path tree, and that the set of edges provided by Kruskal's algorithm has to be converted to a tree with an infrastructure junction node at its source. This is done by depth-first search over the problem graph, traversing only edges present in the minimum spanning tree edge set. The purging phase works exactly like it does in the single source shortest path heuristic.

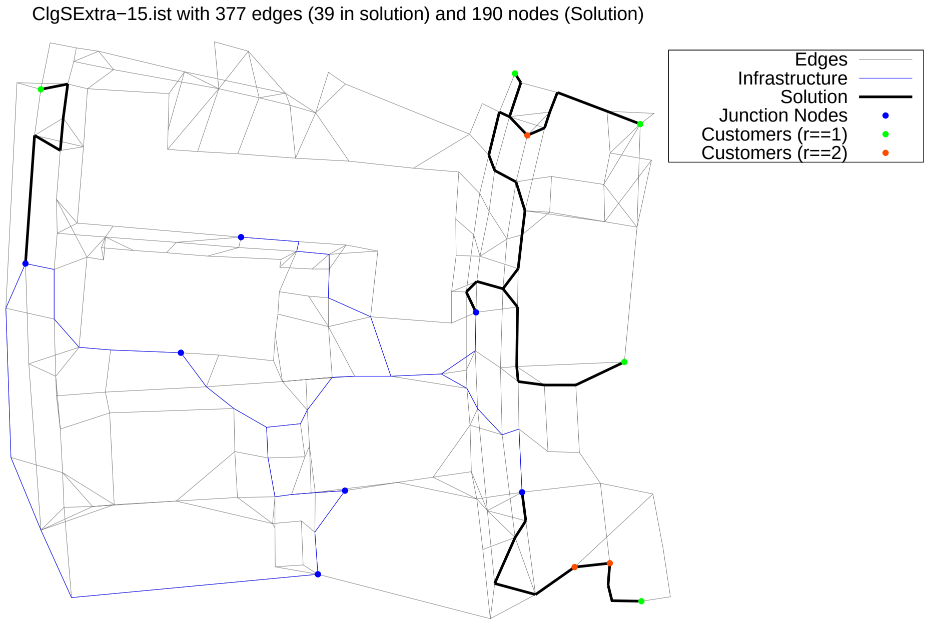


Figure 6: Example of the effect of the MST Steiner tree heuristic on real-world instance *ClgSmall04*

It is obvious that during construction of the minimum spanning tree a lot of nodes might have to be connected that will no longer be in the solution after purging. One approach for handling this effect is presented in [26]. However, in our case this heuristic serves more as a method to diversify the set of solutions we can produce, so we focused on other construction heuristics instead.

4.3.3 Minimum Spanning Tree using All-Pairs-Shortest Path Meta-Edges

Another heuristic algorithm for generating Steiner trees is using the shortest paths between two nodes. This can be done using the Floyd-Warshall algorithm (as described, for example, in [8]), but since in most real-world instances the ratio of terminal nodes to possible Steiner nodes is rather low, and we really only need the shortest paths between terminal nodes for the construction of a Steiner tree, we decided to use Dijkstra's algorithm starting from each terminal node. Thus, access to the shortest paths between each pair of terminal nodes is possible.

We will describe three variants of algorithms that make use of this set of shortest paths between terminal nodes:

All-Pairs-Shortest-Path (APSP) This example of such an algorithm sorts the distances between terminal nodes given by the shortest paths. It then calculates the minimum spanning tree (using Kruskal's algorithm) on these distances, and returns the edges from all paths corresponding to the entries that were selected by the minimum spanning tree algorithm. This edge set is then converted to a tree with a junction node as the root, and thus a Steiner tree is constructed.

All-Pairs-Shortest-Path-extended (APSPe) This algorithm is another example for the use of a minimum spanning tree over a set of edges acquired using an all-pairs-shortest-path algorithm. It uses the set of edges that are contained by any of the paths determined by the all-pairs-shortest-path phase, and uses that set of edges to determine a minimum spanning tree on the problem graph.

All-Pairs-Shortest-Path-adapting (APSPx) This third example of such an algorithm is very similar to the variant we described first (APSP). It is identical except for the fact that it takes into account that some of the shortest paths might share edges. When a path P is added to the minimum spanning tree, each other path R in the queue is checked whether or not it shares any edges with P, and the distance value for R is reduced accordingly. This algorithm can also be said to be an implementation of the shortest path minimum spanning tree algorithm presented, for example, in [26].

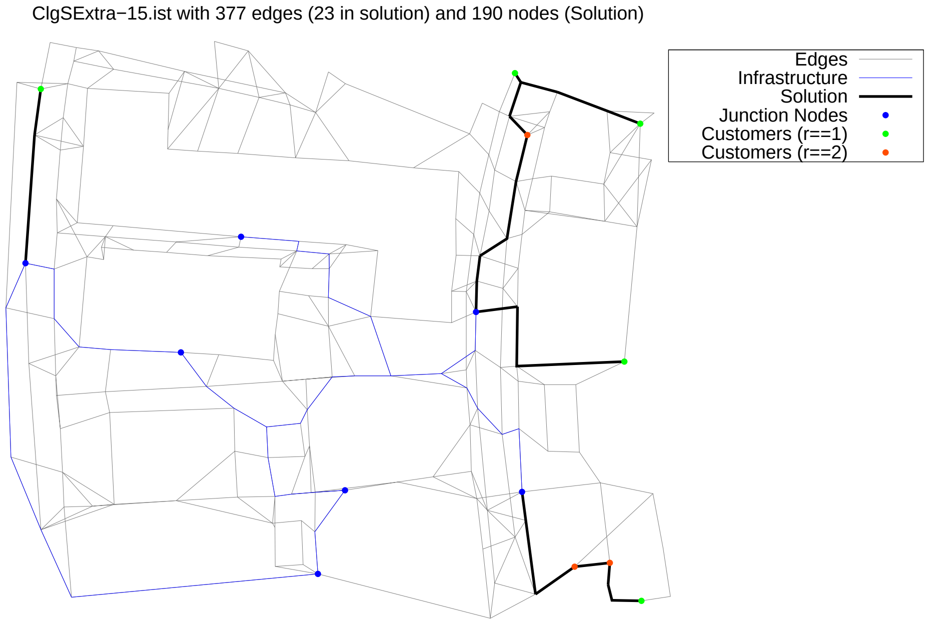

ClgSExtra−15.ist with 377 edges (23 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 7: Example of the effect of the APSPx Steiner tree heuristic on real-world instance *ClgSmall-I1-15*

Algorithm 2 Outline of the APSPx algorithm for heuristic Steiner tree construction

```
solution_graph $G_S = (V_S, E_S)$
solution_graph := $(\emptyset, \emptyset)$
determine shortest path tree on problem graph $G = (V, E)$ for each terminal node
  and a infrastructure junction node
sort shortest pairs of nodes according to distance into queue $Q$
while ($Q \neq \emptyset$) do
  $g$ := get_and_delete_pair_with_least_distance($Q$)
  $P$ := get_shortest_pair_path($g$)
  if $P$ contains a terminal node that was not yet added with another path then
    add $P$ to the solution graph $G_S$
    for all pairs $h$ still in the queue do
      $R$ := get_shortest_pair_path($h$)
      for all edges $e \in P \cap R$ that have not yet been marked do
        set_distance($h$, get_distance($h$) - get_cost($e$)) # Adapt the cost of the
          path. #
        mark $e$
      end for
    end for
  end if
end while
return the solution graph
```

4.4 Heuristic Steiner Tree Augmentation

Once we have determined a (possibly non-crossing) solution-sub-graph $G_S = (V_S, E_S)$, currently containing a Steiner tree $T = (V_T, E_T)$ on the given problem graph $G = (V, E, c)$, we still need to establish redundancy on the connections to all customer nodes $c_2 \in C_2$ that require a redundant connection. We shall now present two approaches, both based on augmentation by shortest path.

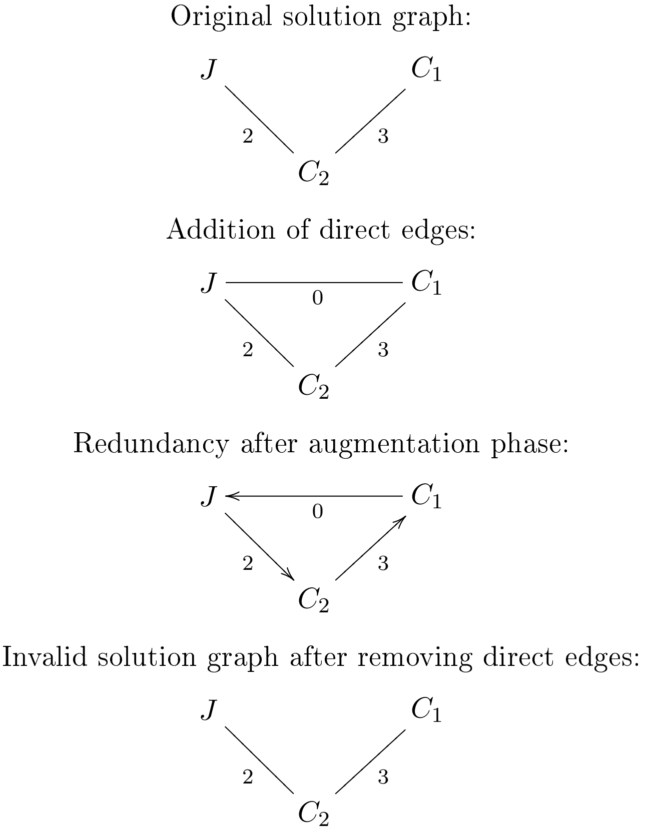
4.4.1 Augmentation by Shortest Path (AugSP)

A simple method (we shall call it *AugSP*, abbreviating ”Augmentation by Shortest Path”) to do this is to search for the shortest path from any infrastructure junction node $j \in J$ to c_2 for each $c_2 \in C_2$, using Dijkstra’s algorithm, respecting non-crossing constraints if needed, and ignoring all nodes $n \in V_P \setminus \{j, c_2\}$ (where $P = (V_P, E_P)$ is the sub-graph that represents the current path in T that connects c_2 to the infrastructure) in order to obtain a path to c_2 that is node-disjunct in respect to the current connection, if such a path exists. These paths are then added to the solution graph G_S.

However, this approach ignores the fact that T already contains edges beside the ones contained in P that can be used for a redundant connection to c_2. The use of such edges does not cause any additional cost to the solution, and might provide a shorter redundant path from j to c_2.

4.4.2 Augmentation by Shortest Path with Pseudo-Infrastructure Extension (AugSPe)

A variation of the AugSP algorithm of this algorithm (we shall call it *AugSPe*, short for ”Augmentation by Shortest Path extended”), for each node $c_2 \in C_2$, introduces a new set of edges E_A that connects all nodes in $V_S \setminus V_P \cup V_{indirect}$ to j, and sets the weight of each edge $e_A \in E_A$ to zero, where $V_{indirect}$ denotes all nodes that are connected to the infrastructure via a path in T containing a node in V_P. The nodes in $V_{indirect}$ must not be connected with such a ”free” edge because of the fact that these edges would lead the algorithm into building a redundant connection that really passes the same node twice. The algorithm then, as before, uses Dijkstra's algorithm to find a new, possibly non-crossing, redundant path to c_2, and adds the resulting path to the solution graph G_S.


Original solution graph:

Addition of direct edges:

Redundancy after augmentation phase:

Invalid solution graph after removing direct edges:

J: junction node, C_1: customer node requiring a single connection, C_2: customer node requiring a redundant connection, S: possible Steiner node

Figure 8: Example of a misleading direct connection during AugSPe augmentation

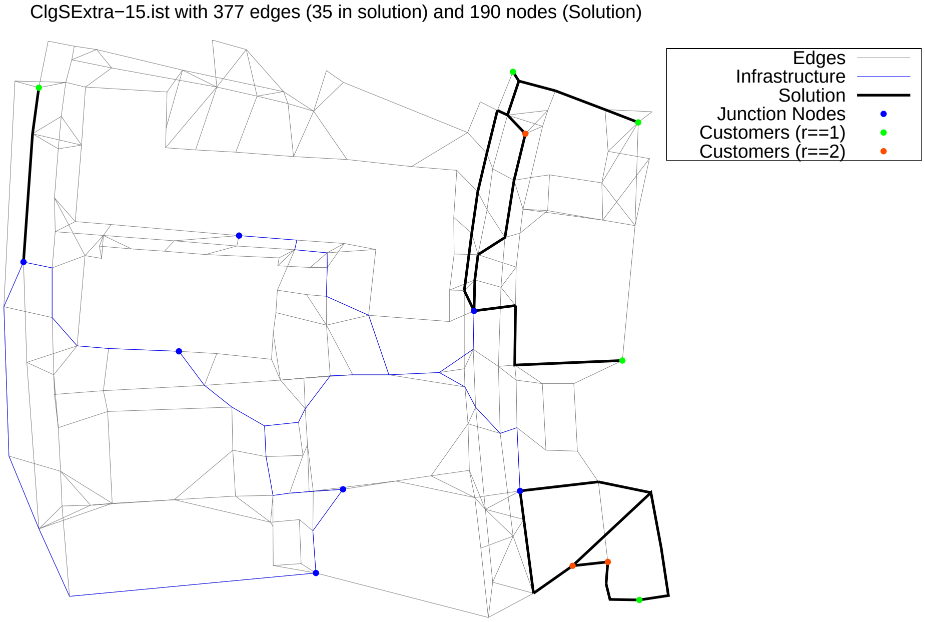

ClgSExtra−15.ist with 377 edges (35 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 9: Example of an AugSP augmented Steiner tree (constructed using APSPx) on the real-world instance *ClgSmall-I1-15*

For both variants, the order in which nodes from the set C_2 are chosen might be critical for finding a solution if the non-crossing constraint is active and the fourth option ”first edge” for modification of Dijkstra’s algorithm is used. When dealing with more than two nodes in C_2 and a non-crossing constraint respected using ”first edge”, more and more edges become crossed and thus unavailable with each added augmentation path. The addition of an augmentation path for a node $c_a \in C_2$ might make the augmentation of another node $c_b \in C_2$ impossible at some point. In some cases, we can overcome this problem by permuting the order in which the nodes are augmented. Therefore, these algorithms might take account trying permutations of the elements of C_2, in these cases, before deeming a node or solution unaugmentable.

Algorithm 3 Heuristic Steiner Tree Augmentation using the AugSPe Algorithm

```
$G_S = (V_S, E_S) := T(V_T, E_T)$
for all nodes $c_2 \in C_2$ do
    $P = (V_P, E_P)$
    $P$ := get_path($c_2$, $J$, $T$) #Get the current path from $c_2$ to the infrastructure#
    $j :=$ infrastructure_node($P$)
    for all nodes $v_s \in (V_S \setminus (V_P \cup V_{indirect}))$ do
        $E_A := E_A \cup (j, v_S, 0)$ #Add a ”free” edge from j to $v_S$#
    end for
    $G_S := G_S \cup$ find_shortest_path($c_2, j, (G \cup E_A) \setminus (P \setminus \{c_2, j\})$) #find the new path using also the edges from $E_A$, ignoring the nodes from the current path, possibly non-crossing#
    if augmentation of $c_2$ was not possible due to an active and violated non-crossing constraint then
        if not all possible permutations have been tried then
            start anew, using a different permutation
        else
            mark augmentation as failed
        end if
    end if
end for
```

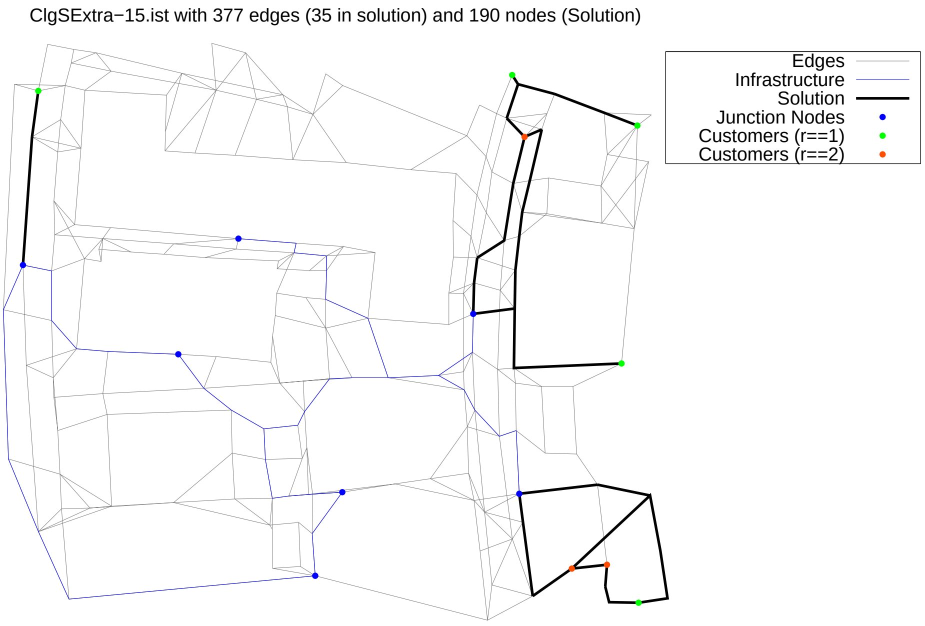

ClgSExtra−15.ist with 377 edges (35 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 10: Example of an AugSPe augmented Steiner tree (constructed using APSPx) on the real-world instance *ClgSmall-I1-15*

4.5 Solution Checking and Purification

4.5.1 Redundancy-Search Solution Validity Check

One possibility for a solution validity checking algorithm we will describe is validity checking by finding all paths in a solution sub-graph $G_S = (V_S, E_S)$ that connects the infrastructure $I = (V_I, E_I)$ of a problem graph $G = (V, E)$ with itself, following a modified depth-first-search algorithm. First, if the non-crossing-constraint is active, the solution is tested for crossing edges. If the given solution passes this test, any node $c_2 \in C_2$ encountered on a cycle without repeated vertices within the solution graph G_S connecting the infrastructure with itself is marked as being redundantly connected, and also, during the same search sweep, any node $c \in C$ encountered as being connected to the infrastructure by solution edges is marked as being connected. After the sweep, if all the nodes $c \in C$ are marked as connected, and all nodes $c_2 \in C_2$ are marked as being redundantly connected, the solution represented by the solution graph is deemed valid.

Finding a cycle without repeated vertices in the solution graph G_S that leads from an infrastructure junction node to an infrastructure junction node shows that all nodes on this cycle are either infrastructure nodes or vertex redundantly connected to the infrastructure. We can show this easily by following the cycle from one of the junction nodes it connects, up to the node $v \in V \setminus V_I$ in question. This proves one path P exists to the infrastructure within G_S, and thus that v is connected to it. Because of the definition of a cycle without repeated vertices, if we continue following it, we will reach an infrastructure junction node, without ever visiting any node $v_p \in P$ again. This shows that a second path to the infrastructure must exist from v, and it thus complies with the vertex redundancy constraint.

Algorithm 4 Redundancy-Search-Based Solution Validity Check Algorithm

```
if (crossing_edges_considered $\wedge$ find_crossing_edges($E_S$)) #Check if the non-crossing constraint is met# then
    return false
    end
end if
CONNECTED_NODES := $\emptyset$
REDUNDANTLY_CONNECTED_NODES := $\emptyset$
mark_nodes(any $j \in J$, $\emptyset$) #Perform the actual redundancy-search#
if (($C \subseteq$ CONNECTED_NODES ) $\wedge$ ( $C_2 \subseteq$ REDUNDANTLY_CONNECTED_NODES)) #Check if the constraints are met for all customer nodes# then
    return true
else
    return false
end if
```

REDUNDANTLY_CONNECTED_NODES and CONNECTED_NODES are to be considered global variables

Algorithm 5 Redundancy-Search Algorithm

```
procedure mark_nodes($v \in V$, path_nodes)
path_nodes = path_nodes $\cup\{v\}$
CONNECTED_NODES := CONNECTED_NODES $\cup$ $\{v\}$ #We reached this node, it is connected to the infrastructure#
return
for all nodes $w$ incident to $v$ in $G_S$ with $w \notin$ path_nodes do
  if $w \in J$ #We found a path from the infrastructure to itself ... # then
    REDUNDANTLY_CONNECTED_NODES := REDUNDANTLY_CONNECTED_NODES $\cup$ path_nodes # ... mark all nodes on it as redundantly connected#
  else
    mark_nodes($w$, path_nodes) # ... so use it to continue searching#
  end if
end for
path_nodes := path_nodes$\setminus\{v\}$ #This node might be contained in another path, remove it from the list#
```

REDUNDANTLY_CONNECTED_NODES and CONNECTED_NODES are to be considered global variables

4.5.2 Path-Node Removal Solution Validity Check

An alternative solution validity check variant is a path-node removal based validity check. It makes use of the fact that the vertex biconnectivity constraint can be shown to be equivalent to an alternative formulation:

A graph $G = (V, E)$ has vertex biconnectivity if for any node $v \in V$, the graph G' representing the original graph $G \setminus \{v\}$ representing G with v and all its incident edges removed, is connected. This is a formulation that directly results from using the vertex version of Menger's theorem (as stated in 1928) using a vertex connectivity requirement of 2. Menger's theorem in the vertex version states that if there exists no set of $k - 1$ nodes within a graph which, when removed, causes two nodes to be disconnected, then the graph has *k-connectivity.* A proof for Menger's theorem can be found in [5].

This variant first checks if the non-crossing constraint is active, and if so, if it is met. If the solution graph passes this preliminary test, a depth-first-search within the solution graph is performed, and all encountered nodes are marked as connected. Also, the resulting depth-first-search tree T is recorded. If all nodes $c \in C$ are marked as connected, the solution is checked for compliance with the redundancy criterion by the following step:

For each node $c_2 \in C_2$, the path within the depth-first-search tree connecting c_2 to the infrastructure is examined by checking each graph G'_S (where G'_S represents G_S with one node is removed from the path in the depth-first-search tree T between the infrastructure and c_2) for an existing alternate connection between the infrastructure and c_2.

If we find such an alternate connection for each node $c_2 \in C_2$, we have shown that they are all redundantly connected: the first path $P = (V_P, E_P)$ that connects c_2 to the infrastructure (excluding c_2 itself and the junction node $j \in J$ it connects to) is the one which we found using the depth-first-search tree T. Its existence shows that we can remove all nodes $v_r \in {}_S \setminus V_P$) and all edges incident to them from the solution graph G_S

(resulting in a graph G'_S), and still have a connection from c_2 to a junction node: the original path P, which is, of course, still completely included in G'_S, and connects c_2 to j.

So what remains to be shown in order to prove that c_2 is connected to the infrastructure in a vertex redundant way, is that we can remove each single node from V_P from G_S and still have a connection to the infrastructure. The algorithm does this by temporarily removing each node $v_p \in V_P$ and all incident edges from G_S, and searching the remaining solution graph G''_S for a connection from c_2 to a node $j \in J$. If such a connection exists in G''_S, then c_2 must have a vertex redundant connection to the infrastructure, and the vertex redundancy constraint is met for that node.

Algorithm 6 Path-Node-Removal Validity Check

```
if ($crossing\_edges\_considered \wedge find\_crossing\_edges(E_S)$) #Check if the non-crossing constraint is met# then
    return false
    end
end if
DFS_tree = $(V_T, E_T)$
DFS_tree := depth-first-search($j \in J$, $G_S$) # Get a depth-first-search tree starting from the infrastructure. #
if ($\neg(C \subseteq V_T)$) # Check if all the terminal nodes are within the solution. # then
    return false
    end
end if
for all paths $P = (V_P, E_P)$ in DFS_tree that lead to a node $c_2 \in C_2$ do
    for all nodes $v \in V_P$ except for $c_2$ and $j$ do
        if ($\neg$connection_exists($c_2$, $v$, $G_S \setminus \{v\}$)) # No alternative connection, the solution is invalid. # then
            return false
            end
        end if
    end for
end for
return true # If we reach this, we know all constraints are met and the solution is valid. #
```

4.5.3 Solution Purification

Since the heuristic vertex redundancy augmentation might use a different base algorithm (for example, constructing a Steiner tree using a minimum spanning tree algorithm and then augmenting it using a shortest path algorithm), redundancy might be introduced where it is not required. Purifying is the process of finding such unneeded redundancy and removing it to the point where the removal of one further element of the solution graph makes the solution invalid.

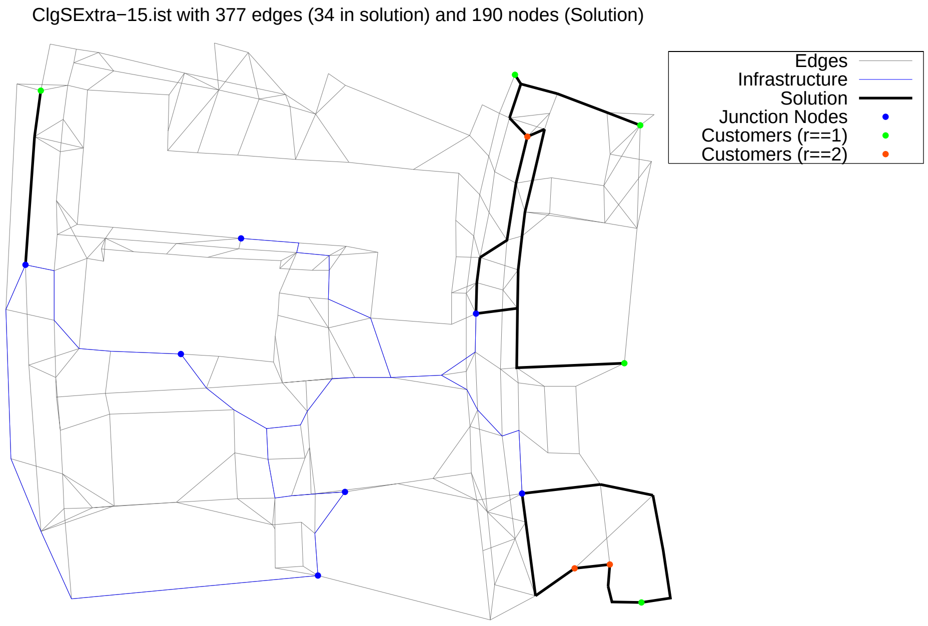

ClgSExtra−15.ist with 377 edges (34 in solution) and 190 nodes (Solution)
Edges
Infrastructure
Solution
Junction Nodes
Customers (r==1)
Customers (r==2)

Figure 11: Example of a purified AugSPe augmented Steiner tree (constructed using APSPx) on the real-world instance *ClgSmall-I1-15*

For our purposes, this was done using validity checking algorithms. For a solution subgraph $G_S = (V_S, E_S)$ of a problem graph $G = (V, E, c)$, each edge is removed if its removal does not cause the solution to no longer comply with one of the given problem constraints. The order in which the edges are attempted to be removed is usually descending cost order, but other strategies might also be useful.

Algorithm 7 Typical purifying algorithm

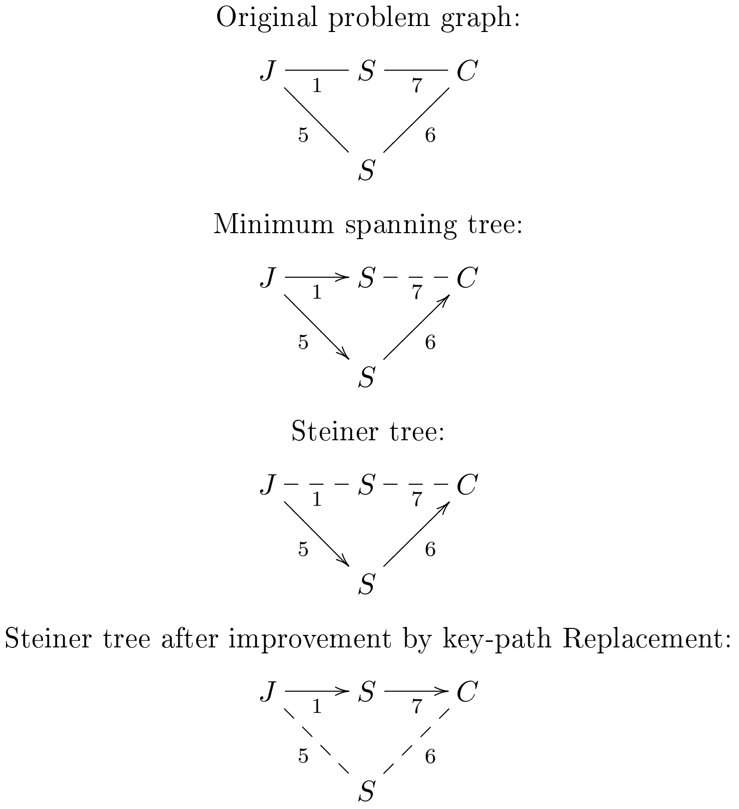
```
queue := $E_S$
queue = order_by_descending_cost(queue) #Sort the edges by cost#
while (queue $\neq \emptyset$) do
    $e$ := get_and_delete_max_cost_edge(queue) #Get the most expensive edge in the queue...#
    remove_edge_from_solution($e$, $S$) #...and remove it#
    if ($\neg$check_solution_validity($S$)) #If the solution is no longer valid...# then
        add_edge_to_solution($e$, $S$) #...put the edge back in the solution#
    end if
end while
```

4.6 Heuristic Solution Improvement Methods

Purification can remove some unnecessary edges, and thus cost from a solution, but the various algorithms and the nature of the two-phase construction process can lead to situations where a better solution can be found without requiring too much computational effort. Therefore, two algorithms to improve solution cost in such cases will be presented.

4.6.1 Key-path Replacement

In some cases our heuristics might end up with a solution on a weighted problem graph $G = (V, E, c)$ that contains unneeded cost in key-paths (as we already defined, these are paths between key-nodes that cross only Steiner nodes of degree two in the solution graph $G_S = (V_S, E_S)$). If we were using, for example, the MST or APSPe heuristics during Steiner tree construction, nodes that never made it into the solution might have been included during building of a minimum spanning tree, and thus the selection of edges might have been made in order to be able to connect these nodes, introducing additional cost to our solution. This additional cost can often not be removed during purifying, since the edges in question might be necessary for the validity of the solution.


Original problem graph:

Minimum spanning tree:

Steiner tree:

Steiner tree after improvement by key-path Replacement:

J: junction node, C: customer node, S: possible Steiner node

Figure 12: Example of an improvement by key-path Replacement

This additional cost can be removed by finding each such key-path $P = (V_P, E_P)$ between solution nodes $v_a \in V_S$ and $v_b \in V_S$ within the solution and replacing the key-path with the shortest possible path while ignoring solution nodes in order to avoid breaking redundancy. Purified solutions consist entirely of key-paths within G_S, so replacing only these paths is sufficient to cover the whole solution sub-graph G_S.

Algorithm 8 Key-Path Replacement Algorithm

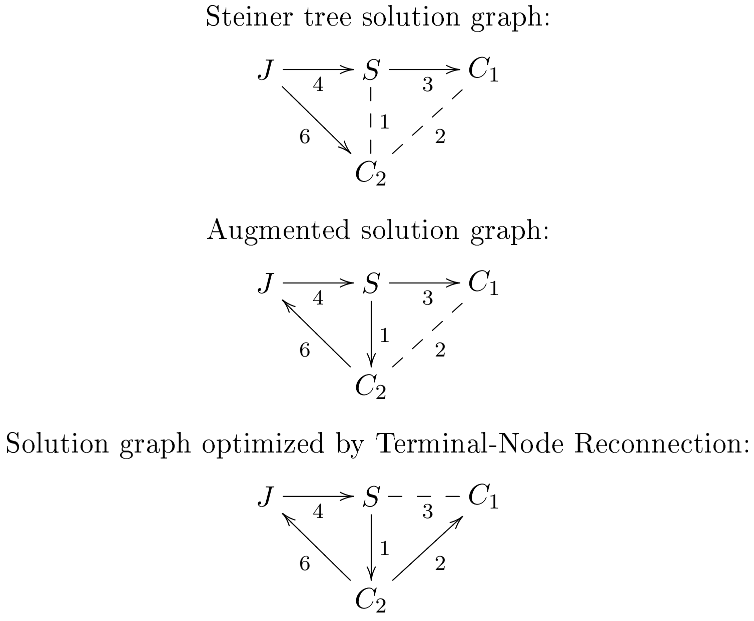
```
boundary_nodes := $C \cup \{v_S \in V_S | degree(v_S) > 2\} \cup$ any $j \in (J \cap G_S)$
for all nodes $v \in$ boundary_nodes do
    for all unmarked nodes $w \in$ boundary_nodes that are directly connected in $G_S$
    via path $P(V_P, E_P)$ do
        remove nodes in $V_P \setminus \{w, v\}$ and all incident edges from $G_S$
        $G_S = G_S \cup$ find_shortest_path($w, v, G \setminus (V_S \setminus \{w, v\}$)
    end for
    mark $v$
end for
```

4.6.2 Terminal-Node Reconnection

In many real-life problem instances good solutions can be constructed by creating cycles that include many of the nodes requiring redundant connections and a junction node. Since these cycles can contain a lot of edges and also a junction node, connecting terminal nodes that do not require a redundant connection to a node of such a cycle instead of directly to a junction node might reduce the total solution cost.

The construction heuristics described here are not aimed at creating such cycles, and initially connect all terminal nodes by creating a Steiner tree, and then augment that tree to provide redundancy, so the possibility of exploiting augmentation edges for cheaper connection of nodes $c_1 \in C_1$ is never explored during construction.

Terminal-Node-Reconnection tries to do this by reconnecting nodes in C_1. The first step is iteratively removing all connections from a node within the solution with a degree higher than two within the solution graph $G_S = (V_S, E_S)$, to nodes in C_1 with a degree of one within G_S. Then direct edges E_A from a junction node $j \in J$ with zero cost are introduced to all solution nodes except for the nodes in C_1. Finally a shortest path tree is calculated using Dijkstra's algorithm, and j as the source, and the paths to all nodes in C_1 are added to the solution.



Steiner tree solution graph:

Augmented solution graph:

Solution graph optimized by Terminal-Node Reconnection:

J: junction node, C_1: customer node requiring a single connection, C_2: customer node requiring a redundant connection, S: possible Steiner node

Figure 13: Example of an improvement by Terminal-Node Reconnection

5 Meta-Heuristic Approaches

5.1 Local Search and Simulated Annealing

5.1.1 Local Search

Local search has been applied as an incomplete algorithm to many computationally hard problems. It generally involves iteratively moving from one solution to a neighborhood, in order to find a better solution, starting from an initial candidate solution. [2]

In our case, a solution is generated on the problem graph $G = (V, E, c)$ using the heuristic approaches we described earlier, and uses one of three move-operators that will be specified later. One of the three operators is chosen at random, and applied to the candidate solution. As soon as a better solution is found, this new solution becomes the candidate solution.

Algorithm 9 Basic Overview of a Local Search Algorithm

```
$G_S = (V_S, E_S)$
$G_S$ := heuristic_solution_construction($G$)
candidate_solution := $G$
while the maximum number of generations is not reached (or some other end condition is not met) do
    new_solution := move_with_a_random_operator(candidate_solution)
    if (cost(new_solution) < cost(candidate_solution)) then
        candidate_solution := new_solution
    end if
end while
```

5.1.2 Simulated Annealing

Simulated Annealing was first introduced by [19] and, independently, two years later by [6]. It mimics annealing in metallurgy, a process involving a controlled cooling process of heated metals which leads to better development of crystals inside the material and thus increasing its quality.

Simulated Annealing generally, and also in our specific case, is very similar to local search, but has one crucial difference:

Instead of only accepting better solutions as the new candidate solution, it also accepts worse solutions with a certain probability, called the *temperature* T. This feature has been introduced in order to overcome local minima. As the iterations progresses, T decreases.

5.1.3 Neighborhoods

For Local Search and Simulated Annealing we need some way to randomly modify our solution without making it invalid. For this purpose, we will describe three algorithms:

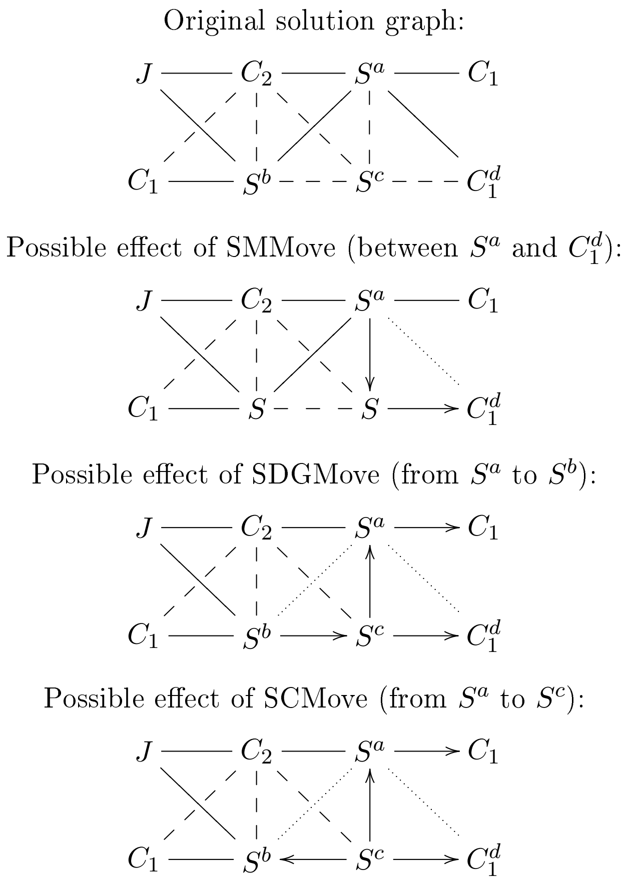
- Single Migration Move (SMMove)
- Single Degree Move (SDGMove)
- Single Connecting Move (SCMove)

Each of these algorithms takes a valid solution graph $G_S = (V_S, E_S)$ on a given problem graph $G = (V, E, c)$ and transforms it into a similar, but randomly modified solution. They are designed to complement each other, with one of these algorithms being chosen randomly during each Local Search or Simulated Annealing iteration. Since they are all based on Dijkstra's algorithm, they can attempt to create solutions complying with a non-crossing constraint using one of the four modifications described earlier. It has to be noted, though, that in this case the order in which the nodes affected by these algorithms are chosen plays an important part in determining the outcome if the fourth modification referred to as "first edge" is to be used.

SMMove This move-operator searches for a key-path $P = (V_P, E_P)$ (in other words, a path where all the nodes except the endpoints have a degree equal to two within the solution graph) from a random node within the solution graph G_S, and replaces it by the shortest path (ignoring all solution nodes except for the endpoints of P) using Dijkstra's algorithm. This operator serves to improve key-paths within the solution, but does not influence the degree within G_S of nodes with an initial degree larger than two.

SDGMove This move-operator takes a random key-node $v \in (C_1 \cup \{v_S \in V_S | degree(v_S) > 2\})$, looks for all direct neighbors N (nodes that are connected to v by a key-path), with $n \notin C_2$ and $n \notin J$. It determines a depth-first search-tree, starting at n and not searching beyond any node $j \in J$, and marks all nodes $R \subseteq V$ that are in the predecessor path to a C_2 node. Then it removes all key-paths to nodes $n \in (N \setminus R)$ originating from v, and attaches n to a random node $w \in G_S$ with $w \notin I$ and $w \notin C_2$ and connected to v by a key-path within G_S using the shortest possible path and ignoring nodes already in the solution. The restriction regarding R is important for conserving redundant paths for nodes in C_2: one path to each node in C_2 is conserved, the other one modified. Finally, if $v \in C$ or not all connections could be moved (for example, because a direct neighbor was a node in C_2), v is itself attached to w by the algorithm (to preserve redundant connections, if present). This operator serves to move the "junction" nodes within G_S around.

SCMove This move-operator is very similar to SDGMove. It also takes a node $v \in (C_1 \cup \{v_S \in V_S | degree(v_S) > 2\})$, marks all depth-first predecessors R of nodes in C_2, searches for all key-paths to a node n, with $n \notin C_2$,$n \notin J$, $n \notin R$ and originating from v, and removes them. However, it then attaches the direct neighbors to a neighboring node $w \in G$ with $w \notin I$ with the new key-node $w \notin G_S$, meaning it is chosen from outside of the original solution. Again, the algorithm will also attach v to w if necessary. This operator serves to also explore the possibility of including nodes that were not in the original solution in such a way that their degree within G_S is larger than 2.



Original solution graph:

Possible effect of SMMove (between S^a and C_1^d):

Possible effect of SDGMove (from S^a to S^b):

Possible effect of SCMove (from S^a to S^c):

J: junction node, C_1: customer node requiring a single connection, C_2: customer node requiring a redundant connection, S: possible Steiner node
Edge weights have been omitted to increase legibility.

Figure 14: Illustration of each of the three move-operators SMMove, SDGMove and SCMove

SCMove and SDGMove have a similar intention (of diversifying solutions by modifying the set of key-nodes within the solution) as the Local Search approach described in [9]. However, our approach does not employ construction heuristics for the entire problem, and instead operates directly on a given solution sub-graph G_S. The reason for choosing this approach is that we are not trying to solve only the Steiner tree problem, but also trying to augment and purify our solution-sub-graphs, thus resulting in a more complex problem with our heuristic construction algorithms having a longer running time than just the time needed for building a minimum spanning tree.

The intuition in our approach is not the one of adding or removing key-nodes but of moving the key-nodes around. They are, however, still able to add and remove key-nodes: When a key-node is moved by one of these two operators, it is possible that the key-paths attached to it are moved to another node which is already a key-node. Thus, the resulting solution contains one less key node than the original solution. An additional key-node can be created when the operator is not able to reconnect all key-paths of a target key-node, since some of the original key-paths remain connected to the original key-node, and some are transferred to the new key-node.

These move-operators have each have very specific effects, which is why we have also considered using more than one of these operators at a time, which we call $multi-move$.

Instead of just applying one of these operators, a random representative of these operators is applied until a better or equal solution is found, or a definable number of operator applications is reached.

5.2 Variable Neighborhood Descent and Variable Neighborhood Search

5.2.1 Variable Neighborhood Descent

Variable Neighborhood Descent is a method that searches various neighborhoods for an improvement of an initial candidate solution. In our case, an initial candidate solution $G_S = (V_S, E_S)$ is created on a problem graph $G = (V, E, c)$ using the heuristic algorithms described earlier. A first neighborhood of G_S is then explored, and each solution that improves upon the candidate solution becomes, in turn, the new candidate solution. Once the search through this neighborhood yields no improvements, other neighborhoods are explored, one after the other. As soon as an improvement upon the candidate solution is found, the first neighborhood is, explored again, until no neighborhood includes any solution better than the current candidate solution. Optionally, the algorithm finds the best neighbor in the neighborhood of the current candidate solution, or any neighbor better than the current candidate solution in the neighborhood.

In our case, the neighborhoods used are:

1. Junction-Move Neighborhood (described later)
2. Direct-Path Replacement
3. Terminal-Node Reconnection

It has to be noted that the last two neighborhoods actually only contain at most one neighbor for each solution.

Algorithm 10 Basic Overview over a Variable Neighborhood Descent Algorithm

```
$G_S = (V_S, E_S)$
$G_S$ := get_heuristic_solution($G = (V, E)$)
while the number of maximum iterations is not reached do
  repeat
    get_move_junction_neighborhood_improvement($G_S$) # Gets a better/the best solution from the Junction-Move neighborhood #
  until the Junction-Move neighborhood contains no solution that is better than the current candidate solution
  $G'_S$ := direct_path_replacement($G_S$) # Try key-path replacement #
  if ($\neg(G'_S < G_S)$) # If this doesn't improve our solution... # then
    $G_S$' := terminal_node_reconnection($G_S$) # ...try terminal node reconnection. #
    if ($\neg(G'_S < G_S)$) # If this doesn't help either... # then
      break # ...we stop. #
    end if
  end if
  := $G'_S$# # We found a better solution, use it as a candidate solution from now on. #
end while
```

5.2.2 Variable Neighborhood Search

Variable Neighborhood Search is a meta-heuristic method that uses neighborhood-changes systematically within a Local Search. [22]

In our case, variable neighborhood search uses Variable Neighborhood Descent, as described in the last section, within a local search. An initial candidate solution $G_S = (V_S, E_S)$ on a problem graph $G = (V, E, c)$ is created using the heuristic methods described earlier. Variable Neighborhood Descent is then applied to this solution sub-graph, possibly resulting in an improvement in G_S. Once it no longer yields any improvement, G_S is randomly modified using one of the three move-operators we described earlier or a multi-move. This action is generally called *shaking*. Variable Neighborhood Descent then is applied again. This process of shaking G_S and applying Variable Neighborhood Descent to it is repeated, with the difference that the amount of times G_S is shaken each time (in other words, how often a random move-operator is applied to it sequentially) is steadily increased, up to a maximum. When the maximum is reached, the number of shakes to apply to G_S is set to one again, and the process starts anew. This cycle is performed until a maximum number of iterations is reached.

Algorithm 11 An Overview over a Basic Variable Neighborhood Search Algorithm

```
$G_S = (V_S, E_S)$
$G_S$ = get_heuristic_solution($G(V, E)$)
num_shakes = 1
while the maximum number of iterations is not reached do
    $G_S$ = variable_neighborhood_descent($G_S$)
    shake($G_S$, num_shakes)
    num_shakes = (num_shakes) mod (max_num_shakes) + 1
end while
```

5.2.3 Key-Node-Move Neighborhood

This neighborhood algorithm (which we shall call *MNOpt*) explores a neighborhood of a solution sub-graph $G_S = (V_S, E_S)$ on a given problem graph $G = (V, E, c)$. This neighborhood is defined by moving connections from one node within the solution to another node, similar to the approach in the move-operators SDGMove and SCMove, but with the difference that the possible moves are explored systematically.

Each node v within the solution sub-graph G_S, $v \in C_1 \cup \{v_S \in V_S | degree(v_S) > 2\}$, is examined for key-path neighbors n (nodes that are connected to v by a key-path), with $n \notin C_2$, $n \notin R$ (R being the set of all neighbors that are predecessors of C_2 nodes in a depth-first search tree starting from v and not searching beyond any node $j \in J$) and $n \notin J$. The restriction regarding R is, as with SCMove and SDGMove, important to conserve vertex-redundant connections to nodes in C_2. Then the algorithm removes all key-paths originating from v, and systematically tries to attach n to all nodes $w \in G_S$ with $w \notin I$ and $w \notin C_2$ and with w contained in a key-path to v within G_S, and all nodes $u \in V \setminus V_S$ connected to v by an edge $e \in E$. It uses the shortest possible path and ignores nodes already in the solution. Finally, as with SDGMove and SCMove, if $v \in C$ or not all connections could be moved (for example, because a direct neighbor was a node in C_2), v is itself attached to w by the algorithm in order to preserve redundant connections.

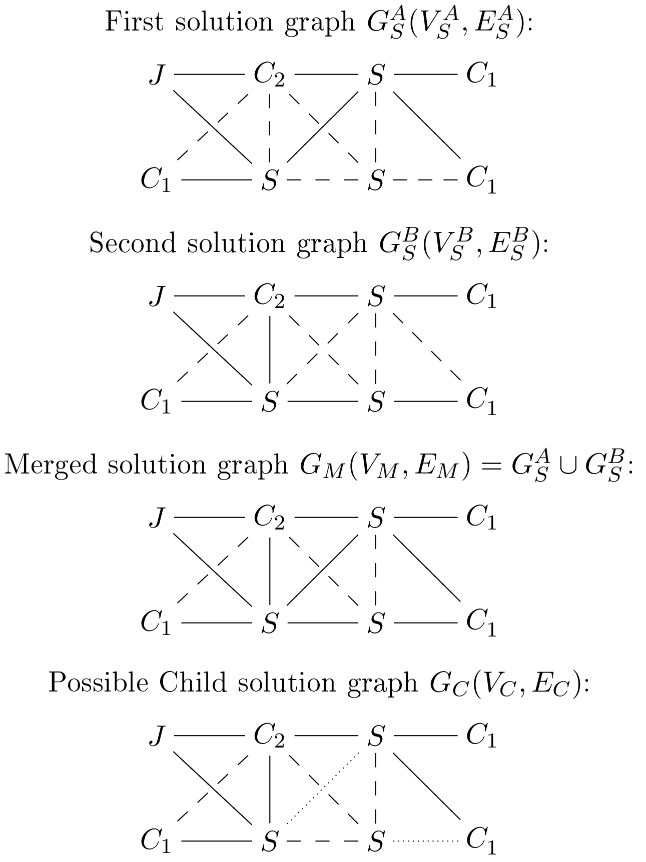
The algorithm can be set to return any solution encountered during this process that is better than the original solution, or it can be set to try all possible moves and in the end return the best solution.

For this neighborhood, the order in which the direct neighbors are reattached plays an important role if a non-crossing constraint is present and the fourth algorithm modification ”first edge” for supporting a non-crossing constraint is to be used.

5.3 Solution Merging

In order to find a good solution for a given problem graph $G = (V, E)$, it might be helpful to not only apply one heuristic to G, which might be limited to one ”view” or ”focus” of the problem because of the algorithms used in it (for example, minimum spanning tree or shortest path), but to try various heuristics and then use the best features of each to construct a solution that is significantly better than any single original solution.

We will describe a method of merging a set of solutions into one sub-graph G_M, and then extracting a valid solution sub-graph $G_S = (V_S, E_S)$ for the problem graph G with $V_S \subseteq V_M$ and $E_S \subseteq E_M$.



First solution graph $G_S^A(V_S^A, E_S^A)$:

Second solution graph $G_S^B(V_S^B, E_S^B)$:

Merged solution graph $G_M(V_M, E_M) = G_S^A \cup G_S^B$:

Possible Child solution graph $G_C(V_C, E_C)$:

J: junction node, C_1: customer node requiring a single connection, C_2: customer node requiring a redundant connection, S: possible Steiner node
Edge weights have been omitted to increase legibility.

Figure 15: Illustration of the merging of two solutions and the extraction of a child solution

5.3.1 Merging Solutions

For purposes of the approaches described in this thesis, it is sufficient to think of a solution as a set of edges (and subsequently, the nodes that are covered by these edges). Merging two solution sub-graphs of a problem instance $G(V, E, c)$, or a solution sub-graph with any sub-graph on G to obtain a third sub-graph is, for our purposes, simply a matter of joining the edge and node sets of the two sub-graphs.

To avoid conflicts with a non-crossing constraint, when two sub-graphs $G_A = (V_A, E_A)$and $G_B = (V_B, E_B)$ are merged, any edge $e \in E_A$ that crosses an edge $f \in E_B$ causes the removal of f, making one sub-graph "dominant" over the other, and thus the order of addition of sub-graphs relevant.

5.3.2 Extracting a Child Solution Using Purification

A sub-graph $G_M = (V_M, E_M)$ consisting of merged solution sub-graphs of a problem graph $G = (V, E, c)$, already is a valid solution sub-graph since each of the underlying sub-graphs is, in itself a valid solution, and at least one of them dominated and is therefore fully included in G_M. However, if the solution sub-graphs were not all exactly the same, there might be unneeded edges in G_M. This is a problem that we have faced before, during heuristic solution construction, and it can be solved by purifying. A purifying algorithm is applied to G_M, and the resulting solution is returned.

5.3.3 Extracting a Child Solution Using Exact Methods

Another possibility to view the generation of a child solution from a sub-graph $G_M = (V_M, E_M)$ consisting of merged solution sub-graphs of a problem graph $G = (V, E, c)$ is to look at G_M as a problem instance by itself, using the cost function c from G. Applying one of the heuristic methods used during the construction of a solution sub-graph that was included in G_M would obviously not amount to much. However, since G_M is itself a sub-graph of the original problem, and might be smaller than G in terms of the number of edges, using an exact approach on G_M might terminate sooner than using the same exact approach on G itself. We obviously don't solve the original problem in an exact way, because edges that might be included in an exact solution for G might not be included in G_M, but we will get a cost-minimal solution using only edges in E_M, and thus originating from our original heuristic solution sub-graphs. A further possibility to further reduce the computation effort for the exact solver would be to force it to include edges that are contained in every solution in the merged set.

5.3.4 Obtaining Various Solution Sub-Graphs for Merging

Many variants of obtaining candidate solutions for merging come to mind. Of these, we have selected three, which we have also implemented and experimented with (as will be described later):

- Using different combinations of heuristics to construct various solutions on the same problem graph

- Collecting solutions at different stages of an iterative or generational meta-heuristic approach
- Generating solutions for multiple problem graphs that are all derived from the original problem graph G by randomly modifying c.

6 Implementation

An implementation of the algorithms listed in this thesis was written in C++, on and for a Linux-based environment.

6.1 Frameworks and Used Components

The algorithms described earlier were implemented using a series of external resources and frameworks, namely LEDA, EALib and some of Daniel Wagner's NQCC components, which in turn make use of CPLEX and LEDA.

LEDA *LEDA* is a commercial C++ class library for efficient data types and algorithms by Algorithmic Solutions Software G.m.b.H. [15]

It is used as the main library for graph-related data structures in the implementation of the employed exact algorithms and problem instance classes, and also in the implementation of the algorithms listed in this thesis. Especially important are the data structures for hash-arrays, sets, ordered lists, priority queues as well as, of course, the constructs for graphs, graph nodes and graph edges.

EALib *EALib 2.0* is a class library designed at the department for Algorithms and Computer Graphics. It provides base classes and algorithm implementations for meta-heuristic approaches [30]. It is used in the implementations of all meta-heuristics except for solution sub-graph merging with exact solution extraction. The components related to Local Search, Simulated Annealing, Variable Neighborhood Descend and Variable Neighborhood Search are used (that includes algorithm classes as well as chromosome classes).

NQCC Components *NQCC* is the suite for solving the OPT problem and related heuristic and meta-heuristic approaches into which the implementation of the algorithms described in this thesis are integrated. The framework we used is the same that was used in [29]. The algorithms make use of some existing components of that framework, specifically the following:

- *nqProblem* is a class for containing and pre-processing problem instances, and is used as such. Especially frequently used are the features for loading a problem instance, reducing the infrastructure to a single infrastructure junction node, determining crossing edge pairs and writing a solution sub-graph to a GNUplot file.
- *nqLogging* is used for text output in all components. This includes debugging output, main text output and output of warnings and error messages.
- *MCFRedSolver* is an exact, multi-commodity-flow Integer Linear Programming based solver for the simplified redundant OPT problem as defined earlier. It uses an algorithm from the CPLEX optimizer suite as a base and Integer Linear Programming solver, and is used for extracting solutions from a merged set of solution sub-graphs on the same problem instance graph. This is not actually accomplished

by creating a new problem instance, but by adding constraints to exclude certain edges from being included in the solution or to force them into the solution.

CPLEX *CPLEX* by ILOG Inc. is a commercial suite of optimizers targeted at solving large, real-world optimization problems. It provides optimal solutions and can, depending on use, also return approximate solutions after a certain running time (of which we made use during our experiments). It is used by Daniel Wagner's MCFRedSolver (which is described in [29]) which, in turn, was used by the implementation of an exact algorithm for extracting a child solution from a merged set of solution sub-graphs on the same problem graph [16].

6.2 Class Diagram

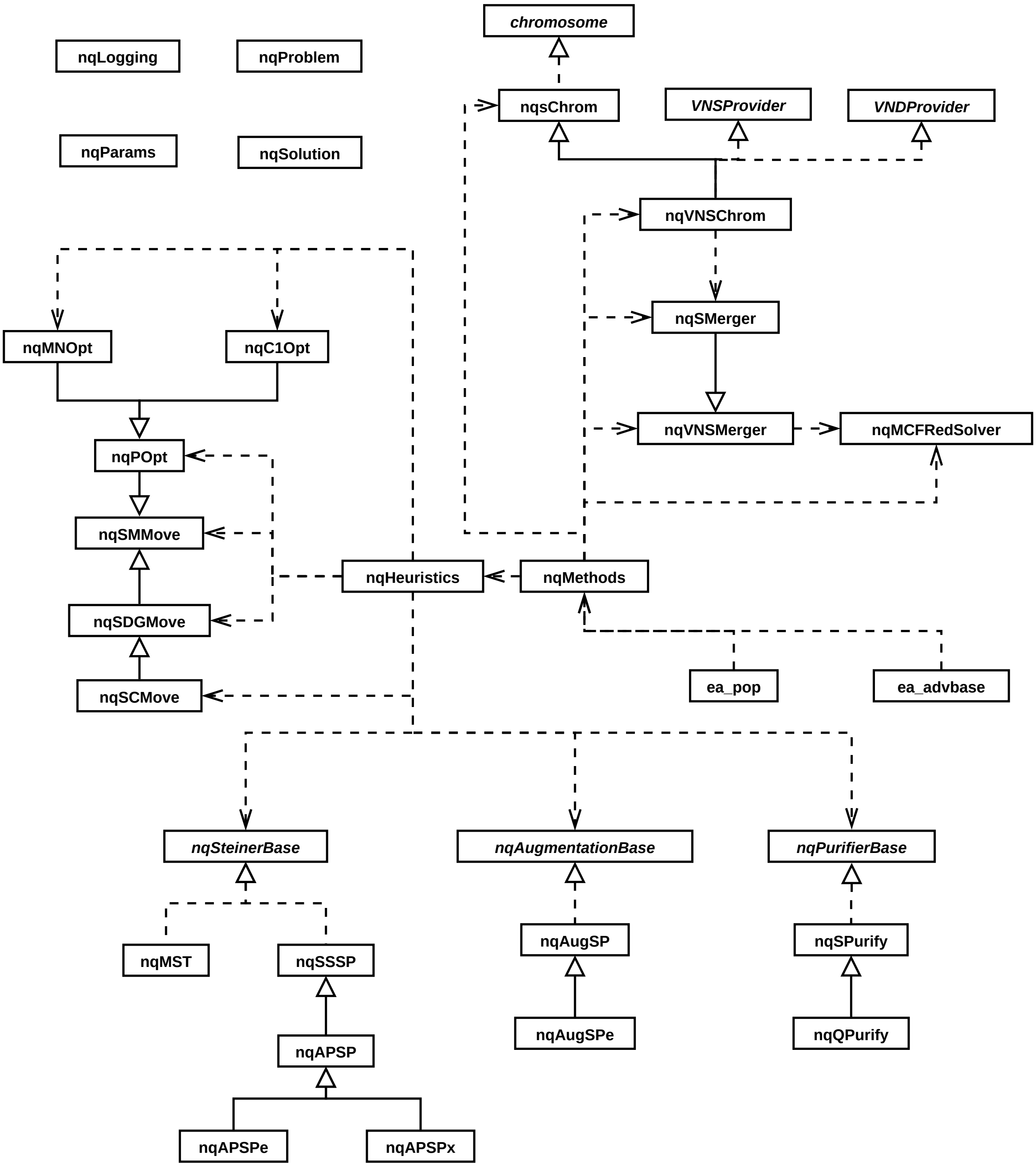


Figure 16: Class Overview Diagram

6.3 Class Descriptions

nqSolution A class for containing a solution sub-graph of a problem instance graph. It also provides access to basic information about the contents of the solution sub-graph, such as cost of the solution or the degree of a node within the solution graph. It also contains features to determine the set of edges crossed by the contained solution, and for basic solution cleaning (in other words, iteratively removing non-terminal nodes of degree zero or one within the solution sub-graph).

nqHelpers A class implementing various base algorithms: Dijkstra's shortest path algorithm, Kruskal's minimum spanning tree algorithm and modified versions of these algorithms to enable them to ignore or force nodes, and to respect a non-crossing constraint.

nqSteinerBase An abstract base class for minimum Steiner tree heuristics.

nqSSSP A class implementing the Single Source Shortest Path minimum Steiner tree heuristic described earlier. It uses Dijkstra's algorithm for determining a shortest path tree.

nqMST A class implementing the Minimum Spanning Tree heuristic for the Steiner problem we described earlier. It uses Kruskal's algorithm for determining a minimum spanning tree.

nqAPSP A class implementing the All-Pairs-Shortest-Path minimum Steiner tree heuristic described earlier (APSP). It uses multiple runs of Dijkstra's algorithm for determining the shortest paths between all pairs of terminal nodes and Kruskal's algorithm for determining a minimum spanning tree on the resulting set of paths.

nqAPSPe A class implementing the All-Pairs-Shortest-Path minimum Steiner tree heuristic described earlier (APSPe). It uses multiple runs of Dijkstra's algorithm for determining the shortest paths between all pairs of terminal nodes and Kruskal's algorithm for determining a minimum spanning tree on the resulting edge set.

nqAPSPx A class implementing the All-Pairs-Shortest-Path minimum Steiner tree heuristic described earlier (APSPx) It uses multiple runs of Dijkstra's algorithm for determining the shortest paths between all pairs of terminal nodes and Kruskal's algorithm (modified to accommodate an adapting queue) for determining a minimum spanning tree on the resulting set of paths.

nqAugmentationBase An abstract base class for Steiner tree augmentation algorithms.

nqAugSP A class implementing the AugSP augmentation algorithm described earlier. It uses a modified version Dijkstra's algorithm to find an alternate path to nodes requiring redundant connections to the infrastructure.

nqAugSPe A class implementing the AugSPe augmentation algorithm described earlier. It uses a modified version Dijkstra's algorithm to find an alternate path to nodes requiring redundant connections to the infrastructure and temporarily adds direct edges with zero weight to lower the solution cost by reusing nodes already in the solution. This implementation temporarily modifies the nqProblem instance it is working on, so caution is advised when using it in a multi-threaded environment.

nqPurifierBase An abstract base class for purifying and solution checking algorithms.

nqSPurifier A class implementing a purifier and solution checker using the Redundancy-Search Validity Check algorithm.

nqQPurifier A class implementing a purifier and solution checker using the Path-Node Removal Solution Validity Check algorithm.

nqPOpt A class implementing the key-path Replacement optimizing algorithm.

nqC1Opt A class implementing the Terminal-Node Reconnection optimizing algorithm. This implementation temporarily modifies the nqProblem instance it is working on, so caution is advised when using it in a multi-threaded environment.

nqSMMove A class implementing the SMMove move-operator described earlier.

nqSDGMove A class implementing the SDGMove move-operator described earlier.

nqSCMove A class implementing the SCMove move-operator described earlier.

nqHeuristics A class for easier access to heuristic construction algorithms, purifiers, optimizers and move-operators.

nqsChrom A class derived from the EALib chromosome class, used for Local Search and Simulated Annealing and makes use of the construction heuristics implementations and move-operators.

nqsMerger A class for containing merged solutions, and solving them using Daniel Wagner's MCFRedSolver. It is not able to respect the non-crossing constraint. It does not actually create a new problem instance out of the merged solution, but fixes nodes and edges of the problem graph that are not contained in any of the solutions not to be considered as candidates by the solver in order to speed up the exact algorithm. It can, if so configured, also force nodes and edges that are contained in all the solutions to be taken into the solution by the solver, in order to further speed up the exact algorithm.

nqVNSMerger A class derived from nqsMerger to merge intermediate solutions during a Variable Neighborhood Descent or Variable Neighborhood Search.

nqsMNOpt A class implementing a search through the MNOpt neighborhood described earlier. It is not able to respect a non-crossing constraint.

nqVNSChrom A class derived from nqsChrom with additional functionality needed for Variable Neighborhood Descent (a method to find a better/the best neighbor) and Variable Neighborhood Search (a method to shake the solution). Also notifies an instance of VNSMerger when the candidate solution is modified.

nqMethods A class for easier access to heuristic and meta-heuristic algorithms.

6.4 Use of Components

nqHeuristics This is a class that can be used for easy access to everything related to heuristic solution construction and modification. This means it is able to control Steiner tree heuristics, augmentations, solution checkers, purifiers, move-operators, optimizers and neighborhood operators. It provides means to safely move a solution (meaning the move is only applied if the solution is not destroyed), which was used in the experiments during Local Search, Simulated Annealing and for Variable Neighborhood Search shaking.

An object of this class can be created, with an nqProblem instance, an nqSolution instance and a nqLogging instance. Non-crossing constraints and move-operator-ratios can also be set, if needed. The object is able to perform the actions listed above on the given solution object using the given problem instance and outputs all logging information to the given logger object.

Whenever needed, the solution object can be extracted and the sets of nodes and edges can be set as the solution for the given nqProblem.

nqMethods This class is meant to facilitate access to the heuristic and meta heuristic methods provided by the implementation. This includes construction heuristics, Local Search, Simulated Annealing, Variable Neighborhood Descent, Variable Neighborhood Search and various Merging and Solving variants. It takes a nqProblem instance and a nqLogging instance, and creates its own solution object. Then all construction heuristics (as contained in a nqHeuristics object) and meta-heuristic approaches contained within the implementation can be used by calling them as a method of this object.

Again, whenever needed, the solution object can be extracted and the sets of nodes and edges can be set as the solution for the given nqProblem.

6.5 Definable Parameters

The list of definable parameters for the implementation of the algorithms described in this thesis are defined in a set of files called *nqParams*. The implementations of the algorithms use the parameter group "*mheur*" to avoid complications with other solvers and implementations.

The following parameters are related to these implementations:

augmaxperms The maximum number of permutations the augmentation algorithm will try until it finds a valid order of customer nodes that require a redundant connection to the infrastructure.

augmentation This determines augmentation algorithm to be used. 0 for none, 1 for AugSP and 2 for AugSPe.

c1opt Determines whether the terminal-reconnection-optimizer will be used.

c1opt_iterrem Determines whether or not the terminal-reconnection-optimizer will remove customer nodes without a redundancy requirement all at once, or one after the other before connecting them again.

c1opt_iteradd Determines whether or not the terminal-reconnection-optimizer will reconnect customer nodes without a redundancy requirement all at once, or one after the other.

cplextilim The maximum time the CPLEX-based exact solver will run before returning the best solution that was encountered.

conn The desired connectedness. For the OPT-problem, this should always be 2, but can be set to 1 for the use of just the Steiner tree problem heuristics.

ifile The input file. This is the name of the file containing the problem graph instance.

itype The input file-type of which the input file is an instance (IST, SteinLib, PCSTP, TSPLIB)

initial_heuristic This parameter determines the initial Steiner tree problem heuristic that is used for meta-heuristic algorithms. In the case of just applying heuristic methods, this determines the Steiner tree problem heuristic that is used for creating a Steiner tree. 0 for none, 1 for MST, 2 for SSSP, 3 for APSP, 4 for APSPe, 5 for APSPx.

maxi Determines whether the chosen meta-heuristic maximize or minimize the target function. Should be set to 0 for the purposes of our algorithms, since the OPT-problem is a minimization problem.

merge_apsp Add a solution constructed with APSP to the merged solution set if merging was chosen as the method to apply.

merge_apspe Add a solution constructed with APSPe to the merged solution set if merging was chosen as the method to apply.

merge_apspx Add a solution constructed with APSPx to the merged solution set if merging was chosen as the method to apply.

merge_force Forces edges that are present in all of the solutions that have been merged to be in the extracted solution as well if merging or a Variable Neighborhood meta-heuristic was chosen as the method to apply.

merge_modifier The amount of perturbation applied to the problem instance before each merging pass if merging was chosen as the method to apply.

merge_mst Add a solution constructed with MST to the merged solution set if merging was chosen as the method to apply.

merge_passes The number of times a solution is constructed using the set heuristic methods and added to the set of merged solutions, when merging is the active meta-heuristic method.

merge_sssp Add a solution constructed with SSSP to the merged solution set if merging was chosen as the method to apply.

method The desired heuristic or meta-heuristic method to employ. Can be set to 0 for none (does nothing), to 1 for just using the heuristic methods, 2 for enabling merging, 4 for Local Search, 5 for Simulated Annealing, 6 for Variable Neighborhood Descent and 7 for Variable Neighborhood Search.

mnopt Determines whether the best key-node-move neighbor in the solution will be used to optimize the solution.

mnopt_inside Determines whether the key-node-move neighborhood will look outside the solution for new possible key-nodes.

mnopt_outside Determines whether the key-node-move neighborhood will look inside the solution for new possible key-nodes.

moveop0prob This parameter is relevant for Local Search, Simulated Annealing and Variable Neighborhood Search. It determines the relative probability with which no move-operator will be used for mutation or shaking.

moveop1prob This parameter is relevant for Local Search, Simulated Annealing and Variable Neighborhood Search. It determines the relative probability with which SMMove will be used for mutation or shaking.

moveop2prob This parameter is relevant for Local Search, Simulated Annealing and Variable Neighborhood Search. It determines the relative probability with which SDGMove will be used for mutation or shaking.

moveop3prob This parameter is relevant for Local Search, Simulated Annealing and Variable Neighborhood Search. It determines the relative probability with which SCMove will be used for mutation or shaking.

multi_move This parameter sets the maximum number of times the solution is moved when a multiple move is applied to a solution. Local Search, Simulated Annealing, Variable Neighborhood Search and Variable Neighborhood Descent all use multiple moves, although the default value for it is 1, which is equivalent to a regular move.

noncross Determines whether a non-crossing constraint is active in for the problem graph or not.

noncross_method The method with which the minimum spanning tree and shortest path algorithms will be modified to accommodate a non-crossing constraint, if present. Can be set to 0 for none, 1 for cheapest edge, 2 for most expensive edge, 3 for random edge and 4 for first edge.

osfile The name of the file to which the solution is to be written.

pfy_order The order in which the purifier tries to eliminate solution edges. Can be set to 0 for random order, or to 1 for cost decreasing order.

popt Determines whether the key-path-optimizer will be used.

purifier This determines the solution checking algorithm to be used for purifying. Can be set to 0 for none, 1 for Redundancy-Search Solution Validity Check (nqSPurifier) and 2 for Path-Node Removal Solution Validity Check (nqQPurifier).

saca The slope for geometric cooling in Simulated Annealing.

sacint The interval between cooling steps in Simulated Annealing.

satemp The initial temperature for Simulated Annealing.

sub.eamod Relevant for Variable Neighborhood Search. For our purposes, it should be set to either 4 (Local Search), 5 (Simulated Annealing), or 10 (Variable Neighborhood Descent) when using Variable Neighborhood Search.

test_purifier This determines the solution checking algorithm used for a final check of the output solution. Set to 0 for none, 1 for Redundancy-Search Solution Validity Check (nqSPurifier) and 2 for Path-Node Removal Solution Validity Check (nqQPurifier).

tgen The number of generations/iterations to be performed by generational meta heuristic approaches.

vnd_mergeinterval Variable Neighborhood Descent (also when used as a sub-algorithm during Variable Neighborhood Search) will add each nth solution it encounters to a merged solution set, if it differs at least by an absolute amount of d from the last solution in terms of cost. This parameter sets n. Set to 0 for no merging during Variable Neighborhood Descent or Variable Neighborhood Search.

vnd_mergemincostdiff Variable Neighborhood Descent (also when used as a sub-algorithm during Variable Neighborhood Search) will add each nth solution it encounters to a merged solution set, if it differs at least by an absolute amount of d from the last solution in terms of cost. This parameter sets d.

vns_max_neigh This parameter sets the maximum neighborhood distance that is traversed by the solution when shaking during Variable Neighborhood Search.

7 Experimental Results

In this section we will show the results acquired using the implementation and methods described in the last sections. We will first give an overview about the details and configurations we tested, and then evaluate the results acquired by applying them to a set of problem instances.

7.1 Overview of Conducted Experiments

We tested various types of instances, and applied our methods using the described implementation using several configurations and combinations, which we will describe in detail.

7.1.1 Tested Instances

We performed our experiments on three types of instances. For two of these, we had subsets of instances, with each subset having a different infrastructure sub-graph. Our experiments were conducted using the following four large sets of problem graph instances:

GridGraph100/400 A collection of randomly generated weighted graphs, arranged in a grid consisting of connected graphs with four nodes each. Edge costs are also randomly assigned. Each graph contains a hundred/four hundred vertices. Three sets of instances containing 100 nodes and (after preprocessing by the nqProblem class) 342 edges were tested, two of which contained 15 instances, and one containing 20 instances. Also, two sets of instances containing 400 nodes were tested, each containing fifteen instances, of which each (again, after nqProblem preprocessing) contained 1482 edges. Of the nodes, at most 10 were customer nodes, and around 50% of these required a redundant connection to the infrastructure. Each instance contained from 10 to 20 junction nodes. Exact results for these instances were obtained using a multi-commodity flow approach described in [29], and all have gaps below 10%, with optimal solutions for most instances.

ClgSmall/Medium Extracted from parts of the spatial topology of the city of Cologne, Germany and processed for the NETQUEST project, these instances describe parts of the possible cable routes through the city. The "Extra"-Instances of these sets were tested. The ClgSmall set contains three subsets, with the first subset containing twenty-five, and the second and third subset containing fifteen instances. After nqProblem preprocessing, each of these instances contained 377 edges. The ClgMedium set consists of two subsets, with the first subset containing twenty-five and the second subset containing fifteen instances. Each instance contains, after nqProblem preprocessing, 3877 edges. At most 10 nodes were set a customer nodes, and around 50% of these required a redundancy in their connection to the infrastructure, and instances contained between 10 to 20 junction nodes. Exact results for these instances were obtained using a multi-commodity flow approach described in [29], and all have gaps below 10%, with optimal solutions for most instances.

TSPLib These instances were acquired by applying a transformation to various instances intended for testing of approaches for solving the Travelling Salesperson Problem. This transformation converted each instance into a connected graph by triangulation, using geometric distance to derive costs for each resulting edges. Graph sizes vary from 29 to 226 nodes, and from 77 to 586 edges. One node was chosen as root node to represent an existing infrastructure. 33% of nodes were declared customer nodes, and 25% as customer nodes requiring a redundant connection to a given infrastructure. Exact results for these instances were obtained using a multi-commodity flow approach described in [29] with a time limit of 900 seconds for algorithm execution, and have an average gap of 19% when compared to the best target function value that was encountered during the process, with optimal solutions for the smaller instances.

Most experiments were performed on real-world and grid-graph based instances without a non-crossing constraint. Result analyses and interpretations contained in the evaluation do not apply to TSPLib instances unless explicitly stated.

7.1.2 Measured Values

All experiments measured the cost of the cheapest solution obtained during a heuristic or meta-heuristic run. Additionally, we measured total running time (excluding only preprocessing done by the nqProblem class) in seconds, to the second decimal point. Therefore running times, for example in the case of purifying, also include the measurements of Steiner tree construction and augmentation.

For experiments using approaches involving solution merging, we also recorded how many edges were forced into the solution, and how many edges were excluded from being in the solution.

7.1.3 Used Hardware

The experiments were conducted on a machine provided by the Institute for Computer Graphics and Algorithms of the Vienna University of Technology. It uses a 2.8 Gigahertz processor and 2 Gigabytes of RAM with 512k L2 cache.

7.1.4 Experiments using Heuristic Approaches

Heuristic Steiner Tree Construction Algorithms The whole test-set of problem instances was entered into each of the implemented Steiner tree heuristics. The resulting Steiner trees were then augmented by the AugSPe heuristic, but not purified or improved. These approaches were tested on all instances without a non-crossing constraint, and on the smaller instances of the real-world and grid-graph based sets with a non-crossing constraint.

Heuristic Augmentation Algorithms Steiner trees were constructed on all the test problem instances using the APSPx heuristic, and then augmented using each of the two augmentation heuristics AugSP and AugSPe. The algorithms were set to try maximally 10 permutations of customer nodes requiring redundant connections to the infrastructure.

We also used these algorithms for constructing solutions with an active non-crossing constraint for our smaller test instances. These approaches were tested on all instances without a non-crossing constraint, and on the smaller instances of the real-world and grid-graph based sets with a non-crossing constraint.

Purifying Algorithms Solutions were generated for all test problem instances using the APSPx Steiner tree heuristic and the AugSPe augmentation heuristic. The valid results were then purified using each of the two purifying algorithms. Solutions for instances that were not valid have been taken out of the comparisons details and statistics, but are included in the tables in the Appendix. We also purified solutions for instances with a non-crossing constraint. These approaches were tested on all instances without a non-crossing constraint, and on the smaller instances of the real-world and grid-graph based sets with a non-crossing constraint.

Heuristic Improvement Algorithms The improvements strategies were applied with standard settings, using a purified solution constructed using APSPx and AugSP. C1Opt was used with iterative addition and non-iterative removal of C_1-Nodes with a degree of 1 within the solution sub-graph. These approaches were tested using all instances of the real-world and grid-graph based sets without a non-crossing constraint.

7.1.5 Experiments using Local Search and Simulated Annealing

These approaches were tested using all instances of the real-world and grid-graph based sets without a non-crossing constraint.

Local Search The experiments using Local Search used the three move-operators (SMMove, SDGMove, SCMove) in a 1:1:1 ratio, over 3000 iterations, using APSPx for constructing a Steiner tree and AugSPe for augmentation.

For the multi-move variant (with multi-move set to move the solution a maximum number of 10 times), 700 iterations were performed, with the initial solution being constructed in the same way.

The number of iterations was chosen using results from preliminary tests.

Simulated Annealing The experiments using Simulated Annealing used the three move-operators (SMMove, SDGMove, SCMove) in a 1:1:1 ratio, over 3000 iterations, using APSPx for constructing a Steiner tree and AugSPe for augmentation.

As with Local search, for the multi-move variant (with multi-move set to move the solution a maximum number of 10 times), 700 iterations were performed. Again, APSPx and AugSPe were used for constructing the initial solution.

As with Local Search, the number of iterations was chosen using results from preliminary tests.

7.1.6 Experiments using Variable Neighborhood Descent and Variable Neighborhood Search

These approaches were tested using InstClgSmall, InstClgMedium, G0100 and G0400 instances without a non-crossing constraint.

Variable Neighborhood Descent The experiments involving Variable Neighborhood Descent were conducted on all instances, using MNOpt, POpt and C1Opt as a neighborhood operator. We configured the neighborhood operator to return the next best result encountered. If MNOpt returned no better result, POpt was used. If POpt did not return a better result, C1Opt was applied to the best solution so far encountered.

Variable Neighborhood Search For the experiments using Variable Neighborhood Search, Variable Neighborhood Descent, as described in the previous paragraph, was used as sub-heuristic. Between runs of Variable Neighborhood Descents, SMMove, SDGMove and SCMove were used for shaking, starting from a shaking neighborhood-distance of size 1 upwards to a neighborhood-distance of size 15. The algorithm was performed for 100 iterations. This termination condition was selected because of the results of preliminary tests.

7.1.7 Experiments using Solution Merging

Since these experiments involved an exact solver, and running times would be unacceptably high for larger instances, running times for the exact solver were capped at 15 minutes and the best result encountered by the solver was used. The option of forcing edges to the solver was not used for the purpose of these experiments, since its use would increase solution cost and the set cap time was more than enough for all instances to be solved in time without this further speed-up. These approaches were tested on all instances without a non-crossing constraint, and on the smaller instances of the real-world and grid-graph based sets with a non-crossing constraint.

Merging of Various Heuristic Solutions For the experiments, one solution was created using each Steiner tree heuristic with AugSPe augmentation and no purification. The solutions were merged using the SMerger method, and the sub-graph constructed using the merged set of solutions was entered into a multi-commodity flow solver. The extracted solution was then checked for validity. We also used this approach to solve the smaller instances of our test-set with an active non-crossing constraint.

Merging of Various Heuristic Solutions Built Using Modified Edge Weights This was conducted for all test problem instances. For each instance, three randomized instances were generated by randomly modifying the weights by a maximum amount of 20% for each edge. The original and the three derived instances were each entered into each Steiner tree heuristic and augmented using the AugSPe augmentation heuristic. For each problem instance, the twenty resulting solution sub-graphs (one of each Steiner tree heuristic augmented with AugSPe, and three constructed the same way using modified edge weights) were merged and entered into a multi-commodity flow solver. The extracted solution was checked for validity. We also applied this approach to the smaller instances of our test-set using an active non-crossing constraint.

7.2 Evaluation of Experimental Results

In this section we will list the results acquired using the methods, implementations and configurations we described earlier. Gaps are measured in percent against the best known solution (see the appendix for a complete listing) using the following formula:

$$Gap\% = 100 \cdot \frac{(c_{best\_known} - c_{heuristic})}{c_{best\_known}}$$

Percentages are calculated against the larger value, averages and standard deviations are calculated on percentage values if they are used to describe general properties of a series of percentages, and are given in brackets next to the average value.

7.2.1 Ability of Heuristics to Generate Valid Solutions

Augmented Steiner Tree Heuristics Since the Steiner tree heuristics are augmented by searching for the shortest path to the customer nodes that require redundancy that does not share any nodes with the path in the Steiner tree itself, it can happen that the path in the Steiner tree was chosen in a manner that makes augmentation of certain nodes impossible. Another possible case is that a path added during augmentation itself now blocks another path from becoming augmented by a non-crossing constraint.

When we look at the ability to generate valid solutions when using various Steiner Tree heuristics and the AugSPe augmentation, it is obvious that the SSSP algorithm performed best on our test instances with no non-crossing constraint, since it was the only heuristic that was able to provide a valid solution on all of the 175 test instances. It is closely followed by the APSPx heuristic, which was able to provide 173 valid solutions (98.86% success rate), and the APSP heuristic, which managed to generate valid solutions for 172 instances (98.29% success rate). The algorithms employing Kruskal's minimum spanning tree algorithm directly on edges were only able to generate valid solutions for 161 (APSPe, 92% success rate) and 160 (MST, 91,42% success rate) instances.

On the TSPLib instances we found that SSSP returned the most valid solutions, with a success rate of 97%, followed by APSPx with 79% success rate. The edge-MST-based heuristics were only able to provide valid results for 60%-64% of instances.

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	80	100	100	84	100
ClgSmall-I2	73.33	100	93.33	73.33	93.33
ClgSmall-I3	73.33	100	100	73.33	100
ClgMedium-I1	100	100	100	100	100
ClgMedium-I2	93.33	100	93.33	93.33	93.33
G0100-I1	100	100	100	93.33	100
G0100-I2	100	100	93.33	100	93.33
G0100-I3	100	100	95	100	100
G0400-I1	93.33	100	100	93.33	93.33
G0400-I2	100	100	100	100	100
All above	91.43	100	98.29	92	98.86
TSPLib instances	60.61	96.97	72.73	63.64	78.79

Table 1: Percentage of valid solutions acquired with various augmented Steiner tree heuristics

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	80	100	100	88	100
ClgSmall-I2	73.33	100	80	73.33	80
ClgSmall-I3	73.33	100	100	73.33	100
G0100-I1	93.33	100	100	93.33	100
G0100-I2	93.33	93.33	93.33	93.33	93.33
G0100-I3	65	100	95	100	100
All above	79.04	99.05	95.23	87.61	96.19

Table 2: Percentage of valid solutions acquired with various augmented Steiner tree heuristics and an active non-crossing constraint

Augmentation Heuristics As expected, both heuristics performed the same (using APSPx as a Steiner tree heuristic) in terms of solution augmentation validity, because both heuristics only fail in the case that no path to the augmentation target can be found that is node disjunct to the path within the Steiner tree. It terminated with 173 out of 175 tested instances being augmented successfully (98.86% success rate).

For tested TSPLib instances, AugSPe was by far more reliable, providing valid results for 79% of the instances, while AugSP was only able to augment 30% of the instances successfully.

Also in the case of having a non-crossing constraint, both heuristics performed the equally and were able to augment 96.19% of our small instance sets.

7.2.2 Cost of Solutions Provided by Construction Heuristics

Augmented Steiner Tree Heuristics We shall now compare the Steiner tree heuristics in terms of cost (again, the Steiner trees were augmented using the AugSPe heuristic algorithm).

The MST Steiner tree heuristic proved to be the worst one in terms of cost, with only returning the cheapest result in 2 cases, both of which were invalid solutions. This is, however, acceptable since this heuristic was not designed with competitivity in mind, but with the concept of diversifying the set of solutions we can generate using greedy Steiner tree heuristics.

The Steiner tree heuristic APSPe was the second worst, with 5 results being the cheapest on an instance. Of out of these solutions, 1 proved to be invalid.

The SSSP Steiner tree heuristic proved to be a lot better for our test instances. It provided the best solution in 45 cases, of which none were invalid.

The heuristic APSPx is ranked number two when comparing solution cost, with the cheapest solution in 58 cases. Of these, 1 was invalid.

APSP was the Steiner tree heuristic that returned the cheapest result in 65 cases. Of these solutions, 1 proved to be invalid.

In total, the cheapest result was valid in 169 of 175 cases (96.57% success rate).

For TSPLib instances, APSPx was the heuristic that provided the cheapest results, with an average gap of 20% to the exact solution. Since the exact method was capped during the runs for TSPLib instances, and thus gaps exist for many instances, optimality of solutions can not be concluded in any case. In four cases, it yielded results that were better than those provided by the exact method. In SSSP solutions had the highest gap, with an average of 51%.

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	59.78 (39.11)	8.99 (13.01)	9.08 (15.02)	54.91 (40.56)	8.60 (15.11)
ClgSmall-I2	48.84 (23.09)	24.62 (12.67)	15.60 (10.19)	46.18 (22.62)	15.56 (10.25)
ClgSmall-I3	33.06 (20.38)	12.90 (9.74)	10.04 (9.04)	31.29 (18.83)	13.77 (11.79)
ClgMedium-I1	66.37 (28.33)	17.52 (11.45)	14.16 (8.50)	54.92 (20.36)	12.55 (8.39)
ClgMedium-I2	63.43 (35.44)	22.31 (13.72)	11.43 (9.36)	57.30 (26.97)	10.67 (9.61)
G0100-I1	45.77 (13.13)	32.89 (8.27)	32.48 (11.25)	40.50 (11.10)	31.53 (12.51)
G0100-I2	50.83 (21.99)	30.50 (10.49)	25.87 (11.54)	45.30 (19.08)	24.60 (9.10)
G0100-I3	18.26 (11.96)	12.31 (12.31)	11.12 (11.34)	15.66 (9.81)	12.49 (11.45)
G0400-I1	43.63 (11.61)	24.29 (7.95)	19.58 (6.76)	43.23 (11.76)	17.17 (6.06)
G0400-I2	63.38 (7.87)	35.12 (6.16)	30.41 (8.45)	61.60 (7.02)	29.89 (9.17)
TSPLib	43.43 (19.48)	50.59 (25.10)	22.29 (17.31)	42.84 (19.05)	20.42 (15.02)

Table 3: Average gap percentages of solution costs acquired with various augmented Steiner tree heuristics

Instance set	Tot.	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	25	0	0	1	0	1
ClgSmall-I2	15	0	0	0	0	0
ClgSmall-I3	15	0	0	0	0	0
ClgMedium-I1	25	0	0	0	0	0
ClgMedium-I2	15	0	0	0	0	0
G0100-I1	15	0	0	0	0	0
G0100-I2	15	0	0	0	0	0
G0100-I3	20	0	1	3	0	3
G0400-I1	15	0	0	0	0	0
G0400-I2	15	0	0	0	0	0
Total	175	0	1	4	0	4

Table 4: Number of optimal solutions for the OPT-problem found using various augmented Steiner tree heuristics

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	64.43 (42.37)	22.91 (21.34)	19.52 (18.73)	58.18 (42.58)	16.12 (16.21)
ClgSmall-I2	44.20 (23.34)	29.30 (12.46)	20.72 (11.69)	51.28 (21.96)	19.54 (10.68)
ClgSmall-I3	33.61 (20.07)	25.97 (16.56)	21.24 (15.54)	29.29 (19.07)	22.83 (15.81)
G0100-I1	46.01 (16.36)	41.76 (11.64)	37.20 (15.24)	46.37 (16.88)	37.89 (17.90)
G0100-I2	49.43 (21.76)	39.40 (16.51)	26.42 (10.67)	43.55 (14.74)	27.66 (12.26)
G0100-I3	17.98 (16.67)	18.26 (12.33)	16.38 (14.15)	21.81 (14.54)	17.74 (13.73)

Table 5: Average gap percentages of solution costs acquired with various augmented Steiner tree heuristics using an active non-crossing constraint

Instance set	Tot.	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	25	0	0	0	0	0
ClgSmall-I2	15	0	0	0	0	0
ClgSmall-I3	15	0	0	0	0	0
G0100-I1	15	0	0	0	0	0
G0100-I2	15	0	0	0	0	0
G0100-I3	20	0	0	2	0	2
Total	105	0	0	2	0	2

Table 6: Number of optimal solutions OPT-problem found using various augmented Steiner tree heuristics with a non-crossing constraint

From the average results on the classes, it can be said that APSPx seemed to yield the best results for all larger instance classes (ClgMedium and G0400), while APSP and SSSP found cheaper results on some of the smaller classes. With a non-crossing constraint, on average, APSPx yielded the best results on all large instance sets and APSP on the smaller instance sets. In some cases, augmented SSSP, APSP and APSPx solutions had costs equal to or better than the cost of the best known solution. This also occurred in some cases when a non-crossing constraint was present.

Augmentation Heuristics AugSPe proved to be the heuristic acquiring better results. It provided the cheaper solution in 105 cases (exactly 60%) . In 58 cases (33.14%), the results returned by AugSPe and AugSP were equal. The average difference between the results of the two approaches was 2.80% of the results provided by the AugSP heuristic, in favor of the AugSPe augmentation heuristic, showing a standard deviation of 5.52.

Instance set	AugSP	AugSPe
ClgSmall-I1	9.48 (15.51)	8.60 (15.11)
ClgSmall-I2	18.12 (13.24)	15.56 (10.25)
ClgSmall-I3	15.56 (13.09)	13.77 (11.79)
ClgMedium-I1	13.68 (8.11)	12.56 (8.39)
ClgMedium-I2	17.46 (17.55)	10.68 (9.62)
G0100-I1	35.47 (15.60)	31.53 (12.51)
G0100-I2	30.81 (11.18)	24.60 (9.10)
G0100-I3	14.43 (11.57)	12.49 (11.45)
G0400-I1	26.14 (10.56)	17.17 (6.07)
G0400-I2	38.57 (10.02)	29.89 (9.17)
TSPLib	14.79 (32.07)	20.42 (15.82)

Table 7: Average gap percentages of solution costs acquired with various augmentation heuristics

Instance set	Tot.	AugSP	AugSPe
ClgSmall-I1	25	1	1
ClgSmall-I2	15	0	0
ClgSmall-I3	15	0	0
ClgMedium-I1	25	0	0
ClgMedium-I2	15	0	0
G0100-I1	15	0	0
G0100-I2	15	0	0
G0100-I3	20	1	3
G0400-I1	15	0	0
G0400-I2	15	0	0
Total	175	2	4

Table 8: Number of optimal solutions found using various augmentation heuristics

AugSPe, on average, was better for every problem class except for the TSPLib instances, which indicates that including information about parts that are already covered by the solution in a heuristic does indeed yield an advantage. Using AugSP, an optimal solution was found in less cases than using AugSPe. for TSPLib instances, AugSP provided a valid solution in very little cases, but when it did, the solution was often one that was better than the one provided by exact methods. Optimality of solutions can not be determined, as in the last section, due to gaps in the solutions provided by exact methods.

Instance set	AugSP	AugSPe
ClgSmall-I1	17.68 (16.30)	16.12 (16.21)
ClgSmall-I2	25.48 (13.21)	19.54 (10.68)
ClgSmall-I3	35.98 (21.90)	22.83 (15.81)
G0100-I1	44.25 (14.75)	37.89 (17.90)
G0100-I2	35.39 (10.91)	27.66 (12.26)
G0100-I3	20.08 (12.98)	17.74 (13.73)

Table 9: Average gap percentages of solution costs acquired with various augmentation heuristics with an active non-crossing constraint

Instance set	Tot.	AugSP	AugSPe
ClgSmall-I1	25	0	0
ClgSmall-I2	15	0	0
ClgSmall-I3	15	0	0
G0100-I1	15	0	0
G0100-I2	15	0	0
G0400-I2	15	0	2
Total	105	0	2

Table 10: Number of optimal solutions found using various augmentation heuristics with a non-crossing constraint

When the non-crossing constraint was enabled, the differences between the two approaches were similar, with AugSPe outperforming AugSP on average in every tested problem class.

7.2.3 Heuristic Construction Algorithm Running Times

Augmented Steiner Tree Heuristics When considering all Steiner tree construction heuristics, SSSP was clearly the fastest, with having the shortest solution construction time in 168 cases (exactly 96%). MST was the second fastest, with having the same or better running times in 41 cases (23.43%). Both were faster than the all-pairs-shortest-path-based heuristics in all cases. Of these, APSP was clearly the fastest, with reaching the best running time in 167 cases (95.43%). APSPe shared that time or had a better running time in 49 cases (28%), and APSPx was the fastest or shared the first place in 32 cases (18.29%).

Average running times showed some interesting results: the minimum-spanning-tree-based construction heuristics that operated directly on edges had strongly varying times on the instances. This was especially true for MST, with an average of 2.28 seconds and a standard deviation of 19.60. APSPe had an average running time of 0.76 seconds with a standard deviation of 4.16. This can be explained by two things: the fact that the MST implementation used provides an edge-set as solution, which then has to be converted to a rooted tree, and the fact that the base MST algorithm, as noted earlier, tends to include edges for reaching nodes that are not in included in the resulting solution. This has an effect on augmentation path lengths as well as on solution checking, and on purification (since this does solution checking and checks the solution once for each solution edge). APSPx had an average running time of 0.24 seconds with a standard deviation of 0.30, and APSP had an average running time of 0.20 seconds with a standard deviation of 0.27. Clearly fastest was SSSP with an average running time of 0.04 seconds and a standard deviation of 0.06.

For TSPLib instances, APSPx proved to be the fastest method, with an average running time of 6.80 seconds and a standard deviation of 24.87. This can be attributed to the fact that the Steiner tree phase of SSSP generally tends to include more nodes than a minimum spanning tree on the shortest paths, which can lead to a high number of permutations being attempted during augmentation phase due to blocked paths. MST was slowest on TSPLib instances, with an average running time of 97.42 seconds and a standard deviation of 430.49.

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	0.02 (0.005)	0.09 (0.003)	0.02 (0.002)	0.03 (0.003)	0.02 (0.004)
ClgSmall-I2	0.04 (0.02)	0.02 (0.01)	0.04 (0.007)	0.06 (0.01)	0.07 (0.012)
ClgSmall-I3	0.02 (0.005)	0.01 (0.003)	0.03 (0.005)	0.04 (0.006)	0.04 (0.01)
ClgMedium-I1	15.09 (49.95)	0.10 (0.03)	0.72 (0.04)	3.99 (10.38)	0.76 (0.06)
ClgMedium-I2	0.65 (0.10)	0.11 (0.06)	0.06 (0.04)	1.12 (0.16)	0.78 7(0.11)
G0100-I1	0.03 (0.01)	0.02 (0.006)	0.04 (0.01)	0.04 (0.01)	0.07 (0.02)
G0100-I2	0.02 (0.01)	0.02 (0.006)	0.03 (0.004)	0.03 (0.01)	0.04 (0.01)
G0100-I3	0.01 (0.004)	0.007 (0.004)	0.02 (0.004)	0.03 (0.004)	0.02 (0.005)
G0400-I1	0.10 (0.04)	0.043 (0.02)	0.11 (0.01)	0.17 (0.05)	0.14 (0.03)
G0400-I2	0.05 (0.77)	0.09 (0.10)	0.19 (0.06)	0.66 (0.79)	0.26 (0.09)
TSPLib	97.42 (430.49)	17.17 (59.43)	8.84 (29.44)	95.65 (429.22)	6.80 (24.87)

Table 11: Average running times of solution construction using various Steiner tree heuristics in seconds (std. deviation in brackets)

Instance set	MST	SSSP	APSP	APSPe	APSPx
ClgSmall-I1	0.02 (0.005)	0.008 (0.003)	0.02 (0.005)	0.03 (0.003)	0.03 (0.005)
ClgSmall-I2	0.03 (0.01)	0.02 (0.008)	0.05 (0.01)	0.06 (0.02)	0.08 (0.02)
ClgSmall-I3	0.02 (0.005)	0.01 (0.005)	0.04 (0.006)	0.04 (0.007)	0.05 (0.01)
G0100-I1	0.02 (0.009)	0.02 (0.006)	0.05 (0.01)	0.05 (0.01)	0.08 (0.02)
G0100-I2	0.02 (0.004)	0.01 (0.005)	0.03 (0.006)	0.03 (0.005)	0.04 (0.01)
G0100-I3	0.01 (0.003)	0.008 (0.004)	0.02 (0.03)	0.02 (0.004)	0.03 (0.01)

Table 12: Average running times of solution construction using various Steiner tree heuristics in seconds using a non-crossing constraint

Augmentation Heuristics AugSP was clearly faster than AugSPe. In 81 cases (46.29%) it returned a solution quicker than AugSPe, which was faster in only 16 cases (9.14%). In 78 cases (44.57%) running times were equal.

On average, a solution obtained using AugSP could be constructed 4.68% faster than with AugSPe, with a standard deviation of 16.13.

For TSPLib instances AugSPe proved to be faster, with an average running time of 15.82 seconds (std. dev. 6.80) versus an average running time of AugSP of 32.07 seconds (std. dev. 7.91).

Instance set	AugSP	AugSPe
ClgSmall-I1	0.02 (0.0032)	0.024 (0.005)
ClgSmall-I2	0.07 (0.02)	0.071 (0.02)
ClgSmall-I3	0.04 (0.01)	0.04 (0.01)
ClgMedium-I1	0.76 (0.05)	0.77 (0.06)
ClgMedium-I2	0.77 (0.11)	0.78 7(0.11)
G0100-I1	0.07 (0.02)	0.07 (0.02)
G0100-I2	0.04 (0.01)	0.04 (0.009)
G0100-I3	0.02 (0.004)	0.02 (0.005)
G0400-I1	0.13 (0.03)	0.14 (0.03)
G0400-I2	0.025 (0.08)	0.26 (0.09)
TSPLib	32.07 (7.91)	15.82 (6.80)

Table 13: Average running times of solution construction various augmentation heuristics in seconds (std. deviation in brackets)

Instance set	AugSP	AugSPe
ClgSmall-I1	0.02 (0.003)	0.03 (0.005)
ClgSmall-I2	0.07 (0.02)	0.08 (0.02)
ClgSmall-I3	0.04 (0.01)	0.05 (0.01)
G0100-I1	0.07 (0.02)	0.08 (0.02)
G0100-I2	0.04 (0.01)	0.04 (0.01)
G0100-I3	0.02 (0.004)	0.03 (0.01)

Table 14: Average running times of solution construction various augmentation heuristics with a non-crossing constraint in seconds

More or less the same differences between the two heuristic augmentation algorithms can be seen when a non-crossing constraint is set, with AugSP being generally faster than AugSPe.

7.2.4 Purification and Improvement Heuristics

Purification As expected, both purification algorithms had the same effect on all solutions they were applied to. Purification yielded an improvement in 152 of 173 cases (86.86%). The purified solution was, on average 6.59% cheaper than the original solution, with a standard deviation of 7.01. An optimal solution was found in more cases than using only construction heuristics. For TSPLib instances, average solution improvement was 17.49%, with a standard deviation of 5.26.

For the two instances for which APSPx in combination with AugSPe did not provide a valid solution, obviously no improvement was found.

Running times were similar in many cases, but showed some differences. The SPurifier implementation clearly showed and advantage, with shorter running times in 118 of 173 cases (68.20%). In 37 cases (21.39) measured total running times were equal to the 2nd decimal point, and in 18 cases (10.40%), all of which were in the ClgMedium instance set, QPurifier showed a shorter running time. On average, SPurifier outperformed QPurifier by 29.64% of its running time (in terms of total running time), with a standard deviation of 51.27. For TSPLib instances, QPurify had the shorter running time, on average.

Instance set	% impr.
ClgSmall-I1	2.03 (3.14)
ClgSmall-I2	5.99 (6.73)
ClgSmall-I3	7.94 (7.77)
ClgMedium-I1	1.31 (2.13)
ClgMedium-I2	4.43 (6.87)
G0100-I1	16.73 (6.55)
G0100-I2	9.20 (5.57)
G0100-I3	5.03 (6.03)
G0400-I1	6.39 (4.49)
G0400-I2	13.04 (4.84)
All above	6.59 (7.02)
TSPLib	17.49 (5.26)

Table 15: Average cost improvements using purification

Instance set	Tot.	Purifier
ClgSmall-I1	25	1
ClgSmall-I2	15	0
ClgSmall-I3	15	0
ClgMedium-I1	25	0
ClgMedium-I2	15	1
G0100-I1	15	0
G0100-I2	15	0
G0100-I3	20	6
G0400-I1	15	0
G0400-I2	15	0
Total	175	8

Table 16: Number of optimal solutions found using APSPx, AugSPe and Purification

Instance set	% impr.
ClgSmall-I1	3.58 (4.45)
ClgSmall-I2	6.67 (6.97)
ClgSmall-I3	7.44 (7.41)
G0100-I1	17.49 (5.78)
G0100-I2	13.21 (8.64)
G0100-I3	7.54 (4.98)
All Instances	7.85 (7.23)

Table 17: Average cost improvements using purification with an active non-crossing constraint

Instance set	Tot.	Purifier
ClgSmall-I1	25	0
ClgSmall-I2	15	0
ClgSmall-I3	15	0
G0100-I1	15	2
G0100-I2	15	2
G0400-I2	15	6
Total	105	10

Table 18: Number of optimal solutions found using APSPx, AugSPe and Purification with a non-crossing constraint

On average, with an active non-crossing constraint, purification manages to reduce the cost of our solutions provided by AugSPe and APSPx even more. Purification time is a bit higher in this case, since the solution is also checked for a violation of the non-crossing constraint.

Instance set	SPurify	QPurify
ClgSmall-I1	0.03 (0.007)	0.03 (0.07)
ClgSmall-I2	0.10 (0.03)	0.14 (0.06)
ClgSmall-I3	0.06 (0.01)	0.08 (0.25)
ClgMedium-I1	2.15 (0.93)	2.296 (1.81)
ClgMedium-I2	1.25 (0.25)	1.82 (0.61)
G0100-I1	0.10 (0.04)	0.16 (0.07)
G0100-I2	0.06 (0.01)	0.08 (0.04)
G0100-I3	0.03 (0.007)	0.03 (0.008)
G0400-I1	0.25 (0.11)	0.35 (0.21)
G0400-I2	0.50 (0.27)	1.02 (0.61)
TSPLib	30.25 (83.77)	17.37 (50.29)

Table 19: Average running times of construction and purification using various purification algorithms in seconds

Instance set	SPurify	QPurify
ClgSmall-I1	0.03 (0.007)	0.04 (0.01)
ClgSmall-I2	0.04 (0.14)	0.14 (0.07)
ClgSmall-I3	0.02 (0.08)	0.08 (0.02)
G0100-I1	0.03 (0.15)	0.15 (0.05)
G0100-I2	0.02 (0.07)	0.07 (0.03)
G0100-I3	0.006 (0.03)	0.03 (0.008)

Table 20: Average running times of construction and purification using various purification algorithms with a non-crossing constraint in seconds

Improvement Heuristics Both improvement algorithms on their own had little effect on the quality of most solutions, since they are also based on shortest paths and use the existing solution as a pseudo-infrastructure. They are, however, quite effective when applied to solutions generated using the two Steiner tree construction heuristics that are minimum-spanning-tree based and apply Kruskal's minimum spanning tree directly to instance edges and not shortest-path meta-edges, since they tend to ”straighten out” meandering key-paths that were constructed while trying to include additional nodes. However, to preserve comparability, we chose to reflect only the effects on already purified solutions constructed with the APSPx Steiner tree construction heuristic. For the two instances that a Steiner tree constructed with APSPx did not return a valid solution for when augmented with AugSPe, no valid solution was found, as expected.

POpt was able to cheapen purified solutions by 0.52% on average with a standard deviation of 2.51, while C1Opt was able to reduce the cost of the given purified solution by 0.95% on average with a standard deviation of 2.62.

Total running times (including construction and purification) were 0.65 seconds for POpt, with a standard deviation of 1.08 and 0.65 seconds for C1Opt, with a standard deviation of 1.07.

On some instances, both optimizers even worsened the solution by introducing unneeded redundant edges, thus canceling the effects of purification.

ClgSmall-I1	% Impr. POpt	% Impr. C1Opt
ClgSmall-I2	0.02 (0.09)	0.10 (0.34)
ClgSmall-I3	0.00 (0.00)	0.80 (0.90)
ClgMedium-I1	0.00 (0.00)	0.41 (0.74)
ClgMedium-I2	0.00 (0.00)	0.37 (0.70)
G0100-I1	0.06 (0.16)	0.12 (0.20)
G0100-I2	2.84 (3.75)	3.08 (3.78)
G0100-I3	1.59 (3.62)	2.42 (3.69)
G0400-I1	1.09 (5.27)	1.16 (5.29)
G0400-I2	0.09 (0.25)	0.77 (1.12)
G0400-I2	0.03 (0.10)	1.12 (1.16)
All above	0.52 (2.51)	0.95 (2.62)

Table 21: Average cost improvements using various improvement heuristics algorithms of purified solutions in percent

Instance set	POpt	C1Opt
ClgSmall-I1	0.0364 (0.0074)	0.0364 (0.0069)
ClgSmall-I2	0.1373 (0.0593)	0.1407 (0.0580)
ClgSmall-I3	0.0827 (0.0252)	0.0840 (0.0263)
ClgMedium-I1	2.2880 (1.6562)	2.2696 (1.6403)
ClgMedium-I2	1.2507 (1.3591)	1.8227 (0.5790)
G0100-I1	0.1547 (0.0638)	0.1600 (0.0632)
G0100-I2	0.0873 (0.0406)	0.0873 (0.0364)
G0100-I3	0.0310 (0.0089)	0.0340 (0.0073)
G0400-I1	0.3627 (0.2075)	0.3627 (0.2075)
G0400-I2	1.0307 (0.6029)	1.0327 (0.6053)

Table 22: Average running times of construction, purification and heuristic improvement using various improvement heuristics in seconds

7.2.5 Meta-Heuristics

Local Search As expected, this approach returned invalid solutions for only two instances, which were the instances for which augmentation with AugSPe of a Steiner tree constructed using APSPx did not provide a valid initial solution. They were ignored during evaluation. The variant without multi-moves showed little improvement over the initial construction heuristics. This might be due to the move-operators having too little effect on the overall structure of the solution. In cases where an improvement was found (27% of the tested initial solutions), the new solution was 3.46% better than the initial solution, on average. Average running time was 52.42 seconds, and average improvement when considering all instances was 0.95% with a standard deviation of 2.94. However, additional experiments have shown that the effect is larger when using initial solutions with a larger gap.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	5.97 (13.86)	8.26 (1.68)	0.24 (1.08)
ClgSmall-I2	7.03 (5.80)	16.56 (3.14)	0.97 (1.72)
ClgSmall-I3	3.81 (3.57)	12.69 (2.96)	0.06 (0.19)
ClgMedium-I1	10.99 (8.34)	161.70 (42.97)	0.06 (0.28)
ClgMedium-I2	4.98 (5.92)	150.27 (40.85)	0.26 (0.96)
G0100-I1	5.24 (5.71)	14.88 (4.26)	3.25 (3.63)
G0100-I2	9.49 (7.84)	12.50 (3.49)	2.94 (3.96)
G0100-I3	5.02 (9.91)	6.12 (1.19)	1.27 (5.29)
G0400-I1	9.10 (6.05)	45.56 (10.17)	0.43 (1.21)
G0400-I2	11.55 (6.10)	67.66 (17.02)	0.97 (1.93)
All above	7.38 (8.77)	52.42 (63.42)	0.95 (2.78)

Table 23: Average details using Local Search without multi-moves with APSPx and AugSPe as construction heuristics

The variant with multi-moves gained better results, but not by much, indicating that stronger modification does indeed yield better results, but that the neighborhood operators did not change the solution enough to search through the solution space successfully. Both variants seem to yield a lot more improvement on the small, randomly generated instances of G0100. Improvement was still rather small, but roughly 29% better than using the variant without multi-move at comparable run-times due to the smaller amount of generations that had to be performed.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	5.65 (12.53)	5.88 (2.16)	0.43 (1.40)
ClgSmall-I2	7.03 (5.80)	11.12 (5.96)	0.97 (1.72)
ClgSmall-I3	3.81 (3.57)	9.88 (5.39)	0.06 (0.19)
ClgMedium-I1	10.97 (8.53)	190 (80.55)	0.08 (0.29)
ClgMedium-I2	4.75 (6.00)	158.58 (58.68)	0.47 (1.10)
G0100-I1	4.59 (5.43)	11.55 (6.09)	3.83 (3.79)
G0100-I2	8.93 (8.01)	8.40 (4.02)	3.45 (4.17)
G0100-I3	3.91 (8.34)	3.67 (1.48)	2.13 (6.32)
G0400-I1	8.32 (6.11)	54.43 (22.29)	1.12 (2.09)
G0400-I2	11.49 (6.09)	61.48 (15.58)	1.01 (1.93)
All above	7.01 (8.33)	55.45 (78.49)	1.25 (3.43)

Table 24: Average details using Local Search with multi-moves with APSPx and AugSPe as construction heuristics

Simulated Annealing Using Simulated annealing without multi-move, the results were identical to Local Search, except for a slightly higher running time. This indicates, again, that the move operators had little effect on the structure of the solution, since they did not overcome local minima. As with Local Search, two instances could not be tested successfully because APSPx with AugSPe augmentation was not able to deliver a valid initial solution.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	5.97 (13.86)	9.12 (1.69)	0.24 (1.08)
ClgSmall-I2	7.03 (5.80)	17.63 (3.14)	0.97 (1.72)
ClgSmall-I3	3.81 (3.57)	13.63 (2.65)	0.06 (0.19)
ClgMedium-I1	10.99 (8.34)	177.41 (43.17)	0.06 (0.28)
ClgMedium-I2	4.98 (5.92)	160.37 (41.06)	0.26 (0.96)
G0100-I1	5.24 (5.71)	15.63 (4.22)	3.25 (3.63)
G0100-I2	9.49 (7.84)	13.32 (3.45)	2.94 (3.96)
G0100-I3	5.02 (9.91)	6.86 (1.16)	1.27 (5.29)
G0400-I1	9.10 (6.05)	48.78 (10.18)	0.43 (1.21)
G0400-I2	11.55 (6.10)	70.47 (17.29)	0.97 (1.93)
All above	7.38 (8.77)	65.50 (68.39)	0.95 (2.78)

Table 25: Average details using Simulated Annealing without multi-moves with APSPx and AugSPe as construction heuristics

With multi-move, better solutions could be found for many problem instances. On average, the improvement percentage was, albeit still rather small, 27% larger than with using the variant without multi-moves. As with Local Search, run times were comparable and even shorter than in the variant without multi-moves on smaller instances.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	5.60 (12.54)	6.17 (2.22)	0.47 (1.60)
ClgSmall-I2	7.03 (5.80)	11.51 (5.98)	0.97 (1.72)
ClgSmall-I3	5.15 (3.81)	9.90 (5.15)	0.06 (0.19)
ClgMedium-I1	10.94 (8.36)	192.41 (86.19)	0.10 (0.29)
ClgMedium-I2	4.66 (6.04)	159.35 (60.59)	0.57 (1.18)
G0100-I1	5.72 (4.63)	11.17 (5.72)	3.81 (3.54)
G0100-I2	3.68 (9.09)	8.39 (3.68)	3.31 (4.24)
G0100-I3	1.54 (3.97)	3.84 (1.53)	2.08 (6.31)
G0400-I1	8.65 (6.00)	54.49 (8.65)	0.83 (1.34)
G0400-I2	11.36 (6.22)	15.44 (11.36)	1.14 (1.98)
All above	7.03 (8.37)	56.02 (80.04)	1.24 (3.38)

Table 26: Average details using Simulated Annealing with multi-moves with APSPx and AugSPe as construction heuristics

Variable Neighborhood Descent We can see that the improvement on the cost of the initial solution is rather small, and, similarly to Local Search and Simulated Annealing, larger in the areas of the randomly constructed instance set of rather small problem graphs G0100. However, in contrast to Local Search and Simulated Annealing, it terminated a lot faster. This is a direct consequence of the absence of a fixed number of generations and due to the fact that this algorithm stops when none of the neighborhoods yield any improvements. As with Local Search and Simulated Annealing, no valid solutions for two instances were found, since their initial solutions were already invalid. A similar experiment letting the neighborhood operator return the best neighbor yielded exactly the same results with a slightly higher running time.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	6.10 (13.90)	0.06 (0.02)	0.13 (0.34)
ClgSmall-I2	7.12 (5.04)	0.20 (0.06)	0.86 (1.03)
ClgSmall-I3	3.42 (3.17)	0.12 (0.03)	0.42 (0.74)
ClgMedium-I1	10.62 (8.23)	3.03 (1.75)	0.38 (0.70)
ClgMedium-I2	5.03 (5.83)	2.65 (0.71)	0.20 (0.25)
G0100-I1	5.42 (5.67)	0.24 (0.09)	3.08 (3.78)
G0100-I2	9.93 (7.65)	0.15 (0.07)	2.55 (3.71)
G0100-I3	5.14 (9.92)	0.049 (0.01)	1.16 (5.29)
G0400-I1	8.62 (5.91)	0.74 (0.33)	0.86 (1.14)
G0400-I2	11.33 (5.90)	1.48 (0.69)	1.16 (1.20)
All above	7.33 (8.64)	0.93 (1.35)	0.99 (2.80)

Table 27: Average details using Variable Neighborhood Descent

Variable Neighborhood Search Here we can see that the process of shaking the solution multiple times seem to have had a large effect on results. This approach yielded almost 60% more solution improvement, on average, and reduced the gap to the best known solution even more. However, running times were significantly higher because of the fixed number of iterations that was used during the experiments. As with the previous optimization approaches, for two instances no valid solutions could be found, since the initial solutions were already invalid. A similar series of experiments, with the difference being that the neighborhood operator returns the best neighbor instead of the next better neighbor it encounters, yielded slightly more expensive solutions (on average, results using this method were 0.02% more expensive than with the version presented here), suggesting that the algorithm got trapped in local minima more often.

Instance set	% Gap	Run time	% Improvement
ClgSmall-I1	5.40 (12,53)	3.66 (1.86)	0.66 (1.67)
ClgSmall-I2	6.42 (5.62)	10.36 (3.20)	1.52 (1.27)
ClgSmall-I3	3.21 (3.27)	7.13 (3.66)	0.62 (0.84)
ClgMedium-I1	10.56 (8.28)	130.11 (47.71)	0.44 (0.75)
ClgMedium-I2	4.60 (5.84)	141.57 (37.14)	0.61 (1.00)
G0100-I1	4.81 (5.56)	13.35 (6.89)	3.63 (3.81)
G0100-I2	8.06 (7.87)	9.24 (5.57)	4.22 (3.96)
G0100-I3	3.91 (8.36)	2.98 (1.54)	2.14 (6.32)
G0400-I1	8.08 (5.96)	86.30 (30.23)	1.35 (1.42)
G0400-I2	10.67 (5.96)	74.20 (30.94)	1.74 (1.55)
All above	6.65 (8.21)	48.78 (59.32)	1.57 (3.50)

Table 28: Average details using Variable Neighborhood Search

Merging Using Multiple Augmented Steiner Trees Since a valid solution was included for each instance, this approach, as expected, returned only valid solutions. This approach had the best results so far. While the exact solver with the same settings had run times that were longer than our maximum running time of 900 seconds for most instances, this approach terminated well in time for all tested instances (according to a separate series of experiments which we will not describe here). On average, 62.60% of the edges were excluded by this algorithm, with an average running time of 30.17 seconds.

The average gap was 3.36% to the best known solution. Additional experiments show that, as expected, activation of the option to force edges which are present in all solutions that are entered into the nqSMerger class reduced running times, but also increased the gap. We got the same result as the exact solver for 26.43% of the tested instances, most of these from the smaller sets and one instance from the set G0400-I1.

For TSPLib instances a valid solution could not be determined for one instance. For the other instances, this approach was able to return a better solution than the exact method alone (with similar time constraints) in 68% of the tested cases. The solution was better by 10.19% (std. dev. 12.40) on average, when calculated over all the results on tested TSPLib instances. on average, 53.57% (std. dev 5.38) of the edges in the problem were excluded by the merge-preprocessing. Running times were a bit higher than for the exact methods, since one solution for each Steiner tree construction heuristic was created, additionally.

Instance set	% Gap	Cost ≤ Best known	Run time	% edges excluded
ClgSmall-I1	2.96 (11.03)	12/25	0.49 (0.11)	52.55 (2.53)
ClgSmall-I2	1.61 (2.18)	6/15	2.73 (0.78)	49.05 (3.52)
ClgSmall-I3	2.34 (3.10)	6/15	1.13 (0.32)	55.07 (1.98)
ClgMedium-I1	7.67 (7.74)	0/25	93.89 (60.02)	82.82 (1.74)
ClgMedium-I2	1.27 (2.21)	1/15	131.52 (31.20)	63.57 (1.19)
G0100-I1	2.29 (2.18)	4/15	2.45 (1.08)	61.66 (1.91)
G0100-I2	1.92 (3.74)	5/15	1.21 (0.25)	64.23 (2.05)
G0100-I3	5.79 (9.03)	11/20	0.37 (0.69)	43.98 (1.05)
G0400-I1	1.77 (2.57)	0/15	16.84 (3.82)	76.86 (1.60)
G0400-I2	2.38 (1.93)	0/15	36.26 (15.20)	71.53 (1.51)
All above	3.36 (2.80)	46/175	30.17 (51.14)	62.60 (12.76)
TSPLib	-10.19 (12.40)	22/32	693 (1137)	53.57 (5.38)

Table 29: Details using merging of heuristic solutions constructed with AugSPe and all five Steiner heuristics

When a non-crossing constraint was active, no valid solution could be found for the problem instance (G0100-I2-08). We found the same solutions as the exact solver itself for 18.09% of the tested instances. We can also see that, on average, less edges were included than in the variant with crossing allowed. The gap, on average is the double of that of the variant without such a constraint.

Instance set	% Gap	Cost ≤ Best known	Run time	% edges excluded
ClgSmall-I1	5.54 (11.91)	6/25	0.49 (0.12)	51.76 (2.46)
ClgSmall-I2	4.61 (4.61)	1/15	3.43 (1.44)	46.45 (3.30)
ClgSmall-I3	15.35 (10.25)	1/15	1.28 (0.51)	52.94 (3.70)
G0100-I1	6.20 (4.23)	0/15	2.57 (1.15)	58.15 (2.09)
G0100-I2	11.95 (23.19)	0/15	1.26 (0.36)	62.90 (2.46)
G0100-I3	10.24 (12.69)	5/20	0.37 (0.07)	40.85 (1.25)
All above	6.40 (11.95)	13/105	1.41 (1.32)	51.60 (7.56)

Table 30: Details using merging of heuristic solutions constructed with AugSPe and all five Steiner heuristics and a non-crossing constraint

Merging Using Random Weight Modification As with merging using only the heuristically constructed solutions without modifying edge weights, this method returned only valid solutions. What we could immediately see is that, as expected, this method produces results at least as good as the ones produced without using random weights, since it does not exclude any edges that the other method would exclude. In these experiments, it even reduced the gap by about 28%. It is also obvious that this method excludes less edges, which is also clearly reflected in the data. Running times are usually higher, since more solutions have to be generated (20 instead of 5) and as a direct consequence of less edges being excluded. Increasing the maximum weight modification and using more passes improves results, but also increases running times of the algorithm. For 32% of the instances, we even got the same result as the exact solver, mostly on small instances and instances from the set G0400-I1.

For TSPLib instances, this approach returned valid solutions for all instances. Because of 20 solutions being constructed using various Steiner tree heuristics, running time was substantially higher than that of the time-capped exact method in some cases. It was able to provide better or equal solutions as the exact method in 82% of the tested cases, and on average, the solution provided by this approach was 10.12% (std. dev. 12.00) better than the one provided by the exact method.

Instance set	% Gap	Cost ≤ Best known	Run time	% edges excluded
ClgSmall-I1	2.55 (11.00)	15/25	0.78 (0.17)	49.35 (2.91)
ClgSmall-I2	0.81 (1.32)	8/15	3.75 (0.85)	45.50 (4.25)
ClgSmall-I3	2.18 (1.32)	7/15	1.65 (0.45)	51.09 (2.78)
ClgMedium-I1	6.02 (6.47)	0/25	141.00 (81.43)	80.29 (2.06)
ClgMedium-I2	0.86 (2.20)	0/15	175.57 (65.38)	56.82 (15.17)
G0100-I1	1.54 (1.71)	5/15	3.93 (2.73)	58.44 (2.48)
G0100-I2	1.10 (1.98)	6/15	1.74 (0.35)	61.60 (2.52)
G0100-I3	4.09 (7.14)	11/20	0.56 (0.09)	42.91 (1.48)
G0400-I1	1.11 (1.96)	3/15	20.26 (5.00)	75.20 (2.00)
G0400-I2	0.95 (0.97)	1/15	50.46 (24.03)	69.64 (1.91)
All above	2.42 (2.54)	56/175	42.38 (72.86)	59.66 (12.60)
TSPLib	-10.12 (12.00)	27/33	1125 (2037)	47.63 (6.40)

Table 31: Details using merging of heuristic solutions constructed with AugSPe and all five Steiner heuristics, with weight modification and 4 construction passes

When the non-crossing constraint was enabled, we got the same results as an exact solver for 24.76% of the instances. As with the variant without random weight modification, we could not find a valid solution for one instance (G0100-I2-08).

Instance set	% Gap	Cost $\leq$ Best Known	Run time	% edges excluded
ClgSmall-I1	4.13 (11.68)	10/25	0.78 (0.90)	48.32 (3.02)
ClgSmall-I2	3.68 (4.56)	2/15	3.75 (2.13)	43.77 (3.20)
ClgSmall-I3	13.89 (10.15)	1/15	1.65 (1.33)	50.22 (4.02)
G0100-I1	5.22 (3.90)	0/15	3.93 (2.07)	55.77 (3.05)
G0100-I2	19.47 (55.52)	0/15	1-74 (1.40)	59.79 (3.08)
G0100-I3	8.73 (9.75)	5/20	0.55 (0.77)	39.87 (1.91)
All above	6.35 (21.95)	18/105	2.10 (1.90)	49.03 (7.25)

Table 32: Details using merging of heuristic solutions constructed with AugSPe and all five Steiner heuristics with a non crossing constraint and weight modification with 4 construction passes

7.2.6 Solution Cost vs. Running Time

This essential tradeoff in computation is also represented in our experiments. The better heuristics generally yielded results with a gap between 10% and 30%, usually within a lot less than a second in the case of smaller instances to a few seconds in the larger instances we tested. This gap could be reduced using purification and heuristic local improvement algorithms, as well as using generational and non-generational meta-heuristic approaches. The generational approaches, in our case, yielded little improvement, while the non-generational approaches terminated a lot faster and yielded comparable results. Solution merging, for the tested instances, proved to be the most obvious choice if a best approach was to be chosen and a low gap was essential, since it yielded the best results of all approaches we tested within a reasonable time frame.

7.3 Comparison of Results to Exact Approaches

As previously mentioned, our construction heuristics show a gap of 10% to 30% for most tested instances. Improvement and purification decreased that gap by a certain amount, with purification having the largest impact by far, leading to gaps usually between 7% and 15%. For instances with a non-crossing constraint, these gaps were usually higher. Construction and purification is, however, of course very fast when compared to exact approaches, with calculation times being around 0.05 seconds for smaller, and around 0.2 to 1.5 seconds for the larger tested instances on the tested hardware. For TSPLib instances, running times were higher (around 17 seconds on average), but still only required a only very little computation time when compared to the exact approach used for comparison. The tested TSPLib instances, modified using triangulation to fit the problem treated in this thesis, proved to be the hardest instances for both heuristic approaches, hybrid and exact methods when compared to randomly generated grid-based graphs or real-time instances.

The neighborhood-based approaches we described did little to improve solution quality, even with allowing the generational approaches to perform several thousand generations.

The smallest gap to the exact (and approximative) solutions was found in the solutions provided by the approach involving merging and solving of a set of heuristically constructed solutions. This, in some cases, encountered the optimal solution. The average gap of the experimental results (using several sets of solutions based on a problem graph, with each set having randomly modified edges), was roughly 3.4% on the tested instances without a non-crossing constraint, and about 6.4% when a non-crossing constraint was active. Running time was significantly lower than the running times of the same algorithm on the same computer, since they all were well below the maximum running time we allowed, which the exact solver alone usually exceeded for the medium and larger experimental instances.

For TSPLib instances, the exact solutions were afflicted with a rather large gap, due to the limited run-time the exact algorithm was allowed to use, which could be improved by over 10% using the merge-and-solve based approaches. The construction heuristics were usually a lot faster than the maximum running time of the exact algorithm, and were in many cases able to produce results with gaps around 15%, or in some cases even better solutions than the exact method was able to determine during its allowed run-time.

8 Summary

Increasing distribution of fiber-optic networks, and its fairly high cost of installation makes good planning of cable routes essential for maintaining affordability. One way this problem can be formalized is reducing it to two sub-problems: construction and augmentation of a Steiner tree.

A lot of work has been done in the field of the approximative, exact, heuristic and meta-heuristic construction of Steiner trees, vertex connectivity augmentation, the vertex biconnectivity problem and the SDNP-{0,1,2} problem. In this thesis, we first give an overview over previous work in these areas. We also present various algorithms for Steiner tree construction, augmentation of customer nodes requiring redundant connections to an infrastructure, algorithms for removing unnecessary redundant edges and a few other algorithms for decreasing solution cost in detail. These approaches are heavily based on previous work in both areas, especially our construction and augmentation heuristics. On some cases, these heuristics were able to determine an optimal solution.

We also present a series of neighborhood-based, meta-heuristic approaches. The neighborhoods we define are also based on previous work, but are modified in order not to destroy the validity of a solution that they are applied to, resulting in neighborhoods where the neighbors of a solution are very similar to the initial solution.

A further approach is presented which involves merging of multiple heuristic solutions into one set of edges, with the resulting sub-graph induced by that set of edges having considerably less edges than the original problem graph. This sub-graph was then fed into an exact solver.

The algorithms were implemented using C++, LEDA for data structures, CPLEX for a linear programming solving and EALib2 for meta-heuristic approaches. As an exact solver, a multi-commodity-flow based approach from [29] was used.

Experimental results using this set of implementations show that the heuristic methods described in this thesis are able to construct valid solutions for all instances without a non-crossing constraint, and for most instances with a non-crossing constraint. Sometimes the constructed solution was almost a third more expensive than an optimal solution determined using exact methods. However, for some of the smaller instances we could find an optimal solution using only our construction heuristics. This number could be increased using heuristic improvement and purification heuristics, which also yielded a significant improvement for many test instances.

Experiments involving the presented neighborhoods show these neighborhoods seem to have a significantly reduced degree of diversity when compared to the neighborhoods for Steiner tree construction that their intuitive concepts were based on. This seems to have considerable impact on their improvement capabilities. The experiments also show that the approaches involving the merging of multiple heuristic solutions into a new problem sub-graph lowered computation times by a large factor, and kept the gap to the best known solutions rather small.

9 Conclusion

Since our heuristic approaches were able to determine valid solutions for almost every instance we considered, and in some cases even an optimal solution, approaches involving the construction of a Steiner tree using multiple applications of minimum spanning tree algorithms could be evaluated and used to extend the set of construction heuristics with heuristics that are not shortest-path-based. These construction heuristics could then be coupled with the other improvement, optimization and merge-and-extract approaches presented in this thesis.

Since the neighborhoods we chose for Local Search and Simulated Annealing showed only small improvements of the original solution, an neighborhood similar to the ones described in [9] and [26] might be interesting. In contrast to the neighborhoods we chose, they construct an entirely new solution based on the removal or addition of key-nodes, leading to a solution with a possibly completely different structure. This would be possible by generating a modified problem instance and applying the heuristic solution construction algorithms provided here. Moreover, approaches based on genetic algorithms might prove successful. Since Variable Neighborhood Search and Variable Neighborhood Descent showed slightly better results in our experiments, further exploration into more radical neighborhoods might be fruitful, maybe using construction of an entirely new solution as well.

The experiments involving the approximative or exact solution of a merged set of heuristically generated solutions show that, on the instances we tested, the problem instance size could be decreased by half using heuristic methods, speeding up the solving process. The gap to the best known solutions was the smallest we encountered during the experiments described in this thesis. This causes us to consider this approach the most effective one we described in this thesis when applied to the instances we used for testing. More experiments of that kind could include, for example stronger weight modification or more passes. Also, using different weight modification strategies (for example involving penalization of edges that are already within the set of merged solutions) or different sources for solutions to merge (new construction heuristics etc.) might lead to further improvement of this technique.

Another feasible approach involving the exact solution of a sub-graph would be to destruct parts of a heuristically created solution, and then rebuild the destroyed area using exact methods. Developing effective strategies for selecting areas of the solution that are to be destructed are crucial for the success of such an approach.

References

[1]

[2] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* Princeton University Press, 2003.

[3] Peter Bachhiesl. The OPT- and the SST-problems for real world access network design – basic definitions and test instances. Working Report Nr. 01/2005, Carinthia Tech Institue, Department of Telematics and Network Engineering, Primoschgasse 8, 9020 Klagenfurt, Austria, October 2005.

[4] J.E. Beasley. An algorithm for the steiner problem in graphs. *Networks*, 14(1):147–159, 1984.

[5] T. Böhme, F. Göring, and J. Harant. Menger's theorem.

[6] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1990.

[9] M. Poggi de Aragao, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. Hybrid local search for the steiner problem in graphs. In *4th Metaheuristics International Conference*, 2001.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[11] T. Feo and M. Resende. Greedy randomized adaptive search procedures, 1995.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* A series of books in the mathematical sciences. W.H. Freeman and Company, New York, NY, 1979.

[13] P. C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *SIAM Journal on Applied Mathematics*, 10:305–313, 1962.

[14] F. Glover, J. P. Kelly, and M. Laguna. Genetic algorithms and tabu search: hybrids for optimization. *Comput. Oper. Res.*, 22(1):111–134, 1995.

[15] Algorithmic Solutions Software GmbH. Leda description. http://www.algorithmic-solutions.com, 2006.

[16] ILOG Inc. Cplex. http://www.ilog.com, 2006.

[17] Carinthia Tech Institute. Netquest - simulation and optimization of access networks. http://www.cti.ac.at/netquest/index.htm, 2006.

[18] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 1972.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[20] Jr. Kruskal, J. B. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[21] I. Ljubić and G. R. Raidl. A memetic algorithm for minimum-cost vertex-biconnectivity augmentation of graphs. *Journal of Heuristics*, 9(5):401–427, 2003.

[22] N. Mladenović and P. Hansen. Variable neighborhood search. *Comps. in Opns. Res.*, 24:1097–1100, 1997.

[23] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P 826, Pasadena, CA, 1989.

[24] T. Polzin. Algorithms for steiner problems in networks, dissertation, 2003.

[25] Ravi and Williamson. An approximation algorithm for minimum-cost vertex-connectivity problems. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1995.

[26] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid grasp with perturbations for the steiner problem in graphs. *INFORMS J. on Computing*, 14(3):228–246, 2002.

[27] G. Singh, S. Das, S. Gosavi, and S. Pujar. Ant colony algorithms for steiner trees: an application to routing in sensor networks, 2003.

[28] H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. 6:573–577, 1990.

[29] D. Wagner, G. R. Raidl, U. Pferschy, P. Mutzel, and P. Bachhiesl. A multi-commodity flow approach for the design of the last mile in real-world fiber optic networks. In *Operations Research Proceedings 2006, Karlsruhe, Germany, to appear.* Springer, 2006.

[30] Daniel Wagner. Eine generische bibliothek für metaheuristiken und ihre anwendung auf das quadratic assignment problem. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, August 2005. supervised by Günther Raidl.

List of Algorithms

List of Figures

List of Tables

Index