TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

D I P L O M A R B E I T

Solving Two Generalized Network Design Problems with Exact and Heuristic Methods

ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von

Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
und
Univ.-Ass. Dipl.-Ing. Bin Hu

durch

Markus Leitner
Matrikelnr. 0025315
Palffygasse 27/11
1170 Wien

Mai 2006 _____

Abstract

This thesis considers two NP hard generalized network design problems, where the nodes of a graph are partitioned into clusters and exactly one node from each cluster must be connected. The objective of both problems is to identify for a given graph a subgraph with minimal total edge costs and satisfying certain constraints.

The Generalized Minimum Spanning Tree (GMST) problem extends the classical Minimum Spanning Tree problem by seeking a connected cycle-free subgraph containing exactly one node from every cluster. This problem is solved by a Variable Neighborhood Search (VNS) approach which uses three different neighborhood types. The first one is focused on the nodes selected within a concrete solution, while the second one first selects the global edges between the clusters. Both are large in the sense that they contain exponentially many candidate solution, but efficient polynomial-time algorithms are used to identify best neighbors. The third neighborhood type uses Integer Linear Programming (ILP) to solve parts of the problem to provable optimality. Tests on Euclidean and random instances with up to 1280 nodes indicate especially on instances with many nodes per cluster significant advantages over previously published metaheuristic approaches.

Extending the classical Minimum Edge Biconnected Network problem, goal of the Generalized Minimum Edge Biconnected Network (GMEBCN) problem is to obtain a subgraph connecting exactly one node from each cluster and containing no bridges. Two different Variable Neighborhood Search (VNS) approaches using four different neighborhood types, are presented for this problem. The first one focuses on optimizing the used nodes, while the second one puts more emphasis on the arrangement of them. Two different versions exists for both neighborhoods. The simpler ones operate in a straightforward way on the nodes of the solution-graph while the more sophisticated versions consider the so-called "reduced graph". Using this significant smaller graph, it is easy to determine the best used nodes for the majority of clusters in an optimal way. The third neighborhood type optimizes a solution by first adding a new edge and then removing as many unnecessary edges as possible. Finally the last neighborhood optimizes both the used nodes as well as the edges. It is based on changing the used nodes within exactly one cluster and removing all incident edges which divides the graph into several edge-biconnected components. Afterwards the solution is heuristically augmented until the edge biconnectivity property holds again. Comparing these two approaches on Euclidean and random instances with up to 1280 nodes indicate that the second approach, using the more sophisticated neighborhoods and having higher computational complexity, is able to outperform the simpler, but much faster one significantly with respect to solution quality.

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit zwei NP schweren, generalisierten Netzwerkdesignproblemen, deren Knotenmenge in so genannte Cluster unterteilt ist. Eine zulässige Lösung eines solchen Problems enthält jeweils genau einen Knoten aus jedem dieser Cluster, während die optimale Lösung jene mit der geringsten Summe an Kantenkosten ist.

Das Generalisierte Minimale Spannbaum (GMST) Problem, bei dem es sich um eine Erweiterung des klassischen minimalen Spannbaum Problems handelt, sucht einen möglichst kostengünstigen, kreisfreien Teilgraph, der eben aus jedem Cluster genau einen Knoten enthält und alle gewählten Knoten verbindet. Dieses Problem wird mit Variabler Nachbarschaftssuche (VNS) mittels drei verschiedener Nachbarschaftstypen gelöst. Die erste Nachbarschaft stellt die zu wählenden Knoten in den Mittelpunkt, während die zweite zunächst die globalen Kanten zwischen den Clustern auswählt. Trotz der Tatsache, dass der Suchraum für beide Nachbarschaften exponentiell groß ist, kann die jeweils beste Lösung effizient in polynomieller Zeit ermittelt werden. Der dritte Nachbarschaftstyp verwendet ganzzahlige lineare Programmierung (ILP) um Teile des Problems exakt zu lösen. Tests, die sowohl für den euklidischen als auch den nicht euklidischen Fall durchgeführt wurden, zeigen, dass dieser Ansatz speziell dann, wenn die Anzahl der Knoten pro Cluster groß ist, bessere Ergebnisse als bisherige heuristische Ansätze liefert.

Beim zweiten Problem, dem Generalisierten Minimalen Kanten-Zweizusammenhaengenden Netzwerk (GMEBCN) Problem, wird ein möglichst kostengünstiger, bezüglich der Kanten zweifach zusammenhängender Teilgraph gesucht, der wiederum aus jedem Cluster genau einen Knoten enthält. Zwei verschiedene Ansätze, basierend auf VNS mit vier Nachbarschaftstypen, werden für dieses Problem gegenübergestellt. Die erste Nachbarschaft optimiert die ausgewählten Knoten, während die zweite deren Anordnung betrachtet. Für diese beiden existiert jeweils noch eine aufwändigere Version, welche auf einem reduzierten Graph operiert und die gewählten Knoten für die meisten Cluster exakt berechnet. Die dritte Nachbarschaft erweitert eine Lösung zunächst durch hinzufügen einer neuen Kante und versucht anschließend, so viele Kanten wie möglich zu entfernen, ohne den zweifachen Zusammenhang zu zerstören, während die letzte Nachbarschaft sowohl Knoten als auch Kanten gleichzeitig optimiert. Dazu entfernt sie zunächst einen Knoten mit allen inzidenten Kanten aus der Lösung und wählt stattdessen einen anderen Knoten aus dem gleichen Cluster. Anschließend werden heuristisch solange möglichst kostengünstige Kanten hinzugefügt, bis der geforderte zweifache Zusammenhang wieder gegeben ist. Tests zeigen, dass die Ergebnisse für jenen Ansatz, der die komplizierteren, aber natürlich auch langsameren Nachbarschaftsstrukturen verwendet, eindeutig besser sind als jene des einfacheren und schnelleren Ansatzes.

Acknowledgments

First of all I want to thank my advisors Prof. Günther Raidl and Bin Hu for supervising my master thesis, providing so many interesting ideas and being a great help during the last months.

My special thanks also belong to my family, especially to my parents for always believing in me and for all their support during the last years.
Additionally I want to say thank you to all my friends for so many interesting discussions and for all the fun we had together.

Last but not least I want to thank Romana for the wonderful time we had together, as well as for all her understanding and her outstanding support during the last years.

Contents

List of Tables

List of Figures

1. Introduction

This thesis is located in the area of combinatorial optimization, focusing on NP hard generalizations of two classical network design problems that occur in real world where multiple local area networks are interconnected by a backbone network.
Depending on the demands of such a network, the underlying problem can either be formulated as the Generalized Minimum Spanning Tree (GMST) problem or the even harder Generalized Minimum Edge Biconnected Network (GMEBCN) problem.

The **Generalized Minimum Spanning Tree (GMST) problem** is an extension of the classical Minimum Spanning Tree (MST) problem and is defined as follows.
Consider an undirected weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. The node set V is partitioned into r pairwise disjunct clusters $V_1, V_2, \ldots, V_r$ containing $d_1, d_2, \ldots, d_r$ nodes, respectively.
A spanning tree of a graph is a cycle-free subgraph connecting all nodes. A solution to the GMST problem defined on G is a graph $S = \langle P, T \rangle$ with $P = \{p_1, p_2, \ldots, p_r\} \subseteq V$ containing exactly one node from each cluster, i. e. $p_i \in V_i$ for all $i = 1, \ldots, r$, and $T \subseteq P \times P \subseteq E$ being a tree spanning the nodes P, see Figure 1.1. The costs of such a tree are its total edge costs, i. e. $C(T) = \sum_{(u,v)\in T} c(u, v)$, and the objective is to identify a solution with minimum costs.

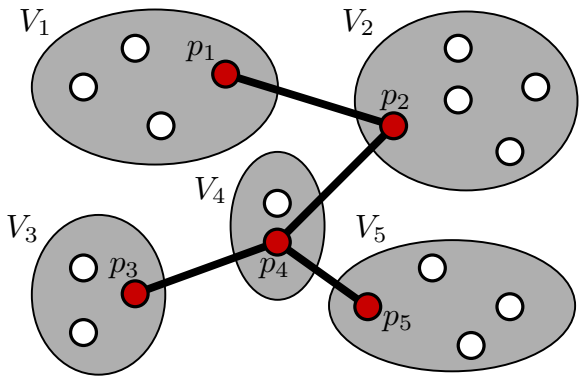


Fig. 1.1: Example for a GMST solution.

In case each cluster contains only one node, i. e. $d_i = 1$ for all $i = 1, \ldots, r$, the problem reduces to the simple MST problem, which can be efficiently solved in polynomial time. In general, however, here the GMST problem is NP hard [28].
As mentioned before there are several real world applications of the GMST problem, e. g. in the design of backbones in large communication networks. Devices belonging to the same local area network can be seen as a cluster, and the global network connects one device per local network. For a more detailed overview on the GMST problem, see [28, 5, 30].

A variant to the GMST problem is the less restrictive At-Least GMST (L-GMST) problem where more than one node is allowed to be connected from each cluster [21, 3]. However, this thesis concentrates on the GMST problem whose solutions contain exactly one node for each cluster.

Similar to Generalized Minimum Spanning Tree Problem, the **Generalized Minimum Edge Biconnected Network (GMEBCN) problem** is an extension of the classical Minimum Edge Biconnected Network Problem (MEBCN).
The following definition can be derived directly from the GMST by requiring an edge biconnected subgraph instead of a spanning tree.
Consider a weighted complete graph $G = \langle V, E, c\rangle$ with node set V, edge set E and edge cost function $c : E \to \mathbb{R}^+$. The node set V is partitioned into r pairwise disjunct clusters $V_1, V_2, \ldots, V_r$ containing $d_1, d_2, \ldots, d_r$ nodes, respectively.
An edge biconnected network of a graph is a subgraph connecting all nodes and containing no bridges. Therefore a solution to the GMEBCN problem defined on G is a subgraph $S = \langle P, F\rangle$ with $P = \{p_1, p_2, \ldots, p_r\} \subseteq V$ containing exactly one node from each cluster, i. e. $p_i \in V_i$ for all $1 \leq i \leq r$, and $F \subseteq P \times P \subseteq E$ being an edge biconnected graph (i. e. does not contain any bridges), see Figure 1.2.
The costs of such a subgraph are its total edge costs, i. e. $C(F) = \sum_{(u,v)\in F} c(u, v)$, and the objective is to identify a solution with minimum costs.

Another important consideration for the GMEBCN problem will be the concept of an edge-minimal edge-biconnected graph which is defined as a graph containing no *redundant edges*. A redundant edge is an edge whose removal does not violate the edge-biconnectivity property of the graph. Obviously, an optimal solution to the GMEBCN problem is always edge-minimal.

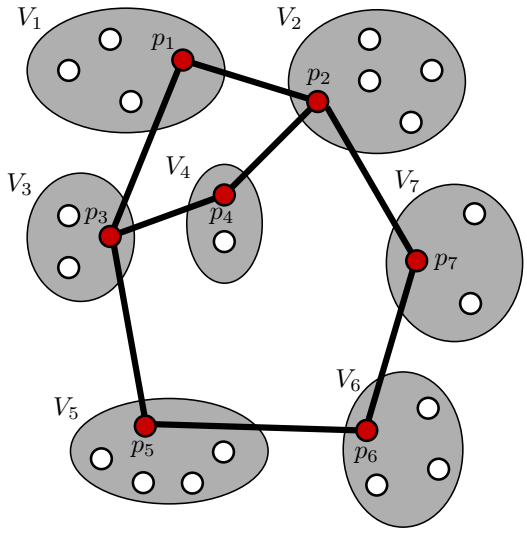


Fig. 1.2: Example for a GMEBCN solution.

The GMEBCN problem is NP hard even for the case each cluster contains only one node,

i. e. $d_i = 1$ for all $1 \leq i \leq r$, which is the classical minimum edge biconnected network (MEBCN) problem.

As for the GMST problem, real world application of the GMEBCN problem occur for example in the design of backbones in large communication networks as it makes sense to interconnect local area networks by a global network that can survive a single link outage though of edge redundancy [5]. In that way all real world applications to the GMST problem can be adapted to the GMEBCN problem whenever reliability is relevant.

This thesis attacks both problems using heuristic methods, by means of Variable Neighborhood Search (VNS) as well as combined with Integer Linear Programming (ILP).

1.1 Guide to the Thesis

Chapter 2 starts with introducing the necessary theoretical fundamentals, terms, definitions and methods used in this thesis before proceeding with a summary of the classical versions of the two considered network design problems. Afterwards, Chapter 3 presents previous and related research done so far.
Chapter 4 gives the details of the hybrid VNS approach developed for the GMST while Chapter 5 discusses its experimental results in comparison to previous approaches as well as the instance types used for testing. Previous versions of all chapters regarding the GMST problems have been published in [18, 19].
Chapter 6 analysis if and how ideas similar to those developed for the GMST problem can be utilized for the GMEBCN problem and presents two exact formulations for the GMEBCN problem. Afterwards, Chapter 7 presents the details of two different VNS approaches for the GMEBCN problem, while Chapter 8 discusses the experimental results gained for these approaches.
Finally Chapter 9 presents some implementation details and is followed by a summary and outlook for possible further work in Chapter 10.

2. Preliminaries

This chapter focuses on presenting the methods used for solving the problems considered in this thesis as well as defining some general terms and notations. Next to that, a short summary of the the classical versions of the both problems and a short introduction to generalized network design problems will be given.

2.1 Graph Theory

Unless explicitly mentioned this thesis considers only simple (i. e. no parallel edges or self loops), undirected graphs $G = \langle V, E, c \rangle$ with edge costs $c : E \to \mathbb{R}$.
Especially for the Generalized Minimum Edge Biconnected Network Problem the following basic terms need to be introduced, whichs definitions are mostly taken from the book of Diestel [2].

Definition 1: For any subset $W \subset V$ of the nodes of a graph $G = \langle V, E \rangle$, the edge set $\delta(W) = \{(i, j) \in E | i \in W, j \in V \setminus W\}$ is called the *cut set* of W in G.

Definition 2: A graph G is called *k-connected* if it cannot be separated by removing less then k vertices and called *k-edge-connected* if it cannot be seperated by removing less than k edges. More formally k-(edge)-connectivity can be defined in the following way.
A graph $G = \langle V, E \rangle$ is called *k-connected* (for $k \in \mathbb{N}$) if $|V| > k$ and $G \setminus X$ is connected for every set $X \subseteq V$ with $|X| < k$. The greatest integer k such that G is k-connected is called the *connectivity* $\kappa(G)$ of G.
Similar a graph $G = \langle V, E \rangle$ is called *k-edge-connected* (for $k \in \mathbb{N}$) if $|V| > 1$ and $G \setminus F$ is connected for every set $F \subseteq E$ with $|F| < k$. The greatest integer k such that G is k-edge-connected is called the *edge connectivity* $\lambda(G)$ of G.

In case of edge-connectivity, the definition given above is equal to requiring k edge disjoint paths between any two nodes, which obviously leads to a k edge-connected graph too. If $k = 2$ it is common to speak of *(edge)-biconnected* graphs.
Obviously k-connectivity implies k-edge-connectivity while the opposite is not true in general.

Definition 3: An edge that separates its ends is called a *bridge*. Thus, the bridges in a graph are precisely those edges that do not lie on any circle.

2.2 Metaheuristics

Metaheuristics for solving hard combinatorial optimization problems (COPs) are typically divided into two groups, local search based metaheuristics (e. g. Variable Neighborhood Search) and population based metaheuristics (e. g. evolutionary algorithms). The latter will not be

considered here. Before moving to basic local search, some terms need to be defined [1]. As this thesis does consider minimization problems only, *minimum* and *optimum* refer to the same term.

Definition 4: A *neighborhood structure* N is a function $N : X \rightarrow 2^X$, assigning each possible solution $x \in X$ a set of neighbors $N(x) \subseteq X$.
$N(x)$ is called the neighborhood of x.

Definition 5: A *local minimum (optimum)* with respect to a neighborhood $N(x)$ of a minimizing combinatorial optimization problem is a solution x, such that $\forall x' \in N(x) : f(x) \leq f(x')$.

Definition 6: A *global minimum (optimum)* of a minimizing combinatorial optimization problem is a solution x^*, such that $f(x^*) \leq f(x),\ \forall x \in X$. Therefore a global optimum is a local optimum for all neighborhood structures N (see Figure 2.1).

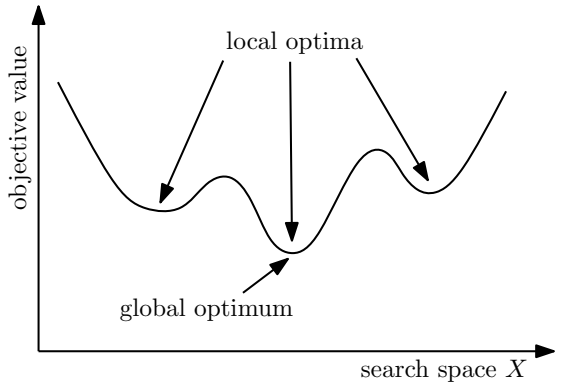


Fig. 2.1: Local and Global Optimas.

2.2.1 Basic Local Search

Basic Local Search [1, 29], which is called hill climbing or iterative improvement is a simple optimization method based on trial and error.
A typical Basic Local Search processes is shown in Algorithm 1. First of all an initial feasible solution x is computed. Afterwards, the process of selecting a new solution x' from the neighborhood $N(x)$ of x is repeated until some termination condition is met. Usually Basic Local Search stops after a predefined time or if no better solution within the neighborhood can be found.
Depending on the way the neighborhood is searched it is common to distinguish between *best improvement, next improvement, and random improvement* local search. Best Improvement (Steepest Descent) Local Search does an exhaustive search on $N(x)$ and selects the best solution $x' \in N(x)$ while Next (or First) Improvement Local Search accepts the first solution $x' \in N(x)$ leading to an improvement of the current solution (i. e. $f(x') < f(x)$). In opposite to those Random Improvement Local Search randomly selects a solution $x' \in N(x)$, instead of using a deterministic predefined order.

Algorithm 1: Basic Local Search

```
x ← initial solution
repeat
    select x' ∈ N(x)
    if f(x') ≤ f(x) then
        x ← x'
until termination condition is met
```

The main weakness of Basic Local Search is the lack of a possibility to overcome a local optima in order to reach better solutions or even the global optimum of a problem. Several extensions to Basic Local Search, such as Iterated Local Search [25, 26], Simulated Annealing [24], or Tabu Search [12] have been proposed to overcome this weakness. A good survey on such metaheuristics can be found in [1]. However we concentrate on the method used in this thesis, Variable Neighborhood Search.

2.2.2 Variable Neighborhood Search

Variable Neighborhood Search (VNS) [14, 15] is a metaheuristic which exploits systematically the idea of neighborhood change to head for local optima, but as well to escape from these valleys in oder to reach under-explored areas to find even better results.
VNS is based on the following three observations [15].

- A local optimum with respect to one neighborhood structure is not necessary a local optimum for another.
- A global optimum is a local optimum with respect to all possible neighborhood structures.
- For many problems local optima with respect to one or several neighborhoods are relatively close to each other.

The basic Variable Neighborhood Descent (VND) scheme (see Algorithm 2) which is based on the first observation, enhances the basic local search scheme by taking more neighborhood structures in to concept.

Algorithm 2: Variable Neighborhood Descent

```
repeat
    l = 1
    while l ≤ l_max do
        find the best neighbor x' ∈ N_l(x)
        if f(x') < f(x) then
            x = x'
            l = 1
        else
            l = l + 1
until no improvement is obtained
```

The basic idea of Variable Neighborhood Search (VNS) is to enhance VND by using a *shake* function to jump to a random new solution within the neighborhood (with increasing size) of the current solution. This procedure enables the possibility to escape from local optima and valleys containing them. Several VNS schemes, such as Reduced VNS, Basic VNS, and General VNS [15] exists. Among these, the general VNS scheme is used through all algorithms in this thesis, and is presented in Algorithm 3.

Algorithm 3: General Variable Neighborhood Search

```
repeat
    k = 1
    while k ≤ k_max do
        generate a point x' from the k^th neighborhood N_k(x) of x // shaking
        // local search by VND
        l = 1
        while l < l_max do
            find the best neighbor x'' ∈ N_l(x')
            if f(x'') < f(x') then
                x' = x''
                l = 1
            else
                l = l + 1
        if f(x') < f(x) then
            new so far best local optimum
            x = x'
            k = 1
        else
            k = k + 1
until termination condition is met
```

2.3 Integer Linear Programs

Combinatorial Optimization Problems (COP) can often be formulated in a mathematical way as Integer Linear Programs (ILP). An ILP is defined by an objective function (2.1), linear constraints (2.2) and constraints forcing the variables to be integral (2.4). The standard form of an ILP is defined as follows.

$$\begin{aligned} &minimize && cx && (2.1)\\ &subject\ to && Ax \geq b && (2.2)\\ & && x \geq 0 && (2.3)\\ & && x\ integral && (2.4) \end{aligned}$$

As solving ILPs is NP hard in general, algorithms based on Branch and Bound, Branch and Cut, or sophisticated ILP solvers such as CPLEX, which is a state-of-the-art Mathematical

Programming Optimizer from ILOG [1], are used to handle them.

2.4 The Minimum Spanning Tree Problem

The Minimum Spanning Tree (MST) problem is one of the best known network design problems and is defined as follows.
Consider a weighted graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. The optimal solution to the MST problem is a cycle free, connected subgraph (a tree) $S = \langle V, T \rangle$ connecting all nodes of G with minimal edge costs $C(T) = \sum_{(u,v)\in T} c(u, v)$. Minimum Spanning Trees are usually computed using one of the classical greedy algorithms of Kruskal (see Algorithm 4) or Prim (Algorithm 5) which always compute the optimal solution. For dense graphs the algorithm of Prim, which has computational complexity $O(|V|^2)$, is faster, while Kruskals' algorithm, which has complexity $O(|V| + |E| \log |E|)$ is the better choice for sparse graphs.

Algorithm 4: Kruskal-MST($G = \langle V, E \rangle$)

```
S = ∅
T = ∅
i = 1
sort edges with increasing costs, i. e. c(e_1) ≤ c(e_2) ≤ ··· ≤ c(e_n)
for v ∈ V do
    add v to S as a single set
while S has more than one set do
    if u_i and v_i do not belong to the same set in S then
        T = T ∪ {(u_i, v_i)}
        union the sets containing u_i and v_i
    i++
```

Algorithm 5: Prim-MST($G = \langle V, E \rangle$)

```
Select an arbitrary node s ∈ V
C = {s}
T = ∅
while |C| ≠ |V| do
    Select an edge e = (u, v) ∈ E, u ∈ C, v ∈ V \ C with minimal weight c(e)
    T = T ∪ {e}
    C = C ∪ {v}
```

2.5 The Minimum Edge Biconnected Network Problem

The Minimum Edge Biconnected Network (MEBCN) problem which is also called the 2-Edge connected network (2-ECN) problem or the bridge connected network problem appears in the

[1] http://www.ilog.com/products/cplex/

area of survivable network design and is defined as follows.
Consider a weighted graph $G = \langle V, E, c\rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. The optimal solution to the MEBCN problem is a subgraph $S = \langle V, F\rangle$ connecting all nodes and containing no bridges, with minimal edge costs, i. e. $C(T) = \sum_{(u,v)\in F} c(u, v)$.
By a reduction of the MEBCN problem to the Hamilton cycle problem (a graph has a Hamilton cycle, if and only if it has an edge biconnected spanning subgraph with $|V|$ edges) it can be seen easily that the MEBCN problem is NP hard [23]. Therefore an optimal solution to the MEBCN problem cannot be computed within reasonable time, unless P=NP.
Algorithm 6 and 7 show how to efficiently determine whether a graph $G = \langle V, E\rangle$ is edge biconnected or not, using a depth first search algorithm [34, 10], which has complexity $O(|V|+|E|)$. Afterwards, the nodes of the edge biconnected components of G can be identified by having the same label I, i. e. $I[v] = I[v']$ if and only if v and v' belong to the same edge biconnected component of G. The number of edge biconnected components of a graph is equal to the value of $c - |V|$ after applying Algorithm 7. If $c - |V| = 1$ we can conclude that the corresponding graph is edge biconnected.

Algorithm 6: Bridge($G = \langle V, E\rangle$)

```
empty stacks S and B
for v ∈ V do
    I[v] = 0
c = |V|
forall v ∈ V do
    if I[v] = 0 then
        DFSBridges(v, 0) //see Algorithm 7
```

Algorithm 7: DFSBridges(v, u)

```
S.push(v)
I[v] = |S|
B.push(I[v])
forall edges (v, w) ∈ E do
    if I[w] = 0 then
        DFSBridges(w, v)
    else
        if w ≠ u then
            while I[w] < top element of B do
                B.pop()
if I[v] = top element of B then
    B.pop()
    c++
    while I[v] ≤ |S| do
        I[top element of S] = c
        S.pop()
```

2.6 Generalization of Network Design Problems

Several Network Design Problems can be generalized by partitioning the node set V into clusters V_k, $k \in K$ [5]. Even if those clusters are not necessarily disjunct, the proposed algorithms in this thesis requires node disjunct clusters (i. e. $V_i \cap V_j = \emptyset,\ \forall i, j \in K, i \neq j$).
A common distinction between three different types (*exactly, at least, at most*) of such generalizations can be done with respect to the number of nodes that need to be in a feasible solution of the problem.
A feasible solution of an *exactly* generalized problem selects exactly one vertex from each cluster V_k ($\forall k = 1, \ldots, r$). Similar to that, the *at least* version requires the solution to contain at least one node from each cluster and the *at most* version requires the solution to contain at most one node from each cluster.

Typically the prefixes *E-, L-*, and *M-* are used to distinguish between the different generalization types of the same problem (i. e. L-GMST stands for the at least version, E-GMST for the exact version, and M-GMST for the at most version of the GMST problem). A missing prefix usually refers to the exact generalization of the network design problem. This notation will be used in this thesis too as only exact generalizations are considered.

3. Previous Work

3.1 Previous Work for the Generalized Minimum Spanning Tree Problem

Reich and Widmayer [33] were the first to study the Grouped Steiner Tree problem, which is a more general variant of the GMST problem. The actual GMST problem was introduced by Myung, Lee, and Tcha [28]. They proved that this problem is NP hard and provided four different ILP formulations. Feremans, Labbe, and Laporte [7] added another four formulations and did some in-depth investigation on all eight ILPs. They have shown that among these formulations, the "Undirected Cluster Subpacking" formulation does not only have the best linear relaxation, but is also the most compact one in the number of variables.

Pop [30] introduced the "Local-Global" formulation, which is also compact in the number of variables. It proved in particular to be more efficient in practice, especially in combination with a relaxation technique called the "Rooting Procedure". Instances with up to 240 nodes divided into 30 clusters or 160 nodes divided into 40 clusters could be solved to optimality. Furthermore, Pop utilized the idea of his ILP formulation in a Simulated Annealing approach in order to heuristically solve larger instances. His work also formed a basis for the design of the neighborhoods presented for the GMST problem in this thesis.
A more complex Branch-and-Cut algorithm which features new sophisticated cuts and detailed separation procedures has been recently presented by Feremans, Labbe, and Laporte [8]. Nevertheless, large instances can still not be solved to optimality in practical time.

Regarding approximation algorithms, Myung, Lee, and Tcha [28] have shown the inapproximability of the GMST problem in the sense that no approximation algorithm with constant quality guarantee can exist unless P = NP. However, there are better results for some special cases of the problem. Pop, Still, and Kern [31] described an approximation algorithm for the case when the cluster size is constant. If $d_{\max}$ is the maximal number of nodes per cluster, the total costs of the resulting solution are at most $2 \cdot d_{\max}$ times the optimal solution value.
Feremans and Grigoriev [6] provided a Polynomial Time Approximation Scheme (PTAS) for the GMST problem in case of grid clustering, i. e. the cluster assignment is based on a $k \times l,\ k \leq l$ integer grid laid on the nodes in the Euclidean plane and all nodes within the same grid cell belong to the same cluster. The authors first presented a dynamic programming algorithm which can solve the problem to optimality in $O(l \cdot d_{\max}^{6k} \cdot 2^{34k^2} \cdot k^2)$ time. Extending this concept by dividing the grid into $O(\varepsilon)$ slices, each containing $O(1/\varepsilon)$ rows to be solved with dynamic programming and then connected together, a PTAS is obtained that provides quality bounds of $O(1+\varepsilon)$.

To approach more general and larger GMST instances, various metaheuristics have

been suggested. Ghosh [11] implemented and compared a Tabu Search with recency based memory (TS), a Tabu Search with recency and frequency based memory (TS2), a Variable Neighborhood Descent Search, a Reduced VNS, a VNS with Steepest Descent and a Variable Neighborhood Decomposition Search (VNDS). For all the VNS approaches, he used 1-swap and 2-swap neighborhoods, which exchange the used nodes within clusters. Comparing these approaches on instances ranging from 100 to 400 nodes partitioned into up to 100 clusters, Ghosh concluded that TS2 and VNDS perform best on average.
Golden, Raghaven, and Stanojevic [13] presented a lower bounding procedure, which basically computes the MST on a derived complete graph $G' = \langle V', E', c' \rangle$ with $v'_i \in V'$ representing the clusters V_i in G. The edge costs $c'(v'_i, v'_j)$ are chosen as $\min\{c(a,b) \mid (a,b) \in E \wedge a \in V_i \wedge b \in V_j\}$. Furthermore, the authors introduced upper bounding procedures by adapting Kruskal's, Prim's, and Sollin's algorithm for the classical MST problem, as well as providing a Genetic Algorithm (GA). The GA encodes candidate solutions by the used nodes of each cluster. Solution reproduction is done by a simple one-point crossover and a separation operator which generates two offsprings from one solution. Local search based on exchanging the used nodes is also applied in addition to a simple mutation procedure which changes the used node of a random cluster.
Hu, Leitner, and Raidl [18] proposed a VNS approach which combines two different neighborhood types using different solution representations and working in complementary directions which is an earlier version of the GMST part of this thesis. Finally the same authors proposed a significant extension by adding one more neighborhood type based on solving parts of the problems via ILP and including more extensive experimental comparisons in [19] which reflects the GMST part of this thesis.

Concerning the L-GMST, Dror, Haouari, and Chaouachi [3] developed two ILPs, four simple construction heuristics and a basic genetic algorithm. In [17], the same authors presented strategies for obtaining upper bounds by means of a sophisticated construction heuristic and a complex genetic algorithm. They also discussed three alternative ILP formulations which provide lower bounds after relaxing them in a Lagrangian fashion. Based on these bounds, Haouari and Chaouachi [16] developed a branch-and-bound algorithm which could solve some instances with up to 250 nodes and 1000 edges to optimality. Unfortunately, many of the more specific concepts behind these algorithms for the L-GMST cannot be applied to the GMST as considered here.

Duin, Volgenanta, and Voß [4] suggested to solve the more general Group Steiner problem (GSP) by transforming it into a classical Steiner tree Problem in Graphs (SPG) and applying effective algorithms for this more common problem. The GSP considers a weighted graph $G = \langle V, E, c \rangle$ with not necessarily disjunct node sets $V_k \subset V$ for indices $k \in K$. In contrast to the GMST problem, there might also exist nodes $v \in V$ with $v \notin \bigcup_{k \in K} V_k$. The objective is to find a minimum cost tree spanning at least one node of each set V_k. The transformation to SPG constructs a graph G' by adding for each cluster V_k an artificial node v_k and edges (v_k, u), $\forall u \in V_k$ with "large" costs. The Steiner tree on G' connecting all terminal nodes v_k corresponds to a solution for the GSP on G after removing the artificial nodes and edges. However, when applying this transformation to the GMST problem, the resulting clusters can contain more than one used node. Therefore, this method can be used to solve the L-GMST problem, but is not directly applicable here. It would be necessary to consider additional constraints preventing the application of standard algorithms for SPG.

3.2 Previous Work for the Generalized Minimum Edge Biconnected Network Problem

Not much research has been done on the GMEBCN problem so far. Feremans [5] described the problem and several real world application to it while Huygens [20] studied the polytypes of both the exact and the at least formulation of the GMEBCN problem.
Other than that, no research has been done on this problem.

4. Variable Neighborhood Search for the GMST Problem

In this chapter, the new VNS approach for the GMST problem will be described in detail. First, two constructive heuristics to produce initial solutions are considered. Chapter 4.2 describes the used neighborhoods and the search techniques applied to them, while Chapter 4.3 provides the details of their arrangement within the Variable Neighborhood Descent (VND). Finally, Chapter 4.4 describes the shaking procedure, and Chapter 4.5 explains a memory function to substantially reduce the number of evaluations for the same solutions.

4.1 Initialization

To compute an initial feasible solution for the GMST problem, either a specialized heuristic or an adaption of a standard algorithm for the classical MST problem can be used. Golden, Raghavan, and Stanojevic [13] give a comparison between three simple and three improved adaptions of Kruskal's, Prim's and Sollin's MST algorithms for the GMST problem. While all three improved adaptions produce comparable results, the variant based on Sollin's algorithm in general has the highest computational effort. Therefore the improved version based on Kruskal's MST heuristic is adopted for initialization and will be compare to the rather simple minimum distance heuristic which was also used by Ghosh [11] to generate initial solutions.

4.1.1 Minimum Distance Heuristic

The Minimum Distance Heuristic (MDH) for computing a feasible initial solution for the GMST problem is shown in Algorithm 8. For each cluster, the node with the lowest sum of edge costs to all nodes in other clusters is used and a MST is calculated on these nodes. Using Krukal's algorithm for computing the MST, the complexity of MDH is $O(|V|^2 + r^2 \log r^2)$.

Algorithm 8: Minimum distance heuristic

for $i = 1, \ldots, r$ **do**
  choose $p_i \in V_i$ with minimal $\sum_{v \in V \setminus V_i} c(p_i, v)$ as the used node
determine MST T on the used nodes $P = \{p_1, \ldots, p_r\}$
return solution $S = \langle P, T \rangle$

4.1.2 Improved Adaption of Kruskal's MST Heuristic

Creating a feasible solution for the GMST by an adaption of Kruskal's algorithm for the classical MST problem is rather straightforward [13]. The basic idea is to consider edges in increasing cost-order. An edge is added to the solution iff it does not introduce a cycle and does not connect a second node of any cluster. Obviously, this adaption does not change the time complexity of Kruskal's original algorithm, which is $O(|V| + |E| \log |E|)$.
By fixing an initial node to be in the resulting generalized spanning tree, different solutions can be obtained. The Improved Adaption of Kruskal's MST Heuristic (IKH), as it is called by Golden, Raghavan, and Stanojevic [13], is shown in Algorithm 9 and follows this idea by running the simple version $|V|$ times, once for each node to be initially fixed. Due to the fact that the sorting of edges needs to be done only once, the computational complexity is $O(|V|^2 + |E| \log |E|)$.

Algorithm 9: Improved Kruskal heuristic

```
for v ∈ |V| do
    fix v to be in the generalized spanning tree
    compute generalized spanning tree with the adaption of Kruskal's MST algorithm
return solution with minimal costs
```

4.2 Neighborhoods

The implemented VNS algorithm uses three types of neighborhoods. The first two are based on local search concepts of Ghosh [11] and Pop [30]. Ghosh represents solutions by the used nodes and defines neighborhoods on them. Optimal edges are derived for a given selection on nodes by determining a classical MST.
On the other hand, Pop approaches the GMST problem from the other side by representing a solution via its "global connections" – the pairs of clusters which are directly connected. The whole solution is obtained by a decoding function which identifies the best suited nodes and associated edges for the given global connections. The neighborhood of a solution contains all solutions obtained by replacing a global connection by another feasible one.
In the third neighborhood type reasonably small subgraphs of the whole instance are selected and solved independently to optimality via an ILP formulation from Pop [30]. These solved parts are then reconnected in a best possible way. The following chapters describe the three neighborhood types and the ways they search in detail.

4.2.1 Node Exchange Neighborhood

In this neighborhood, which was originally proposed by Ghosh [11], a solution is represented by the set of used nodes $P = \{p_1, \ldots, p_r\}$ where p_i is the node to be connected from each cluster V_i, $i = 1, \ldots, r$. Knowing these nodes, there are r^{r-2} possible spanning trees, but one with smallest costs can be efficiently derived by computing a classical MST on the subgraph of G induced by the chosen nodes.
The Node Exchange Neighborhood (NEN) of a solution P consists of all node vectors (and corresponding spanning trees) in which for precisely one cluster V_i the node p_i is replaced by a

different node p'_i of the same cluster. This neighborhood therefore consists of $\sum_{i=1}^{r}(|V_i|-1) = O(|V|)$ different node vectors representing in total $O(|V| \cdot r^{r-2})$ trees. Since a single MST can be computed in $O(r^2)$ time, e.g. by Prim's algorithm, a straight-forward generation and evaluation of the whole neighborhood in order to find the best neighboring solution can be accomplished in $O(|V| \cdot r^2)$ time.
Using an incremental evaluation scheme, the computational effort can be reduced significantly. The goal is to derive from a current minimum-cost tree S represented by P a new minimum-cost tree S' when node p_i is replaced by node p'_i. Removing p_i and all its incident edges from the initial tree S results in a graph consisting of $k \geq 1$ connected components $T_1, \ldots, T_k$ where usually $k \ll r$.
The new minimum-cost tree S' will definitely not contain new edges within each component $T_1, \ldots, T_k$, because they are connected in the cheapest way as they where optimal in S. New edges are only necessary between nodes of different components and/or p'_i. Furthermore, only the shortest edges connecting any pair of components must be considered. So, the edges of S' must be a subset of

- the edges of S after removing p_i and its incident edges,
- all edges (p'_i, p_j) with $j = 1, \ldots, r \wedge j \neq i$, and
- the shortest edges between any pair of the components $T_1, \ldots, T_k$.

To compute S', we therefore have to calculate the MST of a graph with $(r-k-1)+(r-1)+(k^2-k)/2 = O(r+k^2)$ edges only. Unfortunately, this does not change the worst case time complexity, because identifying the shortest edges between any pair of components may need $O(r^2)$ operations. However, in most practical cases it is substantially faster to compute these shortest edges and to apply Kruskal's MST algorithm on the resulting thin graph. Especially when replacing a leaf node of the initial tree S, results in only a single component plus the new node and the incremental evaluation's benefits are largest.

Exchanging More Than One Node

The above neighborhood can be easily generalized by simultaneously replacing $t \geq 2$ nodes. The computational complexity of a complete evaluation raises to $O(|V|^t \cdot r^2)$. While an incremental computation is still possible in a similar way as described above, the complete evaluation of the neighborhood becomes nevertheless impracticable for larger instances even when $t = 2$. Therefore a Restricted Two Nodes Exchange Neighborhood (RNEN2) is used in which only pairs of clusters that are adjacent in the current solution S are simultaneously considered. In this way, the time complexity for a complete evaluation is only $O(|V| \cdot r^2)$.
Nevertheless, RNEN2 is in practice still a relatively expensive neighborhood. Since its complete evaluation consumes too much time in case of large instances, its exploration is aborted after a certain time limit returning the so-far best neighbor instead of following a strict best-neighbor strategy.

4.2.2 Global Edge Exchange Neighborhood

Inspired by Pop's local-global ILP and his Simulated Annealing approach [30], the Global Edge Exchange Neighborhood (GEEN) is defined on a so-called "global graph". This graph

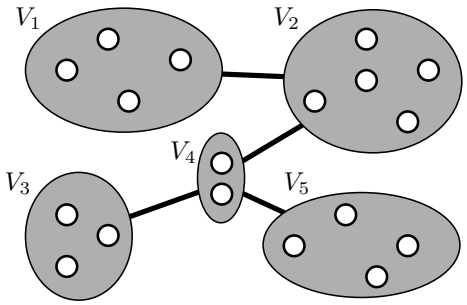


Fig. 4.1: A global graph G^g.

$G^g = \langle V^g, E^g \rangle$ consists of nodes corresponding to the clusters in G, i .e. $V^g = \{V_1, V_2, \ldots, V_r\}$, and edge set $E^g = V^g \times V^g$, see Figure 4.1.
Consider a spanning tree $S^g = \langle V^g, T^g \rangle$ with $T^g \subseteq E^g$ on this global graph. This tree represents the set of all feasible generalized spanning trees on G which contain for each edge $(V_a, V_b) \in T^g$ a corresponding edge $(u, v) \in E$ with $u \in V_a \wedge v \in V_b \wedge a \neq b$. Such a set of trees on G that a particular global spanning tree represents is in general exponentially large with respect to the number of nodes. However, dynamic programming can be used to efficiently determine a minimum cost solution from this set. In this process, the global spanning tree is rooted at an arbitrary cluster $V_{\text{root}} \in V^g$ and all edges are directed towards the leafs. This tree is traversed in a recursive depth-first way calculating for each cluster $V_k \in V^g$ and each node $v \in V_k$ the minimum costs for the subtree rooted in V_k when v is the node to be connected from V_k. These minimum costs of a subtree are determined by the following recursion:

$$C(T^g, V_k, v) = \begin{cases} 0 \text{ if } V_k \text{ is a leaf of the global spanning tree} \\ \sum\limits_{V_l \in Succ(V_k)} \min\limits_{u \in V_l} \{c(v,u) + C(T^g, V_l, u)\} \text{ else,} \end{cases}$$

where $Succ(V_k)$ denotes the set of all successors of V_k in T^g. After having determined the minimum costs for the whole tree, the nodes to be used can be easily derived in a top-down fashion by fixing for each cluster $V_k \in V^g$ the node $p_k \in V_k$ yielding the minimum costs. This dynamic programming algorithm requires in the worst case $O(|V|^2)$ time and is illustrated in Figure 4.2.
As Global Edge Exchange Neighborhood (GEEN) for a given global tree T^g, any feasible spanning tree differing from T^g by precisely one edge is considered. If the best neighbor is determined by evaluating all possibilities of exchanging a global edge and naively perform the whole dynamic programming for each global candidate tree, the time complexity is $O(|V|^2 \cdot r^2)$. For a more efficient evaluation of all neighbors, the whole dynamic programming is performed only once at the beginning, keep all costs $C(T^g, V_k, v), \forall k = 1, \ldots, r,\ v \in V_k$, and incrementally update our data for each considered move. According to the recursive definition of the dynamic programming approach, we only need to recalculate the values of a cluster V_i if it gets a new child, loses a child, or the costs of a successor change.
Moving to a solution in this neighborhood means to exchange a single global connection (V_a, V_b) by a different connection (V_c, V_d) so that the resulting graph remains a valid tree, see Figure 4.3. By removing (V_a, V_b), the subtree rooted at V_b is disconnected, hence V_a loses a

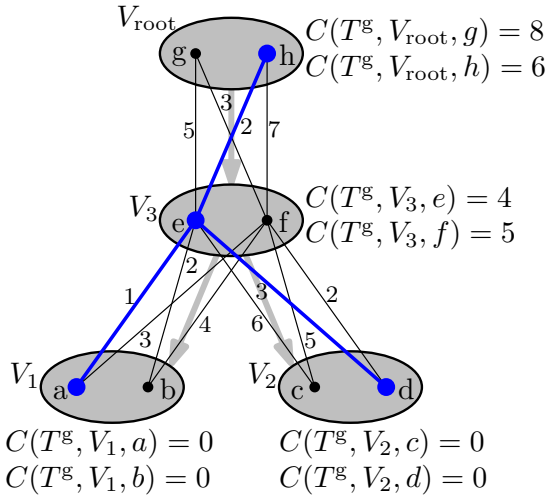


Fig. 4.2: Determining the minimum-cost values for each cluster and node. The tree's total minimum costs are $C(T^g, V_{\text{root}}, h) = 6$, and the finally selected nodes are printed bold.

child and V_a, as well as all its predecessors, must be updated. Before adding (V_c, V_d), one first needs to consider the isolated subtree. If $V_d \neq V_b$, the subtree at cluster V_d has to be re-rooted first. Thereby, the old root V_b loses a child. All other clusters which get new children or lose children are on the path from V_b up to V_d, and they must be reevaluated. Otherwise, if $V_d = V_b$, nothing changes within the subtree. When adding the connection (V_c, V_d), V_c gets a new successor and therefore must be updated together with all its predecessors on the path up to the root. In conclusion, whenever we replace a global connection (V_a, V_b) by (V_c, V_d), it is enough to update the costs of V_a, V_b, and all their predecessors on the ways up to the root of the new global tree.

If the tree is not degenerated, its height is $O(\log r)$, and one only need to update $O(\log r)$ clusters of G^g. Suppose each of them contains no more than $d_{\max}$ nodes and has at most $s_{\max}$ successors. The time complexity of updating the costs of a single cluster V_i is $O(d_{\max}^2 \cdot s_{\max})$ and the whole process needs time that is bounded by $O(d_{\max}^2 \cdot s_{\max} \cdot \log r)$. The incremental evaluation is therefore much faster than the complete evaluation with its time complexity of $O(|V|^2)$ as long as the trees are not degenerated. An additional improvement is to further avoid unnecessary update calculations by checking if an update actually changes costs of a cluster. If this is not the case, the update of the cluster's predecessors may be omitted as long as they are not affected in some other way.

To examine the whole neighborhood of a current solution by using the improved method described above, it is a good idea to choose a processing order that supports incremental evaluation as well as possible. Algorithm 10 shows how this is done in detail.

Removing an edge (V_i, V_j) splits the rooted tree into two components: K_1^g containing V_i and K_2^g containing V_j. The algorithm iterates through all clusters $V_k \in K_1^g$ and makes them root. Each of these clusters is iteratively connected to every cluster of K_2^g in the inner loop. The advantage of this calculation order is that none of the clusters in K_1^g except its root V_k has

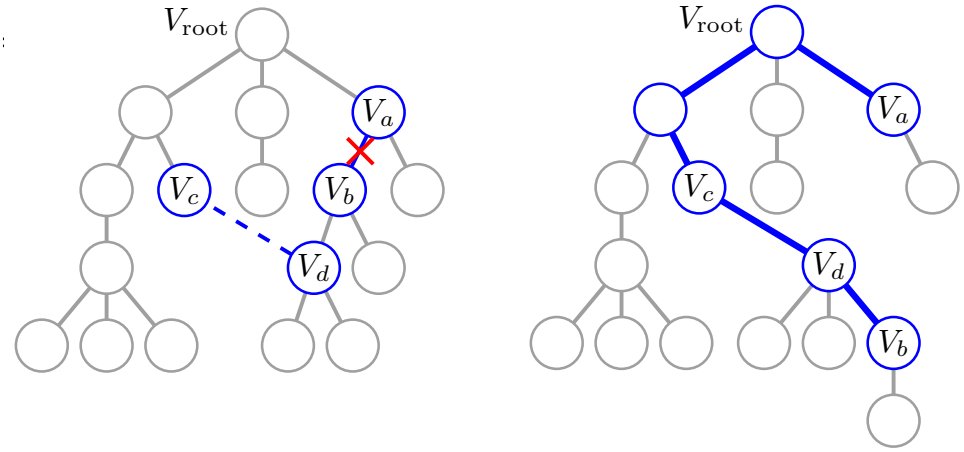


Fig. 4.3: After removing (V_a, V_b) and inserting (V_c, V_d), only the clusters on the paths from V_a to V_{root} and V_b to V_{root} must be reconsidered.

Algorithm 10: Global edge exchange (solution S)

```
forall global edges (V_i, V_j) ∈ T^g do
    remove (V_i, V_j)
    M_1 = preorder list of clusters in component K_1^g containing V_i
    M_2 = preorder list of clusters in component K_2^g containing V_j
    forall V_k ∈ M_1 do
        root K_1^g at V_k
        forall V_l ∈ M_2 do
            root K_2^g at V_l
            add (V_k, V_l)
            use incremental dynamic programming to determine the complete solution
                and the objective value
            if current solution better than best then
                save current solution as best
            remove (V_k, V_l)
restore and return best solution
```

to be updated more than once, because global edges are only added between the roots of K_1^g and K_2^g. Processing the clusters in preorder has another additional benefit: Typically, most of the time very few clusters have to be updated when re-rooting either K_1^g or K_2^g.

4.2.3 Global Subtree Optimization Neighborhood

This neighborhood follows the idea of selecting subproblems of reasonable size, solving them to provable optimality via ILP and merging the results to an overall solution as well as possible. The current GMST $S = \langle P, T \rangle$ is considered with its corresponding global spanning tree $S^g = \langle V^g, T^g \rangle$ defined on the global graph G^g as described in Chapter 4.2.2, i. e. for each

edge $(u, v) \in T$ with $u \in V_i \wedge v \in V_j$, there exists a global edge $(V_i, V_j) \in T^g$. After rooting S^g at an randomly chosen cluster V_{root}, a depth-first search to determine all subtrees $Q_1, \ldots, Q_k$ containing at least $N_{\min}$ and no more than $N_{\max}$ clusters is performed. Figure 4.4 shows an example for this subtree selection mechanism with $N_{\min} = 3$ and $N_{\max} = 4$ with its resulting subtrees $Q_1, \ldots, Q_4$ rooted at $V_1, \ldots, V_4$.

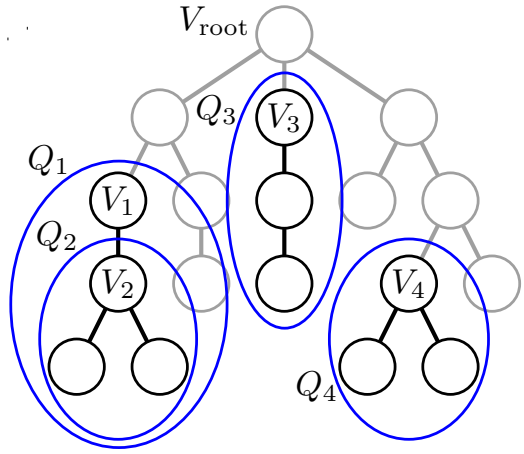


Fig. 4.4: Selection of subtrees to be optimized via ILP.

Moving to a solution in the Global Subtree Optimization Neighborhood (GSON) means to optimize one subtree Q_i as an independent GMST problem on the restricted graph induced by the clusters and nodes of Q_i. After solving this subproblem via ILP, the new subtree is reconnected to the remainder of the current overall tree in the best possible way. This can be achieved by considering all possible global edges that connect both components, which is similar as in GEEN. Algorithm 11 summarizes the evaluation of this neighborhood in pseudo-code.

Algorithm 11: Global exact subtree (solution S)

```
V_1, ..., V_k = roots of the subtrees Q_1, ..., Q_k containing at least N_min and
      no more than N_max clusters
forall i = 1, ..., k do
    remove the edge (parent of V_i, V_i) // separate subtree Q_i from S
    optimize Q_i via ILP
    reconnect Q_i to S in a best possible way // as GEEN reconnection mechanism
    if current solution better than best then
        save current solution as best
    restore initial solution
restore and return best solution
```

Whether or not to also consider contained subtrees as Q_2 in addition to Q_1 in Figure 4.4 was a difficult question while designing GSON. In general, if Q_i contains Q_j, it is not guaranteed that optimizing and reconnecting Q_i would always yield a better result than optimizing and reconnecting only the smaller subtree Q_j. This is possible in particular if the connection between Q_i's root cluster V_i and its predecessor is cheap, but Q_j fits better at a different

location. So it has been decided to include contained subtrees. If $N_{\min}$ and $N_{\max}$ are close, the additional computational effort caused by contained subtrees is relatively low.
The computational complexity of GSON is hard to determine due to the optimization procedure via ILP. If overlapping subtrees are not allowed, the number of subtrees to be considered is bounded below by 0 and above by $\lfloor \frac{r}{N_{\min}} \rfloor$. In our case, of allowing contained subtrees, the number of subtrees to be optimized can be as large as $\lfloor \frac{r}{N_{\max}} \cdot (N_{\max} - N_{\min} + 1) \rfloor$. Experiments showed, that choosing $N_{\min} = 5$ and $N_{\max} = 6$ produced in general the best results.

Local-Global ILP Formulation

In order to solve the subproblems on restricted sets of clusters to optimality, GSON utilizes Pop's local-global ILP formulation [30], which turned out to be more efficient than other formulations when using a general purpose ILP solver as CPLEX. This formulation is based on the fact that for each cluster $V_k,\ k = 1, \ldots, r$, there must be a directed global path from V_k to each other cluster $V_j,\ j \neq k$. For each k, these paths together form a directed tree rooted at V_k. The following binary variables are used.

$$y_{ij} = \begin{cases} 1 & \text{if cluster } V_i \text{ is connected to cluster } V_j \text{ in the global graph} \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i, j = 1, \ldots, r, \\ i \neq j \end{array}$$

$$\lambda_{kij} = \begin{cases} 1 & \text{if cluster } V_j \text{ is the parent of cluster } V_i \text{ when we root the} \\ & \text{tree at cluster } V_k \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i, j, k = 1, \ldots, r, \\ i \neq j,\ i \neq k \end{array}$$

$$x_e = \begin{cases} 1 & \text{if the edge } (u, v) \in E \text{ is included in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall e = (u, v) \in E$$

$$z_v = \begin{cases} 1 & \text{if the node } v \text{ is connected in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V$$

Pop could prove that if the binary incidence matrix y describes a spanning tree of the global graph, then the local solution is integral. Therefore it is sufficient to only force y to be integral in the following local-global ILP formulation.

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{4.1}$$

$$\text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, r \tag{4.2}$$

$$\sum_{e \in E} x_e = r - 1 \tag{4.3}$$

$$\sum_{u \in V_i, v \in V_j} x_{uv} = y_{ij} \qquad \forall i, j = 1, \ldots, r,\ i \neq j \tag{4.4}$$

$$\sum_{u \in V_i} x_{uv} \leq z_v \qquad \forall i = 1, \ldots, r,\ \forall v \in V \backslash V_i \tag{4.5}$$

$$y_{ij} = \lambda_{kij} + \lambda_{kji} \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k \tag{4.6}$$

$$\sum_{j \in \{1,\ldots,r\} \backslash \{i\}} \lambda_{kij} = 1 \qquad \forall i, k = 1, \ldots, r,\ i \neq k \tag{4.7}$$

$$\lambda_{kij} \geq 0 \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k \tag{4.8}$$

$$x_e,\ z_v \geq 0 \qquad \forall e = (i, j) \in E, \forall v \in V \tag{4.9}$$

$$y_{lr} \in \{0, 1\} \tag{4.10}$$

Constraints (4.2) guarantee that only one node is selected per cluster. Equality (4.3) forces the solution to contain exactly $r - 1$ edges while constraints (4.4) allow them only between nodes of clusters which are connected in the global graph. Inequalities (4.5) ensure that edges only connect nodes which are selected. Constraints (4.6) and (4.7) force the global graph to be a directed spanning tree: Equalities (4.6) ensure that a global edge (i, j) is selected iff i is the parent of j or j is the parent of i in the tree rooted at k. Constraints (4.7) guarantee that every cluster except the root k has exactly one parent.

Alternative Neighborhoods

When designing GSON several other possible neighborhoods that combine the concepts of global graph and exact ILP formulations have been considered. One possible adaption of GSON would be to solve all subtrees of a limited size exactly first and then iterate through a neighborhood structure which examines all variations of reconnecting these parts. As the number of possibilities for these reconnections is exponential, the exhaustive exploration turned out to be too expensive in practice.

Another idea for enhancing GSON is to select the clusters inducing a subproblem to be solved exactly not just from the subtrees connected via a single edge to the remaining tree, but from any connected subcomponent of limited size. However, the number of such components is in general too large for a complete enumeration. A practical possibility is to consider the restricted set formed by choosing each cluster as root exactly once and adding $N_{\max} - 1$ further clusters identified via breadth first search. The motivation behind is that it might be more meaningful to consider components of the current global tree where the clusters are close to each other. Unfortunately, the performed experiments indicated that the gain of this variant of GSON could not cover its high computational costs.

4.3 Arrangement of the Neighborhoods

Our approach uses the general VNS scheme with VND as local improvement [14, 15]. In VND, an alternation between NEN, GEEN, RNEN2, and GSON is performed in this order, see Algorithm 12. This sequence has been determined according to the computational complexity of evaluating the neighborhoods.

Algorithm 12: VND (solution $S = \langle P, T \rangle$)

```
l = 1
repeat
    switch l do
        case 1: // NEN
            for i = 1, ..., r do
                forall v ∈ V_i \ p_i do
                    change used node p_i of cluster V_i to v
                    recalculate the MST T by means of Kruskal's algorithm
                    if current solution better than best then
                        save current solution as best
            restore best solution
        case 2: // GEEN
            Global edge exchange (S) //see Algorithm 10
        case 3: // RNEN2
            forall clusters V_i and V_j adjacent in the current solution do
                forall v ∈ V_i \ p_i and u ∈ V_j \ p_j do
                    change used node p_i of cluster V_i to v
                    change used node p_j of cluster V_j to u
                    recalculate the MST T by means of Kruskal's algorithm
                    if current solution better than best then
                        save current solution as best
            restore best solution
        case 4: // GSON
            Global exact subtree (S) //see Algorithm 11
    if solution improved then
        l = 1
    else
        l = l + 1
until l > 4
```

4.4 Shaking

It turned out that using a shaking function which puts more emphasis on diversity yields good results for our approach, see Algorithm 13. This shaking process uses both, the NEN and the GEEN structures. For NEN, the number of random moves for shaking starts at three because a restricted 2-Opt NEN improvement is already included in the VND; thus, shaking in NEN

with smaller values would mostly lead to the same local optimum as reached before. Shaking in GEEN starts with two random moves for the same reason. The number k of random moves increases in steps of two up to $\lfloor \frac{r}{2} \rfloor$.

Algorithm 13: Shake (solution $S = \langle P, T \rangle$, size k)

for $i = 1, \ldots, k+1$ **do**
  randomly change the used node p_i of a random cluster V_i
recalculate the MST T and derive T^g
for $i = 1, \ldots, k$ **do**
  remove a randomly chosen global edge $e \in T^g$ yielding components K_1^g and K_2^g
  insert a randomly chosen global edge e' connecting K_1^g and K_2^g with $e' \neq e$
  recalculate the used nodes $p_1, \ldots, p_r$ by dynamic programming

4.5 Memory Function

There is a common situation where VNS unnecessarily spends much time on iterating through all neighborhoods. Suppose, a local optimum reached by VND is a dead end for all neighborhoods and VNS uses shaking to escape from it. Sometimes, applying VND on the new solution soon leads to the same local optimum. Nevertheless, VND spends much time on iterating through all remaining neighborhoods again.
To avoid this situation a hash memory is used. Each deterministic neighborhood keeps a hash value for the best solution obtained so far. Before VND tries to improve a solution, it compares the current and the memorized hash values first. The computation is skipped if these values are identical, i. e. the solution cannot be improved by this neighborhood. This simple concept turned out to save much time in practice.

5. Computational Results for the GMST Problem

In the following, first a detailed description on the test instances used for testing the approaches for both problems is presented, followed by a summary for an experimental comparison of the two constructive heuristics described in Chapter 4.1, which are considered for the creation of initial solutions for the VNS. The computational results of the VNS approach on the different test data sets follow in Chapter 5.3. Finally, Chapter 5.4 analyzes the individual contributions of the different neighborhoods within the VND.
All experiments were performed on a Pentium 4, 2.8GHz PC with 2GB RAM using CPLEX 9.03 to solve the ILP subproblems within GSON.

5.1 Test Instances

Testing has been done on instances used by Ghosh [11], some further large instances of the same types but with more nodes per cluster created by myself, and most of the large Euclidean TSPlib[1] instances with geographical clustering [5].
Ghosh [11] created so called grouped Euclidean instances. In this type of instances, squares with side length *span* are associated to clusters and are regularly laid out on a grid of size $col \times row$ as shown in Figure 5.1. The nodes of each cluster are randomly distributed within the corresponding square. By changing the ratio between cluster separation *sep* and cluster span *span*, it is possible to generate instances with clusters that are overlapping or widely separated. The second type are so called random Euclidean instances where nodes of the same cluster are not necessarily close to each other. Such instances are created by simply scattering the nodes randomly within a square of size 1000×1000 and making the cluster assignment independently at random. Finally, non-Euclidean random instances are generated by choosing all edge costs randomly from the integer interval $[0, 1000]$. All graphs are fully connected. The complete benchmark set used for testing contains instances with up to 1280 nodes, 818560 edges, and 64 clusters; details are listed in Table 5.1. For each type and size, three different instances are considered. The values in the columns denote names of the sets, numbers of nodes, numbers of edges, numbers of clusters and numbers of nodes per cluster. In case of grouped Euclidean instances, numbers of columns and rows of the grid, as well as the cluster separation and cluster span values are additionally given.
Applying geographical clustering [9] on the TSPlib instances is done as follows. First, r center nodes are chosen to be located as far as possible from each other. This is achieved by selecting the first center randomly, the second center as the farthest node from the first center, the third center as the farthest node from the set of the first two centers, and so on. Then, the clustering is done by assigning each of the remaining nodes to its nearest center node. The largest of such TSPlib instances with up to 442 nodes, 97461 edges, and 89 clusters are considered; details are listed in Table 5.1. The values in the columns denote names of

[1] http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html

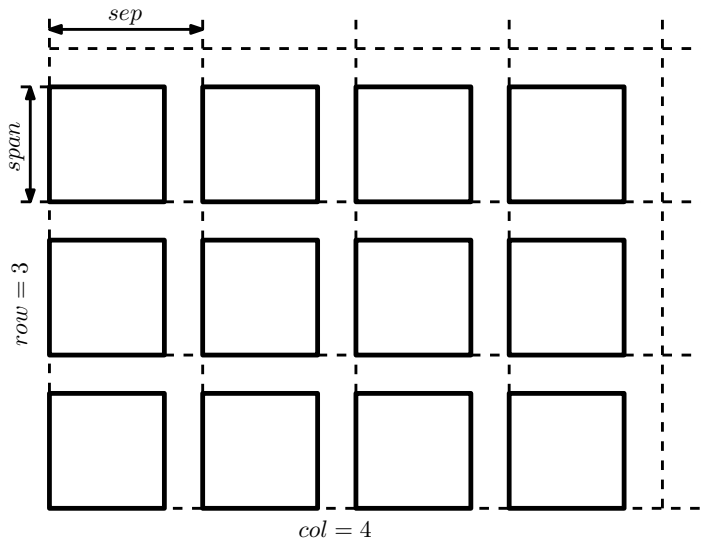


Fig. 5.1: Creation of Grouped Euclidean Instances.

Instance set	$\|V\|$	$\|E\|$	r	$\frac{\|V\|}{r}$	col	row	sep	$span$
Grouped Eucl 125	125	7750	25	5	5	5	10	10
Grouped Eucl 500	500	124750	100	5	10	10	10	10
Grouped Eucl 600	600	179700	20	30	5	4	10	10
Grouped Eucl 1280	1280	818560	64	20	8	8	10	10
Random Eucl 250	250	31125	50	5	-	-	-	-
Random Eucl 400	400	79800	20	20	-	-	-	-
Random Eucl 600	600	179700	20	30	-	-	-	-
Non-Eucl 200	200	19900	20	10	-	-	-	-
Non-Eucl 500	500	124750	100	5	-	-	-	-
Non-Eucl 600	600	179700	20	30	-	-	-	-

Tab. 5.1: Benchmark instance sets adopted from Ghosh [11] and extended with new sets. Each instance has a constant number of nodes per cluster.

the instances, numbers of nodes, numbers of edges, numbers of clusters, and the average, minimal, and maximal numbers of nodes per cluster.

Instance name	$\|V\|$	$\|E\|$	r	$\frac{\|V\|}{r}$	$d_{\min}$	$d_{\max}$
gr137	137	9316	28	5	1	12
kroa150	150	11175	30	5	1	10
d198	198	19503	40	5	1	15
krob200	200	19900	40	5	1	8
gr202	202	20301	41	5	1	16
ts225	225	25200	45	5	1	9
pr226	226	25425	46	5	1	16
gil262	262	34191	53	5	1	13
pr264	264	34716	54	5	1	12
pr299	299	44551	60	5	1	11
lin318	318	50403	64	5	1	14
rd400	400	79800	80	5	1	11
fl417	417	86736	84	5	1	22
gr431	431	92665	87	5	1	62
pr439	439	96141	88	5	1	17
pcb442	442	97461	89	5	1	10

Tab. 5.2: TSPlib instances with geographical clustering [5]. Numbers of nodes vary for each cluster.

5.2 Comparison of the Constructive Heuristics

Table 5.2 summerizes the comparison of MDH and IKH on all considered input instances. It turned out that IKH performs consistently better than MDH on the TSPlib based, grouped Euclidean, and non-Euclidean instances. Only on random Euclidean instances, MDH could outperform IKH on 70% of the instances. The ratios $\overline{\text{IKH}/\text{MDH}}$ indicate the average factor between the objective values of the solutions generated by IKH and MDH. Interestingly, the two heuristics never obtained the same solutions or solutions of the same quality. As the required CPU-times of both heuristics are very small (less than 80ms for the largest instances with 1280 nodes), I decided to run both, MDH and IKH, and to choose the better result as initial solution for VNS.

Instance Type	MDH better %	IKH better %	$\overline{\text{IKH/MDH}}$
TSBlib based	0	100	0.89
Grouped Euclidean	0	100	0.85
Random Euclidean	70	30	1.36
Non-Euclidean	0	100	0.16

Tab. 5.3: Comparison of MDH and IKH.

5.3 Computational Results for the VNS

The results of the Variable Neighborhood Search (VNS) are compared to Tabu Search with recency and frequency based memory (TS2) [11], Variable Neighborhood Decomposition Search (VNDS) [11], the Simulated Annealing (SA) approach from Pop [30], and, in case of TSPlib instances, also to the Genetic Algorithm (GA) from Golden, Raghavan, and Stanojevic [13]. While TS2 is deterministic, average results over 30 runs for VNDS and VNS and over at least 10 runs for SA (due to its long running times) are provided. For TS2, VNDS, and our VNS, runs were terminated when a certain CPU-time limit had been reached. In contrast, SA was run with the same cooling schedule and termination criterion as specified by Pop [30], which led to significantly longer running times compared to the other algorithms. The results for the GA are directly taken from Golden, Raghavan, and Stanojevic [13].
In Table 5.4 and 5.5, instance names, numbers of nodes, numbers of clusters, (average) numbers of nodes per cluster and the (average) objective values of the final solutions of the different algorithms are shown. Best values are printed in bold. In case of SA and VNS, the corresponding standard deviations are provided too. VNDS produces very stable results as the standard deviations are always zero, except for the second instance of the set "Random Eucl 400" where it is 0.34. For GA, standard deviations are not available as they are not listed by Golden, Raghavan, and Stanojevic [13].
Table 5.4, compares VNS to TS2, VNDS, and SA on grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances. Additionally, relative values grouped by the different sets are illustrated in Figure 5.2, 5.3, and 5.4 where the results of our VNS are taken as base (100%).
The time limit was set to 600s for TS2, VNDS, and VNS. In fact, none of the tested algorithms practically needs that much time on smaller instances to find the finally best solutions, but Ghosh [11] used this time limit as termination criterion, so it is retained. SA required 150s for small instances with 125 nodes and up to about 40000s for the largest instances with 1280 nodes.
When comparing the VNS approach with SA, it can be observed that VNS consistently finds better solutions. Wilcoxon rank sum tests yield error probabilities of less than 1% for the assumptions that the observed differences in the mean objective values are significant. Also in comparison to VNDS, our VNS is the clear winner. There are only two instances where VNS and VNDS obtained exactly the same mean results and one instance (the second of set "Grouped Eucl 500") on which VNDS performed better. In all other cases, VNS' solutions are superior with high statistical significance (error levels less than 1%). The results of VNS

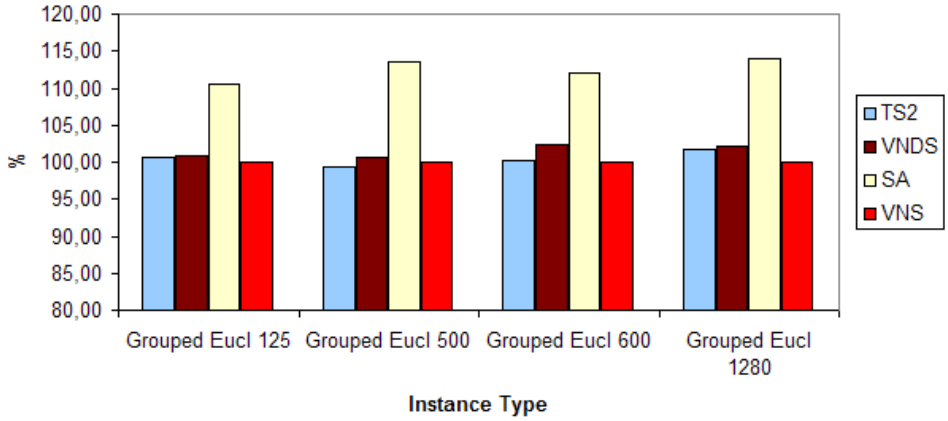


Fig. 5.2: Relative results on grouped euclidean instances for each set (VNS = 100%).

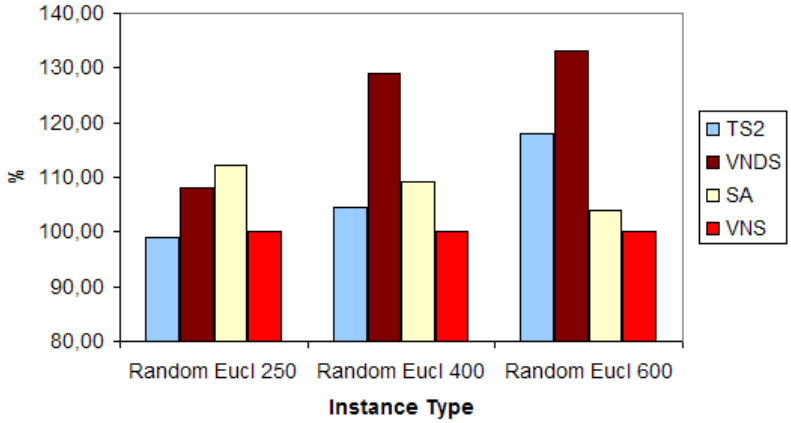


Fig. 5.3: Relative results on random euclidean instances for each set (VNS = 100%).

and TS2 are ambiguous. While TS2 usually produces better results on instances with few nodes per cluster, VNS is typically superior when the number of nodes per cluster is higher. This can in particular be observed on the instances with 30 nodes per cluster.
On grouped Euclidean instances, the objective values of the final solutions obtained by the considered algorithms, especially those by TS2 and VNS, are relatively close. It seems that these instances are easier to handle as the quality of the solutions are less affected by the differences of the approaches. On random Euclidean instances, especially when the number of nodes per cluster is higher, VNS produces substantially better results than TS2 and VNDS; e.g. for the third instance of set "Random Eucl 600", the solutions obtained by VNS are on average 34.4% better than those of TS2. It can be observed too that SA, which is usually worst, is able to outperform TS2 and VNDS on some of these instances. One can conclude that the neighborhood type GEEN, which is also the main component of SA, is very effective

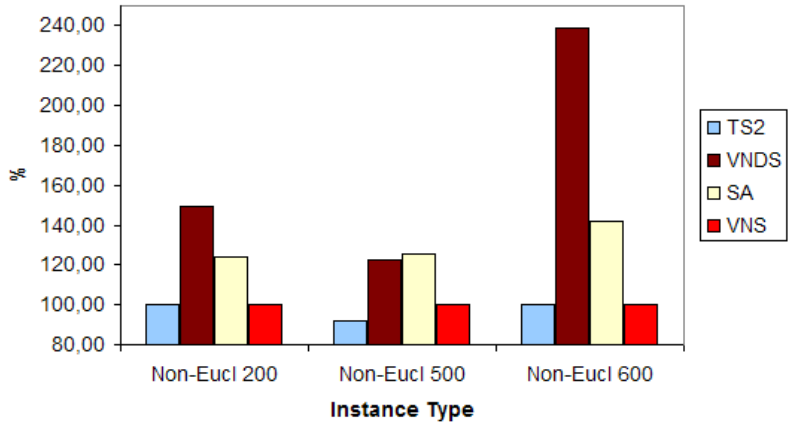


Fig. 5.4: Relative results on non-euclidean instances for each set (VNS = 100%).

on random Euclidean instances and on instances with higher number of nodes per cluster. On non-Euclidean instances, TS2 mostly outperforms all the other algorithms.
Table 5.5, compares our VNS to TS2, VNDS, SA, and also the GA on the TSPlib based instances. The results for the GA are adopted from Golden, Raghavan, and Stanojevic [13], where only smaller instances up to pr226 have been considered. The listed CPU-times were the stopping criteria for TS2, VNDS and VNS. SA needed up to 10000s for large instances as pcb442. The test runs indicate that our VNS outperforms VNDS and SA significantly. Wilcoxon rank sum tests again yield error probabilities of less than 1% for the assumptions that the observed differences in the mean objective values are significant. Judging by the few results for GA, VNS produces results which are at least as good as those of GA. Considering VNS and TS2, clear conclusions cannot be drawn. Most of the time, these two algorithms generate comparable results under the same conditions. Smaller TSPlib instances are omitted in Table 5.5 as the most capable algorithms TS2, GA and VNS were all able to (almost) always provide optimal solutions as found by the Branch-and-Cut algorithm [8]. The latter could solve all instances with up to 200 nodes except d198 to provable optimality in up to 5254s.
In overall, VNS and TS2 are the most powerful algorithms among all considered approaches. Out of 46 instances that have been tested, VNS produces strictly better results in 19 cases, TS2 is better in 17 cases, and on 10 instances, they are equally good.

5.4 Contributions of the Neighborhoods

In order to analyze how the different neighborhood searches of VNS contribute to the whole optimization, a logging how often each one was able to improve on a current solution has been done. Table 5.6 shows the ratios of these numbers of improvements to how often they were called. These values are grouped by the different types of input instances. Additionally, these values are illustrated in Figures 5.5, 5.6, and 5.7.
In general, each neighborhood search contributes substantially to the whole success. NEN and RNEN2 are most effective in terms how often they improve on a solution, whereas the differences in the objective values achieved by single improvements are typically larger in case

of GEEN and GSON. Considering that GSON operates on solutions which are already local optima with respect to all other neighborhoods, its improvement ratios are quite remarkable.

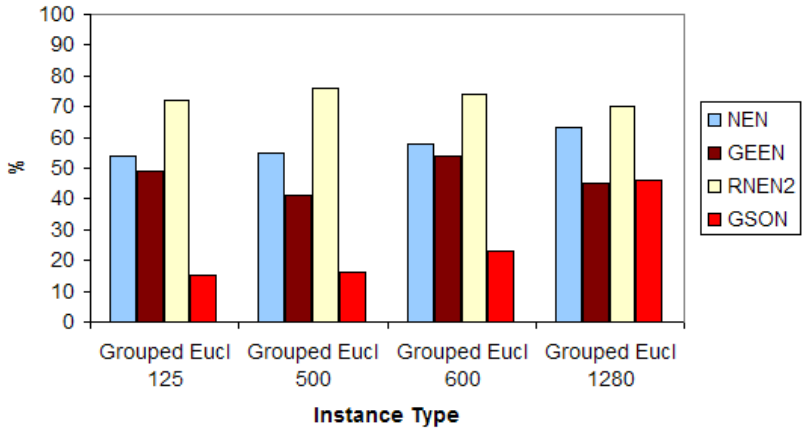


Fig. 5.5: Contributions of the neighborhoods on grouped euclidean instances.

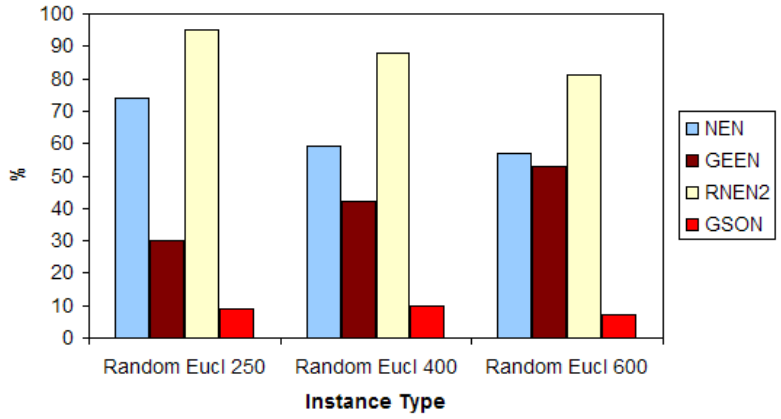


Fig. 5.6: Contributions of the neighborhoods on random euclidean instances.

Regarding the different instance sets, it can be observed that the improvement ratio of GEEN generally increases with the size of nodes per cluster. Furthermore, one can see that on large grouped Euclidean instances containing 1280 nodes, GSON performs substantially better than on other instances.

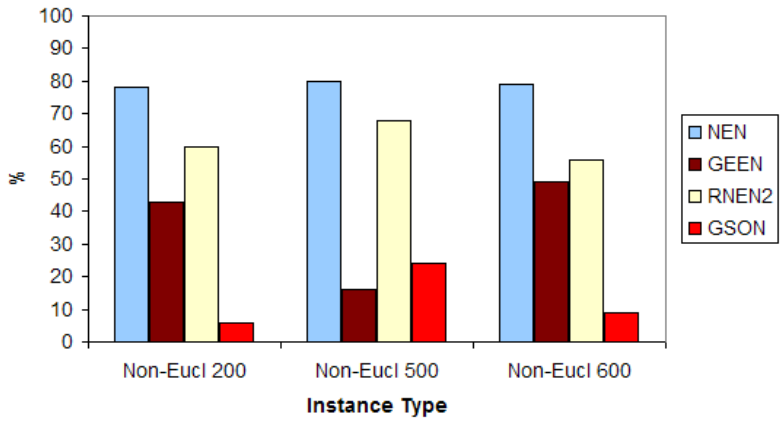


Fig. 5.7: Contributions of the neighborhoods on non-euclidean instances.

Instances				TS2	VNDS	SA		VNS	
Set	$\|V\|$	r	$\|V\|/r$	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
	125	25	5	**141.1**	**141.1**	152.3	0.52	**141.1**	0.00
Grouped Eucl 125	125	25	5	**133.8**	**133.8**	150.9	0.74	**133.8**	0.00
	125	25	5	143.9	145.4	156.8	0.00	**141.4**	0.00
	500	100	5	**566.7**	577.6	642.3	0.00	567.8	1.08
Grouped Eucl 500	500	100	5	**578.7**	584.3	663.3	1.39	585.7	1.02
	500	100	5	**581.6**	588.3	666.7	1.81	584.2	1.93
	600	20	30	85.2	87.5	93.9	0.00	**84.6**	0.00
Grouped Eucl 600	600	20	30	**87.9**	90.3	99.5	0.28	**87.9**	0.00
	600	20	30	88.6	89.4	99.2	0.17	**88.5**	0.00
	1280	64	20	327.2	329.2	365.1	0.46	**316.3**	2.20
Grouped Eucl 1280	1280	64	20	322.2	322.5	364.4	0.00	**318.7**	1.57
	1280	64	20	332.1	335.5	372.0	0.00	**330.8**	1.70
	250	50	5	**2285.1**	2504.9	2584.3	23.82	2318.5	39.27
Random Eucl 250	250	50	5	**2183.4**	2343.3	2486.7	0.00	2201.3	24.54
	250	50	5	**2048.4**	2263.7	2305.0	16.64	2060.8	34.32
	400	20	20	**557.4**	725.9	665.1	3.94	620.4	14.34
Random Eucl 400	400	20	20	724.3	839.0	662.1	7.85	**595.3**	0.00
	400	20	20	604.5	762.4	643.7	14.54	**588.4**	6.08
	600	20	30	541.6	656.1	491.8	7.83	**443.5**	0.00
Random Eucl 600	600	20	30	540.3	634.0	542.8	25.75	**535.5**	10.66
	600	20	30	627.4	636.5	469.5	2.75	**469.0**	11.98
	200	20	10	**71.6**	94.7	76.9	0.21	**71.6**	0.00
Non-Eucl 200	200	20	10	**41.0**	76.6	41.1	0.02	**41.0**	0.00
	200	20	10	**52.8**	75.3	86.9	5.38	**52.8**	0.00
	500	100	5	**143.7**	203.2	200.3	4.44	155.8	3.07
Non-Eucl 500	500	100	5	**132.7**	187.3	194.3	1.20	154.1	4.89
	500	100	5	**162.3**	197.4	205.6	0.00	168.8	3.25
	600	20	30	**14.5**	59.4	22.7	1.49	15.5	1.17
Non-Eucl 600	600	20	30	17.7	23.7	22.0	0.82	**16.3**	1.33
	600	20	30	**15.1**	29.5	22.1	0.44	15.3	0.90

Tab. 5.4: Results on instance sets from [11] and correspondingly created new sets, 600s CPU-time (except SA). Three different instances are considered for each set.

TSPlib Instances				TS2	VNDS	SA		GA	VNS	
Name	$\|V\|$	r	time	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	$\overline{C(T)}$	std dev
gr137	137	28	150s	**329.0**	330.0	352.0	0.00	**329.0**	**329.0**	0.00
kroa150	150	30	150s	**9815.0**	**9815.0**	10885.6	25.63	**9815.0**	**9815.0**	0.00
d198	198	40	300s	7062.0	7169.0	7468.73	0.83	**7044.0**	**7044.0**	0.00
krob200	200	40	300s	11245.0	11353.0	12532.0	0.00	**11244.0**	**11244.0**	0.00
gr202	202	41	300s	**242.0**	249.0	258.0	0.00	243.0	**242.0**	0.00
ts225	225	45	300s	62366.0	63139.0	67195.1	34.49	62315.0	**62268.3**	0.45
pr226	226	46	300s	**55515.0**	**55515.0**	56286.6	40.89	**55515.0**	**55515.0**	0.00
gil262	262	53	300s	**942.0**	979.0	1022.0	0.00	-	942.3	0.78
pr264	264	54	300s	**21886.0**	22115.0	23445.8	68.27	-	21888.6	5.03
pr299	299	60	450s	20339.0	20578.0	22989.4	11.58	-	**20316.7**	2.03
lin318	318	64	450s	18521.0	18533.0	20268.0	0.00	-	**18513.0**	11.87
rd400	400	80	600s	**5943.0**	6056.0	6440.8	3.40	-	5954.3	10.77
fl417	417	84	600s	7990.0	7984.0	8076.0	0.00	-	**7982.0**	0.00
gr431	431	87	600s	1034.0	1036.0	1080.5	0.51	-	**1033.0**	0.18
pr439	439	88	600s	**51852.0**	52104.0	55694.1	45.88	-	51868.2	47.58
pcb442	442	89	600s	**19621.0**	19961.0	21515.1	5.15	-	19746.2	55.26

Tab. 5.5: Results on TSPlib instances with geographical clustering, $\frac{|V|}{r} = 5$, variable CPU-time.

Instance Type	$\|V\|$	r	$\|V\|/r$	NEN	GEEN	RNEN2	GSON
TSBlib based	n.a.	n.a.	5	0.55	0.44	0.67	0.18
Grouped Euclidean	125	25	5	0.54	0.49	0.72	0.15
	500	100	5	0.55	0.41	0.76	0.16
	600	20	30	0.58	0.54	0.74	0.23
	1280	64	20	0.63	0.45	0.70	0.46
Random Euclidean	250	50	5	0.74	0.30	0.95	0.09
	400	20	20	0.59	0.42	0.88	0.10
	600	20	30	0.57	0.53	0.81	0.07
Non-Euclidean	200	20	10	0.78	0.43	0.60	0.06
	500	100	5	0.80	0.16	0.68	0.24
	600	20	30	0.79	0.49	0.56	0.09

Tab. 5.6: Relative effectivity of NEN, GEEN, RNEN2, and GSON.

6. The Generalized Minimum Edge Biconnected Network Problem

This chapter starts with the definition of the Generalized Minimum Edge Biconnected Network (GMEBCN) problem which has already been presented in Chapter 1. Afterwards, Chapter 6.2 analyses if the effective concepts of either fixing the selected nodes or the global edges and computing a concrete solution out of them as used for the GMST problem can be applied in an appropriate way for the GMEBCN problem too. Unfortunately both subproblems turn out to be NP hard for the GMEBCN problem. Therefore Chapter 6.3 discusses one possible way to deal with this complexity by presenting a technique to substantially reduce a current solution to a significant smaller graph in terms of the amount of clusters. Finally, Chapter 6.4 provides two ILP formulations for the GMEBCN problem.

6.1 Problem Definition

The *Generalized Minimum Edge Biconnected Network (GMEBCN)* problem considers a weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E and edge cost function $c : E \rightarrow \mathbb{R}^+$. The node set V is partitioned into r pairwise disjunct clusters $V_1, V_2, \ldots, V_r$ containing $d_1, d_2, \ldots, d_r$ nodes, respectively.
As introduced in Chapter 2.1, an edge biconnected network of a graph is a subgraph connecting all nodes and containing no bridges. Therefore a solution to the GMEBCN problem defined on G is a subgraph $S = \langle P, F \rangle$ with $1 \leq i \leq r$, and $F \subseteq P \times P \subseteq E$ being an edge biconnected graph (i. e. does not contain any bridges).
The costs of such an edge biconnected network are its total edge costs, i. e. $C(F) = \sum_{(u,v) \in F} c(u, v)$, and the objective is to identify a solution with minimum costs.

6.2 Subproblems and their Complexity

The main ideas of the neighborhoods described for the GMST problem are to either compute the best possible MST on a fixed set of selected nodes for NEN and RNEN2 on the one hand and to compute the best nodes to select for a fixed set of global edges on the other hand (GEEN, GSON). As these concepts turned out to be very efficient it makes sense to apply them on the GMEBCN problem, too. Theorem 1 starts by formally describing the obvious fact of computing the optimal solution while fixing the selected nodes is NP hard.

Theorem 1: Given an instance $G = \langle V, E, c \rangle$ of the GMEBCN Problem. If the selected nodes

$P = \{p_1, \ldots, p_r\}$ of a solution $S = \langle P, F \rangle$ are fixed, then the problem of identifying the optimal edges $F \subseteq E$ is NP hard.

Proof Identifying the optimal edges for a given set of nodes P becomes the classical Minimum Edge Biconnected Network (MEBCN) problem which is known to be NP hard (compare Chapter 2.5). □

In order to discuss the remaining subproblem, we consider an edge-minimal edge-biconnected global graph $G^{\mathrm{g}}_{\mathrm{bic}}$ with fixed global edges $E^{\mathrm{g}}_{\mathrm{bic}}$. The NP hardness of the subproblem of looking for the optimal selected nodes of each cluster yielding the minimum costs will be proved in two steps.
First, the graph coloring problem will be reduced to the problem of finding the optimal selected nodes of a general global graph $G^{\mathrm{g}}_{\mathrm{gen}}$ which is not necessarily edge-minimal edge-biconnected. In the second step, we transform $G^{\mathrm{g}}_{\mathrm{gen}}$ into an edge-minimal edge-biconnected global graph $G^{\mathrm{g}}_{\mathrm{bic}}$ by adding artificial clusters so that finding the optimal selected nodes in $G^{\mathrm{g}}_{\mathrm{bic}}$ yields the minimum cost solution to $G^{\mathrm{g}}_{\mathrm{gen}}$.

Theorem 2: If the global edges of a global graph $G^{\mathrm{g}}_{\mathrm{gen}} = \langle V^{\mathrm{g}}_{\mathrm{gen}}, E^{\mathrm{g}}_{\mathrm{gen}} \rangle$ are fixed, the problem of identifying optimal selected nodes P yielding the minimum costs is NP hard.

Proof Consider the graph coloring problem on an undirected graph $H = \langle U, F \rangle$ (Figure 6.1a). This graph is transformed into a clustered graph $G^{\mathrm{g}}_{\mathrm{gen}} = \langle V^{\mathrm{g}}_{\mathrm{gen}}, E^{\mathrm{g}}_{\mathrm{gen}} \rangle$ by the following procedure: Each node $i \in U$ becomes a cluster $V_i \in V^{\mathrm{g}}_{\mathrm{gen}}$ and for each possible color c of i, we introduce a node v^c_i in cluster V_i (Figure 6.1b). For each edge $(i, j) \in F$, we create in the clustered graph edges $(v^c_i, v^d_j)\ \forall\ v^c_i \in V_i,\ \forall\ v^d_j \in V_j$ (Figure 6.1c). An edge's costs are 1 if $c \neq d$ and ∞ otherwise.
If we are able to solve the problem of identifying the optimal nodes of each cluster in order to minimize the total network costs of $G^{\mathrm{g}}_{\mathrm{gen}}$ (Figure 6.1d), we also solve the original graph coloring problem on H. Suppose v^c_i is the selected node in cluster V_i, then c becomes the color of node $i \in U$ (Figure 6.1e). □

Theorem 3: If the global edges of an edge-minimal edge-biconnected global graph $G^{\mathrm{g}}_{\mathrm{bic}} = \langle V^{\mathrm{g}}_{\mathrm{bic}}, E^{\mathrm{g}}_{\mathrm{bic}} \rangle$ are fixed, the problem of identifying optimal selected nodes of each cluster yielding the minimum costs is NP hard.

Proof If $G^{\mathrm{g}}_{\mathrm{gen}} = \langle V^{\mathrm{g}}_{\mathrm{gen}}, E^{\mathrm{g}}_{\mathrm{gen}} \rangle$, after the previous transformation, is not edge-minimal edge-biconnected, $E^{\mathrm{g}}_{\mathrm{gen}}$ consists of at least one redundant edge (Figure 6.2a). For each redundant global edge $e = (V_i, V_j) \in E^{\mathrm{g}}_{\mathrm{gen}}$, we insert additional clusters V^e_i and V^e_j, which are exact copies of V_i and V_j. The global edge (V_i, V_j) gets replaced by (V_i, V^e_i), (V^e_i, V^e_j) and (V^e_j, V_j) (Figure 6.2b). Let $G^{\mathrm{g}}_{\mathrm{bic}} = \langle V^{\mathrm{g}}_{\mathrm{bic}}, E^{\mathrm{g}}_{\mathrm{bic}} \rangle$ denote the resulting graph, which is obviously edge-minimal edge-biconnected.
By adding the clusters V^e_i and V^e_j, we have to modify the local connections E as well. We replace each edge $(u, v) \in E \mid u \in V_i \wedge v \in V_j$ by $(u^e, v^e) \mid u^e \in V^e_i \wedge v^e \in V^e_j$ with u^e and v^e being the copies of u and v, respectively. Between V_i and V^e_i, we add edges (u, u^e) with costs 0

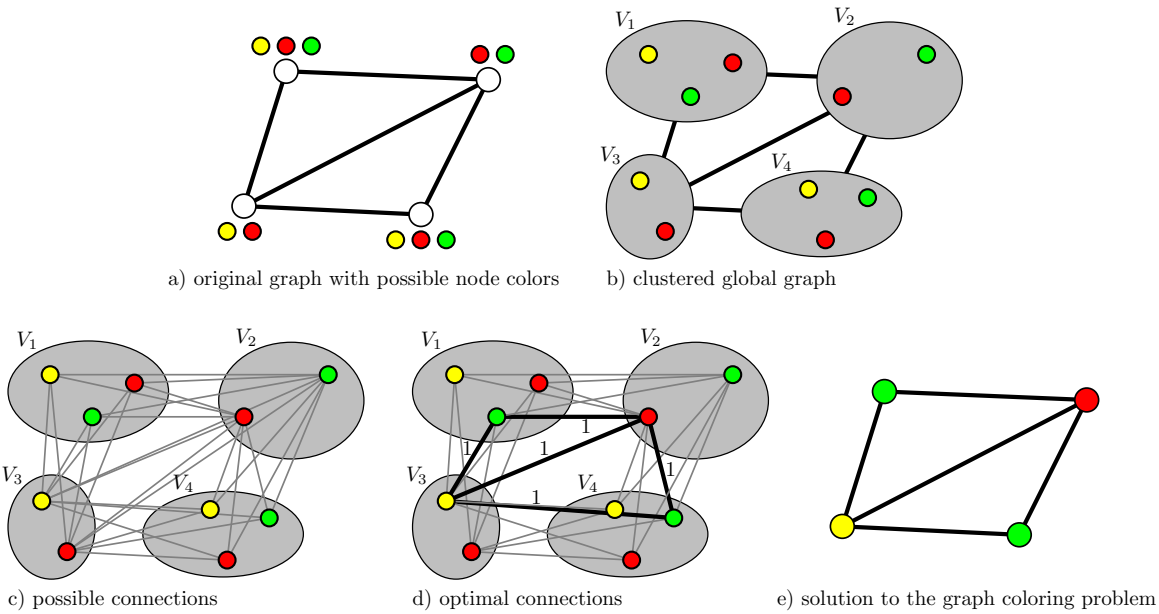


Fig. 6.1: Transformation of Graph Coloring to GMEBCN - Part 1.

for all $u \in V_i$. The same procedure is applied for V_j and V_j^e (Figure 6.2c). Let $G' = \langle V', E', c \rangle$ denote the resulting local graph of $G^{\mathrm{g}}_{\mathrm{bic}}$. Note that after this extension of polynomial size, G' is edge-minimal edge-biconnected.
By determining the optimal selected nodes on G' subject to the global edges of $G^{\mathrm{g}}_{\mathrm{bic}}$, we get the optimal selected nodes on G subject to the global edges of $G^{\mathrm{g}}_{\mathrm{gen}}$ by removing the artificial clusters (Figure 6.2d). Thus, we also obtain the solution of the graph coloring problem on H by choosing the edges $F = \{(u, v) \in E \mid \exists (u^e, v^e) \in F'\}$ with F' being the edge set of the solution to the MEBCN problem on G'.
The backward transformation is valid because only one node (hence one color) is chosen per cluster as we solve the GMEBCN problem containing exactly one node per cluster. Furthermore, the cloning process only creates edges from nodes $u \in V$ to its copies $v_i^e \in V^e$ and thus only copies of the same node are selected in the copied clusters. □

6.3 Graph Reduction

Due to the unpleasant results regarding the complexity of the subproblems, the concept of utilizing a global graph for defining neighborhoods cannot be adapted straightforward to the GMEBCN problem. Nevertheless it is still possible to compute the best possible nodes for a given global graph by additionally fixing the used nodes in a few clusters as described in this chapter.
This so called graph reduction is based on the observation that good solutions to the GMEBCN problem typically consist of only few clusters with high degrees and long paths of clusters having degree two between them. As shown in Figure 6.3 the best nodes to select for such a path between two clusters of higher degrees can be determined efficiently using either a

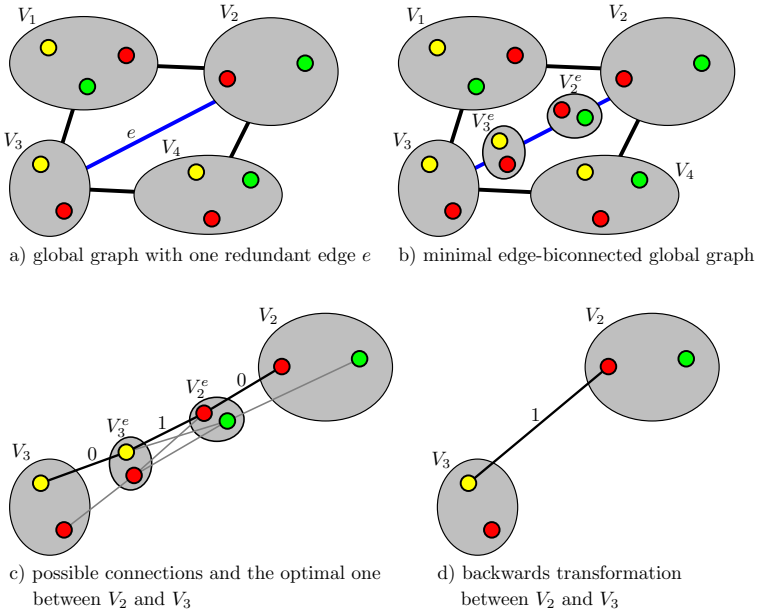


Fig. 6.2: Transformation of Graph Coloring to GMEBCN - Part 2.

shortest path algorithm or a dynamic programming approach similar as described for GEEN in Chapter 4.2.2.

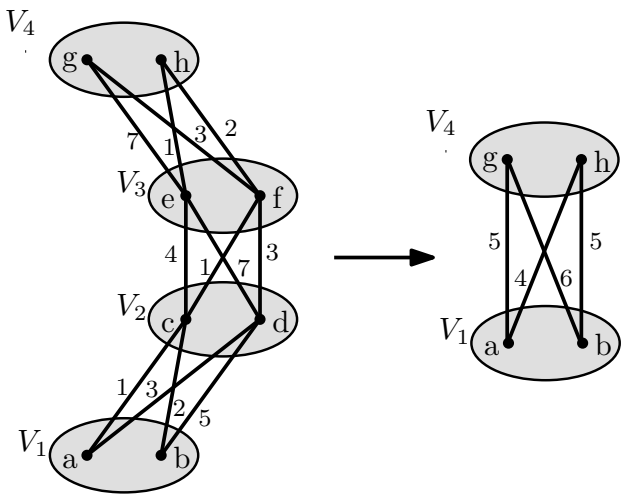


Fig. 6.3: Reducing a Global Path.

Before going into the details how to reduce a graph, the concept of a global path and the terms inner nodes and end nodes of a path have to be introduced.

Definition 7: A global path $P^{\mathrm{g}} = \langle V_{\mathrm{p}}^{\mathrm{g}}, E_{\mathrm{p}}^{\mathrm{g}} \rangle$ of a global graph $G^{\mathrm{g}} = \langle V^{\mathrm{g}}, E^{\mathrm{g}} \rangle$ is defined by a

subset of clusters $V_{\mathrm{p}}^{\mathrm{g}} = \{V_1, \ldots, V_{\mathrm{n}}\} \subseteq V^{\mathrm{g}}$ and includes all edges of G^{g} between them, i. e. $E_{\mathrm{p}}^{\mathrm{g}} = \{(u, v) \in E^{\mathrm{g}} | u, v \in V_{\mathrm{p}}^{\mathrm{g}}\}$, where all inner clusters have degree two, i. e. $\deg(V_i) = 2$, (i=2, ..., n-1) in G^{g}.

As a global path is already defined by its edges, a simplified term will be used afterwards, e. g. $P = \{(V_1, V_2), \ldots, (V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$ is a short notation for $P = \langle V_{\mathrm{P}}, E_{\mathrm{P}} \rangle$ with $V_{\mathrm{P}} = \{V_1, \ldots, V_{\mathrm{n}}\}$ and $E_{\mathrm{P}} = \{(V_1, V_2), \ldots, (V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$.

Definition 8: Given a path $P = \langle V_{\mathrm{p}}, E_{\mathrm{p}} \rangle$ with $V_{\mathrm{p}} = \{v_0, \ldots, v_{\mathrm{n}}\}$ and $E_{\mathrm{p}} = \{(v_0, v_1), \ldots, (v_{\mathrm{n}-1}, v_{\mathrm{n}})\}$ with $v_0 \neq v_1 \neq v_{\mathrm{n}}$ and $\deg(v_i) = 2,\ \forall i = 1, \ldots, n-1$. We call $v_0, v_{\mathrm{n}} \in V_{\mathrm{p}}$ the *end nodes* of P and $v_1, \ldots, v_{\mathrm{n}-1} \in V_{\mathrm{p}}$ the *inner nodes* of P.

By computing the shortest paths between every pair of nodes of the end clusters of a global edge, it is possible to replace each of these paths by a single global edge. The costs between every pair of these nodes are the costs of the shortest paths between them.
For the edges of a given global path $E_{\mathrm{p}}^{\mathrm{g}} = \{(V_1, V_2), \ldots, (V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$, the shortest path from each node $w \in V_1$ of the start cluster V_1 to each node $v \in V_k$ and $V_k \in V_{\mathrm{p}}^{g}$ can be determined by the following recursion:

$$C_w(P^{\mathrm{g}}, V_k, v) = \begin{cases} 0 \text{ if } k = 1 \text{ (V_k is the the start cluster of the global path)} \\ \min\limits_{u \in V_{k-1}} \{c(v, u) + C_w(P^{\mathrm{g}}, V_{k-1}, u)\} \text{ else.} \end{cases} \quad \forall w \in V_1 \quad (6.1)$$

The computation of the "Reduced Graph" or "Reduced Solution" is done by first determining all clusters having degree greater than two in the global graph as these are the ones that will be included in the reduced graph. Afterwards, all global paths are determined and reduced one by one.
Several special cases may occur during this process. First of all it happens frequently that two global paths connect the same clusters. Therefore multi-edges would occur in the resulting graph. However these edges can simply be combined to a single edge by adding the corresponding costs.
Secondly, such paths may be closed circles, i. e. $V_1 = V_{\mathrm{n}}$. A reduction of a circle using the approach described above would lead to a single cluster with an edge to itself. As graphs containing self loops are not considered during the algorithms, we need to avoid this situation. Therefore, for each circle an additional cluster is included in the resulting reduced graph as shown in Figure 6.4. Detection of these circles is done while determining all global paths which is accomplished by a depth first search algorithm. For simplicity this algorithm adds the last cluster it passes before reaching the start cluster (i. e. $V_{\mathrm{n}-1}$) to the resulting reduced graph, see Figure 6.4.

Definition 9: All clusters $V_i,\ i = 1, \ldots, r$ in a global graph G^{g} that are included in the corresponding reduced graph $G_{\mathrm{r}}^{\mathrm{g}}$ (i. e. all clusters having $\deg(V_i) \geq 3$ and all clusters that are additionally included in the case of the occurrence of circles) are called *relevant clusters*, while all clusters $V_i \notin G_{\mathrm{r}}^{\mathrm{g}}$ are referred as *irrelevant clusters*.

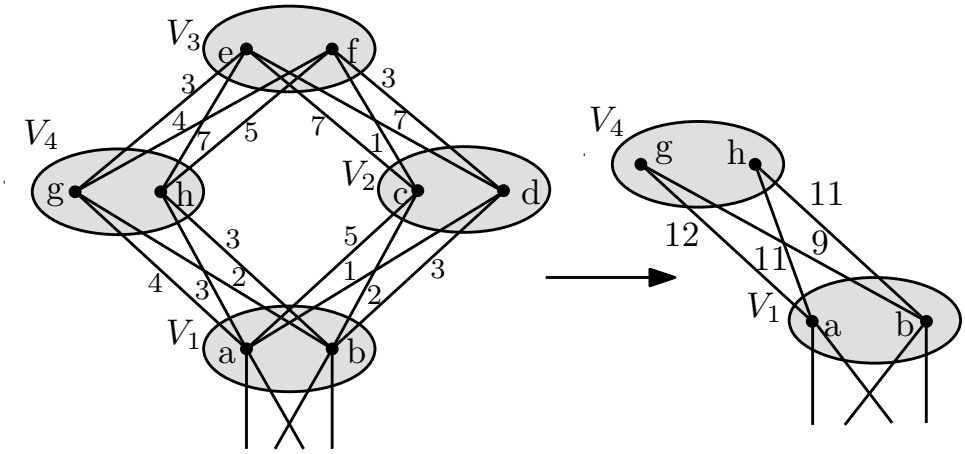


Fig. 6.4: Reducing a Simple Cycle.

Figure 6.5 shows an example how a graph is reduced. Clusters V_1 and V_6 have degree higher than two, and therefore are included in the reduced graph, while all three paths $P_1 = \{(V_1, V_2), (V_2, V_3), (V_3, V_6)\}$, $P_2 = \{(V_1, V_4), (V_5, V_6)\}$, and $P_3 = \{(V_1, V_5), (V_5, V_6)\}$ between them are reduced to a single global edge in the reduced graph. Assuming the circle $P_4 = \{(V_6, V_7), (V_7, V_8), (V_8, V_9), (V_9, V_6)\}$ is examined in this order, V_9 is included additionally in the reduced graph.

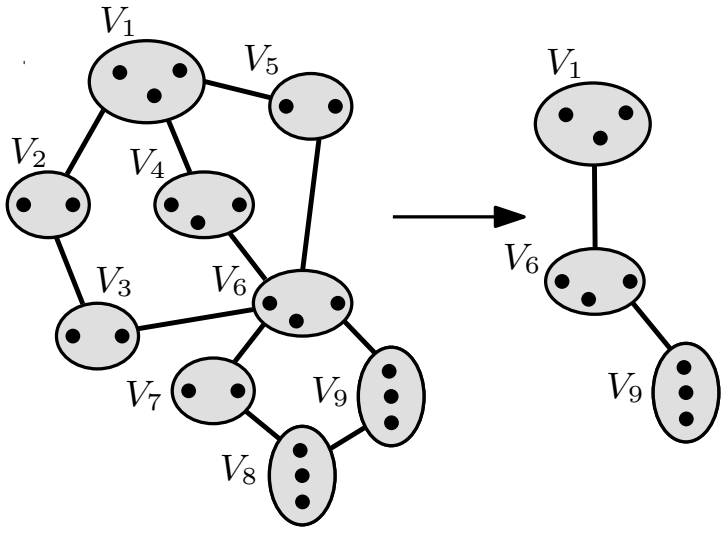


Fig. 6.5: Example of Reducing a Graph.

Algorithm 14 summarizes the steps for computing a reduced graph. Similar to GEEN (compare Chapter 4.2.2) we save all values computed using dynamic programming which has the benefit that decoding the reduced graph to a concrete set of used nodes for the original problem can be done without applying the whole algorithm again. Additionally these values will

be used in intelligent enumeration strategies that update the objective value in an incremental way for two neighborhoods based on graph reduction which will be presented in Chapter 7.3.

Algorithm 14: Compute Reduced Graph(solution S)

$G^{\mathrm{g}} = \langle V^{\mathrm{g}}, E^{\mathrm{g}} \rangle$ is the global graph of $S = \langle P, F \rangle$
$E^{\mathrm{g}}_{\mathrm{red}} = \emptyset$
$V^{\mathrm{g}}_{\mathrm{red}}$ = set of all clusters in G^{g} with $\deg(V^{\mathrm{g}}_{\mathrm{r}}) \geq 3$
forall global paths $P^{\mathrm{g}} = \{(V_1, V_2), \ldots, (V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$ in G^{g} **do**
  if $V_1 = V_{\mathrm{n}}$ **then**
    // P^{g} is a circle
    $P^{\mathrm{g}} = P^{\mathrm{g}} \setminus \{(V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$
    $V^{\mathrm{g}}_{\mathrm{red}} = V^{\mathrm{g}}_{\mathrm{red}} \cup \{V_{\mathrm{n}-1}\}$
    $E^{\mathrm{g}}_{\mathrm{red}} = E^{\mathrm{g}}_{\mathrm{red}} \cup \{(V_{\mathrm{n}-1}, V_{\mathrm{n}})\}$
  else
    if $(V_1, V_{\mathrm{n}}) \notin E^{\mathrm{g}}_{\mathrm{red}}$ **then**
      $E^{\mathrm{g}}_{\mathrm{red}} = E^{\mathrm{g}}_{\mathrm{red}} \cup \{(V_1, V_{\mathrm{n}})\}$
  calculate costs of reduced path P^{g}
reduced global graph $G^{\mathrm{g}}_{\mathrm{red}} = \langle V^{\mathrm{g}}_{\mathrm{red}}, E^{\mathrm{g}}_{\mathrm{red}} \rangle$
// generate corresponding "local" reduced graph $G_{\mathrm{red}} = \langle V_{\mathrm{red}}, E_{\mathrm{red}} \rangle$
$V_{\mathrm{red}} = \{v \in V_{\mathrm{red}} | V_{\mathrm{red}} \in V^{\mathrm{g}}\}$
$E_{\mathrm{red}} = \emptyset$
forall edges $(V_{\mathrm{i}}, V_{\mathrm{j}}) \in E^{\mathrm{g}}_{\mathrm{red}}$ **do**
  forall $v \in V_{\mathrm{i}}$ **do**
    forall $w \in V_{\mathrm{j}}$ **do**
      $E_{\mathrm{red}} = E_{\mathrm{red}} \cup \{(v, w)\}$
      set $c(v, w)$ in G_{red} to costs of reduced path between v and w

While calculating the costs of each reduced path, Algorithm 14 considers each edge of G^{g} mostly once. For each edge considered the concrete costs have to be computed with dynamic programming using equation 6.1 whose complexity is bounded by $O(d^3_{max})$ to compute the costs for all nodes of a single cluster. Thus the first part of Algorithm 14 has computational complexity $O(|E^{\mathrm{g}}| d^3_{max}) \leq O(r^2 d^3_{\max})$.
Finally generating the corresponding "local" reduced graph considers each edge of the reduced global graph $E^{\mathrm{g}}_{\mathrm{red}} = (V_{\mathrm{i}}, V_{\mathrm{j}})$ exactly once. For each of these edges the costs between any pair of nodes $u \in V_{\mathrm{i}}$ and $v \in V_{\mathrm{j}}$ have to be computed, so this part has complexity of $O(|E^{\mathrm{g}}_{\mathrm{red}}| d^2_{\max}) = O(r^2 d^2_{\max})$. Hence the overall complexity of Algorithm 14 is in $O(r^2 d^3_{\max})$ in the worst case. However, as typical solutions to the GMEBCN problem consists of much fewer edges, the computation of a reduced graph should perform substantial faster for practical instances. If $|E^{\mathrm{g}}| = O(r)$ the complexity reduces to $O(r d^3_{\max})$.

6.4 Exact Methods for the GMEBCN problem

Survivable network design problems are often formulated either as cut formulations or using multi-commodity flow based formulations [27]. They will be utilized in this chapter as well

to describe the GMEBCN problem.
Cut based formulations for the minimum edge biconnected network problem are based on the observation that the minimum cut in such a network contains at least two edges.
In order to extend a cutset formulation for the MEBCN problem [22] to the GMEBCN problem, the following binary variables are used.

$$x_e = \begin{cases} 1 & \text{if the edge } (u,v) \in E \text{ is included in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall e = (u,v) \in E$$

$$z_v = \begin{cases} 1 & \text{if the node } v \text{ is connected in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V$$

Using these variables and the cutset $\delta(S)$ as defined in Chapter 2.1 the GMEBCN problem can be formulated as in the following *generalized cutset formulation* where inequality (6.4) ensures that the minimal cut between any two selected nodes consists of at least two edges.

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{6.2}$$

$$\text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, r \tag{6.3}$$

$$x(\delta(S)) \geq 2(z_i + z_j - 1) \qquad \forall i \in S, j \notin S, \emptyset \subset S \subset V \tag{6.4}$$

$$x_e \in \{0,1\} \tag{6.5}$$

$$z_v \in \{0,1\} \tag{6.6}$$

Due to the similar structure it seems reasonable to describe the GMEBCN problem by adapting the "Local-Global" formulation for the GMST as introduced by Pop [30]. Once the global polytype for the GMEBCN problem is described, all constraints to compute the cheapest possible local solution for a given global solution can be adopted.
For formulating the *local global multicommodity based formulation* the following binary variables are used in addtion to the ones defined before.

$$y_{ij} = \begin{cases} 1 & \text{if cluster } V_i \text{ is connected to cluster } V_j \text{ in the global graph} \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i,j = 1, \ldots, r, \\ i \neq j \end{array}$$

$$f_{ij}^k = \begin{cases} 1 & \text{if a flow } f \text{ of commodity } k \text{ exists on the directed edge } (i,j) \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i,j = 1, \ldots, r, \\ \forall k = 2, \ldots, r \end{array}$$

$$g_{ij}^k = \begin{cases} 1 & \text{if a flow } g \text{ of commodity } k \text{ exists on the directed edge } (i,j) \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i,j = 1, \ldots, r, \\ \forall k = 2, \ldots, r \end{array}$$

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{6.7}$$

$$\text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, r \tag{6.8}$$

$$\sum_{u \in V_i, v \in V_j} x_{uv} = y_{ij} \qquad \forall i, j = 1, \ldots, r,\ i \neq j \tag{6.9}$$

$$\sum_{u \in V_i} x_{uv} \leq z_v \qquad \forall i = 1, \ldots, r,\ \forall v \in V \setminus V_i \tag{6.10}$$

$$\sum_i f^k_{i,j} - \sum_l f^k_{j,l} = \begin{cases} -1 & \text{if } j = 1 \\ 1 & \text{if } j = k \\ 0 & \text{else} \end{cases} \qquad \forall k = 2, \ldots, r,\ \forall j = 1, \ldots, r \tag{6.11}$$

$$\sum_i g^k_{i,j} - \sum_l g^k_{j,l} = \begin{cases} -1 & \text{if } j = 1 \\ 1 & \text{if } j = k \\ 0 & \text{else} \end{cases} \qquad \forall k = 2, \ldots, r,\ \forall j = 1, \ldots, r \tag{6.12}$$

$$y_{i,j} \geq f^k_{i,j} \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.13}$$

$$y_{i,j} \geq g^k_{i,j} \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.14}$$

$$f^k_{i,j} + f^k_{j,i} \leq 1 \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.15}$$

$$g^k_{i,j} + g^k_{j,i} \leq 1 \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.16}$$

$$f^k_{i,j} + g^k_{i,j} \leq 1 \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.17}$$

$$f^k_{i,j} + g^k_{j,i} \leq 1 \qquad \forall i, j = 1, \ldots, r,\ \forall k = 2, \ldots, r \tag{6.18}$$

$$x_e,\ z_v \geq 0, \qquad \forall e = (i, j) \in E, \forall v \in V \tag{6.19}$$

$$f^k_{i,j} \geq 0,\ g^k_{i,j} \geq 0 \qquad 1 \leq k, i, j \leq r \tag{6.20}$$

$$y_{lr} \in \{0, 1\} \tag{6.21}$$

This formulation is based on sending two different types of flows (f, g) from a single source V_1 to every other cluster. Flows dedicated to different clusters are distinguished by their commodity k, i. e. $f^k_{i,j}$ will be consumed by cluster V_k.
Therefore conditions 6.11 and 6.12 ensure that one unit of both flows is produced by V_1, preserved by every cluster they are not dedicated for, and consumed by the one having a cluster number equal to the corresponding commodity k. To ensure edge biconnectivity, the circulation of two flows of the same commodity on a single edge must be prohibited. As flows are directed, the circulation of these must be additionally forbidden for the opposite directions of a single edge (i. e. $f^{\mathrm{k}}_{\mathrm{i,j}} = 1$ and $f^{\mathrm{k}}_{\mathrm{j,i}} = 1$ is not allowed). This is achieved by inequalities 6.15, 6.16, 6.17, and 6.18. To describe a global edge biconnected network, each global edge must be included in the solution, where at least one flow circulates, which is ensured by inequalities 6.13 and 6.14.
Finally constraint 6.8 guarantees that only one node is selected per cluster, equality 6.9 allows local edges only between nodes of clusters which are connected in the global graph, and inequality 6.10 ensures that edges only connect nodes which are selected.

7. Variable Neighborhood Search for the GMEBCN Problem

In this chapter, two new VNS approaches for the GMEBCN problem will be described in detail. First, Chapter 7.1 presents a constructive heuristic to produce initial solutions, which is followed by Chapter 7.2 describing four different neighborhoods as well as the search techniques applied to them that try to bypass the high complexity of the GMEBCN problem compare Chapter 6.2). Following the ideas of these fundamental neighborhoods, Chapter 7.3 describes two additional neighborhoods that take advantage of the graph reduction technique described in Chapter 6.3. Afterwards, the concrete arrangement of the neighborhoods for both VNS approaches is described in Chapter 7.4. Finally, Chapter 7.5 describes the shaking procedure, and Chapter 7.6 explains a memory function similar to the one used for the GMST problem.

7.1 Initialization

As no previous research has been done on the GMEBCN problem using heuristic methods (compare Chapter 3.2), no algorithm for computing a feasible solution to the GMEBCN problem has been described yet. It is not possible to extend an algorithm for the classical MEBCN problem to solve the GMEBCN problem due to the high complexity of the classical MEBCN problem compared to the classical MST problem. Therefore a specialized construction heuristic based on combining the ideas of the Improved Kruskal Heuristic (IKH) as used to generated feasible solutions to the GMST problem (see Algorithm 9) with the well known Christofides Heuristic for the Traveling Salesperson Problem (TSP) has been developed. This "Adapted Christofides Heuristic (ACH)" which is shown in Algorithm 15 starts with computing an initial GMST $G = \langle V, E \rangle$ using IKH (see Algorithm 9).

Algorithm 15: Adapted Christofides Heuristic

$G = \langle V, E \rangle$ = feasible GMST computed with IKH *//see Algorithm 9*
S = set of all nodes in G having odd degree
E_{M} = set of edges in a greedy matching of S *//see Algorithm 16*
$G = \langle V, E \cup E_{\mathrm{M}} \rangle$
compute edge biconnected components of G *//see Algorithm 6*
if G has more than one edge biconnected components **then**
  add cheapest edges between all pairs of edge biconnected components
optimize solution *//see Algorithm 17*

Similar to the Christofides Heuristic for the TSP, ACH determines all nodes having odd degree. As the number of nodes having odd degree is even for every graph, a perfect matching on those nodes always exists. However for ACH, which augments the edges of G with a perfect matching, we must prevent including edges that are also part of G. As ACH generates a matching in a greedy way (shown in Algorithm 16) with respect to the edge costs, it might fail to find a perfect matching. To ensure edge biconnectivity in this seldom case, ACH determines the edge biconnected components of G after augmenting it with the matching edges using Algorithm 6 and adds the cheapest possible edges between any two of these components. As the resulting graph is not necessarily edge minimal, a greedy optimization process (see Algorithm 17) is attached afterwards to eliminate the redundant edges.

Algorithm 16: compute matching (GMST $G = \langle V, E \rangle$)

$E_{\mathrm{M}} = \emptyset$
$V_{\mathrm{o}} = \{v \in V | \deg(v)\ is\ odd\}$
$E_{\mathrm{o}} = \{(u, v) \in E | u, v \in V_{\mathrm{o}}\}$
sort E_{o} with increasing costs, i. e. $c(e_1) \leq \ldots c(e_{\mathrm{n}})$
$i = 1$
while $V_{\mathrm{o}} \neq \emptyset \wedge i \neq |E_{\mathrm{o}}|$ **do**
  // current edge $e_i = (u_i, v_i)$
  if $u_i \in V \wedge v_i \in V$ **then**
    if $e_i \notin E$ **then**
      $E_{\mathrm{M}} = E_{\mathrm{M}} \cup \{e_i\}$
      $V_{\mathrm{o}} = V_{\mathrm{o}} \setminus \{u_i, v_i\}$
  i++
  return E_{M}

As Algorithm 16 computes a matching on a solution to the GMST which consists of r nodes, its complexity is $O(r^2 \log r)$ if all nodes have odd degree. Determining the edge biconnected components as well as adding cheapest edges between any two of them can be done within $O(r^2)$. Therefore only IKH and the final optimization procedure have to be considered to determine the overall complexity of ACH.
The worst case for Algorithm 17 occurs in the very unlikely case that all nodes have degree greater than two and $O(|F|)$ edges can be removed without violating the edge biconnectivity property. Under these circumstances its complexity is bounded by $O(|F|(|P|+|F|)+|F|^2(|P|+|F|) = O(|F|^3)$. As $|F| \leq r^2$ this estimation leads to a overall complexity of $O(r^6)$ for Algorithm 17.
However, IKH computes a solution to the GMST problem which consists of exactly $r - 1$ edges and Algorithm 16 therefore adds at most $\lfloor \frac{r-1}{2} \rfloor$ edges. Obviously, even if Algorithm 16 fails to find a perfect matching, the resulting graph consists of at most two edge biconnected components. Therefore only a single edge need to be added to ensure edge biconnectivity. Due to these considerations the amount of edges for the initial solution $S = \langle P, F \rangle$ passed to Algorithm 16 is bounded by $O(r)$, which reduces its complexity to $O(r^3)$ for the worst case that might occur in practice.
Additionally, the set of possible removable edges can be restricted by the observation that an edge e incident to a node v having a degree less than three can never be removed, as $\deg(v)$ would be less than two afterwards which obviously violates the edge biconnectivity property.

Algorithm 17: optimize (solution $S = \langle P, F \rangle$)

```
R = ∅ //set of redundant edges
forall edges (u, v) ∈ F do
    if deg(u) ≥ 3 ∧ deg(v) ≥ 3 then
        if S' = ⟨P, F \ {(u, v)}⟩ is edge biconnected then
            R = R ∪ {(i, j)}
while R ≠ ∅ do
    (u, v) is an arbitrary edge of R
    E = E \ {(u, v)}
    R = R \ {(u, v)}
    forall edges (u, v) ∈ R do
        if deg(u) ≤ 2 ∨ deg(v) ≤ 2 then
            R = R \ {(u, v)}
        else
            if E \ {(u, v)} is not biconnected then
                R = R \ {(u, v)}
```

Therefore the set of removable edges will typically be small and the computation of Algorithm 17 should perform considerable faster than $O(r^3)$ in practice.
Due to these considerations, ACH has a computational complexity of $O(|V|^2+|E|\log|E|+r^3)$.

7.2 Neighborhoods

Similar to the GMST problem, the neighborhoods presented in this chapter try to handle the problem from different point of views. As shown in Chapter 6.2, simply representing a solution by either the set of used nodes $P = \{p_1, \ldots, p_r\}$ or by the set of global edges $E^g \subseteq V^g \times V^g$, $V^g = \{V_1, \ldots, V_r\}$ and finding the optimal solution $S = \langle P, F \rangle$ according to this information cannot be done with reasonable computational effort.
Therefore a solution to the following neighborhoods is represented by both, the set of used nodes as well as by its edges (either local or global, depending on the concrete neighborhood). In that way no decoding procedure is required to obtain the whole solution.

7.2.1 Simple Node Optimization Neighborhood

The Simple Node Optimization Neighborhood (SNON) tries to overcome the hardness of selecting the best nodes $P = \{p_1, \ldots, p_r\}$ for each cluster V_i, $i = 1, \ldots, r$ for a given set of global edges. Moving to a solution in this neighborhood means to change the used node p_i of exactly one cluster V_i to another node p_i' of the same cluster. The whole solution will not be optimized with respect to the selected edges. Therefore if $I = \{p_j \in P | (p_i, p_j) \in F\}$ is the set of nodes incident to p_i in $S = \langle P, F \rangle$, these nodes will be incident to p_i' in the new solution $S' = \langle P', F' \rangle$ with $P' = P \setminus \{p_i\} \cup \{p_i'\}$, $p_i, p_i' \in V_i, p_i \neq p_i'$ and $F' = F \setminus \{(p_i, p) | p \in I\} \cup \{(p_i', p) | p \in I\}$. Figure 7.1 shows the exchange of a node in cluster V_6.
Due to the fact that updating the costs after a move can be done in an incremental way,

the whole neighborhood can be examined in $O(|V|)$ time. Algorithm 18 illustrates the search process in this neighborhood in detail.

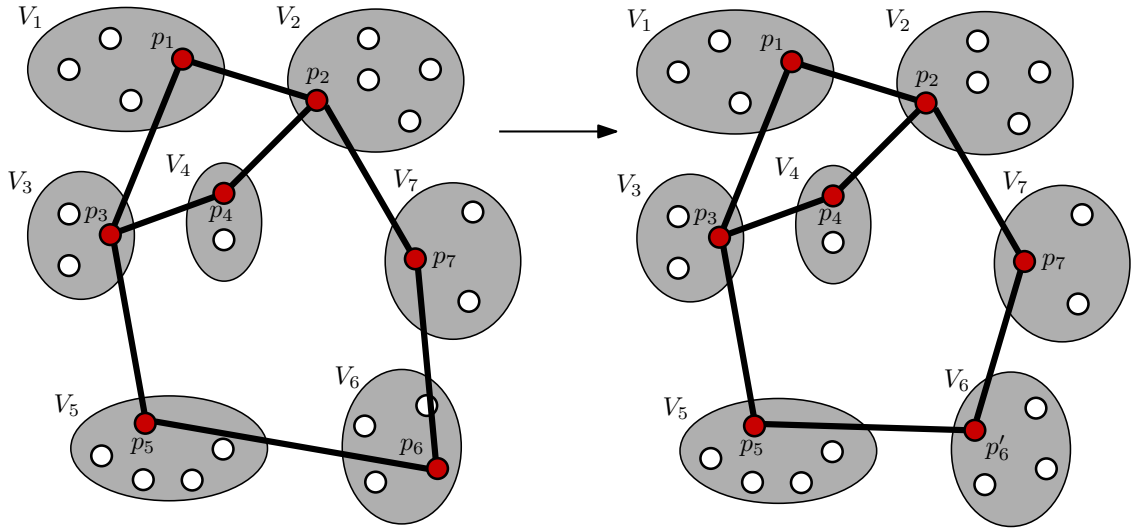


Fig. 7.1: A Node Optimization Move changing the used node of V_6 from p_6 to p'_6.

Algorithm 18: Simple Node Optimization (solution S)

```
for i = 1, ..., r do
    p_i = used node of cluster V_i
    forall v ∈ V_i do
        if v ≠ p_i then
            change used node of cluster V_i to v
            if current solution better than best then
                save current solution as best
            restore initial solution
restore and return best solution
```

7.2.2 Node Re-Arrangement Neighborhood

As for SNON this neighborhood is based on an operation that obtains the edge biconnectivity for a given initial solution. In contrast to SNON, the Node Re-Arrangement Neighborhood (NRAN) is focused on the arrangement of nodes instead of exchanging the used nodes of clusters. Therefore, a neighbor solution differs from the current solution by one swap move between exactly two nodes of different clusters, see Figure 7.2. Swapping p_i and p_j ($p_i \in V_i, p_j \in V_j, i \neq j$) is defined as follows.
Consider a solution $S = \langle P, F \rangle$ and let $I_i = \{p \in P | (p_i, p) \in F\}$ be the set of nodes incident to p_i, and $I_j = \{p \in P | (p_j, p) \in F\}$ the set of nodes incident to p_j in S. A move within SNON transforms $S = \langle P, F \rangle$ into a new solution $S' = \{P, F'\}$ with $F' = F \setminus I_i \setminus I_j \cup \{(p_i, p) | p \in I_j\} \cup \{p_j, p) | p \in I_i\}$. In other words all edges that were incident to p_i are incident to p_j

afterwards, and vice versa.
Updating the objective value for a single move means to subtract the costs of the original edges and add the costs of the new ones, which can be done in constant time. Therefore a complete evaluation of NRAN which is shown in Algorithm 19 can be examined in $O(r^2)$, as there are $O(r^2)$ possible moves.

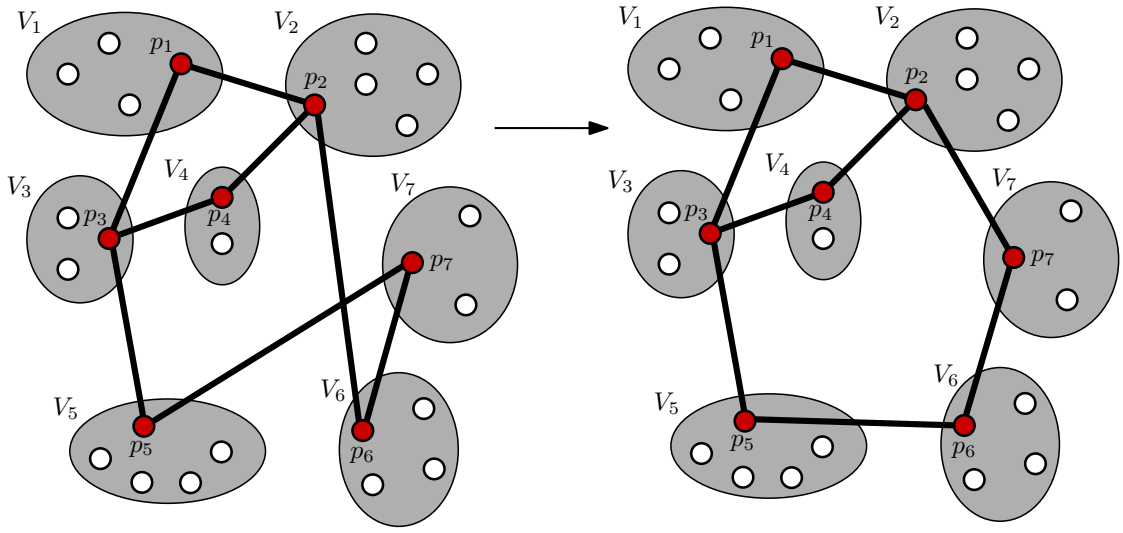


Fig. 7.2: A Node Re-Arrangement Move, swapping p_6 and p_7.

Algorithm 19: Node Re-Arrangement (solution S)

```
for i = 1, ..., r do
    for j = i + 1, ..., r do
        swap nodes of cluster i, j
        if current solution better than best then
            save current solution as best
        restore initial solution
restore best solution
```

7.2.3 Edge Augmentation Neighborhood

Instead of optimizing a given solution while not alternating the connection properties as for SNON and NRAN, the Edge Augmentation Neighborhood (EAN) explores a wider area by adapting the idea of the Edge Exchange Neighborhood introduced for the GMST problem in Chapter 4.2.2.
As edge biconnectivity must be guaranteed after a move, simply exchanging an edge by another one is not possible. Therefore EAN first augments the actual solution $S = \langle P, F \rangle$ by adding a single edge $e \notin F$ to it. The resulting graph $S' = \langle P, F \cup \{e\} \rangle$ is certainly not edge minimal, i. e. there is at least one edge that can be removed without violating the edge biconnectivity property.

Afterwards EAN uses the optimization procedure as introduced in Algorithm 17 to determine the set redundant edges E_r of S' and heuristically remove as many redundant edges as possible from E_r. Needless to say, removing e has to be prevented during this optimization process as this would directly lead to the original solution S. Using this scheme, EAN theoretically consists of all solutions S' that are reachable from S by adding a single edge and removing edges afterwards, until the graph is edge minimal edge biconnected again. However, the search process does not iterate through all these possible solutions but removes the redundant edges heuristically instead.
In the example provided by Figure 7.3, the initial solution is augmented by $e = (p_4, p_5)$ which leads to S' where at least one edge out of $E_r = \{(p_2, p_4), (p_3, p_4), (p_3, p_5), (p_4, p_5)\}$ can be removed. During the optimization process, (p_2, p_4) and (p_3, p_5) are removed which results in a probably better solution S''.

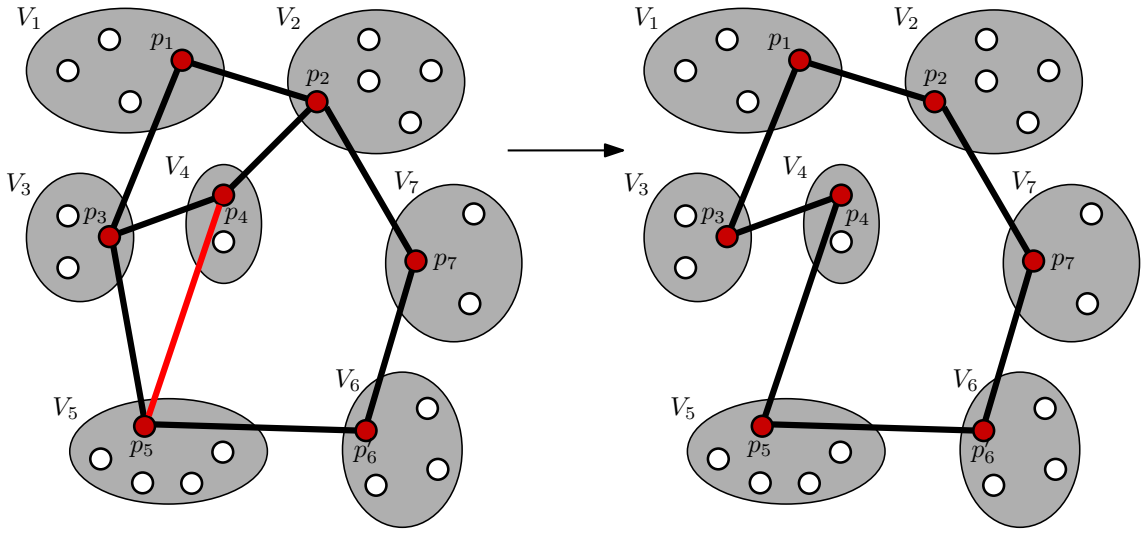


Fig. 7.3: An Edge Augmentation Move, adding (p_4, p_5) and removing the redundant edges (p_2, p_4) and (p_3, p_5).

As it turned out that a considerable amount of augmenting edges e ends in a graph where only e itself is removable, it is crucial for the efficiency of EAN to determine as many of these cases as possible in advance.

Theorem 4: Let $S = \langle P, F \rangle$ be an edge biconnected graph which is edge minimal (i. e. no edges can be removed without violating the edge biconnectivity property) and $F_p = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)\}$, $F_p \subseteq F$ be a path in S $(n \geq 2)$ with disjunct inner nodes (i. e. $v_1 \neq v_2 \neq \cdots \neq v_{n-1}$) and for which the condition $\deg(v_i) = 2$ holds for all inner nodes $v_i,\ i = 1, \ldots, n-1$ (see Definition 8).

After adding a single edge e_a between two arbitrary inner nodes of F_p (i. e. $e_a = (v_i, v_j), i, j \in$

$\{1,\ldots,n-1\},\ e_\mathrm{a} \notin F_p$) no edge apart from e_a can be removed from $S' = \langle P, F \cup \{e_\mathrm{a}\}\rangle$ without violating the edge biconnectivity property.

Proof Let $e_\mathrm{a} = (v_i, v_j)$ be the edge inserted in S and $e_\mathrm{r} = (v_a, v_b)$, $e_\mathrm{a} \neq e_\mathrm{r}$ be removable as shown in Figure 7.4.

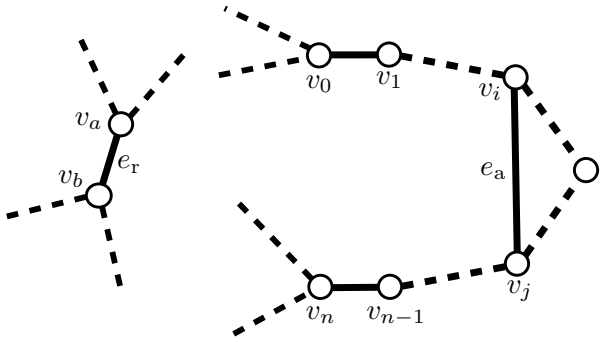


Fig. 7.4: Augmenting a path.

From $e_\mathrm{r} = (v_\mathrm{a}, v_\mathrm{b})$ being removable without violating the edge biconnectivity property follows directly that two edge disjoint paths $P_1 = \{(v_\mathrm{a}, v_1'), (v_1', v_2'), \ldots, (v_{l'}, v_\mathrm{b})\}$ and $P_2 = \{(v_\mathrm{a}, v_1''), (v_1'', v_2''), \ldots, (v_{\mathrm{m}''}, v_\mathrm{b})\}$ must exist in $S' \setminus \{e_\mathrm{r}\}$.
As S was edge minimal the following holds: $e_\mathrm{a} \in P_1 \vee e_\mathrm{a} \in P_2$. Otherwise both P_1 and P_2 would be in S and e_r could therefore be removed in S too. Let $e_\mathrm{a} \in P_1$ without loss of generality.
Obviously no edge within F_p can be removed because if $e_\mathrm{r} \in F_\mathrm{p}$, then at least one node in $\{v_\mathrm{a}, v_\mathrm{b}\}$ is incident to only two edges in S' (we assume $\deg_{S'}(v_\mathrm{a}) = 2$ without loss of generality). Therefore $\deg_{S' \setminus \{e_\mathrm{r}\}}(v_\mathrm{a}) = 1$ which is a contradiction to the edge biconnectivity condition.
Due to $e_\mathrm{a} \in P_1$ and $e_\mathrm{r} \notin F_\mathrm{p}$ we can conclude that $(v_0, v_1) \in P_1 \wedge (v_{\mathrm{n}-1}, v_\mathrm{n})\} \in P_1$. Therefore these edges cannot be part of P_2. Due to this we can conclude that another path $P_3 = P_1 \setminus \{e_\mathrm{r}\} \cup P_\mathrm{o}$, with $P_\mathrm{o} = \{(v_\mathrm{i}, v_{\mathrm{i}+1}, \ldots, (v_{\mathrm{j}-1}, v_\mathrm{j})\}$ exists in the original solution S which is edge disjoint to P_2.
Therefore two edge disjoint paths P_2 and P_3 exist in S and we can remove e_r in S without introducing bridges which is a contradiction to S being edge minimal. □

Good solutions to the GMEBCN problem usually consist of only few clusters with high node degrees and quite long paths consisting of nodes having degree two between them, this observation turns out to be important for practical instances while it unfortunately does not reduce the computational complexity in worst case. Using this restriction, EAN can be described as given in Algorithm 20.
As Algorithm 17 has a complexity of $O(r^3)$, the worst case complexity of a complete evaluation of EAN is $O(r^5)$, which is a very bad estimation for its running time, as good solutions to the GMEBCN problem usually do not lead to situations similar to the theoretical worst case for Algorithm 17.

Algorithm 20: Edge Augmentation (solution S)

```
for i = 1, ..., r − 1 do
    for j = i + 1, ..., r do
        if (i, j) ∉ F then
            if i and j are not inner nodes of path then
                add (i, j)
                optimize(S) //see Algorithm 17
                if current solution better than best then
                    save current solution as best
                restore initial solution
restore best solution
```

7.2.4 Node Exchange Neighborhood

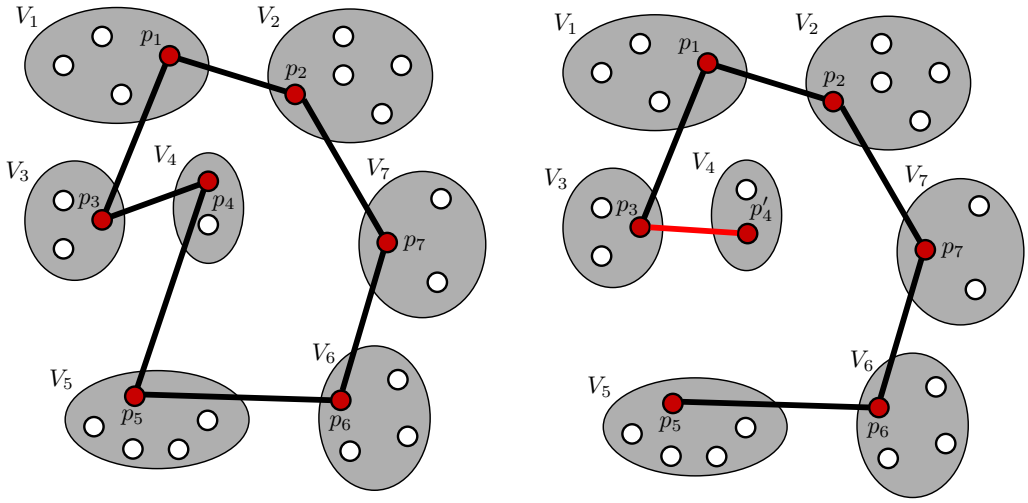
The Node Exchange Neighborhood (NEN) optimizes a solution by applying changes on both the used nodes as well as on the edges connecting them. Similar to EAN, optimization regarding the edges is done in a heuristic way. A solution for NEN may differ by exactly one used node and theoretically an arbitrary number of edges.
A single move within the Node Exchange Neighborhood (NEN) is accomplished by first exchanging the representative node p_i of a cluster V_i to p_i' and removing all edges that were incident to p_i which leads to a graph that consists of at least two components. First of all NEN, which is given in Algorithm 21, reconnects the graph by adding the cheapest edges between any pair of components (there are at least two and at most $\deg(p_i) + 1$ components to connect). Once this step is completed, the edge biconnectivity property must be restored which can principally be done by simply determining all bridges and adding edges between any two edge biconnected components. Finally redundant edges are removed, using the optimization procedure as given in Algorithm 17.

Algorithm 21: Node Exchange (solution S)

```
for i = 1, ..., r do
    forall v ∈ V_i \ p_i do
        remove all edges incident to p_i
        change used node p_i of cluster V_i to v
        add cheapest edges between any two graph components
        restore biconnectivity //see Algorithm 22
        optimize(S) //see Algorithm 17
        if current solution better than best then
            save current solution as best
        restore initial solution
restore best solution
```

Figure 7.5 shows a worst case example of NEN by means of the amount of bridges to be considered. It is essential to apply a more clever bridge covering strategy as the optimization process might take plenty of time when a lot of edges are removable after many edges have been added to cover all bridges. Unfortunately, situations similar to this worst case scenario appear frequently when dealing with concrete solutions. This is due to the fact that removing

an inner node of a path whose inner nodes have degree two, does lead to a graph where all remaining edges of this path are bridges.



a) initial solution

b) after exchanging p_4 by p'_4 and reconnecting V_4, all edges are bridges

Fig. 7.5: Worst Case Example for NEN, exchanging p_4 to p'_4.

Algorithm 22 shows one possible way to reduce the number of edges to be added. It first determines all nodes with degree one in the current solution and connects each one with its cheapest partner. In case only one node v with degree one exists it is connected with the first reachable node having degree greater than two, which is determined by starting a depth first search at v. This strategy helps to cover a lot of bridges by single edges. Remaining bridges are covered by simply adding the cheapest edges between any pair of edge biconnected components.

Algorithm 22: biconnect(solution $S = \langle P, F \rangle$)

```
S_E = {p ∈ P | deg(p) = 1}
if |S_E| = 1 then
    v is the first node with deg(v) ≥ 3 on the path originated at u ∈ S_E
    F = F ∪ {(u, v)}
else
    forall u ∈ S_E do
        select v ∈ S_E with minimal c(u, v)
        F = F ∪ {(u, v)}
calculate edge biconnected components
if S has more than one edge biconnected component then
    add cheapest edges between all pairs of biconnected components
```

Algorithm 17 is definitely the most expensive part of NEN by means of computational complexity which is therefore bounded by $O(|V|r^3)$. Even when using the more clever bridge

covering strategy, NEN is still a rather expensive neighborhood, therefore its exploration is aborted after a certain time limit, similar to RNEN2, using the so-far best neighbor.

7.3 Neighborhoods based on Graph Reduction

Combining the knowledge of the graph reduction technique introduced in Chapter 6.3 with the ideas of the so far described neighborhood operations (see Chapter 7.2) leads to two obvious combinations of these which are described as follows.
Even if reducing the graph of a solution is not too expensive by means of computational effort, it is crucial to avoid applying the complete reduction procedure after every single move within a neighborhood. In the following Extended Node Optimizing Neighborhood, it is sufficient to apply the reduction procedure once to the initial solution while the more complicated Cluster Re-Arrangment Neighborhood shows how and in which cases a complete recomputation can be avoided and how it is done.

7.3.1 Extended Node Optimization Neighborhood

The Extended Node Optimization Neighborhood (ENON) is a straightforward extension of SNON as introduced in Section 7.2.1. In ENON a solution is represented by the set of used nodes within the relevant clusters of the reduced graph S_{red} and the set of edges in S.
Therefore, ENON starts by computing the reduced graph S_{red}, which corresponds to its initial solution S. ENON consists of all solutions S'_{red} that differ from S_{red} by at most two used nodes in the relevant clusters, while the connections between clusters are not changed. Used nodes of irrelevant clusters are selected in an optimal way when a solution is decoded again. Compared to SNON this neighborhood has two advantages. First of all it is possible to change the used nodes of two clusters because of the significant smaller graph in a reduced solution compared to the original solution. Additional to that the used nodes of all irrelevant clusters that do not occur in the reduced solution are selected in the best possible way.
As the reduction process has to be applied only once and updating the objective function can be done with constant effort, ENON (see Algorithm 23) which has complexity $O(r^2 d_{\max}^3)$, is a very efficient neighborhood.

Alternative Implementation of ENON

Other than the so far described functionality ENON is able to solve the problem of selecting the best used nodes exactly if the amount of clusters in the reduced solution is below some predefined threshold. This exact calculation is done by the following integer program, which is just a small adaption of Pop's Local - Global formulation for the GMST problem [30]. As in Chapter 4.2.3 the following binary variables are used.

Algorithm 23: Extended Node Optimization (solution S)

```
S_red = reduced graph of S  // see Algorithm 14
r_g = number of clusters in S_red
for i = 1, ..., r_g − 1 do
    u'_i = used node of cluster V_i
    for j = i + 1, ..., r_g do
        v'_j = used node of cluster V_j
        forall u ∈ V_i do
            if u ≠ u'_i then
                change used node of cluster V_i to u
                forall v ∈ V_j do
                    change used node of cluster V_j to v
                    if current solution better than best then
                        save current solution as best
                    restore initial solution
restore best solution
```

$$y_{ij} = \begin{cases} 1 & \text{if cluster } V_i \text{ is connected to cluster } V_j \text{ in the reduced graph} \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j = 1, \ldots, |V_{\text{red}}|, \; i \neq j$$

$$x_e = \begin{cases} 1 & \text{if the edge } e \in E_{\text{red}} \text{ is included in the reduced graph} \\ 0 & \text{otherwise} \end{cases} \qquad \forall e = (u, v) \in E_{\text{r}}$$

$$z_v = \begin{cases} 1 & \text{if the node } v \text{ is connected in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V_{\text{red}}$$

With these variables the following ILP can be used to determine the best local solution of a given reduced graph.

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{7.1}$$

$$\text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, |V_{\text{red}}| \tag{7.2}$$

$$\sum_{u \in V_i, v \in V_j} x_{uv} = y_{ij} \qquad \forall i, j = 1, \ldots, |V_{\text{red}}|, \; i \neq j \tag{7.3}$$

$$\sum_{u \in V_i} x_{uv} \leq z_v \qquad \forall i = 1, \ldots, |V_{\text{red}}|, \; \forall v \in V_{\text{red}} \setminus V_i \tag{7.4}$$

$$x_e, \; z_v \geq 0 \qquad \forall e = (i, j) \in E_{\text{red}}, \forall v \in V_{\text{red}} \tag{7.5}$$

However tests showed that the amount of clusters in a reduced solution is very low for all considered instances. Due to that the improvement gained by using the ILP above instead of

doing a heuristic optimization is too low to cover its computational effort. Therefore it was not used for testing (i. e. the maximum amount of clusters for exact solving has been set to zero).

7.3.2 Cluster Re-Arrangement Neighborhood

The Cluster Re-Arrangement Neighborhood (CRAN) extends NRAN (see Chapter 7.2.2) by operating on a reduced graph. A solution is represented by the global edges between all clusters and the used nodes of all relevant clusters (i. e. the ones that are included in the reduced graph). The CRAN of a solution S consists of all solutions S' that differ from S by swapping exactly two nodes in the same way as done for NRAN, computing the reduced graph, and selecting the best nodes in all irrelevant clusters. More concrete, CRAN swaps nodes within the S and determines the corresponding changes in the reduced graph which is used to retrieve the objective value. Obviously, recomputing all components of the reduced graph can be avoided in the majority of cases. Figure 7.6 visualizes the changes of a reduced graph in case two nodes of degree two, being part of the same global path are swapped. Figure 7.7 shows the changes if two nodes belong to different paths are swapped. If one of the considered nodes is an element of a relevant cluster, the structure of the reduced solution may change significantly and therefore the whole reduction is applied completely.

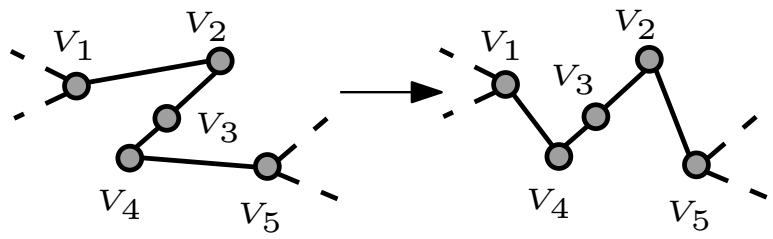


Fig. 7.6: After swapping V_2 and V_4 only the path containing them may change.

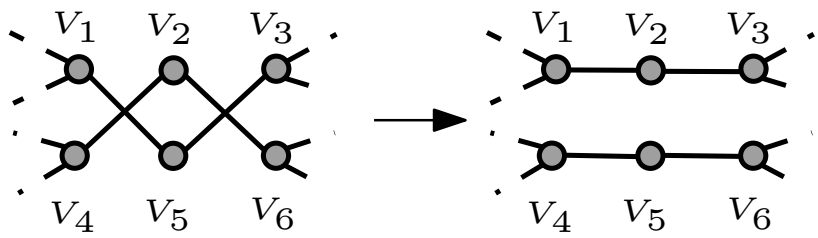


Fig. 7.7: After swapping V_2 and V_5 only the paths containing them may change.

The complexity of evaluating CRAN (Algorithm 24) completely is bounded by $O(r^4 d_{\max}^3)$ if the whole reduction process would be applied after every move. However, CRAN turned out to consume too much time for a complete evaluation for some instances. Therefore, its exploration is aborted after a certain time limit returning the so-far best neighbor instead of following a strict best neighbor strategy.

Algorithm 24: Cluster Re-Arrangement Optimization (solution S)

```
compute reduced graph G_red
for i = 1, ..., r − 1 do
    for j = i + 1, ..., r do
        swap clusters V_i and V_j
        if V_i or V_j is a relevant cluster then
            recompute reduced graph G_red completely
        else
            if V_i and V_j belong to the same global path P then
                update P in G_red
            else
                update the global path containing V_i in G_red
                update the global path containing V_j in G_red
        if current solution better than best then
            save current solution as best
        restore initial solution and G_red
restore best solution
```

7.4 Arrangement of the Neighborhoods

As for the GMST problem the general VNS scheme with VND as local improvement [14, 15] is used. In order to be able to examine the quality of the more complicated neighborhoods based on graph reduction (ENON, CREN), two different VNS approaches are considered. The first one (VND1 - see Algorithm 25) does not use neighborhoods based on graph reduction. Therefore it alternates between SNON, NRAN, EAN, and NEN in this order which has been determined taking both the computational complexity as well as concrete tests into account. VND2, which is shown in Algorithm 26 alternates between SNON, NRAN, CRAN, EAN, and NEN and therefore uses the more sophisticated neighborhoods having higher computational complexity.

7.5 Shaking

Both VNS approaches are effective in selecting used nodes within solutions, but diversity with respect to the included edges should not be neglected. Therefore the shaking procedure for the GMEBCN problem given in Algorithm 27 is based on the Edge Augmentation procedure. Shaking starts by augmenting a single edge and increments the number of edges that are augmented up to a possible maximum of $\lfloor \frac{r}{4} \rfloor$.

7.6 Memory Function

Similar as for the GMST problem in Chapter 4.5 a hash memory to avoid unnecessary iterating through neighborhoods is used. Even if several neighborhoods are not deterministic and might therefore find a better solution using the same initial solution it turned out to be better to apply another neighborhood instead. Therefore the hash memory is used for these too.

Algorithm 25: VND1 (solution $S = \langle P, F \rangle$)

```
l = 1
repeat
    switch l do
        case 1: // SNON
            SimpleNodeOptimization(S) //see Algorithm 18
        case 2: // NRAN
            NodeReArrangmentOptimization(S) //see Algorithm 19
        case 3: // EAN
            EdgeAugmentationOptimization(S) //see Algorithm 20
        case 4: // NEN
            NodeExchangeOptimization(S) //see Algorithm 21
    if solution improved then
        l = 1
    else
        l = l + 1
until l > 4
```

Algorithm 26: VND2 (solution $S = \langle P, F \rangle$)

```
l = 1
repeat
    switch l do
        case 1: // SNON
            ExtendedNodeOptimization(S) //see Algorithm 23
        case 2: // NRAN
            NodeReArrangementOptimization(S) //see Algorithm 19
        case 3: // CRAN
            ClusterReArrangementOptimization(S) //see Algorithm 24
        case 4: // EAN
            EdgeAugmentationOptimization(S) //see Algorithm 20
        case 5: // NEN
            NodeExchangeOptimization(S) //see Algorithm 21
    if solution improved then
        l = 1
    else
        l = l + 1
until l > 5
```

Algorithm 27: Shake (solution $S = \langle P, F \rangle$, size k)

```
for i = 0, ..., k do
    add an arbitrary edge (u, v) to the current solution
optimize solution //see Algorithm 17
```

8. Computational Results for the GMEBCN Problem

In the following a detailed summary and analysis of the computational results for both VNS approaches as well as for the initial values gained by ACH is given. Chapter 8.2 analyzes the individual contributions of all neighborhoods within VND1 and VND2. Testing has been done on most of the instances used for the GMST problem as described in Chapter 5.1.

8.1 Computational Results for the VNS

As no reference values are available, the results of both Variable Neighborhood Search approaches (VNS1, VNS2) for the GMEBCN problem are compared to each other as well as to the initial solutions computed by ACH. The results are shown in Table 8.1 for grouped Euclidean, random Euclidean and non-Euclidean instances and in Table 8.2 for TSPlib based instances which provide instance names, number of nodes, number of cluster, (average) number of nodes per cluster, the average objective values of the final solutions and the corresponding standard deviations over 30 runs. Additionally, relative values grouped by the different sets are illustrated in Figure 8.1, 8.1 and 8.3 where the results of VNS2 are taken as base (100%). The time limit has been set to those used for testing the VNS approach for the GMST problem as it is interesting to see if computation times similar to the GMST problem are reasonable for the GMEBCN problem too. The maximum time limit for a single search of CRAN and NEN has been set to 5s for all tests.

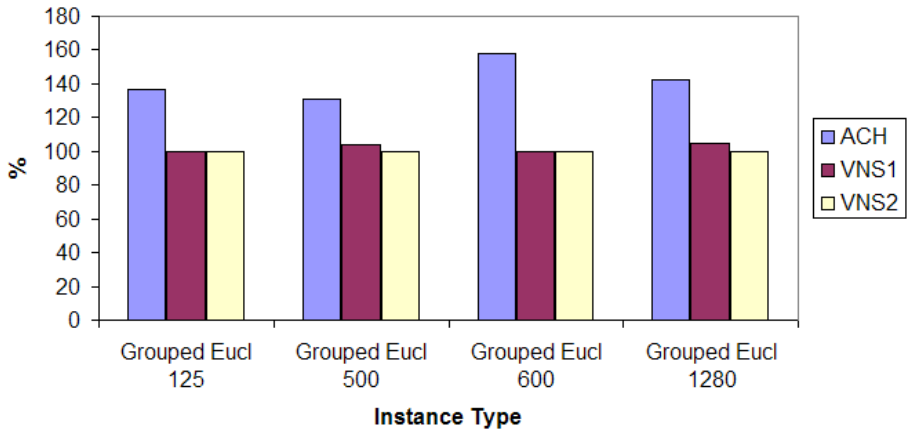


Fig. 8.1: Relative results on grouped euclidean instances for each set (VNS2 = 100%).

Both, VNS1 and VNS2 are able to generate good solutions within the time limit for instances with many nodes per cluster. They need the most computation time for instances with a

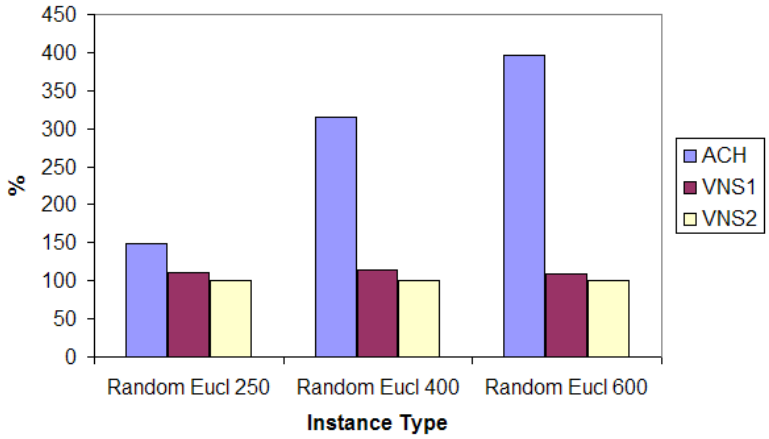


Fig. 8.2: Relative results on random euclidean instances for each set (VNS2 = 100%).

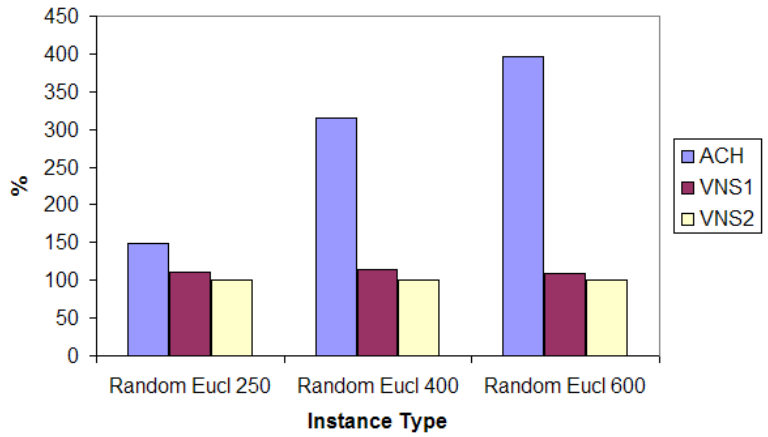


Fig. 8.3: Relative results on non-Euclidean instances for each set (VNS2 = 100%).

high number of clusters, which is the case especially for large TSPlib instances as well es for the grouped Euclidean set with 500 nodes and 100 clusters. The reason for this is probably the number of moves within EAN which is $O(r^2)$. For those instances both approaches might find a better solution if the running time would be increased.
For smaller instances VNS1 performs much faster than VNS2 and it does not need long for finding good solutions, while VNS2 needs considerable effort even for small instances. Interestingly this behavior changes when moving to larger instances. VNS2 benefits from operating on the much smaller reduced graphs for ENON and CRAN which significantly reduced the difference between VNS1 and VNS2 by means of number of iterations. Another unexpected behavior is observed from ACH which generates results with a standard deviation of zero for all but three non-Euclidean instances. As the removement of redundant edges is done in random order within ACH, we can conclude that this order is unimportant in the

majority of cases for solutions generated by ACH.

Comparing the concrete results we can see that both VNS1 and VNS2 are able to improve an initial solution generated by ACH significantly. However ACH produces solutions that are not too bad for TSPlib based and grouped Euclidean instances while the gap between ACH and both VNS approaches increases significantly for random Euclidean instances. Regarding non-Euclidean instances the initial solutions generated by ACH are ten times worse than the final solutions obtained by VNS2.
For TSPlib based instances, VNS1 and VNS2 produce solutions of similar quality. Out of 13 instances of this type VNS1 is better in 7 cases, while VNS2 is better in 6 cases. We can conclude that for those instances having a relatively low number of nodes per cluster the benefit of the graph reduction is relatively low. For grouped Euclidean, random Euclidean and non-Euclidean instances, a completely different result can be observed.
While VNS1 can still produce solutions of similar quality as VNS2 for grouped Euclidean instances, VNS2 is strictly better on all random and all but two non-Euclidean instances. While the gap between VNS1 and VNS2 is relatively small for those instances where VNS1 generated the better results, VNS2 is able to outperform VNS1 significantly especially on random Euclidean instances and on other instance types with a high number of nodes per clusters.

8.2 Contributions of the Neighborhoods

Like the GMST problem, a record of how often each neighborhood was able to improve a current solution has been taken, which allows to analyze how the different neighborhood searches of VNS1 and VNS2 contribute to the whole optimization. Table 8.3 shows the ratios of these numbers of improvements to how often they were called for VNS1, while Table 8.4 contains the same values for VNS2. These values are grouped by the different input instances. Additionally, all values included in those tables are illustrated in Figures 8.4, 8.5, and 8.6 for VNS1 and in Figures 8.7, 8.8, and 8.9 for VNS2.
For VNS1, each neighborhood search contributes substantially to the whole success. While the effectivity of all four neighborhoods is nearly equal for TSPlib based instances, the results are ambiguous for other instance types. For grouped Euclidean instances, SNON is the most effective one while NRAN and EAN produce similar results. The high values of all neighborhoods for grouped Euclidean instances (Grouped Euclidean 500 and 1280) with a high amount of clusters is probably caused by the fact that much fewer iterations could be carried out, and a considerable amount of the total computation time was spent before reaching the first solution that is a local optimum for all four neighborhood structures. For random Euclidean instances, NRAN and EAN were the most effective neighborhoods in terms how often they could improve a solution. Opposite to that, NRAN produces the worst results for non-Euclidean instances. As EAN and NEN were above average for non-euclidean instances it can be concluded that the general structure of the initial solutions computed by ACH is quite bad for those instances.
As for VNS1 all neighborhood searches used by VNS2 contribute to the optimization. Mostly notable CRAN is the most effective one for all instance types. Furthermore, we can see that NRAN could not contribute much to the optimization in case of non-Euclidean instances which has been already observed for VNS1. However, CRAN which extends NRAN could

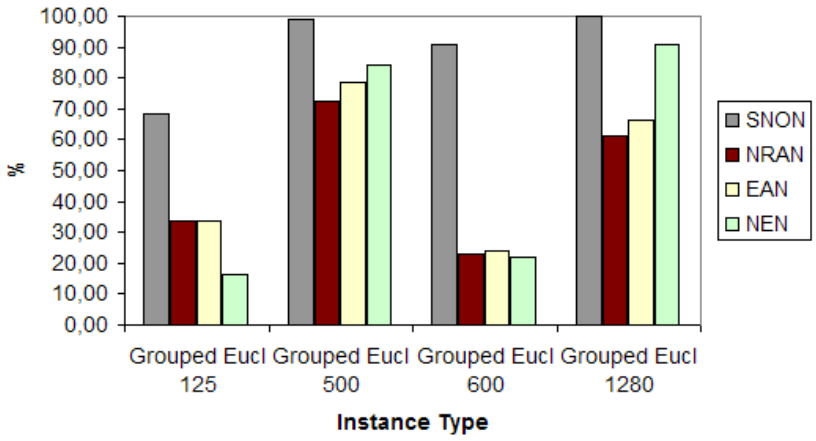


Fig. 8.4: Contributions of the neighborhoods on grouped euclidean instances for VNS1.

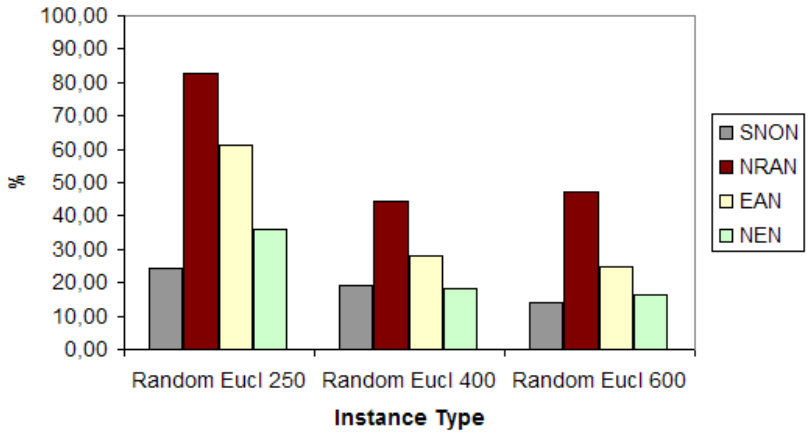


Fig. 8.5: Contributions of the neighborhoods on random euclidean instances for VNS1.

maintain its strength even for instances of those type. Considering that the optima NEN operates on are already local optima with respect to all other neighborhood structures its improvement ratios are quite remarkable.

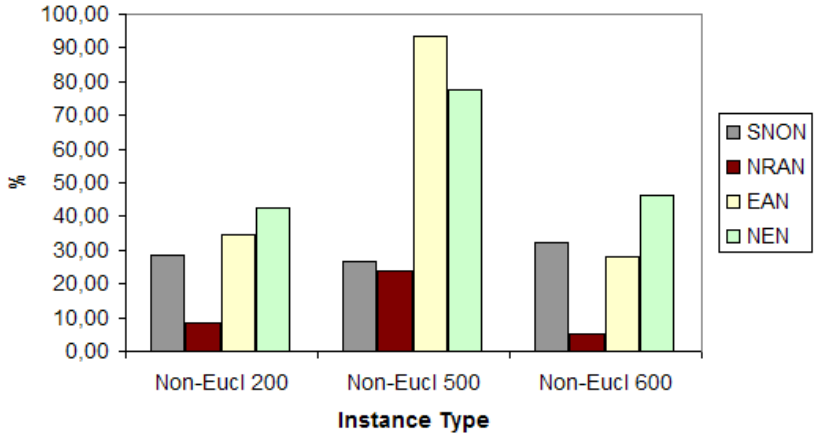


Fig. 8.6: Contributions of the neighborhoods on non-euclidean instances for VNS1.

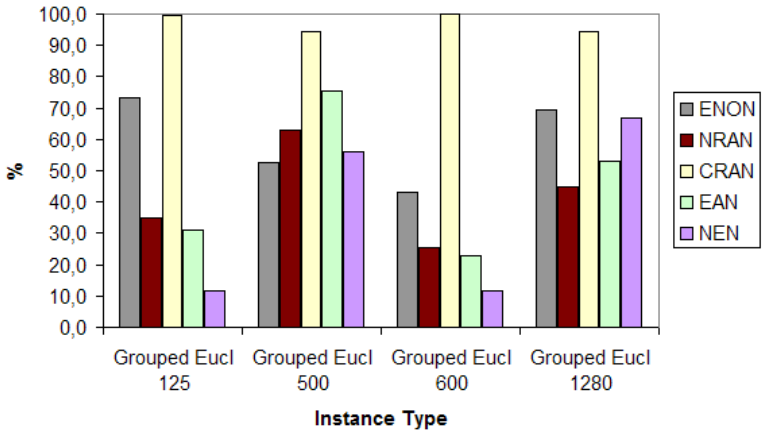


Fig. 8.7: Contributions of the neighborhoods on grouped euclidean instances for VNS2.

Instances				ACH		VNS1		VNS2	
Set	$\|V\|$	r	$\|V\|/r$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
	125	25	5	227.1	0.00	**159.5**	0.00	159.7	0.33
Grouped Eucl 125	125	25	5	209.5	0.00	**163.5**	0.00	**163.5**	0.00
	125	25	5	230.9	0.00	**166.1**	0.00	**166.1**	0.00
	500	100	5	939.6	0.00	717.6	25.78	**712.2**	16.33
Grouped Eucl 500	500	100	5	993.6	0.00	799.0	35.40	**736.7**	33.86
	500	100	5	943.7	0.00	761.7	37.21	**751.5**	38.87
	600	20	30	172.6.	0.00	**105.1**	0.00	105.8	2.66
Grouped Eucl 600	600	20	30	151.0.	0.00	**105.2**	0.00	105.3	0.07
	600	20	30	179.0	0.00	**107.5**	0.00	**107.5**	0.00
	1280	64	8	590.2	0.00	436.8	36.47	**402.2**	22.90
Grouped Eucl 1280	1280	64	8	585.4	0.00	404.6	24.26	**399.9**	14.22
	1280	64	8	562.5	0.00	433.3	35.75	**417.0**	15.83
	250	50	5	4398.9	0.00	3746.6	123.22	**3521.8**	108.83
Random Eucl 250	250	50	5	5110.0	0.00	3661.8	166.90	**3227.5**	222.37
	250	50	5	4975.1	0.00	3403.2	270.07	**3015.2**	139.05
	400	20	20	3237.8.	0.00	1027.2	71.50	**906.8**	61.17
Random Eucl 400	400	20	20	2582.8	0.00	1059.5	138.33	**867.4**	40.78
	400	20	20	2308.6	0.00	858.7	40.51	**802.2**	18.18
	600	20	30	2984.3	0.00	725.1	103.93	**602.6**	4.03
Random Eucl 600	600	20	30	2964.1	0.00	823.7	55.31	**785.4**	39.64
	600	20	30	2550.8	0.00	778.9	60.08	**759.2**	32.63
	200	20	10	1569.7	5.73	244.9	34.41	**237.5**	29.33
Non-Eucl 200	200	20	10	1223.9	0.00	**216.7**	33.83	217.0	22.00
	200	20	10	1465.6	0.00	**179.8**	28.38	195.6	32.20
	500	100	5	2045.9	1.72	1121.3	123.29	**1049.0**	114.92
Non-Eucl 500	500	100	5	2073.6	146.39	1008.2	128.91	**998.1**	94.98
	500	100	5	1565.0	0.00	1025.8	109.49	**1020.4**	78.15
	600	20	30	1469.6	0.00	138.6	25.11	**122.7**	12.43
Non-Eucl 600	600	20	30	1754.6	0.00	132.0	18.98	**118.0**	16.04
	600	20	30	414.3	0.00	119.4	18.98	**117.9**	16.54

Tab. 8.1: Results on instance sets from [11] and correspondingly created new sets, 600s CPU-time (except ACH). Three different instances are considered for each set.

TSPlib Instances				ACH		VNS1		VNS2	
Name	$\|V\|$	r	time	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	137	28	150s	562.0	0.00	**440.8**	2.42	442.2	2.94
kroa150	150	30	150s	17234.0	0.00	**11532.8**	2.44	11542.7	24.74
krob200	200	40	300s	17779.0	0.00	13309.4	79.90	**13300.9**	102.37
ts225	225	45	300s	83729.0	0.00	**68769.7**	305.85	69110.0	641.85
gil262	262	53	300s	1434.0	0.00	1157.1	54.88	**1117.2**	30.70
pr264	264	54	300s	39860.0	0.00	**31639.6**	1449.32	31641.9	1027.65
pr299	299	60	450s	28684.0	0.00	23953.9	792.38	**23397.0**	321.22
lin318	318	64	450s	28039.0	0.00	23101.1	840.31	**22599.6**	780.27
rd400	400	80	600s	9605.0	0.00	**7275.4**	237.96	7291.1	276.53
fl417	417	84	600s	12177.0	0.00	**10636.4**	286.75	10875.7	196.74
gr431	431	87	600s	1681.0	0.00	1408.2	50.76	**1399.6**	45.02
pr439	439	88	600s	86968.0	0.00	**72752.7**	3857.22	73193.7	2941.80
pcb442	442	89	600s	29573.0	0.00	26051.2	197.20	**25960.8**	621.32

Tab. 8.2: Results on TSPlib instances with geographical clustering, $\frac{|V|}{r} = 5$, variable CPU-time.

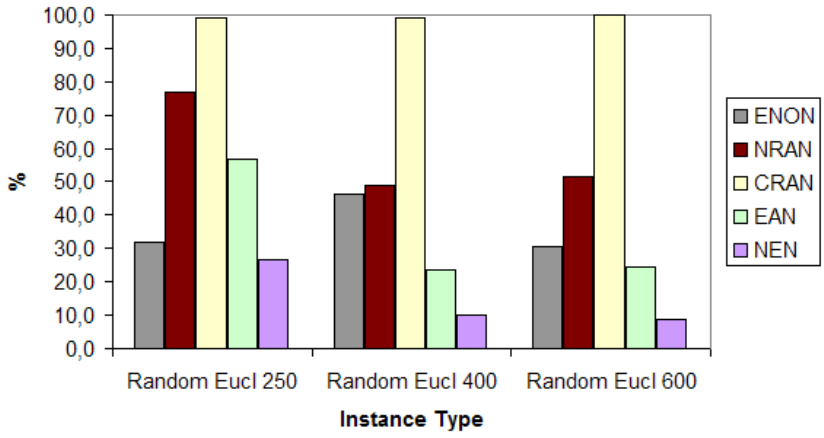


Fig. 8.8: Contributions of the neighborhoods on random euclidean instances for VNS2.

Instance Type	$\|V\|$	r	$\|V\|/r$	SNON	NRAN	EAN	NEN
TSPlib based	n.a.	n.a.	5	0.56	0.49	0.63	0.52
Grouped Euclidean	125	25	5	0.68	0.34	0.34	0.16
	500	100	5	0.99	0.73	0.78	0.84
	600	20	30	0.91	0.23	0.24	0.22
	1280	64	20	1.00	0.61	0.66	0.91
Random Euclidean	250	50	5	0.24	0.83	0.61	0.36
	400	20	20	0.19	0.45	0.28	0.18
	600	20	30	0.14	0.47	0.25	0.17
Non-Euclidean	200	20	10	0.28	0.08	0.35	0.43
	500	100	5	0.27	0.24	0.93	0.78
	600	20	30	0.32	0.05	0.28	0.46

Tab. 8.3: Relative effectivity of SNON, NRAN, EAN, and NEN for VNS1.

Instance Type	$\|V\|$	r	$\|V\|/r$	ENON	NRAN	CRAN	EAN	NEN
TSPlib based	n.a.	n.a.	5	0.31	0.48	0.99	0.61	0.31
Grouped Euclidean	125	25	5	0.73	0.35	1.00	0.31	0.12
	500	100	5	0.53	0.63	0.94	0.75	0.56
	600	20	30	0.43	0.25	1.00	0.23	0.11
	1280	64	20	0.70	0.45	0.94	0.53	0.67
Random Euclidean	250	50	5	0.32	0.77	0.99	0.57	0.27
	400	20	20	0.46	0.49	0.99	0.24	0.10
	600	20	30	0.31	0.51	1.00	0.24	0.09
Non-Euclidean	200	20	10	0.40	0.05	0.75	0.26	0.20
	500	100	5	0.36	0.14	0.78	0.93	0.62
	600	20	30	0.42	0.03	0.88	0.14	0.14

Tab. 8.4: Relative effectivity of ENON, RNAN, CNAN, EAN, and NEN for VNS2.

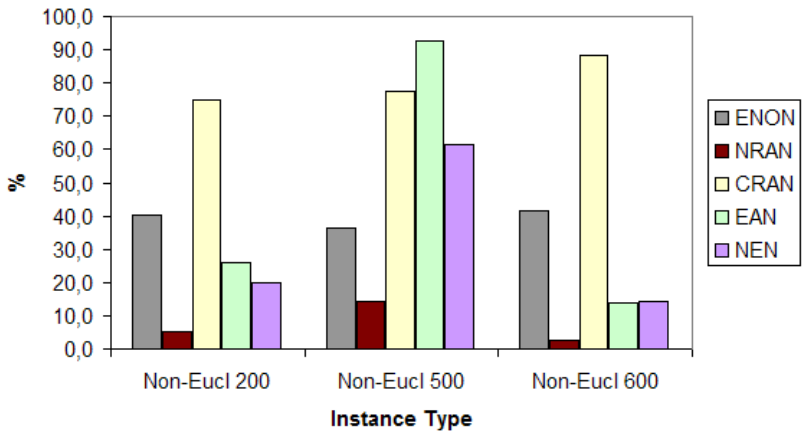


Fig. 8.9: Contributions of the neighborhoods on non-euclidean instances for VNS2.

9. Implementation Details

The algorithms described in this thesis have been implemented with C++ on a Pentium 4, 2.8GHz PC with 2GB RAM and compiled with GCC 3.3.1.
CPLEX 9.03 is used to solve ILP subproblems and EAlib 2.0 [32], which is a problem independent C++ library for the development of efficient metaheuristics.
For running the program for the GMST problem as well as the program for the GMEBCN problem, some command line parameters need to be set in order to initialize EAlib [32] properly (see Table 9.1).

Parameter Name	Value
maxi	0
eamod	9
sub.eamod	10

Tab. 9.1: Program Parameters for EAlib [32].

9.1 Description of the Program for the GMST Problem

For the GMST problem the set of used nodes is represented by a vector whose entries refer to the index of the selected nodes in the corresponding clusters, while (global) edges are represented by an adjacency matrix. The following chapter provides a short description of all classes implemented, while Figure 9.1 presents an UML diagram visualizing correlations and dependencies between them. Table 9.2 shows all parameters to be set additionally to the general ones needed to initialize EAlib [32].

Parameter Name	Description
clusterfile	instance file containing clustering information
edgefile	instance file containing distance matrix
ghosh2_maxtime	maximum computation time for RNEN2
ttime	maximum computation time in seconds

Tab. 9.2: Program Parameters for GMST.

9.1.1 Class Description

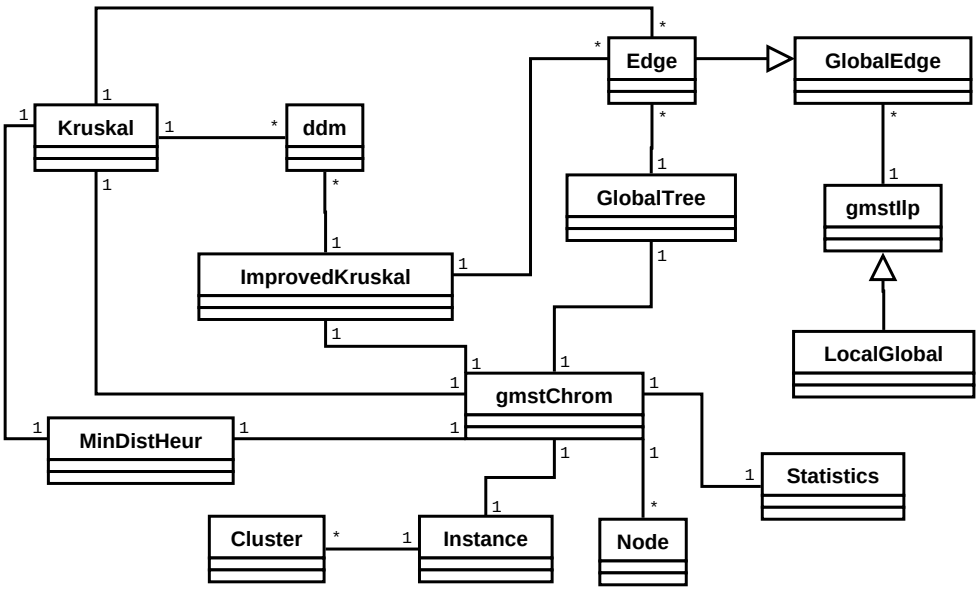


Fig. 9.1: UML Diagram modelling the program for the GMST problem.

- *Cluster*
  Encapsulation of a single cluster. Next to a list of concrete nodes for an actual cluster this class keeps track if a current cluster needs to be recomputed (important for GEEN and GSON).

- *ddm*
  Implementation of the *dynamic disjunct set* data structure in order to compute minimum spanning trees efficiently using the algorithm of Kruskal.

- *Edge*
  Extends *GlobalEdge* by additionally adding edge costs to represent a concrete edge included in a solution.

- *GlobalEdge*
  Representation of an edge within the global graph.

- *GlobalTree*
  Implements all necessary methods to operate on a global graph.

- *gmstChrom*
  Represents the chromosome for the GMST problem and implements all neighborhood structures.

- *gmstIlp*
  Base class to solve parts of a GMST instance exact using *CPLEX*. This class initializes

all variables used for the Local - Global formulation of Pop [30] and describes the global polytype of the problem.

- *ImprovedKruskal*
  Implementation of the Improved Adaption of Kruskal's Algorithm for the MST (IKH) used to generate initial solutions.

- *Instance*
  Creates and stores all necessary information for a concrete instance it reads from the corresponding instance files.

- *Kruskal*
  Implementation of Kruskals algorithm for the classical MST problem.

- *LocalGlobal*
  Extends *gmstIlp* by describing the part of Pops [30] Local - Global ILP formulation responsible for computing the best local solution from a given global one.

- *MinDistHeur*
  Implementation of the Minimum Distance Heuristic (MDH) used to generate initial solutions.

- *Node*
  This class stores all important information for the nodes of a concrete instance during the dynamic programming procedure in GEEN and GSON. This information includes the concrete costs and the direct successors on the path having these costs.

- *Statistics*
  Utility class to store additional information needed to carry out statistical analysis for the neighborhoods.

9.2 Description of the Program for the GMEBCN Program

In opposite to the GMST problem, (global) edges included in a solution are internally represented by storing the adjacency list for each selected node. The corresponding class (*AdjacenceList*) which encapsulates this data structure does ensure that each of these lists is sorted (ascending). Hence adding a new edge and testing if an edge is currently included in the solution cannot be done in constant time anymore, but has computational effort $O(\log(\deg(V_i)))$, $(\forall i = 1, \ldots, r)$. This representation allows iterating through all incident nodes v of an actual node w in $O(\deg w)$ instead of $O(r)$ if an adjacency matrix would be used. As iterating through all neighbors is a frequent operation (e. g. DFSBridges) this advantage does prevail the additional effort while constructing these lists.
Similar to the GMST problem, Table 9.3 presents the program parameters, Figure 9.2 visualizes the dependencies between all classes, which are explained more detailed in the next chapter.

Parameter Name	Description
clusterfile	instance file containing clustering information
edgefile	instance file containing distance matrix
maxClusterSwapTime	maximum computation time for CRAN in seconds
maxLocGlobExactSize	maximum number of clusters in the reduced graph to solve exact in ENON
maxNodeExchangeTime	maximum computation time for NEN in seconds
ttime	maximum computation time in seconds

Tab. 9.3: Program Parameters for GMEBCN.

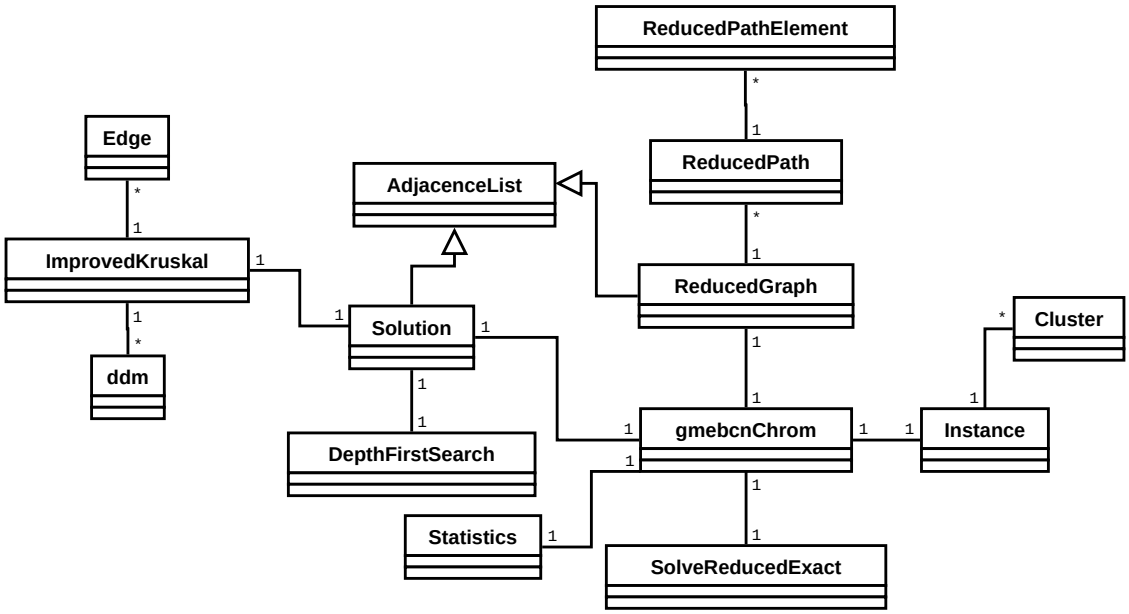


Fig. 9.2: UML Diagram modelling the program for the GMEBCN problem.

9.2.1 Class Description

- *AdjacenceList*
  Implements and encapsulates the adjacency list data structure. All methods that update the list ensure that the list is sorted ascending afterwards.

- *Cluster*
  Encapsulation of a single cluster with its corresponding nodes.

- *DepthFirstSearch*
  Implementation of Algorithm 7, used to test if a graph violates the edge biconnectivity

property as well as to compute its edge biconnected components.

- *ddm*
  Implementation of the *dynamic disjunct set* data structure in order to compute minimum spanning trees efficiently using the algorithm of Kruskal.

- *Edge*
  Internal representation of an edge (u, v) with specified costs. Both u, and v are specified by their cluster number, while the concrete nodes can be queried by additionally taking the vector of used nodes into account. This approach has the benefit that if only the used nodes are changed, nothing has to be updated within the adjacency lists.

- *gmebcnChrom*
  Implementation of the chromosome for the GMEBCN problem and implements all neighborhood structures.

- *ImprovedKruskal*
  Implements the Improved Adaption of Kruskals algorithm for the MST problem, as used within ACH (Algorithm 15).

- *Instance*
  Creates and stores all necessary information for a concrete instance it reads from the corresponding instance files.

- *ReducedGraph*
  Determines and stores a reduced graph. Implements methods to efficiently update a global graph, as well as to decode a global graph to a concrete solution.

- *ReducedPath*
  Internal representation of a global path and its corresponding reduced edge.

- *ReducedPathElement*
  A single element (cluster) of a reduced path. This class additionally stores all information computed with dynamic programming while generating the corresponding reduced edges.

- *Solution*
  Class to store and deal with a concrete solution.

- *SolveReducedExact*
  Implements an Integer Linear Program to compute the best nodes from a given global solution.

- *Statistics*
  Utility class to store additional information needed to carry out statistical analysis for the neighborhoods.

9.3 Instance Generator

For extending the available benchmark set of instances an instance generator has been implemented which supports the creation of *clustered euclidean, random euclidean, and non euclidean* instances which already have been described in Chapter 5.1. Table 9.4 shows how to generate different instance types using the generator, while Table 9.5 explains the concrete parameters.

Instance Type	Generation Command
Grouped Euclidean	./generate GROUPED_EUCL rows cols p sep span clusterfile edgefile
Random Euclidean	./generate NOT_GROUPED_EUCL clustercnt p xmax ymax clusterfile edgefile
Non Euclidean	./generate NOT_EUCL clustercnt p maxdist clusterfile edgefile

Tab. 9.4: Generation of various Instance Types.

Each input instance consists of two files, one containing the distance values and a second one saving the assignment of nodes to clusters. The one containing the distance values ("edgefile") starts with a header of two integral values, specifying the number of nodes and the number of edges, which is followed by an upper triangular matrix containing the concrete edge costs, which are assumed to be symmetric. The file containing the assignment of nodes to their clusters ("clusterfile") starts with a header specifying the node count which is followed by r lists of nodes (one for each cluster), which are separated by zero values.

Parameter	Description
rows	number of rows
cols	number of columns
p	number of nodes per cluster
sep	cluster seperation
span	cluster span
clusterfile	name of file to save clustering information
edgefile	name of file to save distance matrix
xmax	x dimension of surrounding rectangle
ymax	y dimension of surrounding rectangle
maxdist	maximum distance between two nodes
clustercnt	number of clusters

Tab. 9.5: Program Parameters for the Instance Generator

10. Summary and Outlook

This thesis proposed general Variable Neighborhood Search (VNS) approaches for solving the Generalized Minimum Spanning Tree (GMST) problem and the Generalized Minimum Edge Biconnected Network (GMEBCN) problem.

For the GMST problem initialization is done be the Minimum Distance Heuristic and the Improved Adaption of Kruskal's MST Heuristic, which are both based on Kruskal's classical algorithms for determining a MST. Though their performance depends on the instance type, the latter constructive heuristic mostly provides better results.
The Variable Neighborhood Descent for the GMST problem combines three neighborhood types: For the Node Exchange Neighborhood, solutions are represented by the used nodes and one node is replaced by another of the same cluster. Optimal edges are derived by determining a classical MST on these nodes. The Global Edge Exchange Neighborhood works in complementary ways by considering for a solution primarily its global connections, i. e. the pairs of clusters which are directly connected. Neighbors are all solutions differing in exactly one global connection. Knowing this global structure for a solution, dynamic programming is used to determine the best suited nodes and concrete edges. For both of these neighborhoods, incremental evaluation schemes have been described, which speed up the whole computation considerably. The Global Subtree Optimization Neighborhood selects a subset of clusters connected within the current solution and solves the subproblem induced by these clusters to optimality via Integer Linear Programming. The obtained subtree is then reconnected to the remainder as well as possible.

Tests on TSPlib instances, grouped Euclidean instances, random Euclidean instances and non-Euclidean instances show that the proposed VNS algorithm has significant advantages over previous metaheuristic approaches in particular on instances with a large number of nodes per cluster.
On grouped Euclidean and TSPlib based instances, the differences between the objective values of the final solutions obtained by our VNS and the other candidate algorithms are relatively low, which indicates that the structure of these instances are simpler. The differences between the considered algorithms are largest on random Euclidean instances. In this case, VNS produces substantially better results due to the effectiveness of Global Edge Exchange Neighborhood.

A similar approach has been designed for the GMEBCN problem. Initialization is done by the Adapted Christofides Heuristic which augments a solution to the GMST problem generated by the Improved Adaption of Kruskal's MST Heuristic. The augmentation is done by creating a matching on all nodes of odd degree to generate an edge biconnected graph.

In contrast to the GMST problem, fixing the used nodes and computing the best cor-

responding solution based on them cannot be done with reasonable effort as the classical Minimum Edge Biconnected Network problem is known to be NP hard. Analogically, the problem of determining the best used nodes for a given global graph is NP hard too, which can be shown by a reduction from the graph coloring problem. This points out that the GMEBCN is more complex than the GMST problem. To deal with this high complexity, a technique to reduce the size of a solution graph has been introduced. It evaluates the so called *relevant clusters* whose optimal selected nodes are hard to identify while the optimal selected nodes of the remaining clusters, the so called *irrelevant clusters*, can be determined exactly in an efficient way once the used nodes of all relevant clusters are fixed.

Based on these conclusions, six different neighborhoods were presented that consider different aspects of the problem. For all these neighborhoods a solution is represented by the used nodes as well as the selected (global) edges between them.
In the Simple Node Optimization Neighborhood, the used nodes of exactly one cluster gets changed by another node of the same cluster while adopting the global edges of the original solution. Complementary to that the Node Re-Arrangement Neighborhood considers the arrangement of nodes within a solution. A neighbor solution differs by exchanging the sets of incident edges of exactly two nodes.
The Edge Augmentation Neighborhood explores new areas by augmenting a current solution with one new edge and removing redundant edges afterwards. To speed up the examination of this neighborhood a large amount of cases where the solution cannot be improved are determined and their execution prevented in advance.
Optimization on both the used nodes as well as the included edges is done in a heuristic way within the Node Exchange Neighborhood. Moving to a solution is done by exchanging exactly one used node by another node of the same cluster and additionally removing all edges incident to this node. The graph is then heuristically augmented with new edges until the edge biconnectivity property is met again. An efficient way has been developed to re-biconnect a solution as adding too many edges would lead to a higher computational effort when removing redundant edges.
The Extended Node Optimization Neighborhood and the Cluster Re-Arrangement Neighborhood are natural extensions to the Simple Node Optimization Neighborhood and the Node Re-Arrangement Neighborhood by taking the graph reduction technique into concept. The Extended Node Optimization Neighborhood operates on a much smaller graph and changes the used nodes of up to two clusters while computing the best used nodes for all other clusters is done exactly using a dynamic programming procedure. Additionally it is able to determine the optimal used nodes of all clusters using Integer Linear Programming if the amount of clusters in the reduced graph is below some predefined threshold. Similar to that the Cluster Re-Arrangement Neighborhood optimizes the arrangement of clusters in the same way as done in the Node Re-Arrangement Neighborhood. A concrete solution is determined by re-arranging exactly two clusters and computing the best used nodes for all irrelevant clusters while forbearing from changing used nodes within any relevant cluster. As it turned out to be crucial to prevent re-applying the reduction procedure an evaluation scheme has been developed that prevents this process when possible.
Based on these neighborhoods, two different VNS approaches have been presented, one based one the simpler, but much faster neighborhoods, while the second one additionally takes the two neighborhoods utilizing the graph reduction technique in to concept.
Additional to that, some effort has been done on exact methods, by presenting an idea of

adapting a rather successful Integer Linear Programming (ILP) formulation for the GMST to the GMEBCN problem.
Tests on TSPlib instances, grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances show that the second VNS approach (VNS2), utilizing the neighborhoods based on graph reduction, has significant advantages over the simpler version. The superior performance is most evident on random Euclidean instances and instances with a large number of nodes per cluster. On these instances, VNS2 produces substantially better results which is due to the effectiveness of the graph reduction concept used by ENON and CRAN. The differences between the objective values of the final solutions obtained by VNS1 and VNS2 on grouped Euclidean and TSPlib based instances are comparatively lower, which approves the assumption that these instances are easier to deal with.

Further work can be done especially for the GMEBCN problem by introducing additional neighborhoods, particularly those operating on the reduced graph and / or incorporating more sophisticated ILP formulations. It might also be interesting to apply similar concepts to other generalized network design problems.

Bibliography

[1] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparision. *ACM Computing Surveys*, 35(3):268–308, 2003.

[2] R. Diestel. *Graph Theory*. Springer-Verlag, Heidelberg, 2005.

[3] M. Dror, M. Haouari, and J. S. Chaouachi. Generalized spanning trees. *European Journal of Operational Research*, 120:583–592, 2000.

[4] C. W. Duin, A. Volgenanta, and S. Voß. Solving group steiner problems as steiner problems. *European Journal of Operational Research*, 154, issue 1:323–329, 2004.

[5] C. Feremans. *Generalized Spanning Trees and Extensions*. PhD thesis, Universite Libre de Bruxelles, 2001.

[6] C. Feremans and A. Grigoriev. An approximation scheme for the generalized geometric minimum spanning tree problem with grid clustering. Technical Report NEP-ALL-2004-09-30, Maastricht: METEOR, Maastricht Research School of Economics of Technology and Organization, 2004.

[7] C. Feremans, M. Labbe, and G. Laporte. A comparative analysis of several formulations for the generalized minimum spanning tree problem. *Networks*, 39(1):29–34, 2002.

[8] C. Feremans, M. Labbe, and G. Laporte. The generalized minimum spanning tree problem: Polyhedral analysis and branch-and-cut algorithm. *Networks*, 43, issue 2:71–86, 2004.

[9] M. Fischetti, J. J. S. González, and P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45:378–394, 1997.

[10] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.

[11] D. Ghosh. Solving medium to large sized Euclidean generalized minimum spanning tree problems. Technical Report NEP-CMP-2003-09-28, Indian Institute of Management, Research and Publication Department, Ahmedabad, India, 2003.

[12] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533–549, 1986.

[13] B. Golden, S. Raghavan, and D. Stanojevic. Heuristic search for the generalized minimum spanning tree problem. *INFORMS Journal on Computing*, 17(3):290–304, 2005.

[14] P. Hansen and N. Mladenovic. An introduction to variable neighborhood search. In S. Voss, S. Martello, I. H. Osman, and C. Roucairol, editors, *Meta-heuristics, Advances and trends in local search paradigms for optimization*, pages 433–458. Kluwer Academic Publishers, 1999.

[15] P. Hansen and N. Mladenovic. A tutorial on variable neighborhood search. Technical Report G-2003-46, Les Cahiers du GERAD, HEC Montreal and GERAD, Canada, 2003.

[16] M. Haouari and J. S. Chaouachi. Upper and lower bounding strategies for the generalized minimum spanning tree problem. *European Journal of Operational Research*, 171:632–647, 2006.

[17] M. Haouari, J. S. Chaouachi, and M. Dror. Solving the generalized minimum spanning tree problem by a branch-and-bound algorithm. *Journal of the Operational Research Society*, 56(4):382–389, 2005.

[18] B. Hu, M. Leitner, and G. R. Raidl. Computing generalized minimum spanning trees with variable neighborhood search. In *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search*, 2005.

[19] B. Hu, M. Leitner, and G. R. Raidl. Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. Technical Report TR 186-1-06-01, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2006.

[20] D. Huygens. Probléme de conception de réseau k-arête-connexe l-chemin, 2001.

[21] E. Ihler, G. Reich, and P. Widmayer. Class steiner trees and vlsi-design. *Discrete Applied Mathematics*, 90:173–194, 1999.

[22] H. Kerivin and A. R. Mahjoub. Design of survivable networks: A survey. Technical Report LIMOS/RR-05-04, Laboratoire LIMOS, CNRS UMR 6158, University Blaise Pascal, 2005.

[23] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. *J. ACM*, 41(2):214–235, 1994.

[24] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[25] H. R. Lourence, O. Martin, and T. Stützle. A beginner's introduction to iterated local search. In *Proceedings of MIC 2001*, pages 1–6, 2001.

[26] H. R. Lourence, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321 – 353. Kluwer Academic Publishers, 2002.

[27] T. L. Magnati and S. Raghavan. Strong formulations for network design problems with connectivity requirements. *Networks*, 45(2):61–79, 2005.

[28] Y. S. Myung, C. H. Lee, and D. W. Tcha. On the generalized minimum spanning tree problem. *Networks*, 26:231–241, 1995.

[29] C. H. Papdimitriou and K. Steiglitz. *COMBINATORIAL OPTIMIZATION Algorithms and Complexity*. Dover Publications Inc., 1998.

[30] P. C. Pop. *The Generalized Minimum Spanning Tree Problem*. PhD thesis, University of Twente, The Netherlands, 2002.

[31] P. C. Pop, G. Still, and W. Kern. An approximation algorithm for the generalized minimum spanning tree problem with bounded cluster size. In H. Broersma, M. Johnson, and S. Szeider, editors, *Algorithms and Complexity in Durham 2005, Proceedings of the first ACiD Workshop*, volume 4 of *Texts in Algorithmics*, pages 115–121. King's College Publications, 2005.

[32] G. Raidl and D. Wagner. EAlib 2.0 – A Generic Library for Metaheuristics. Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2005.

[33] G. Reich and P. Widmayer. Beyond steiner's problem: A vlsi oriented generalization. In *Graph-Theoretic Concepts in Computer Science WG89*, pages 196–210, 1989.

[34] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

Appendix

A. Curriculum Vitae

Personal Information

- **Full Name:** Markus Karl Leitner
- **Address:** 1170 Wien, Palffygasse 27/11
- **Date and place of birth:** November 02, 1979; Ried im Innkreis, Austria
- **Citizenship:** Austrian

Education

- Since 03/2004: Student of Mathematics, University Vienna
- Since 10/2000: Student of Computer Science, Vienna University of Technology, Austria
- 10/1999 - 09/2000: Civilian Service, Lebenshilfe, Regau
- 09/1994 - 06/1999: Secondary College for Electronics - Special Training Focus Technical Computer Science, school leaving examination passed with distinction
- 09/1990 - 06/1994: Lower secondary school in Ampflwang
- 09/1986 - 06/1990: Primary school in Ampflwang

Work Experience

- Since 08/2005: Siemens AG Austria, PSE, Vienna
- 11/2004 - 12/2004: T-Mobile Austria GmbH, Vienna
- 07/2004 - 08/2004: T-Mobile Austria GmbH, Vienna
- 02/2003 - 09/2003: team-plus! Lernhilfe-Institut, Vienna
- 07/2002 - 08/2002: Funworld AG, Schörfling
- 07/2001 - 08/2001: Kretztechnik AG, Zipf
- 08/1997: OKA (now Energie AG), Gmunden
- 08/1996: OKA (now Energie AG), Vöcklabruck

Publications

- B. Hu, M. Leitner, and G. R. Raidl. Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. Technical Report TR 186-1-06-01, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2006. submitted to Journal of Heuristics.

- B. Hu, M. Leitner, and G. R. Raidl. Computing generalized minimum spanning trees with variable neighborhood search. In P. Hansen, N. Mladenovic, J. A. M. Pérez, B. M. Batista, and J. M. Moreno-Vega, editors, Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search, Tenerife, Spain, 2005.