DISSERTATION

Hybrid Metaheuristics for Generalized Network Design Problems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Univ.-Prof. Dr. Günther Raidl
Institut für Computergraphik und Algorithmen E186
Technische Universität Wien

eingereicht an der Technischen Universität Wien
Fakultät für Informatik von

Bin Hu
Matrikelnummer 9925131
Obere Donaustraße 43/3/54e, 1020 Wien

Wien, am

Bin Hu

Kurzfassung

Diese Dissertation behandelt verschiedene generalisierte Netzwerkdesignprobleme (NDPs), die zu den $\mathcal{NP}$-harten kombinatorischen Optimierungsproblemen gehören. Im Gegensatz zu klassischen NDPs sind die generalisierten Versionen auf Graphen definiert, deren Knotenmengen in Clustern aufgeteilt sind. Das Ziel besteht darin, jeweils einen Subgraphen zu finden, der genau einen Knoten pro Cluster enthält und weitere Nebenbedingungen erfüllt.

Methoden, die zum Lösen von kombinatorischen Optimierungsproblemen eingesetzt werden, können grob in zwei Hauptrichtungen eingeteilt werden. Die erste Klasse besteht aus Algorithmen, die diese Probleme beweisbar optimal lösen können, sofern ihnen ausreichend viel Zeit und Speicher zur Verfügung gestellt werden. Diese Arbeit beginnt mit einer kurzen Einführung in die Techniken der linearen und ganzzahlig linearen Programmierung. Sie bilden die Basis für populäre Algorithmen wie Branch-and-Bound, Branch-and-Cut und viele weitere. Die zweite Klasse besteht aus Metaheuristiken, die zwar Näherungslösungen erzeugen, aber wesentlich weniger Zeit benötigen. Wenn beide Klassen miteinander kombiniert werden, entstehen hybride Algorithmen, die von den Vorteilen beider Richtungen profitieren können. Einige der vielfältigen Kombinationsmöglichkeiten werden untersucht und auf NDPs in dieser Arbeit angewandt.

Das erste Problem, das betrachtet wird, ist das generalisierte minimale Spannbaumproblem. Gegeben ist ein Graph, dessen Knoten in Clustern partitioniert sind. Wir suchen nach einem minimalen Spannbaum, der genau einen Knoten pro Cluster verbindet. Ein Ansatz basierend auf variabler Nachbarschaftssuche (VNS) wird vorgestellt, der drei verschiedene Nachbarschaftstypen verwendet. Zwei davon

arbeiten komplementär, um die Effizienz bei der Suche zu erhöhen. Beide enthalten exponentiell viele Lösungen, aber effektive Algorithmen werden eingesetzt, die die jeweils besten Nachbarlösungen in polynomieller Zeit finden. Für die dritte Nachbarschaft wird ganzzahlige lineare Programmierung verwendet, um Teilbereiche einer Lösung zu optimieren.

Als nächstes betrachten wir das generalisierte Handlungsreisendenproblem (GTSP). Ausgehend von einem geclusteten Graphen suchen wir eine Rundreise minimaler Länge, die von jedem Cluster einen Knoten besucht. Ein VNS Algorithmus basierend auf zwei Nachbarschaftsstrukturen wird vorgestellt. Eine davon ist die bereits bekannte generalisierte 2-opt Nachbarschaft, für die eine neue inkrementelle Auswertungstechnik entworfen wird, die den Suchvorgang wesentlich beschleunigt. Die zweite Nachbarschaft basiert auf dem Austauschen von den in der Lösung vorkommenden Knoten, auf die anschließend die verkettete Lin-Kernighan Heuristik angewendet wird.

Als ein zum GTSP verwandtes Problem untersuchen wir das Eisenbahn-Handlungsreisendenproblem (RTSP). Gegeben ist ein Fahrplan und ein Geschäftsmann, der eine Anzahl von Städten per Bahn besuchen muss, um Aufträge zu erledigen. Die Reise startet und endet an einem bestimmten Ort und die dafür benötigte Gesamtzeit, inklusive den Wartezeiten, soll minimiert werden. Es werden zwei Transformationen präsentiert, die das Problem als asymmetrisches oder symmetrisches Handlungsreisendenproblem (TSP) umformulieren. Damit können für das RTSP bewährte Techniken eingesetzt werden, die für das klassische TSP konzipiert sind.

Schließlich wird das Problem des generalisierten minimalen kantenzweizusammenhängenden Netzwerks betrachtet. Ausgehend von einem geclusteten Graphen wird ein Subgraph gesucht, der genau einen Knoten pro Cluster kantenzweizusammenhängend verbindet, d.h. zwischen je zwei Knoten müssen mindestens zwei kantendisjunkte Wege existieren. Wir betrachten drei VNS Varianten, die mit unterschiedlichen Nachbarschaftsstrukturen arbeiten. Jede adressiert bestimmte Teilaspekte wie die verbundenen Knoten und/oder die Kanten zwischen ihnen. Für komplexere Nachbarschaften werden effiziente Techniken wie Graphreduktion eingesetzt, die den Optimierungsvorgang wesentlich beschleunigen. Für Vergleichszwecke wird eine Formulierung als ganzzahliges lineares Programm entworfen, mit der kleine Instanzen beweisbar optimal gelöst werden können.

Experimentelle Ergebnisse zeigen, dass die grundlegende Strategie, komplementäre Nachbarschaftsstrukturen zu kombinieren, beim Lösen von generalisierten NDPs sehr erfolgreich ist. Insbesondere wird festgehalten, dass jede Nachbarschaftsstruktur signifikante Beiträge zum Optimierungsvorgang leistet.

Abstract

In this thesis, we consider several generalized network design problems (NDPs) which belong to the family of $\mathcal{NP}$-hard combinatorial optimization problems. In contrast to classical NDPs, the generalized versions are defined on graphs whose node sets are partitioned into clusters. The goal is to find a subgraph which spans exactly one node from each cluster and also meets further constraints respectively.

Applicable methodologies for solving combinatorial optimization problems can roughly be divided into two mainstreams. The first class consists of algorithms which aim to solve these problems to proven optimality – provided that they are given enough run-time and memory. This thesis starts with a brief introduction to linear and integer linear programming techniques since popular algorithms like branch-and-bound, branch-and-cut, etc. are based on them. The second class are metaheuristics which compute approximate solutions but usually require significantly less runtime. By combining these two classes, we are able to form collaboration approaches that benefit from advantages of both sides. We will examine various possibilities of such combinations and some of them will be used to solve the NDPs in this thesis.

The first considered NDP is the generalized minimum spanning tree problem. Given a graph whose nodes are partitioned into clusters, we seek a minimum spanning tree which connects exactly one node from each cluster. A variable neighborhood search (VNS) approach will be presented that uses three different neighborhood types. Two of them work in complementary ways in order to maximize search performance. Both are large in the sense that they contain exponentially many candidate solutions, but efficient polynomial-time algorithms are used to identify best neighbors. For the third neighborhood type we apply integer programming to optimize local parts within candidate solution trees.

We then study the generalized traveling salesman problem (GTSP). Given a clustered graph, we seek the minimum-costs round trip visiting one node from each cluster. A VNS algorithm based on two complementary, large neighborhood structures is proposed. One of them is the already known generalized 2-opt neighborhood for which a new incremental evaluation technique is described, which speeds up the search significantly. The second structure is based on node exchanges and the application of the chained Lin-Kernighan heuristic.

As a related problem to the GTSP, we also consider the railway traveling salesman problem (RTSP). We are given a timetable and a salesman who has to visit a number of cities by train to carry out some business. He starts and ends at a specified home city, and the required time for the overall journey, including waiting times, shall be minimized. Two transformation schemes to reformulate the problem as either a classical asymmetric or symmetric traveling salesman problem (TSP) are presented. Using these transformations, established algorithms for solving the TSP can be used to attack the RTSP as well.

Finally, we consider the generalized minimum edge biconnected network problem. For a given clustered graph, we look for a minimum-costs subgraph connecting one node from each cluster in an edge biconnected way, i.e. at least two edge-disjoint paths must exist between each pair of nodes. Three VNS variants are considered that utilize different types of neighborhood structures, each of them addressing particular properties as spanned nodes and/or the edges between them. For the more complex neighborhood structures, efficient techniques – such as a graph reduction – are applied to essentially speed up the search process. For comparison purpose, a mixed integer linear programming formulation based on multi commodity flows is proposed to solve smaller instances of this problem to proven optimality.

Looking at the obtained results, we observe that the fundamental strategy of combining complementary neighborhood structures is very successful for solving generalized NDPs. In particular, all of them are shown to contribute significantly to the search process.

Acknowledgments

First of all I would like to express my gratitude to my supervisor Prof. Günther Raidl, who introduced me into the world of combinatorial optimization, metaheuristics, and integer programming. He provided me with invaluable advices and ideas every time when I was facing difficulties during research. He also gave me the opportunity to publish in journals and conferences all over the world. I further want to thank Prof. Ulrich Pferschy who agreed to be the second assessor of this thesis. Many thanks to Gunnar Klau who was my supervisor of my master thesis and raised my interests for optimization problems.

I owe my gratitude to all of my colleagues from the Algorithms and Data Structures Group of the Vienna University of Technology: Jakob Puchinger was my roommate until he finished his thesis. We not only shared a good time, but also our thoughts on food-philosophy. Martin Schönhacker was an important mentor particularly in teaching and university issues. Martin Gruber is an amazing colleague who knows the proper solutions to almost all problems one could encounter as graduate student. Andreas Chwatal and Markus Leitner are my current roommates and we always have a nice climate in our room. Matthias Prandtstetter, Sandro Pirkwieser, and Daniel Wagner have one of the furthermost rooms from mine, but that certainly could not hinder our numerous and pleasant conversations. Mario Ruthmair is the most recent colleague in our group and he supports us everywhere he can. Phlipp Neuner and Aksel Filipovic are always present when technical problems occurs. Stephanie Wogowitsch and Angela Schabel are of great assistance whenever I have to deal with administrative issues.

Last but not least I want to thank my family, especially my parents whom I owe everything, and my wife whom I love above all else.

Contents

Chapter 1

Introduction

Network design and network optimization problems are of central importance to the modern society. They appear frequently in practical fields such as transportation, telecommunication, facility allocation, resource supply, and many others. Obtaining good solutions with respect to lowering the connection costs, reducing transmission delays, etc. often results in substantial economical, environmental and/or social advantages.

The term "network design" is involved in many contexts and there are several different aspects which deserve attention. In this thesis, they are regarded from a more theoretical point of view as graph theory problems, i.e. networks are modeled as graphs and optimization algorithms are applied on them.

For example, when we look for an efficient way to connect several communication nodes to a local area network, we can regard communication nodes as nodes in a graph and the possible connections between them as edges. The weight of an edge can be represented by the estimated costs for setting up a connection. Possible other requirements on the communication network like fault tolerance, transmission quality, etc. can be modeled as additional constraints in the corresponding graph problem.

Formally, we are given a graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Generally, we are looking for a subgraph $S = \langle P, T \rangle$ with $P \subseteq V$ and $T \subseteq E$ that has minimal total costs $C(T) = \sum_{e \in T} c(e)$ and also satisfies additional constraints depending on the actual problem.

Let us first consider two examples of well known classical Network Design Problems (NDPs) in combinatorial optimization.

Traveling Salesman Problem (TSP): We are given a graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. We seek a minimal costs subgraph $S = \langle V, T \rangle$ with $T \subseteq E$ being a round trip (Hamiltonian cycle) connecting all nodes $v \in V$. This problem is $\mathcal{NP}$-hard, i.e. there are no known algorithms which can solve every instance in polynomial time with respect to the graph's size.

A strongly simplified practical application of this problem would be if a salesman wants to travel through a number of major cities in Europe by airplane and the minimal amount of time and/or money should be spent.

Another example where the TSP model can be used in a more straightforward way appears in the printed circuits manufactory. The route of a drill machine should be scheduled to all drill holes on a printed circuit board. These holes, which can be of different sizes, represent cities in the TSP and the required time for relocating and/or retooling the the drill machine from one drill hole to another represents the distance between them. At the end, we want to minimize the overall time needed to produce such a printed circuit board.

Minimum Spanning Tree Problem (MSTP): We are given a graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. We seek a minimal costs subgraph $S = \langle V, T \rangle$ with $T \subseteq E$ that connects all nodes $v \in V$ to a single component without cycles.

This problem can be solved in polynomial time, e.g. with the well-known algorithms from Kruskal [71] or Prim [95] for the MSTP. However, there are several $\mathcal{NP}$-hard extensions which add additional constraints to the MSTP, such as a maximal degree for the nodes, or a maximal diameter of the tree, etc.

Since real world network systems are becoming larger and more complex, the need of more sophisticated models arises. For example, with increasing number of local networks, it makes sense to connect them to a new global network. This involves choosing one computer from each local network to be used as an entrance gate for the global backbone. Obviously, the old model of MSTP is not sufficient anymore. This motivates the introduction of Generalized Network Design Problems (GNDPs).

1.1 Considered Problems

In contrast to classical NDPs, the generalized versions are defined on graphs whose node sets are partitioned into clusters. We consider the variant where the goal is to find a subgraph connecting *exactly one* node from each cluster, as well as satisfying other constraints adopted from their classical counterparts.

For all GNDPs considered in this thesis, we are given an undirected weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r,\ \bigcup_{i=1,\ldots,r} V_i = V,\ V_i \cap V_j = \emptyset\ \forall i, j = 1, \ldots, r,\ i \neq j$.

A solution defined on G is a subgraph $S = \langle P, T \rangle$ with $P = \{p_1, p_2, \ldots, p_r\} \subseteq V$ containing one node from each cluster, i.e. $p_i \in V_i$ for all $i = 1, \ldots, r$. Depending on the actual problem, different requirements have to be fulfilled by the subset of edges $T \subseteq E$. The costs of S are its total edge costs, i.e. $C(T) = \sum_{e \in T} c(e)$, and the objective is to identify a solution with minimum costs.

Generalized Minimum Spanning Tree Problem (GMSTP): Introduced by Myung et al. [86], the objective is to find a spanning tree of minimal costs which contains one node from each cluster. We consider the clustered graph $G = \langle V, E, c \rangle$. The requirements for a feasible solution $S = \langle P, T \rangle$ is that P contains exactly one node from each cluster and T connects all these nodes without cycles, see Figure 1.1.

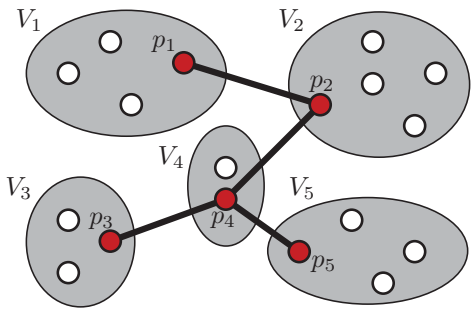


Figure 1.1: Example for a solution to the GMSTP.

Generalized Traveling Salesman Problem (GTSP): This problem was introduced independently by Henry-Labordere [51], Srivastava et al. [112], and Saskena [106]. The goal is to find a node disjoint round trip of minimal costs which spans one node

in each cluster. We consider the clustered graph $G = \langle V, E, c \rangle$. A feasible solution to the GTSP is a subgraph $S = \langle P, T \rangle$ with P containing exactly one node from each cluster and T being a round trip on these nodes, see Figure 1.2.

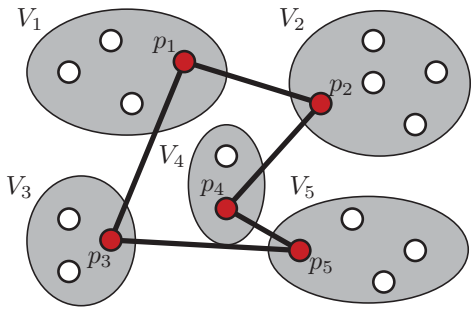


Figure 1.2: Example for a solution to the GTSP.

As a practical extension of the classical TSP and a variant of the GTSP, we also consider the Railway Traveling Salesman Problem (RTSP): We are given a railway network, a train schedule, and a salesman who has to visit a number of cities to carry out some business. He starts and ends at a specified home city, and the required time for the overall journey, including waiting times, shall be minimized.

Generalized Minimum Edge Biconnected Network Problem (GMEBCNP): Huygens [62] was the first one to examine this problem. Extending the GMSTP, we seek an edge biconnected network of minimal costs which connects one node from each cluster. We consider the clustered graph $G = \langle V, E, c \rangle$. A feasible solution to the GMEBCNP is a subgraph $S = \langle P, T \rangle$ with P containing exactly one node from each cluster and T connecting all nodes in P via edge redundancy, i.e. for each pair of nodes $u, v \in P,\ u \neq v$, there must exist at least two edge-disjoint paths, see Figure 1.3.

While for some classical NDPs there exist efficient algorithms, most real-world NDPs and especially the more complex GNDPs are typically difficult to solve. Because of the inherent difficulty and the enormous practical importance of these problems, a large number of techniques for solving them has been proposed in the last decades. It is clear that a very wide range of industries and commercial and government enterprises could find benefits if these NDPs could be solved better – with respect to run-time and/or solution quality – than before.

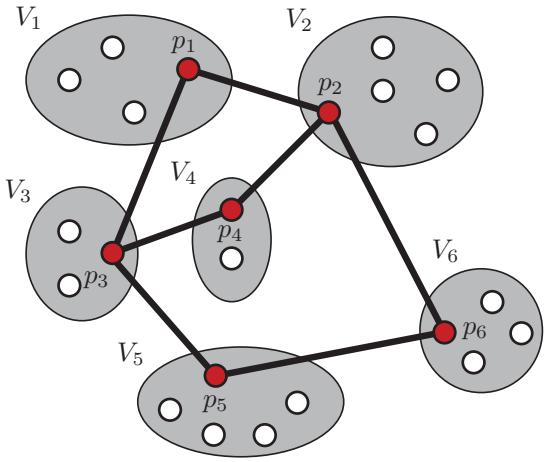


Figure 1.3: Example for a solution to the GMEBCNP.

1.2 Methodology

Techniques for solving these optimization problems can roughly be classified into two main categories: *exact* and *heuristic* algorithms. Exact algorithms are guaranteed to find an optimal solution and to prove its optimality; the run-time, however, often increases dramatically with a problem instance's size, and often only small or moderately-sized instances can be practically solved to provable optimality. For larger instances the only possibility is usually to turn to heuristic algorithms that trade optimality for run-time; i.e., they are designed to obtain good but not necessarily optimal solutions in acceptable time.

When considering exact approaches, the following techniques have had significant success: *Branch-and-Bound* (B&B), *Dynamic Programming* (DP), *Constraint Programming* (CP), and in particular the large class of *Integer Linear Programming* (ILP) techniques including cutting plane methods, linear programming and Lagrangean relaxation based approaches, branch-and-cut, branch-and-price, and branch-and-cut-and-price [87, 90].

On the heuristic side, local search based *metaheuristics* have proven to be highly useful in practice. This category of problem solving techniques include, among others, simulated annealing [68], tabu search [40], iterated local search [81], variable neighborhood search [47], various population-based models such as evolutionary algorithms [3], scatter search [41], and memetic algorithms [85], and estimation of distribution algorithms such as ant colony optimization [21]. See also [39, 54] for more general introductions to metaheuristics.

Looking at the assets and drawbacks of exact techniques and metaheuristics, the approaches can be seen as complementary. As a matter of fact, it appears to be natural to combine ideas from both streams. Puchinger and Raidl [97] present a classification with respect to the hybridization of metaheuristics with exact optimization techniques.

For solving the considered GNDPs in this thesis, we mostly use approaches based on a Variable Neighborhood Search (VNS) framework which combines multiple neighborhood search strategies and even utilizes different solution representations. Some of the considered neighborhood structures can be seen as dual to each other, making their combination particularly powerful. Some of them are exponentially large, rendering naive exhaustive exploration inapplicable. However, we present efficient techniques for finding optimal or near optimal solutions in these neighborhoods via DP, ILP, or other sophisticated methods.

1.3 Overview of the Thesis

An introduction to exact algorithms, in particular linear and integer linear programming is given in Chapter 2. Chapter 3 covers some popular metaheuristics like local search, VNS, tabu search, and evolutionary algorithms. In addition, a new, self-adaptive variant of VNS is proposed. This work was also published in:

> Bin Hu and Günther R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In Carlos Cotta, Antonio J. Fernandez, and Jose E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*. Malaga, Spain, 2006.

Since many approaches for GNDPs in this thesis do not rely on algorithms of a single stream but on combinations of them, we will examine concepts of hybrid approaches, along with some up-to-date representative examples in Chapter 4.

Chapter 5 provides an overview on the GNDPs considered in this thesis and describes the general strategy applied in order to tackle these problems. In the following chapters, actual work on the GNDPs will be presented in detail.

Chapter 6 is dedicated to the generalized minimum spanning tree problem. A VNS approach will be presented that uses three different neighborhood types. Two of them work in complementary ways in order to maximize search performance. Both are large in the sense that they contain exponentially many candidate solutions, but efficient polynomial-time algorithms are used to identify best neighbors. For

the third neighborhood type we apply integer programming to optimize local parts within candidate solution trees.

This chapter was published in:

> Bin Hu, Markus Leitner, and Günther R. Raidl. Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. *Journal of Heuristics*, volume 14(5), pages 473–499, 2008.

An earlier version and preliminary results were published in:

> Bin Hu, Markus Leitner, and Günther R. Raidl. Computing generalized minimum spanning trees with variable neighborhood search. In Pierre Hansen, Nenad Mladenovic, Jose A. Moreno Perez, Belen Melian Batista, and J. Marcos Moreno-Vega, editors, *Proceedings of the 18th Mini Euro Conference on VNS*. Tenerife, Spain, 2005.

Work on the generalized traveling salesman problem is presented in Chapter 7. A VNS algorithm based on two complementary, large neighborhood structures is proposed. One of them is the already known generalized 2-opt neighborhood for which a new incremental evaluation technique is described, which speeds up the search significantly. The second structure is based on node exchanges and the application of the chained Lin-Kernighan heuristic.

This chapter was published in

> Bin Hu and Günther R. Raidl. Effective neighborhood structures for the generalized traveling salesman problem. In Jano van Hemert and Carlos Cotta, editors, *Evolutionary Computation in Combinatorial Optimisation – EvoCOP 2008*, volume 4972 of LNCS, pages 36–47, Springer. Naples, Italy, 2008.

and won the best paper award of this conference.

Chapter 8 covers the generalized minimum edge biconnected network problem. Three VNS variants are considered that utilize different types of neighborhood structures, each of them addressing particular properties as spanned nodes and/or the edges between them. For the more complex neighborhood structures, efficient techniques – such as a graph reduction – are applied to essentially speed up the search process. For comparison purpose, a mixed integer linear programming formulation based on multi commodity flows is proposed to solve smaller instances of this problem to proven optimality.

This chapter was published in:

> Bin Hu, Markus Leitner, and Günther R. Raidl. The generalized minimum edge biconnected network problem: Efficient neighborhood structures for variable neighborhood search. *Networks*. Accepted for publication, 2007.

An earlier version and preliminary results were published in:

> Markus Leitner, Bin Hu, and Günther R. Raidl. Variable neighborhood search for the generalized minimum edge biconnected network problem. In Bernard Fortz, editor, *Proceedings of the International Network Optimization Conference – INOC 2007*, pages 69/1–6. Spa, Belgium, 2007.

Furthermore, a talk with preliminary results was given:

> Bin Hu. Efficient neighborhoods for the generalized minimum edge biconnected network design problem. *Austrian Workshop on Metaheuristics 4*. Vienna, Austria, 2006.

As a related problem to the GTSP, the Railway Traveling Salesman Problem (RTSP) is considered in Chapter 9. We are given a timetable and a salesman who has to visit a number of cities by train to carry out some business. He starts and ends at a specified home city, and the required time for the overall journey, including waiting times, shall be minimized. Two transformation schemes to reformulate the problem as either a classical asymmetric or symmetric traveling salesman problem (TSP) are presented. Using these transformations, established algorithms for solving the TSP can be used to attack the RTSP as well.

This chapter was published in:

> Bin Hu and Günther R. Raidl. Solving the railway traveling salesman problem via a transformation into the classical traveling salesman problem. In Fatos Xhafa, Francisco Herrera, Ajith Abraham, Mario Köppen, and Jose Manuel Benitez, editors, *8th International Conference on Hybrid Intelligent Systems – HIS 2008*, pages 73–77. Barcelona, Spain, 2008.

A talk was given in:

> Bin Hu. Solving the railway traveling salesman problem via a transformation into the classical traveling salesman problem. *Austrian Workshop on Metaheuristics 6*. Vienna, Austria, 2008.

Finally, conclusions are made in Chapter 10.

Chapter 2

Exact Algorithms

Many Combinatorial Optimization Problems (COPs) can be modelled as a (integer) linear program. While Linear Programs (LPs) can be solved efficiently in practice via the well known simplex algorithm and, from a theoretical point, even in polynomial time via the ellipsoid-method [66] and interior-point methods [64], Integer (Linear) Programs (IPs) are in general $\mathcal{NP}$-hard.

Based on books on linear optimization by Bertsimas and Tsitsiklis [7] and combinatorial and integer optimization by Nemhauser and Wolsey [87], this chapter will give a brief introduction to LPs and IPs, as well as algorithms for solving them.

2.1 Linear Programming

A linear program can be defined as:

$$\begin{aligned} \min\, & c^T x \\ & Ax \;\geq\; b \end{aligned}$$

with $c, x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$. While x is the n-dimensional solution vector that should be optimized, vector c with the same dimesions characterizes the optictive function $c^T x$. Similarly, matrix A, together with vector b, form the constraints of the LP.

Equalities can be modelled as two inequalities and for maximization problems, we only need to multiply the objective function with -1. Unless stated otherwise, we will consider minimization problems only, since all of the considered network optimization problems requires to minimize the connection costs. Therefore, we denote the standard of an LP as:

$$z^{\mathrm{LP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{R}^n\} \tag{2.1}$$

2.2 Integer Linear Programming

Consider the LP (2.1), if we ask for integer solutions, i.e. $x \in \mathbb{Z}^n$, we would get an integer linear program

$$z^{\mathrm{IP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{Z}^n\} \tag{2.2}$$

A common and widely-used variant is the so-called 0/1 IP where $x \in \{0,1\}^n$. If some of the variables in IP (2.2) needs to be integers and others not, we call the system a Mixed Integer Program (MIP).

2.3 Geometric Interpretation

Given a linear program $z^{\mathrm{LP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{R}^n\}$, the set of feasible solutions is denoted by a *polyhedron*, defined as

$$P = \{x \in \mathbb{R}^n \mid Ax \geq b,\ A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^n\} \tag{2.3}$$

P is one of following three different types:

- $P = \emptyset \Rightarrow$ the LP is infeasible
- $P \neq \emptyset$ but $\nexists \inf\{c^T x \mid x \in P\} \Rightarrow$ the LP is feasible, but unbounded.
- $P \neq \emptyset$ and $\exists \min\{c^T x \mid x \in P\} \Rightarrow$ the LP is feasible and an optimal solution $x^* \in P,\ c^T x^* = \min\{c^T x \mid x \in P\}$ exists.

This leads to the following definitions.

Definition 1 *A polyhedron $P \subset \mathbb{R}^n$ is* bounded *if there exists a constant k such that $|x_i| < k \forall x \in P, i = 1, \ldots, n$. Such a polyhedron is called a* polytope.

Definition 2 *A set* $S \subset \mathbb{R}^n$ *is* convex *if* $\lambda x + (1-\lambda)y \in S,\ \forall x, y \in S,\ \lambda \in [0,1]$ *holds.*

Definition 3 *Given* $X = \{x^1, \ldots, x^k\}$, *with* $x^i \in \mathbb{R}^n,\ \lambda_i \geq 0,\ i = 1, \ldots, k$ *and* $\sum_{i=1}^{k} \lambda_i = 1$. *Then*

(1) the vector $\sum_{i=1}^{k} \lambda_i x^i$ *is called a* convex combination *of* X*;*

(2) the convex hull *of* X *denoted as* $conv(X)$ *is the set of all convex combinations of* X*.*

Definition 4 *Consider a polyhedron* P *defined by linear equality and inequality constraints, and let* $x^* \in \mathbb{R}^n$*.*

(a) The vector x^* *is a* basic solution *if:*

(1) All equality constraints are satisfied;

(2) There are n *linearly independent constraints that are active (i.e. that hold with equality) at* x^**.*

(b) If x^* *is a basic solution that satisfies all constraints, it is called a* basic feasible solution.

Vertices of a polyhedron have special properties on feasible solutions of the corresponding LP. The following two theorems are essential for the *simplex algorithm*, which will be described in the next section.

Theorem 1 *Let* P *be a nonempty polyhedron and let* $x \in P$*.*
Then the following statements are equivalent:

(a) x *is a vertex;*

(b) x *is a basic feasible solution.*

Theorem 2 *Given an LP (2.1), following statements are true:*

(1) If polyhedron P *in (2.3) is nonempty, there exists a basic feasible solution.*

(2) If (2.1) has an optimal solution, then there is an optimal basic feasible solution.

2.4 Simplex Algorithm

The simplex algorithm is one of the most popular methods to solve linear programs. The optimal solution of an LP is usually found by a *two-phase simplex* approach:

1. Find an initial basic feasible solution, which is a vertex of the given polyhedron.
2. Iteratively move to an adjacent vertex while gradually improve the solution.

Finding an initial basic feasible solution can already be a difficult task for some optimization problems. This is why we use a two-phase simplex, where a preliminary problem is solved in the first phase to obtain a feasible solution.

The algorithm terminates after reaching a vertex representing the optimal solution, where none of the adjacent vertices results in an improvement, or if an unbounded facet is found.

Consider an LP in standard form. For the polyhedron $Ax \geq b$ we first introduce a vector of slack variables $\sigma \in \mathbb{R}^m$, resulting in an equation system $Ax + \sigma = b$ which e.g. can be solved via Gaussian elimination method. This yields the initial basic feasible solution x^0. Let $B(1), \ldots, B(m)$ denote the indices of the basic variables and $B = [A_{B(1)} \ldots A_{B(m)}]$ denote the corresponding basic matrix. To move from one vertex of the polyhedron to an adjacent one means to introduce a new variable into the basis and simultaneously removing an existing variable. This procedure is also called *pivoting*.

Definition 5 *Let x^0 be a basic solution, B be the associated basis matrix, and c_B be the vector of costs of the basic variables. For each j, we define the* reduced cost $\overline{c_j}$ *of the variable x_j according to*

$$\overline{c}_j = c_j - c_B B^{-1} A_j \tag{2.4}$$

The reduced costs $\overline{c_j}$ change with the variable x_j, so when we put variables with negative reduced costs (in case of a minimization problem) into the basis, the basic feasible solution will be improved. If no such variables exist, the current solution is already optimal.

The simplex algorithm was developed by George Dantzig in 1947 and is one of the most efficient methods to solve LPs in practice. However, there are worst-case examples which cannot be solved by simplex in polynomial-time. Nevertheless the LP problem is solvable in polynomial-time, which was first shown by Khachiyan [66] in 1979 using the so-called *ellipsoid-method*. Other polynomial-time algorithms of more practical interest are the *interior-point* methods, introduced by Karmakar [64]

in 1984. More highly-competitive algorithms were developed from then on. Most state-of-the-art commercial LP-solvers incorporate interior-points method such as barrier or primal-dual algorithms.

2.5 Duality

The duality property are of central importance for some advanced methods like primal/dual or column generation algorithms when solving LPs. Given a *primal* problem z^{LP} stated as in 2.1, its *dual* problem is denoted as:

$$w^{\mathrm{LP}} = \max\{ub \mid uA \leq c,\ u \in \mathbb{R}^m\} \tag{2.5}$$

Based on this definition, we can formulate the following theorems.

Proposition 1 *The dual of the dual problem is the primal problem.*

Proposition 2 (Weak Duality) *If x is primal feasible and u is dual feasible, then*

$$cx \leq ub.$$

The next two theorems are fundamental results of LP duality and therefore exploited by primal/dual algorithms. If the primal problem and the dual problem are both feasible, their optimal values are equal.

Theorem 3 (Strong Duality) *If z^{LP} or w^{LP} is finite, then both (2.1) and (2.5) have the same finite optimal value*

$$z^{\mathrm{LP}} = w^{\mathrm{LP}}.$$

Proposition 3 (Complementary slackness) *If x^* and u^* are feasible solutions for the primal (2.1) and the dual (2.5) problem respectively, then x^* and u^* are optimal solutions if and only if*

$$\begin{aligned} u_i(b - Ax)_i &= 0, \quad \forall i, \\ x_j(uA - c)_j &= 0, \quad \forall j. \end{aligned}$$

2.6 LP-based Branch-and-Bound

While it is possible to solve LPs in polynomial time, solving IPs and MIPs is $\mathcal{NP}$-hard in general. A possible way to handle $\mathcal{NP}$-hard problems are the Branch-and-Bound (B&B) approaches, which follow the idea of divide and conquer. The basic principle of B&B is to divide the problem into subproblems for which we calculate bounds. For minimization problems, lower bounds $\underline{z}_i$ can be obtained by solving *relaxations* of the subproblems P_i. Upper bounds $\overline{z}$ are calculated by solving the (sub)problems heuristically. For each subproblem, further procedures are decided according the relation of these bounds:

- $\underline{z}_i = \overline{z}$: optimal solution for P_i is found.
- $\underline{z}_i < \overline{z}$: divide P_i further into subproblems and continue B&B process.
- $\underline{z}_i > \overline{z}$: since lower bound is higher than upper bound, subproblem P_i does not contain optimal solution and thus we can prune it.

Relaxation are methods of approximating a complex problem by simplifying some constraints. For solving IPs, LP relaxations are commonly used.

Definition 6 *The LP relaxation for the IP* $z^{\mathrm{IP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{Z}^n\}$ *is the LP* $z^{\mathrm{LP}} = \min\{cx \mid Ax \geq b,\ x \in \mathbb{R}^n\}$

Proposition 4 *If a LP is the relaxation of an IP, then* $z^{\mathrm{LP}} \leq z^{\mathrm{IP}}$.

Furthermore, any feasible solution of the IP is a valid upper bound. Detailed B&B algorithm is described in Algorithm 1.

Branching

Usually the search space is partitioned into two parts during branching. When we solve the LP relaxation of problem P_i to obtain x_i^{LP} and it is not integral, it contains at least one fractional value. Let $x_j(x_i^{\mathrm{LP}})$ denote such a value. Then we get two new subproblems by rounding it up and rounding it down, respectively.

$$\begin{aligned} P_{k+1} &= P_i \cap \{x : x_j \leq \lfloor x_j(x_i^{\mathrm{LP}}) \rfloor\} \\ P_{k+2} &= P_i \cap \{x : x_j \geq \lceil x_j(x_i^{\mathrm{LP}}) \rceil\} \end{aligned}$$

Algorithm 1: LP-based Branch-and-Bound

Input: Initial problem P
Initialization: Upper bound $\overline{z} := \infty$
List of problems $L := \{P\}$
while $L \neq \emptyset$ **do**
  Choose and remove problem P_i from L
  Solve LP relaxation over P^i yielding solution x_i^{LP} with objective value $\underline{z}_i$
  if P_i *is infeasible* **then**
    Prune by infeasibility
  else if $\underline{z}_i \geq \overline{z}$ **then**
    Prune by bound
  else if x_i^{LP} *integer* **then**
    $\overline{z} := \underline{z}_i$
    Incumbent $x^* := x_i^{\mathrm{LP}}$
    Prune by optimality
  else
    Branching: put subproblems P_{i_1} and P_{i_2} into L
Incumbent x^* is optimal solution to p.

One possibility to choose the branching variable is to take a fractional value which is as close to 0.5 as possible. However, there are more sophisticated methods like *strong branching* [120], which calculates bounds for all possible variables which come into question for branching. The value with most promising bounds is then actually taken.

Choosing the next subproblem

Quickly obtaining feasible and good integral solutions most likely leads to good upper bounds which helps at pruning the search space. Strategies like *Dept-First Search* meets this purpose, where newly generated subproblems are favored for further advancing. On the other hand, it can be advantageous to keep the total number of considered subproblems low. This can be done by following *Best-First Search* strategy where the subproblem with lowest lower bound is chosen first. Most commercial IP solvers use more sophisticated combinations of different strategies to make this decision.

2.7 Cutting Plane Algorithms and Branch-and-Cut

Branch-and-Cut (B&C) algorithms are among the most powerful and popular methods for solving IPs and MIPs. Basically, B&C results from enhancing the performance of LP based B&B by making use of *Cutting Plane Algorithms*.

2.7.1 Cutting Plane Algorithms

Solving complex IPs with many constraining (in)equations can be very difficult. In practice, it is often not necessary to take all constraints into account, but only a possibly small subset of them. Cutting plane algorithms exploit this observation and tries to solve a simplified model of the IP with only a small number of constraints. If the emerging solution does not violate any constraints, it is valid and therefore optimal already. If some constraints are violated, we need to add them to the model. This process is repeated until no more constraints need to be added. A pseudocode of the cutting plane algorithm is described in Algorithm 2.

Definition 7 *The* separation problem *associated with IP (2.2) is defined as: Given* $\hat{x} \in \mathbb{R}^n$*, if* $\hat{x} \notin conv(X)$*, find a valid inequality* $a^T x \geq b_j$ *violated by* $\hat{x}$*.*

Algorithm 2: Cutting plane algorithm

```
Start with simplified model, only containing a subset of constraints
loop
    Solve model, yielding solution x*
    Solve the separation problem for x*
    if ∃ constraint a^T x ≥ b_j such that a^T x* < b_j then
        Add constraint a^T x ≥ b_j to the model
    else
        Return x* as the optimal solution for the original model
```

2.7.2 Branch-and-Cut

Branch-and-Cut B&C is a hybridization of B&B with the cutting plane algorithm to solve IP models. For each subproblem handled by the B&B algorithm, it tries to apply the cutting plane algorithm in order to get a tight bound.

To make the B&C algorithm efficient, following issues are of great importance:

- Good LP relaxations
- Efficient algorithm for separation problem
- Cut strategies
- Cut pool management

2.8 Column Generation and Branch-and-Price

In some cases, the IP model contains exponentially many variables, but a manageable amount of constraints. In such cases, column generation algorithms can be very efficient, which add variables to the model dynamically. Branch-and-Price (B&P) is the combination between B&B and column generation.

2.8.1 Column Generation

In the simplex algorithm, variables (columns) with negative reduced costs are added to the basis in each iteration to improve the solution. If the number of variables is very large, we might want to start with only few of them in the basis at the beginning. Afterwards, we must solve the so-called *pricing problem* for which we seek the variable with minimal reduced costs to add it to the basis. This is repeated until no more variables with negative reduced costs exist, see Algorithm 3.

Algorithm 3: Column generation

```
Start with a subset of the variables: Restricted Master Problem (RMP)
Solve RMP
while variable with negative reduced costs c̄_j exists do
    Determine such a variable
    Add it to the RMP
    Resolve RMP
```

2.8.2 Branch-and-Price

With the column generation method, we are able to enhance the performance of the simplex method when solving LPs. However, when we seek integral solutions of IPs or MIPs, we still need to utilize B&B. Just like B&C in the previous chapter, Branch-and-Price (B&P) is the hybridization of B&B with column generation. The later solves the LP relaxation by adding variables to the model until we get an

optimal LP solution. If it is not integral, we need to branch at fractional variables, and the relaxations of resulting subproblems will be solved via column generation again.

The branching process is more difficult now, since it makes no sense to address the exponentially many variables directly.

B&C algorithms and B&P algorithms theoretically can be seen as dual to each other. To some extend, they are comparable, since variables in the primal problem correspond to constraints in the dual problem, and vice versa. Hence, column generation can be seen as the dual problem to separating cutting planes.

Chapter 3

Metaheuristics

For $\mathcal{NP}$-hard optimization problems, it is often impossible to apply exact methods to large instances in order to obtain optimal solutions in acceptable time. In such cases, metaheuristics can be seen as alternatives, which are often able to provide excellent, but not necessarily optimal solutions in reasonable time. The term *metaheuristic* was first introduced by Glover [38] and refers to a number of high-level strategies or concepts of how to solve optimization problems. It is somewhat difficult to specify the exact boundaries of this term. Voss [118] gives the following definition:

> A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method.

This chapter will first introduce the basic concepts of constructive heuristics and local search. Then, we will consider some popular examples of the family of metaheuristics.

3.1 Constructive Heuristics

Constructive or construction heuristics are mostly used for creating initial solutions in a very short time. These solutions can be used for other algorithms which improve them iteratively, or even for exact algorithms as bounds. Constructive heuristics

usually work straightforward, i.e. once they make a decision to build up a partial solution, they never reconsider it. Because of their simple nature, the computational complexity can usually be evaluated accurately. In many cases, not only the complexity, but also the solution quality can be estimated.

Approximation Algorithms

When we can provide the solution quality of a constructive heuristics in term of bounds, we also call them approximation algorithms [117].

Let A be a heuristic which provides for each instance I of an optimization problem Π a valid solution. Let $c_A(I)$ denote the objective value of the solution generated by heuristic A on instance $I \in \Pi$ and $c_{\text{opt}}(I)$ be the objective value of the optimal solution to I.

Definition 8 *For a minimization problem, if*

$$\exists \varepsilon > 0 : \quad \frac{c_A(I)}{c_{\text{opt}}(I)} \leq \varepsilon, \quad \forall I \in \Pi$$

then A is an ε-approximation algorithm and ε is approximation factor of A.

There are different classes of approximation algorithm: those with absolute and relative approximation factors and (fully) polynomial time approximation schemes. We can also classify the complexity of COPs based on if they are approximable or not.

3.2 Local Search

In contrast to constructive heuristics which generate initial solutions for COPs, Local Search (LS) algorithms improve existing solutions further by applying local changes. The main idea is to create *neighborhood structures* which can be defined for each COP specifically. LS iteratively moves from one solution x to another within the so-called neighborhood $\mathcal{N}(x)$. The final goal is to reach the optimal solution, measured by objective function $f(x)$. There is no guarantee that the optimum will be found, though.

Definition 9 *A neighborhood structure $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ is a function associating a set of neighbors, called neighborhood $\mathcal{N}(x) \subseteq \mathcal{S}$ to every solution $x \in \mathcal{S}$.*

The procedure of changing a solution within its neighborhood $\mathcal{N}(x)$ is also called as applying a move. Algorithm 4 shows the the basic LS in detail.

Algorithm 4: Basic local search (x)

Input: Initial solution x
Output: Improved solution x

repeat
  choose $x' \in \mathcal{N}(x)$
  if $f(x')$ *better than* $f(x)$ **then**
    $x := x'$
until *termination condition(s) met*

There are three possibilities of how to choose x' among neighbor solution $\mathcal{N}(x)$.

- **Random neighbor**: A random solution $x' \in \mathcal{N}(x)$ is chosen. This is the fastest variant, but x' is often even worse than x.
- **Best improvement**: We search the neighborhood $\mathcal{N}(x)$ completely and the best solution x' is chosen.
- **Next improvement**: We search the neighborhood $\mathcal{N}(x)$ systematically and the first solution x' better than x is chosen.

In practice, LS with random neighbor strategy requires most number of iterations and LS with best improvement strategy requires least number of iterations to get to the final solution. However, the computational effort required for each iteration differ greatly. While for the random neighbor strategy only one neighbor solution has to be computed, the best improvement strategy usually requires to systematically check all solutions in the neighborhood in order to determine the best solution. The next improvement strategy sometimes can be advantageous if searching the neighborhood is done in a clever way, e.g. by using incremental evaluation schemes.

Basic LS is often terminated when there is no better solution in neighborhood $\mathcal{N}(x)$ than x itself. In this case, LS cannot improve x anymore and x is *local optimal* in regard to $\mathcal{N}(x)$.

Definition 10 *x is a local optimum* $\leftrightarrow \forall x' \in \mathcal{N}(x) : f(x') \geq f(x)$

A local optimum is not necessarily a *global optimum* which we are ultimately looking for. However, every global optimum is a local optimum.

3.3 Simulated Annealing

Since basic local search terminates after reaching the first local optimum in the optimization process, it highly depends on the initial solution whether the global optimum can be found or not. One straightforward approach for escaping local optima is to not only allow improvements, but also accept solutions that are worse than the current one under certain conditions.

Simulated Annealing (SA) [68] follows this concept. The name was inspired by the annealing process in metallurgy. It uses the random neighbor strategy to generate a new solution x' in the neighborhood of the current solution x in each iteration. If x' is better than x, SA proceeds with the new solution. On the other hand, if x' is worse, then it is not discarded immediately, but it can be accepted with a certain probability that depends on the difference in the objective function $f(x') - f(x)$ and on a *temperature* value. The lower the gap and the higher the temperature, the more likely the worse solution will be accepted. The temperature is initialized with a sufficiently hight value and it decreases after each iteration depending on the *cooling* strategy that is used. Hence SA accepts worse solutions more often during the beginnings of the optimization process but it becomes more and more similar to LS towards the end.

3.4 Variable Neighborhood Search

Another way to enhance basic local search is to make use of following considerations:

- Use multiple neighborhood structures instead of a single one
- Include techniques to escape local optima

Variable neighborhood search, introduced by Hansen and Mladenovic [48] follows these ideas. When designing this metaheuristic, additional, partially empirical observations have been taken into account:

- A local optimum with respect to one neighborhood structure is not necessarily so 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.

3.4.1 Variable Neighborhood Descent

Generally speaking, Variable Neighborhood Descent (VND) is similar to basic LS, but it uses more than one neighborhood structure. Let $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$ be the neighborhood structures. VND changes between them in order to improve the initial solution x until it reaches a local optimum with respect to all of these neighborhood structures, see Algorithm 5.

Algorithm 5: Variable Neighborhood Descent (x)

Input: Initial solution x
Given neighborhoods $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$
Output: Improved solution x

```
k := 1
repeat
    choose x' ∈ N_k(x)
    if f(x') better than f(x) then
        x := x'
        k := 1
    else
        k := k + 1
until k = k_max
return x
```

How to order the neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k_{\max}}$ is important for the performance of VND, but this is not trivial. Following criteria my be taken into consideration when fixing the order.

- The relationship of neighborhoods: They can be
  - overlapping
  - one (entirely) including an other
  - mutual exclusive
  - a mixture between these, etc.
- The complexity of neighborhoods
- The coverage of solution space

If the relationship is overlapping or even mutual including, it is often a good idea to start with the smallest one and gradually moving on to the larger ones. If the neighborhood structures are rather independent, it is standard to order them by increasing complexity at evaluation.

Looking at the VND procedure, it obviously cannot escape local optima. Hence we might want to include some mechanisms to do so.

3.4.2 Self-Adaptive Variable Neighborhood Descent

When using VND it is often difficult to decide the ordering of neighborhoods which are considered during the search procedure. This arrangement typically strongly affects the computation time as well as the quality of the finally obtained solution. In this section, which is based on [58], a new VND variant is presented that orders the neighborhoods dynamically in a self-adaptive way during the optimization process. Each neighborhood structure has associated a rating which is updated according to observed success probabilities and required times for evaluation.

Obviously, neighborhoods ranked in the front are searched more often than others at the end of the queue. If the times required for examining the neighborhoods differ substantially, it is reasonable to order them according to increasing complexity. However, this criterion is not always applicable, in particular when the times for searching the neighborhoods are similar, or if they are unpredictable. The latter case appears often when next-improvement strategy is used instead of best-improvement.

The best suited neighborhood ordering may also depend on specific properties of the particular problem instance and the current state of the search process. Research in the direction of controlling and dynamically adapting the ordering of neighborhood structures is yet limited. For example, Puchinger and Raidl [98] presented a variant of VNS in which relaxations of the neighborhoods are quickly evaluated in order to choose the most promising neighborhood next. This method is effective, however, it requires the existence of fast methods for solving relaxations of the neighborhoods. A more general variant is the "choice function" which is often used in hyperheuristics for selecting low-level heuristics [11, 65].

For the Self-Adaptive Variable Neighborhood Descent (SAVND), neighborhood structures are dynamically rearranged according to their observed benefits during the search process. An initial neighborhood ordering, i.e., a permutation $\lambda = (\lambda_1, \ldots, \lambda_n)$ of $\{1, \ldots, n\}$ is chosen in some intuitive way (or even at random). Each neighborhood structure $\mathcal{N}_i$, $i = 1, \ldots, n$, gets assigned a rating $w_i > 0$, which is initially set to some constant value W being a rough estimation of the average time for evaluating a neighborhood. During the search process, when a neighborhood $\mathcal{N}_{\lambda_i}(x)$ of a current solution x has been investigated, rating w_{λ_i} is updated in dependence of the success and the computation time t_{λ_i} required for evaluation: If an improved solution has been found in $\mathcal{N}_{\lambda_i}(x)$, w_{λ_i} becomes halved and $\frac{t_{\lambda_i}}{\alpha}$ is added; α is a strategy parameter controlling the influence of the evaluation time in

this case. If the search of $\mathcal{N}_{\lambda_i}(x)$ was not able to identify a superior solution, we add time t_{λ_i} to w_{λ_i}. Depending on how much time the evaluation of the neighborhoods is generally required, we initialize $w_i = \varepsilon,\ i = 1, \ldots, n$ with $\varepsilon > 0$ being a small number.

Algorithm 6: Self Adaptive Variable Neighborhood Descent (x)

Input: Initial solution x
Output: Improved solution x

$w_1 := w_2 := \ldots := w_n := W$
$w_{\min} := w_{\max} := W$
$\lambda := (1, 2, \ldots, n)$
$i := 1$
repeat
  find the best neighbor $x' \in \mathcal{N}_{\lambda_i}(x)$, requiring time t_{λ_i}
  if $f(x')$ better than $f(x)$ **then**
    $x := x'$
    $w_{\lambda_i} := \frac{w_{\lambda_i}}{2} + \frac{t_{\lambda_i}}{\alpha}$
    $i := 1$
  else
    $w_{\lambda_i} := w_{\lambda_i} + t_{\lambda_i}$
    $i = i + 1$
  if $w_{\lambda_i} < w_{\min} \vee w_{\lambda_i} > w_{\max}$ **then**
    nextN $:= \lambda_i$ // *store the neighborhood to be considered next*
    sort $\lambda_1, \ldots, \lambda_n$ s.t. $w_{\lambda_1} \leq w_{\lambda_2} \leq \ldots \leq w_{\lambda_n}$
    $w_{\min} := w_{\lambda_1}$
    $w_{\max} := w_{\lambda_n}$
    reset i s.t. $\lambda_i =$ *nextN*
until $i > n$

Permutation λ is not immediately updated after processing a neighborhood in order to avoid too rapid and strong adaptions in case of a temporarily limited extraordinary good or bad behavior. Only when an updated rating w'_{λ_i} is smaller than the so far minimum rating $\min_{j=1,\ldots,n} w_j$ or larger than the maximum rating $\max_{j=1,\ldots,n} w_j$, we redetermine permutation λ by sorting the neighborhood structures according to increasing ratings. SAVND then continues with the structure that would have also been chosen according to the old ordering. Algorithm 6 shows the whole procedure in detail.

3.4.3 Basic Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a simple high level metaheuristic which follows similar ideas as VND. It can collaborate with almost any LS algorithms, such as VND itself. Based on empirical observation that local optima lie near to each other, VNS uses random moves to get from one solution to a neighboring one. These moves are generated by systematically considering a given set of neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{l_{\max}}$ which are usually ordered by size. After applying a random move – this is also called *shaking* – it is common to apply LS to improve the solution. Algorithm 7 shows how VNS works in detail.

Algorithm 7: Basic Variable Neighborhood Search (x)

Input: Initial solution x
Given neighborhoods $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{l_{\max}}$
Output: Improved solution x

```
repeat
    l := 1
    repeat
        x' := Shake(l, x), i.e. choose random solution from N_l(x)
        x' := Local Search(x')
        if f(x') better than f(x) then
            x := x'
            l := 1
        else
            l := l + 1
    until l = l_max
until stopping conditions are met
return x
```

There are many variants of VNS. The most simple form is the *reduced VNS* which does not use any local search at all but only relies on random shaking. On the other hand, the most popular variant is probably *general VNS* which uses VND as local improvement procedure.

Although VNS is a rather new metaheuristic, it is very easy to use. Compared to other metaheuristics, especially tabu search, it has only few adjustable parameters. Hence, creating reasonable neighborhood structures can be sufficient to obtain good results.

3.5 Tabu Search

Tabu Search (TS) [40] can be seen as another popular extension of local search. The central component of TS is the *tabu list* which supports escaping local optima in order to reach for the global optimum. Tabu list is a memory which keeps track of the search progress so far to avoid cycling by trying not to reconsider areas which have already been searched.

Algorithm 8: Tabu Search (x)

Input: Initial solution x
Output: Improved solution x

$x_{\text{best}} := x$
Tabu List $TL := \{x\}$
repeat
  $X' :=$ reduced subset of $\mathcal{N}(x)$ with respect to TL
  choose best $x' \in X'$
  add x' to TL
  remove from TL all elements which are older than t_L iterations
  $x := x'$
  if $f(x)$ better than $f(x_{\text{best}})$ **then**
    $x_{\text{best}} := x$
until termination condition(s) met
return x_{best}

Algorithm 8 shows the procedure of TS in detail. Tabu list TL maintains history of which solutions have been already considered in the past t_L iterations and narrows the neighborhood $\mathcal{N}(x)$ accordingly. The new solution is usually chosen via best improvement strategy, and in contrast to classical LS, inferior ones are accepted as well as superior ones. The overall best solution is kept in x_{best}.

There are two ways to store informations of the solutions in TL:

- Store complete solutions
  This basic approach is illustrated in Algorithm 8. While it is advantageous that previously solutions are certainly not visited again, the drawbacks are the high memory requirement which grows with parameter t_L and the computational effort of verifying if a solution is in TL or not.
- Store solution attributes only
  The more common strategy is to only store solution attributes in TL. All neighbor solutions which contain a tabu attribute are forbidden. To be precise,

when moving from x to its neighbor x', the changed attribute is stored in TL and thus a reverting move is tabu for the next t_L iterations. In contrast to the previous approach, memory consumption and computational effort are significant lower. However, a tabu attribute forbids lots of solutions, some of them have possibly not considered at all. This makes the decision of choosing the right t_L more difficult.

While the tabu list is essential for preventing cycling, it sometimes can become too powerful and thus stagnates the search progress. When only solution attributes are stored, forbidden moves do not necessarily lead to solutions which have been considered. Therefore, *aspiration criteria* are used to cancel tabus. A classical criterion is to perform a tabu restricted move if it would lead to a new overall best solution. Other more sophisticated aspiration criteria have been used in [18, 52].

A parameter of significant importance is the length of the tabu list or *tabu tenure* t_L. If it is too small, cycling can still occur. On the other hand, if it is too long, the search process will be restricted too strongly. As a matter of fact, finding the right tabu tenure is a difficult task in practice and it depends on the problem nature as well as the problem size. Instead of using a statical value for this parameter, there are many approaches of how to change the tabu tenure dynamically [19]:

- depending on elapsed time or iterations [83]
- randomly chosen between an interval for each iteration [114]
- adaptive to the search progress [5]

3.6 Evolutionary Algorithms

Evolutionary Algorithms (EA) are a family of metaheuristics which are inspired by basic principles of natural evolution according to Darwin's theory [15]. First algorithms which use evolution theory to solve combinatorial optimization problems were proposed by Fogel, Owens and Walsh [35] in the 1960s. Two other well known variants are the genetic algorithm by Holland [53] and the evolution strategies by Rechenberg and Schwefel [101, 108]. Since then, these algorithms were improved and extended for many application areas in manifold ways [3].

The main difference between EAs and LS based metaheuristics is the *population* concept. While LS, VNS, etc. only maintain a single current solution during the optimization process, an EA usually operates on a large set of solutions called population. The diversity in this population helps to search in wider areas of the solution space, thus it increases the chance to escape from local optima in order to produce more robust solutions. The pseudocode of a basic EA is given in Algorithm 9.

Algorithm 9: Basic Evolutionary Algorithm

Input: Optimization problem
Output: best solution found during optimization

$P :=$ population set, containing initial solutions
Evaluate(P)
repeat
  $Q :=$ Recombination(P)
  $Q :=$ Mutation(Q)
  Evaluate(Q)
  $P :=$ Selection(P, Q);
until *stopping conditions are met*
return x

Selection

The main idea of selection procedure is to choose $|P|$ solutions of a larger pool of solutions. Like in the nature, better solutions have a better chance of being selected. However, it still should be possible to include inferior solutions. This can be useful when we try to escape from local optima. We will briefly describe three basic selection mechanisms: fitness-proportional selection [53], rank-based selection [4], and tournament selection [25].

Fitness-proportional selection: Let $f(x_i) > 0$ be the objective value (fitness) of solution $x_i \in P, P = \{x_1, \ldots, x_{|P|}\}$. For a maximization problem, the probability of selecting solution x_i is

$$p_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{|P|} f(x_j)}$$

Let $p_s^{\max} = \max\{p_s(x_1), \ldots, p_s(x_{|P|})\}$ and $\overline{p_s} = 1/|P|$. The *selection pressure* is defined as the ratio $p_s^{\max}/\overline{p_s}$, which is the expected probability of how much the best solution is being preferred over an average one.

If selection pressure is too low, good solutions will not be favored enough and thus the search process degenerates to a random search. On the other hand, if the pressure is too high and superior solutions are too much preferred, then diversity will decrease and it will be harder to escape from local optima.

In order to control selection pressure, it is possible to scale the objective value. The most simple way is to use a linear scaling $g(x_i) = a \cdot f(x_i) + b$ with a and b being adjustable parameters. Scaling is also necessary when we consider minimization problems (a is negative in this case) or if $f(x_i)$ can be negative.

Rank-based selection: Instead of using concrete objective values of solutions for the selection probability, it is possible to only consider their ranks. This is done by ordering all solutions $x_i \in P$ according their objective values. The resulting order determines their selection probability.

Tournament selection: This popular selection variant operates quite differently. The following procedure is applied until the new population is completely filled up.

1. Choose k solutions from the population set randomly
2. Add the best of these k solutions to the new population

Selection pressure can easily controlled via parameter k. Unlike the previous two selection mechanisms, no knowledge of the whole population is necessary. Tournament selection can also handle some special optimization problems where actual solutions cannot be rated directly. These problems often appear in game theory where solutions represent game strategies.

Recombination

The purpose of recombination is to generate new offspring solutions based on two (or more) parental solutions. The concept of neighborhood structure is important, since the new solution should inherit attributes of its parents and not purely random. When solutions are encoded as bit-strings, the most simple approach is the one point crossover. It chooses a crossover point randomly and splits each parent solutions into two parts. For the resulting offsprings, the first part of one parental solution is combined with the second part of another parental solution. This one point crossover can be easily extended to two point crossover, multipoint crossover, or uniform crossover.

Permutation is another popular solution representation for COPs. When applying crossover on it, it is important that that resulting offsprings are still valid permutations. Following recombination operators are common examples: partially mapped crossover [42], order crossover [16], and uniform order based crossover [113].

For network problems such as the TSP, when solutions are represented by a set of edges, the edge recombination crossover [13] is also popular. It creates an offspring by taking over edges of parental solutions iteratively according to certain criteria. On network problems, this recombination model performs considerably well since it usually can exploit problem specific informations.

Mutation

Mutation is similar to applying a random move in a certain neighborhood to a solution in LS. This way, lost attributes which do not appear in the whole population have a chance of being introduced again. Usually, mutations are not applied to every solution in the population each iteration, but they only occur with a small probability. For a permutation based representation, a possible mutation could be to exchange the attributes of two positions.

3.7 Memetic Algorithms

A common drawback of EAs is that there is no guarantee for the global best solution to be even local optimal. Though good diversification is present due to a large population, recombination and mutation mechanisms, EAs lack intensification in overall.

Therefore, many successful EAs for complex combinatorial optimization problems additionally use *hybridization* to improve solution quality and/or running time. Pablo Moscato [84] introduced the term Memetic Algorithm (MA) for local search and problem specific knowledge enhanced EAs. The term “meme” corresponds to a unit of imitation in cultural transmission [17]. So while genetic algorithm are inspired by biological evolution, MAs attempts to mimic cultural evolution.

In MAs, While the outer metaheuristic is an EA, individual solutions of the population are further improved e.g. via local search heuristics. If each intermediate solution is always turned into a local optimum, the EA would exclusively search the space of local optima (w.r.t. the neighborhood structure(s) of the local improvement procedure). So by adjusting how much effort is spent in the local improvement, it is possible to tune the balance between intensification and diversification.

Chapter 4

Hybrid Algorithms

Looking at the various exact techniques and metaheuristics described in the previous chapters, each of them has its assets and drawbacks. As a matter of fact, it appears to be natural to combine ideas from multiple algorithmic streams. Several publications of the last years describe different kinds of such hybrid optimizers that are often significantly more effective in terms of running time and/or solution quality since they benefit from synergy. See [24, 97] which illustrates the many different possibilities of combinations and the huge potential they have.

This chapter will focus on embedded techniques since they are implemented in this thesis. They are possibly the most straightforward way of how to combine different approaches. The basic idea is to let one algorithm act as a subordinate of another one. One popular strategy is to apply some local search or more complex algorithms within an outer metaheuristic for "fine-tuning". Variable neighborhood search or memetic algorithms introduced in the previous chapter are typical examples for such a collaboration – while the outer algorithms creates diversity, the inner local search heuristics emphasize intensification. To go one step further, such collaborations between exact and heuristic approaches seem to provide even more hybridization possibilities.

4.1 Exact Algorithms as Subordinates of Metaheuristics

In order to enhance the performance of metaheuristics, exact algorithms can be used to solve parts or subproblems during the optimization process.

4.1.1 Explore Large Neighborhoods by Exact Methods

Numerous local search based algorithms use neighborhoods $\mathcal{N}_k$ that lead to moves referred to as k-*exchange* or k-*opt* with $k = 1$ or $k = 2$. These simple neighborhoods are characterized by the fact that they consider only the change of one or two component(s) of the current configuration vector at once.

Such algorithms are fast but often produce poor suboptimal solutions. To improve this behavior, one can increase k, the number of variables to be concurrently considered at each move, beyond one or two. However, as the number of neighboring solutions in $\mathcal{N}_k$ typically increases exponentially with k, a complete enumeration and evaluation of all neighbors of the current configuration can usually only be done for small k.

Instead of naively enumerating and evaluating all the solutions in a larger neighborhood in order to identify the best move (or any improving move) to be performed, we can consider more sophisticated exact algorithms for this task. So-called *very large scale neighborhood search methods* [1] have been described for a few selected problems, in which large neighborhoods are defined in special ways allowing to identify the best neighboring solution in reasonable (i.e. polynomial) time without explicitly considering each neighbor. For example, Ergun and Orlin [26] presented such approaches for the traveling salesman problem, Congram [9] explored large neighborhoods efficiently by means of dynamic programming, and for a class of partitioning problems, Thompson et al. [116] defined the concept of cyclic exchange neighborhoods.

Such approaches are also highly promising for many other classes of problems. However, the design of successful large neighborhoods, is not trivial, since it goes hand in hand with the design of an efficient algorithm for searching it.

For several generalized network design problems in this thesis, we will use techniques such as dynamic programming and integer linear programming to efficiently search large neighborhoods.

4.1.2 Merge Solutions by Exact Methods

In evolutionary algorithms, a traditional operator is recombination, which derives a new offspring solution by merging properties of two or more selected candidate solutions. This operator traditionally relies on random decisions and often poor offsprings cannot be avoided.

This motivates approaches where new offspring are derived by exactly solving the subproblem of finding the best solution made up only or mostly of parental properties [10].

Therefore, one can interpret this operation, called *merging*, as the exploration of a large neighborhood defined by two or more given solutions. How the neighborhood should be defined exactly is again a non-trivial question, going again hand in hand with the design of an appropriate method for searching it. Note that this technique can also be seen as a sophisticated extension of the successful *path-relinking* operator which is commonly used in scatter search [41].

4.2 Metaheuristics as Subordinates of Exact Algorithms

Commercial ILP solvers, such as ILOG's CPLEX, are based on a branch-and-bound framework and cutting plane techniques. Within this framework, heuristics are typically used to quickly obtain a promising feasible initial solution (if possible) in order to start with a meaningful global bound. Furthermore, heuristics are usually applied to open subproblems within B&B in order to find better feasible solutions and to improve on this global bound. Beside this obvious application of heuristics, there are other more sophisticated hybridization possibilities.

4.2.1 Guiding Branching and Enumeration Rules by Metaheuristics

B&B strategies have several degrees of freedom that substantially influence performance. In particular, it is often crucial on which discrete entity/entities branching is performed, and in which order the outstanding subproblems (open nodes) are tackled. It is possible to use metaheuristics to control these decisions. For example, Kostikas and Fragakis [70] concluded that such an approach can be effective.

4.2.2 Column Generation by Metaheuristics

If written down completely, some ILP models contain a very large number of variables which precludes the direct application of LP/ILP solvers. Furthermore, in order to provide better bounds, decomposition techniques such as Dantzig-Wolfe decomposition are often applied to strengthen ILPs, yielding, however, models with an exponential number of variables.

As introduced in Chapter 2, this problem can be overcome by starting with a small set of variables and iteratively adding new ones to the model – the pricing problem.

However, determining a variable whose addition would improve the current solution is often a difficult task. Therefore, trying to solve this pricing problem first with fast heuristics is a common approach to speed up column generation. When applied properly, metaheuristics can be very useful in this task [100].

Puchinger and Raidl [96, 99] developed a B&P approach for the two-dimensional bin packing problem which uses a hierarchy of four sub-algorithms for column generation: A fast greedy heuristic, an EA, an algorithm for the restricted pricing problem, and an an exact pricing algorithm.

4.2.3 Cut Generation by Metaheuristics

ILP solvers typically rely on cut generation, i.e. the dynamic addition of extra constraints that are satisfied by a feasible optimal solution, but which are violated by the current solution to the linear programming relaxation. Adding such cuts strengthens the formulation and yields a better LP relaxation and therefore a tighter bound. Finding such cuts is, however, often a difficult optimization problem by its own. Similarly, as metaheuristics can help in solving pricing problems, they are also well suited to help in solving the separation problem.

For the bounded diameter minimum spanning tree problem, Gruber and Raidl [44, 45] presented a B&C algorithm based on a novel type of inequalities. Since the separation problem was very hard to solve, they used construction heuristics with local search and tabu search to locate violated constraints in the LP relaxation.

4.2.4 Applying the Spirit of Metaheuristics in B&B Approaches

Fischetti and Lodi [32] have introduced the concept of *local branching* in B&B-based ILP-solvers. Given an incumbent solution, B&B branches by splitting off a relatively small subproblem corresponding to a k-opt neighborhood around the incumbent solution. This subproblem is forced to be solved (either to optimality or truncated through a time or node limit) by B&B itself before considering the remaining "big" problem. If an improved solution is identified in this way, it is used as a new incumbent solution for continuing a virtual local search within the B&B framework.

Furthermore, *guided diving* and *relaxation induced neighborhood search* [14] are techniques with a similar spirit as local branching: Branching and node-selection rules of a B&B-based ILP-solver are modified in such a way that the neighborhood of promising incumbent solutions and the space around the solution of the LP-relaxation is investigated first.

Chapter 5

Generalized Network Design Problems

This section describes the general strategy that is primarily used in this thesis to solve the Generalized Network Design Problems (GNDPs). In addition, we will also give a brief overview of some other GNDPs which are not further considered here. Finally, we will introduce the instances which are used to test the algorithms for the GNDPs.

5.1 Strategies for Solving Generalized Network Design Problems

The GNDPs considered in this thesis are

- the Generalized Minimum Spanning Tree Problem (GMSTP),
- the Generalized Traveling Salesman Problem (GTSP), and
- the Generalized Edge Biconnected Network Problem (GMEBCNP).

As introduced in Chapter 1, all of these GNDPs are regarded in a more abstract way as graph theory problems. We are given an undirected weighted complete graph $G = \langle V, E, c \rangle$ where node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r,\ \bigcup_{i=1,\ldots,r} V_i = V,\ V_i \cap V_j = \emptyset\ \forall i, j = 1, \ldots, r,\ i \neq j$.

Since the goal is to find a subgraph $S = \langle P, T \rangle$ with $P = \{p_1, p_2, \ldots, p_r\}$ containing one node per cluster, i.e. $p_i \in V_i$ for all $i = 1, \ldots, r$ and T meeting additional constraints of the actual GNDPs, we can approach this problem from two directions.

5.1.1 Emphasizing Spanned Nodes

This strategy is probably the more straightforward one. The idea is to first fix P, i.e. the nodes of all clusters that have to be spanned, and then connect them as good as possible with respect to the constraints of the actual GNDP, see Figure 5.1.

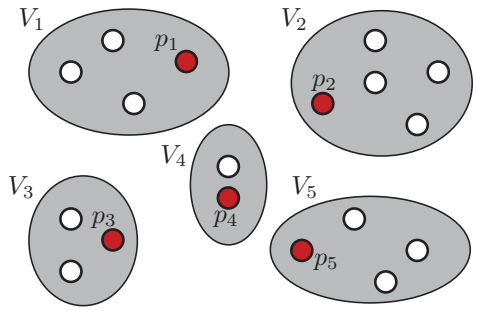


Figure 5.1: Problem of finding the optimal connections for a given set of spanned nodes P.

When designing neighborhood structures for local search based metaheuristics, we can define a move as changing the spanned node of one or more clusters. For the resulting new solution characterized by P', the problem is how to find the optimal connections T'. Obviously, this is equal to solving the classical version of the corresponding GNDP. Depending on the problem, it can be optimally solved in polynomial time (e.g. MSTP), or it can still be $\mathcal{NP}$-hard (e.g. TSP).

This strategy, for example, has already been successfully applied to the MSTP by Ghosh [37]. He implemented different variants of tabu search and variable neighborhood search which use neighborhood structures based on changing one or two spanned nodes and applying Kruskal's MST heuristic to augment the solutions.

5.1.2 Emphasizing Global Connections

The idea behind the previous strategy can also be inverted: When first fixing the general adjacency relations of clusters without choosing particular edges, we aim to

compute an optimal selection of spanned nodes in the second phase. For a formal description, following terminology is used.

Global graph: Given a clustered graph $G = \langle V, E \rangle$, the *global graph* denoted by $G^{\mathrm{g}} = \langle V^{\mathrm{g}}, E^{\mathrm{g}} \rangle$ consists of nodes corresponding to clusters in G, i.e. $V^{\mathrm{g}} = \{V_1, V_2, \ldots, V_r\}$, and edge set $E^{\mathrm{g}} = \{(V_i, V_j) \mid \exists (u, v) \in E \wedge u \in V_i \wedge v \in V_j\}$. Hereby, each *global connection* (V_i, V_j) represents all edges $\{(u, v) \in E \mid u \in V_i \wedge v \in V_j\}$ of graph G.

Global Structure: When given a feasible candidate solution $S = \langle P, T \rangle \subseteq G$, its corresponding *global structure* is defined as the induced global graph's subgraph $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ with global connections $T^{\mathrm{g}} = \{(V_i, V_j) \in E^{\mathrm{g}} \mid \exists (u, v) \in T \wedge u \in V_i \wedge v \in V_j\}$. Figure 5.2 shows an example of a global structure.

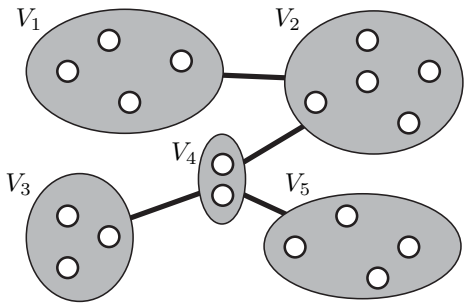


Figure 5.2: Example for a global structure S^{g}

Based on the global connections T^{g} of a given global structure $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$, the idea is to compute an optimal selection of spanned nodes P.

When designing neighborhood structures with respect to this strategy, we can define a move as changing one or more global connections in T^{g}, yielding $T^{\mathrm{g}\prime}$. To obtain the implied solution for $T^{\mathrm{g}\prime}$, the problem is to find an optimal selection of spanned nodes P'. Just as in the previous case, depending on the structure of the global connections, this problem can be optimally solvable in polynomial time or may be $\mathcal{NP}$-hard.

This strategy has already been successfully applied to the MSTP by Pop [91] and to the GTSP by Renaud et al. [103]. They designed neighborhood structures based on modifying the global structure and applying dynamic programming to evaluate the solutions.

5.1.3 Combining both Strategies

The two strategies for approaching GNDPs can be seen as dual to each other and they are based on complementary incomplete representations of the actual solutions. As a result, it appears to be reasonable to combine them.

For solving the GNDPs in this thesis, approaches based on VNS are primarily used. The essential idea behind this metaheuristic is to switch between different neighborhood structures in order to compute solutions that are local optimal with respect to all neighborhoods. This concept is very useful when we design multiple neighborhood structures which are complementary and thus complete each other well. Of course, these neighborhood structures can be used in other metaheuristics besides VNS as well.

5.1.4 Complexity of the Subproblems

For the two strategies, we give the complexity of the corresponding subproblems depending on the GNDP in Table 5.1.

Table 5.1: Complexity of subproblems of GNDPs

	fix spanned nodes and compute connections	fix global connections and compute spanned nodes
Generalized minimum spanning tree problem	polynomial (classical MST)	polynomial (dynamic programming)
Generalized traveling salesman problem	$\mathcal{NP}$-hard (classical TSP)	polynomial (shortest paths)
Generalized minimum edge biconnected network problem	$\mathcal{NP}$-hard	$\mathcal{NP}$-hard

The complexities for fixing the spanned nodes and computing the connections are more obvious since they are derived from the corresponding classical NDP. For determining the complexities when fixing the global connections and computing the spanned nodes, we will either describe polynomial time algorithms (e.g. for the GMSTP and GTSP) or prove $\mathcal{NP}$-hardness (e.g. for the GMEBCNP) in the following chapters.

5.2 Other Generalized Network Design Problems

In addition to the GNDPs that are covered in this thesis and introduced in Section 1.1, there are many others. We will give a short overview on some of them.

The At-Least Variant: For each GNDP there are two problem variants: While we consider the variant where *exactly* one node has to be spanned from each cluster, there is also the variant where *at-least* one node has to be spanned. In some cases, the optimal solutions of both variants are equal. For example, if we consider the GTSP on Euclidean instances, even if we allow more than one node per cluster to be connected, the optimal solution will only contain one node from each cluster [74]. However, for most of the GNDPs, optimal solutions of the less restrictive at-least version can have lower objective values.

Generalized Minimum Vertex Biconnected Network Problem: This is an obvious variation to the GMEBCNP considered in Chapter 8. Based on a clustered graph $G = \langle V, E, c \rangle$, we seek a subgraph $S = \langle P, T \rangle$ of minimal costs with P containing exactly one node from each cluster and T connects all nodes in P via vertex redundancy, i.e. for each pair of nodes $u, v \in P,\ u \neq v$ there must exist at least two node-disjoint paths. Note that vertex redundancy implies edge redundancy, but not conversely. This GNDP has not been considered yet.

Generalized Minimum Clique Problem: Given a clustered graph $G = \langle V, E, c \rangle$, this problem consists of finding the optimal spanned nodes P so that the induced complete graph has minimal costs. Although this problem appears to be less complex since the global structure is always a complete graph, Koster et al. [69] showed that it is still $\mathcal{NP}$-hard to determine the optimal spanned nodes P.

Generalized Minimum Cost Perfect Matching Problem: Given a clustered graph $G = \langle V, E, c \rangle$, the objective is to find a subgraph $S = \langle P, T \rangle$ of minimal costs with P containing exactly one node from each cluster and T being a perfect matching on P, i.e. each node in P is adjacent to only one edge of M. This problem can be solved in polynomial time by precomputing the shortest connections between each pair of clusters.

Generalized Steiner Tree Problem: The generalization of the Steiner tree problem, as it usually appears in literature [102, 105, 122], is rather different compared to other GNDPs. First of all, not all nodes of the given graph $G = \langle V, E, c \rangle$ are assigned to clusters, but only a subset $U \subseteq V$. The remaining nodes are regarded as so-called Steiner nodes, i.e. they do not necessarily have to be connected in the solution. Secondly, the standard generalization is the *at least* version, i.e. the solution $S = \langle P, T \rangle$ must connect at least one node from each cluster. Hence P can contain more than one node for some clusters. This problem was introduced by Reich and Widmayer [102]. They showed that this problem is $\mathcal{NP}$-hard, even if G does not contain any Steiner nodes.

Generalized Node Weighted Steiner Tree Problem: An interesting variation to the previous problem emerges when costs are assigned to the nodes in V as well. These can be negative in case they represent profits for connecting particular nodes. The classical node weighted Steiner tree problem was introduced by Segev [109], but the generalized version has not been considered yet.

Generalized Shortest Paths Problem: Given a graph $G = \langle V, E, c \rangle$, this problem consists of finding a shortest path from u to v, both being nodes of V. It is assumed that $V \setminus \{u, v\}$ is partitioned into clusters and the path must contain *at most* one node from each cluster. There are several different problem formulations. Some can require at least one node from each cluster to be connected, others demand exactly one node from each cluster. For these two variants, it is also necessary to specify whether all clusters have to be included in the path or not. Depending on the particular problem variant, it can be $\mathcal{NP}$-hard or solvable in polynomial time. Li et al. [78] considered a variation of this problem where each node is assigned a non-negative weight value and the shortest path may contain several nodes from a single cluster if the sum of their weights does not exceed a given limit l.

For a more detailed description with an overview on the existing works of various GNDPs, we refer to Feremans et al. [30].

5.3 Test Instances for Generalized Network Design Problems

For testing the algorithms on the GNDPs, we use several instance sets of different types. First of all, we consider Euclidean TSPlib[1] instances with geographical clustering which was used by Feremans [27]. They are based on real world data.

Applying geographical clustering [34] on TSPlib instances is done as follows. A total of 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, clustering is done by assigning each of the remaining nodes to its nearest center node. We consider the largest of such TSPlib instances with up to 442 nodes, 97461 edges, and 89 clusters; details are listed in Table 5.2. The values in the columns denote names of the instances, numbers of nodes, numbers of edges, numbers of clusters, and the average, minimal, and maximal numbers of nodes per cluster.

Table 5.2: TSPlib instances with geographical clustering. Numbers of nodes vary for each 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

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

Besides TSPlib instances, we also consider instance which are created by Ghosh [37] for the GMSTP. First, there are 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 5.3. 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.

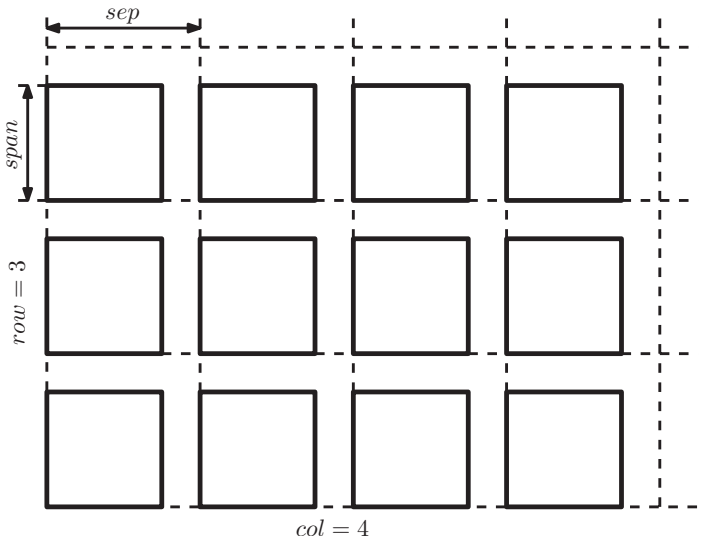


Figure 5.3: Creation of Grouped Euclidean Instances.

The second type of benchmark instances are so-called random Euclidean where nodes of the same cluster are not necessarily close to each other. Such instances are created by simply scattering nodes randomly within a square of size 1000×1000 and making the cluster assignment independently at random.

Finally, Ghosh also generated non-Euclidean random instances by choosing all edge costs randomly from the integer interval $[0, 1000]$. All graphs have a complete set of edges. His benchmark set contains instances with up to 1280 nodes, 818560 edges, and 64 clusters; details are listed in Table 5.3. For each type and size, we consider three different instances.

Expanding this benchmark library, we analogously generated further large instances with 600 nodes and 20 clusters, yielding 30 nodes per cluster, using the same algorithms. 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.

Table 5.3: Benchmark instance sets adopted from [37] and correspondingly created new sets (marked by *). Each instance has a constant number of nodes per cluster.

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

Chapter 6

The Generalized Minimum Spanning Tree Problem

6.1 Introduction

The Generalized Minimum Spanning Tree Problem (GMSTP) is an extension of the classical Minimum Spanning Tree (MST) problem on a graph and is defined as follows. We consider an undirected weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r,\ \bigcup_{i=1,\ldots,r} V_i = V,\ V_i \cap V_j = \emptyset\ \forall i, j = 1, \ldots, r,\ i \neq j$. We write d_i for the number of nodes in $V_i,\ i = 1, \ldots, r$.

A spanning tree of a graph is a cycle-free subgraph connecting all nodes. A solution to the GMSTP 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 $T \subseteq E$ being a tree spanning the nodes in P, see Figure 6.1. 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. We only consider undirected graphs, thus (u, v) is essentially $\{u, v\}$. For better readability, we use (u, v) throughout the chapter.

Parts of this chapter appeared in [55, 57]

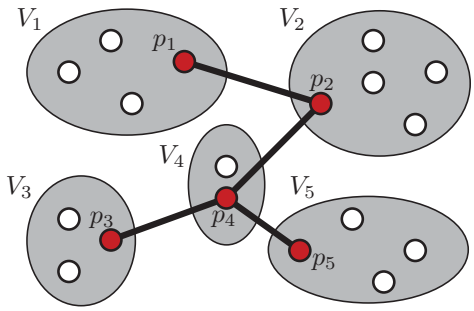


Figure 6.1: Example for a solution to the GMSTP.

In case each cluster contains only one node, i.e. $|V_i| = 1$ 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, the GMSTP is strongly $\mathcal{NP}$-hard [86].

There are several real world applications of the GMSTP, e.g. in the design of backbones in large communication networks. Devices belonging to the same existing local area network correspond to nodes within the same cluster, and the backbone is required to connect one device per local network. For a more detailed overview on the GMSTP, see [86, 27, 91].

A variant of the GMSTP is the less restrictive At-Least GMSTP (LGMSTP) where more than one node is allowed to be connected from each cluster [63, 22]. The GMSTP, as well as the LGMSTP, can further be considered as special cases of the Group Steiner Problem (GSP) introduced by Reich and Widmayer [102]. In this more general problem, clusters are replaced by groups of nodes, which are not required to be disjoint, nor do they have to cover all nodes. The objective is to find a subgraph which spans at least one node of each group.

For solving the GSP, Duin et al. [23] described a transformation to the classical Steiner tree problem on graphs. It is also possible to transform the GMSTP into a (not further constrained) GSP, and therefore, we can solve the GMSTP in principle by means of algorithms for the Steiner tree problem. However, to guarantee that only one node is connected for each cluster, a large constant has to be added to the edge costs, resulting in a GSP instance which is more difficult than general.

We propose a general Variable Neighborhood Search (VNS) approach for solving the GMSTP. As local improvement within VNS, we use Variable Neighborhood Descent (VND) utilizing three different types of exponentially large neighborhoods. Two of them are based on complementary representations of candidate solutions. For the third neighborhood we make use of Mixed Integer Programming (MIP) applying

partial reoptimization, a metaheuristic technique also proposed by Taillard and Voss [115].

This chapter based on [57] and is organized as follows. In Section 6.2, we give an overview on research done on the GMSTP so far. In Section 6.3, we describe the components of our VNS approach in detail. We show experimental results including a comparison to previous approaches in Section 6.4 and conclude in Section 6.5.

6.2 Previous Work

The GMSTP was introduced by Myung et al. [86]. They proved that this problem is $\mathcal{NP}$-hard and provided four different Integer Linear Programming (ILP) formulations. Feremans et al. [29] added another four formulations and performed an in-depth investigation on all eight ILPs. Pop [91] introduced the "Local-Global" MIP formulation. It proved in particular to be more efficient in practice, especially in combination with a relaxation technique called "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 underlying idea of his MIP formulation in a Simulated Annealing approach in order to heuristically solve larger instances. His work also formed a basis for the design of one of the neighborhoods we present in this chapter. A more complex Branch-and-Cut algorithm which features new sophisticated cuts and detailed separation procedures has been recently presented by Feremans et al. [31]. Nevertheless, large instances can still not be solved to optimality in reasonable time.

Regarding approximation algorithms, Myung et al. [86] have shown the inapproximability of the GMSTP 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 et al. [93] described an approximation algorithm for the case when the cluster size is constant. Moreover, Feremans and Grigoriev [28] provided a Polynomial Time Approximation Scheme (PTAS) for the special case of the GMSTP with so-called grid-clustering.

To approach more general and larger GMSTP instances, various metaheuristics have been suggested. Ghosh [37] 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 are based on the exchange of the spanned 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 et al. [43] presented lower and upper bounding procedures, considering the graph $G' = \langle V', E' \rangle$ with nodes v'_i of V' being the clusters V_i of G and with edge set E' being complete. A lower bound arises when one determines an MST on G' with respect to edge costs $c'(v'_i, v'_j)$ defined as $\min\{c(a,b) \mid (a,b) \in E \wedge a \in V_i \wedge b \in V_j\}$. Furthermore, the authors introduced construction heuristics by adapting Kruskal's [71], Prim's [95], and Sollin's algorithm for the classical MST problem, and described a Genetic Algorithm (GA).

A preliminary version of the VNS approach has been described in [55]. The current work has been extended by by exploiting an additional neighborhood type that is based on the idea of solving small parts of an instance via MIP to optimality.

Concerning the LGMSTP, [22] developed two ILPs, four simple construction heuristics and a basic genetic algorithm. [50] 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, [49] 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 LGMSTP cannot be applied to the GMSTP as we consider it here.

6.3 Variable Neighborhood Search for the GMSTP

In this section, we present our VNS approach. First, we consider two constructive heuristics to produce initial solutions. Then, after describing our neighborhoods and the search techniques applied to them, we specify the shaking procedure and a memory function to substantially reduce the number of evaluations for the same solutions.

6.3.1 Initialization

To compute an initial feasible solution for the GMSTP, either a specialized heuristic or an adaption of a standard algorithm for the classical MST problem can be used. Golden et al. [43] give a comparison between three simple and three improved adaptions of Kruskal's, Prim's, and Sollin's MST algorithms for the GMSTP. While all three improved adaptions produce comparable results, the variant based on Sollin's algorithm in general has the highest computational effort. We therefore adopt the improved version based on Kruskal's MST heuristic and compare it to the rather

simple minimum distance heuristic which was also used by Ghosh [37] to generate initial solutions.

Minimum Distance Heuristic

The Minimum Distance Heuristic (MDH) for computing a feasible initial solution for the GMSTP is shown in Algorithm 10. For each cluster, the node with the lowest sum of edge costs to all nodes in other clusters is determined, and a MST is calculated on these nodes. Using Kruskal's algorithm for computing the MST, the complexity of MDH is $O(|V|^2 + r^2 \log r)$ where r is the number of clusters.

Algorithm 10: 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 node to be spanned
determine MST T on the subgraph induced by node set $P := \{p_1, \ldots, p_r\}$
return solution $S := \langle P, T \rangle$

Improved Adaption of Kruskal's MST Heuristic

Creating a feasible solution for the GMSTP by an adaption of Kruskal's algorithm for the classical MST problem is straightforward. 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 in [43], is shown in Algorithm 11 and follows this idea by running the simple version $|V|$ times, once for each node to be initially fixed. Due

Algorithm 11: Improved Kruskal heuristic

forall $v \in |V|$ **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

to the fact that sorting of edges needs to be done only once, the computational complexity is $O(|V|^2 + |E| \log |E|)$.

6.3.2 Neighborhood Structures

Our VNS algorithm applies three types of neighborhoods. The first two are based on local search concepts from [37] and [91]. Ghosh represents solutions by the spanned nodes and defines neighborhood structures on them. Optimal edges are derived for a given selection of nodes by determining a classical MST. On the other hand, Pop approaches the GMSTP from an alternative side by representing a solution via its "global connections" – the pairs of clusters which are directly connected. The complete 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. For our third neighborhood type we consider reasonably small parts of a candidate solution, each part inducing a smaller GMSTP (on a subgraph of the whole instance). We solve these smaller GMSTP's independently to optimality by means of the MIP in [91]. After reconnecting these solved parts we obtain a neighbor of the candidate solution.

Node Exchange Neighborhood

In this neighborhood, which was originally proposed by Ghosh [37], a solution is represented by the set of spanned 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, we can reduce the computational effort 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 some 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 optimization does not change the worst case time complexity, because identifying the shortest edges between any pair of components may require $O(r^2)$ 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, we only get 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 $t = 2$. We therefore apply a Restricted Two Nodes Exchange Neighborhood (RNEN2) in which only pairs of clusters that are adjacent in the current solution S are simultaneously considered. Supposing the clusters are of similar size, the time complexity for a complete evaluation is then 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, we abort its exploration after a certain time limit returning the best neighbor identified so far.

Global Edge Exchange Neighborhood

For a given selection of nodes, optimal edges can be determined by an MST algorithm. Pop [91] has shown that this process can also be reversed: Starting from a global structure $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ which was introduced in Section 5.1.2, we can determine optimal vertices (one for each cluster) by another efficient algorithm. Figure 6.2 shows an example of a global structure for the GMSTP, which is also called a *global spanning tree*.

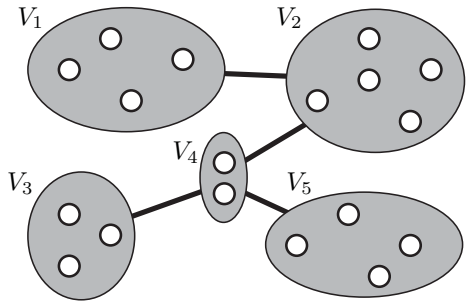


Figure 6.2: A global spanning tree S^{g}.

The global spanning tree represents the set of all feasible generalized spanning trees on G which contain for each global connection $(V_a, V_b) \in T^{\mathrm{g}}$ 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, we can use dynamic programming to efficiently determine a minimum cost solution from this set. We start by rooting the global spanning tree at an arbitrary cluster $V_{\mathrm{root}} \in V^{\mathrm{g}}$ and directing all edges towards the leafs. Then, we traverse this tree in a recursive depth-first way calculating for each cluster $V_k \in V^{\mathrm{g}}$ 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:

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

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 $V_k \in V^{\mathrm{g}}$ the node $p_k \in V_k$ yielding minimum costs. This dynamic programming algorithm requires in the worst case $O(|V|^2)$ time and is illustrated in Figure 6.3.

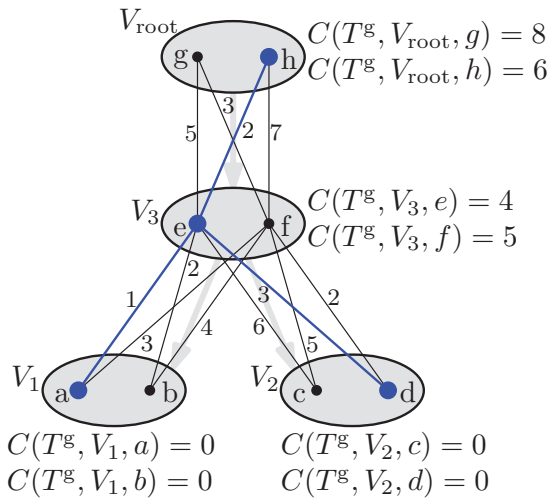


Figure 6.3: Determining the minimum-cost values for each cluster and node. The tree's total minimum costs are $C(T^{\text{g}}, V_{\text{root}}, h) = 6$, and the finally selected nodes are printed bold.

As Global Edge Exchange Neighborhood (GEEN) for a given global tree T^{g}, we consider any feasible global tree differing from T^{g} by precisely one global connection. There are $O(r)$ edges which can be removed and $O(r^2)$ feasible ways of reconnecting the resulting two components. If we determine the best neighbor by evaluating all possibilities and naively perform the whole dynamic programming for each global candidate tree, the total time complexity is $O(|V|^2 \cdot r^3)$.

Incremental Dynamic Programming: For a more efficient evaluation of all neighbors, we perform the whole dynamic programming only once at the beginning, store all costs $C(T^{\text{g}}, V_k, v)$, $\forall k = 1, \ldots, r,\ v \in V_k$, 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 one (V_c, V_d) so that the resulting graph remains a valid tree, see Figure 6.4. By removing (V_a, V_b), the subtree rooted at V_b is disconnected, hence V_a loses a child and V_a, as well as all its predecessors, must be updated. Before we add (V_c, V_d), we first need to consider the isolated subtree. If $V_d \neq V_b$, we have to re-root the subtree at cluster V_d. 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.

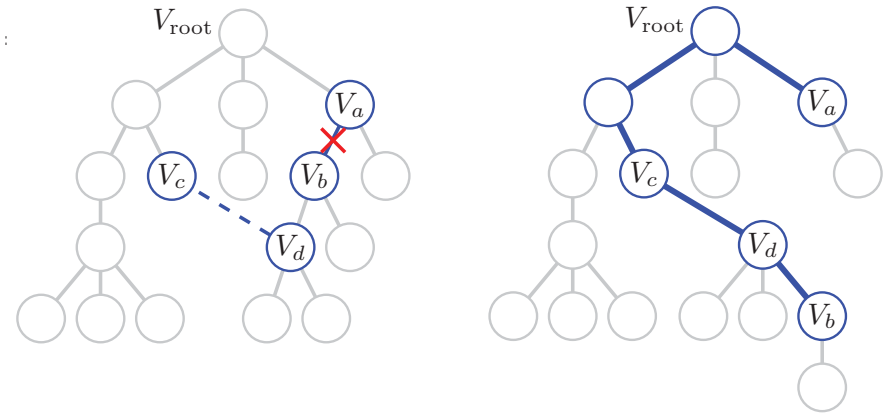


Figure 6.4: After removing (V_a, V_b) and inserting (V_c, V_d), only the clusters on the paths from V_a to V_{root} and V_b to V_{root} must be reconsidered.

If the tree is not degenerated, its height is $O(\log r)$, and we 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, we may skip the update of the cluster's predecessors 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 further supports incremental evaluation. Algorithm 12 shows how this is done in detail.

Removing an edge (V_i, V_j) splits our rooted tree into two components: K_1^{g} containing V_i and K_2^{g} containing V_j. The algorithm iterates through all clusters $V_k \in K_1^{\text{g}}$ 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

Algorithm 12: Global Edge Exchange Neighborhood (solution $S = \langle P, T \rangle$)

```
forall global connections (V_i, V_j) ∈ T^g do
    remove (V_i, V_j)
    M_1 := list of clusters in component K_1^g containing V_i (traversed in
    preorder)
    M_2 := list of clusters in component K_2^g containing V_j (traversed in
    preorder)
    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
```

the clusters in K_1^{g} except its root V_k has to be updated more than once, because global edges are only added between the roots of K_1^{g} and K_2^{g}. Processing 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}.

Global Subtree Optimization Neighborhood

This neighborhood follows the idea of selecting subproblems of reasonable size, solving them to provable optimality via MIP and merging the results to an overall solution as well as possible. We consider the current solution $S = \langle P, T \rangle$ with its corresponding global spanning tree $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ defined on the global graph G^{g}, i.e. for each edge $(u, v) \in T$ with $u \in V_i \wedge v \in V_j$, there exists a global connection $(V_i, V_j) \in T^{\mathrm{g}}$. After rooting S^{g} at a randomly chosen cluster V_{root}, we perform 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. Figure 6.5 shows an example for this selection mechanism with $N_{\min} = 3$ and $N_{\max} = 4$ yielding subtrees $Q_1, \ldots, Q_4$ rooted at $V_1, \ldots, V_4$.

Moving to a solution in the Global Subtree Optimization Neighborhood (GSON)

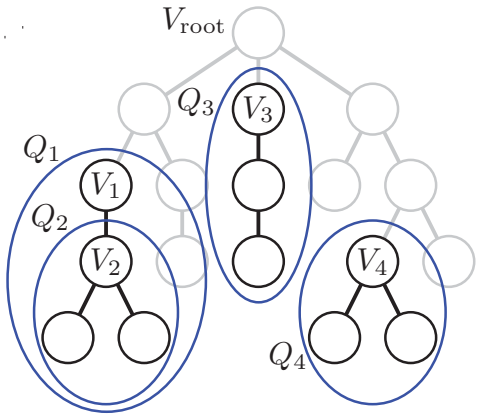


Figure 6.5: Selection of subtrees to be optimized via MIP.

means to optimize one subtree Q_i as an independent GMSTP on the restricted graph induced by the clusters and nodes of Q_i. After solving this subproblem via MIP, we reconnect the new subtree to the remainder of the current overall tree in the best possible way. This can be achieved by inspecting all global connections between both components, which is similar as in GEEN. Algorithm 13 summarizes the evaluation of this neighborhood in pseudo-code.

Algorithm 13: Global Subtree Opt. Neighborhood (solution $S = \langle P, T \rangle$)

```
V_1, ..., V_k := roots of the subtrees Q_1, ..., Q_k containing at least N_min and
    no more than N_max clusters
for i := 1, ..., k do
    remove the edge (parent of V_i, V_i) // separate subtree Q_i from S
    optimize Q_i via MIP
    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 6.5 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 we 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 MIP. If we do not allow overlapping subtrees, the number of subtrees to be considered is bounded below by 0 and above by $\lfloor \frac{r}{N_{\min}} \rfloor$. In our case, we allow contained subtrees, and the number of subtrees to be optimized can be as large as $\lfloor \frac{r}{N_{\max}} \cdot (N_{\max} - N_{\min} + 1) \rfloor$. In our experiments, choosing $N_{\min} = 5$ and $N_{\max} = 6$ yielded the best results.

Local-Global MIP Formulation

In order to solve the subproblems on restricted sets of clusters to optimality, GSON utilizes Pop's local-global MIP formulation [91], which turned out to be more efficient than other formulations when using a general purpose MIP 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. We use the following binary variables.

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

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

$$x_e = \begin{cases} 1 & \text{if edge } e \in E \text{ appears in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall e \in E$$

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

Pop proved 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_{ij} to be integral in the following local-global MIP formulation.

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{6.1}$$

$$\text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, r \tag{6.2}$$

$$\sum_{e \in E} x_e = r - 1 \tag{6.3}$$

$$\sum_{e=(u,v) | u \in V_i, v \in V_j} x_e = y_{ij} \qquad \forall i, j = 1, \ldots, r,\ i \neq j \tag{6.4}$$

$$\sum_{e=(u,v) | u \in V_i} x_e \leq z_v \qquad \forall i = 1, \ldots, r,\ \forall v \in V \backslash V_i \tag{6.5}$$

$$y_{ij} = \lambda_{kij} + \lambda_{kji} \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k \tag{6.6}$$

$$\sum_{j \in \{1,\ldots,r\} \backslash \{i\}} \lambda_{kij} = 1 \qquad \forall i, k = 1, \ldots, r,\ i \neq k \tag{6.7}$$

$$\lambda_{kkj} = 0 \qquad \forall j, k = 1, \ldots, r,\ j \neq k \tag{6.8}$$

$$\lambda_{kij} \geq 0 \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k \tag{6.9}$$

$$x_e,\ z_v \geq 0 \qquad \forall e \in E, \forall v \in V \tag{6.10}$$

$$y_{lr} \in \{0, 1\} \tag{6.11}$$

Constraints (6.2) guarantee that only one node is selected per cluster. Equality (6.3) forces the solution to contain exactly $r-1$ edges, while constraints (6.4) allow them only between nodes of clusters which are connected in the global graph. Inequalities (6.5) ensure that edges only connect nodes v for which $z_v = 1$. For each $k = 1, \ldots, r$, constraints (6.6) and (6.8) force variables λ_{kij} to represent a spanning tree directed out of V_k: Equalities (6.6) ensure the selection of a global connection (i, j) iff i is parent of j or j is parent of i in a spanning tree directed out of V_k. Constraints (6.7) guarantee that each cluster except root k has exactly one parent, while Equalities (6.8) make sure that root k has no parents.

Alternative Neighborhoods

When designing GSON we considered several alternative large neighborhoods combining the concepts of the global graph with an exact MIP. One variation of GSON is to first solve all subtrees of limited size exactly and then iterate through a neighborhood structure in which we consider all possibilities of reconnecting these parts. As there are exponentially many such possibilities, the exhaustive exploration turned out to be too expensive in practice.

Another idea for enhancing GSON was 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 found via breadth first search. Thus one considers components of the current global tree where the clusters are close to each other. Unfortunately, experiments we performed indicated that the gain of this variant of GSON could not cover its high computational costs.

6.3.3 Variable Neighborhood Search Framework

We use the basic VNS scheme with VND as local improvement [48], see Section 3.4. In VND, we alternate between NEN, GEEN, RNEN2, and GSON in this order, see Algorithm 14. This sequence has been determined according to the computational complexity of evaluating the neighborhoods.

Algorithm 14: VND (solution $S = \langle P, T \rangle$)

```
l := 1
repeat
    switch l do
        case 1: // NEN
            S' := Node Exchange Neighborhood (S)
        case 2: // GEEN
            S' := Global Edge Exchange Neighborhood (S) //see Algorithm 12
        case 3: // RNEN2
            S' := Restricted Two Nodes Exchange Neighborhood (S)
        case 4: // GSON
            S' := Global Subtree Optimization Neighborhood (S) //see
            Algorithm 13
    if solution improved then
        S := S'
        l := 1
    else
        l := l + 1
until l > 4
return S
```

Shaking

It turned out that using a shaking function which puts more emphasis on diversity yields good results for our approach, see Algorithm 15. This shaking process uses both, the NEN and the GEEN structures. For NEN, the number of random moves for shaking starts at three because we have a restricted 2-Opt NEN improvement already included in 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 a similar reason. The number k of random moves increases in steps of two up to $\lfloor \frac{r}{2} \rfloor$.

Algorithm 15: Shake (solution $S = \langle P, T \rangle$, size k)

```
for i := 1, ..., k + 1 do
    randomly change the spanned node p_i of a random cluster V_i
determine the MST T and derive T^g
for i := 1, ..., k do
    remove a randomly chosen global connection e ∈ T^g yielding components
    K_1^g and K_2^g
    insert a randomly chosen global connection e' connecting K_1^g and K_2^g with
    e' ≠ e
    determine the spanned nodes p_1, ..., p_r by dynamic programming
return S
```

Memory Function

There is a common situation where VNS unnecessarily spends much time on iterating through all neighborhoods. 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 iterates through all neighborhoods again, trying to improve the solution with no success.

We use a hash memory to avoid such situations. For each deterministic neighborhood structure N_i, we store a hash value h_{N_i} of the best solution obtained by it. Before VND tries to improve a solution within N_i, it compares the hash value of the current solution with the memorized hash value h_{N_i}. If they are equal, the evaluation of the neighborhood is skipped, as the current solution cannot be improved by searching through N_i. Since only one hash value per neighborhood structure is memorized at a time, it is not comparable with full-fledged Tabu Search. Nevertheless, this simple concept turned out to save much time in practice.

Experimental VND Variation

We tried out a supposedly promising variation of VND where we remain at one neighborhood until it cannot improve the solution anymore and only then move on to the next one. This approach has the advantage that by consecutively running the GEEN, we do not need to recalculate the connection values of all nodes each time in the dynamic programming process. Unfortunately this was not such a good idea because the gain was not too big. As GEEN still consumes much more time than NEN, it is better to apply NEN whenever possible.

6.4 Computational Results

We tested our algorithms on Euclidean TSPlib instances with geographical center clustering, grouped Euclidean, random Euclidean, and non-Euclidean instance sets as introduced in Section 5.3. All experiments were performed on a Pentium 4, 2.8GHz PC with 2GB RAM, and we used CPLEX 9.03 to solve the MIP subproblems within GSON.

In the following, we first present a summary for an experimental comparison of the two constructive heuristics described in Section 6.3.1, which we consider for the creation of initial solutions for VNS. Computational results of our VNS approach on the different test data sets follow in Section 6.4.2. Finally, Section 6.4.3 analyses the individual contributions of the different neighborhoods within VND.

6.4.1 Comparison of Construction Heuristics

Table 6.1 summarizes 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. Ratios $\overline{\text{IKH/MDH}}$ indicate the average factor between the objective values of solutions generated by IKH and MDH. Interestingly, the two heuristics never obtained the same solution or solutions of the same quality. As the required CPU-times of both heuristics are very small (less than 80ms for our largest instances with 1280 nodes), we decided to run both, MDH and IKH, and to choose the better result as initial solution for VNS.

Table 6.1: Comparison of the construction heuristics MDH and IKH.

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

6.4.2 Computational Results for VNS

We compare the results of our VNS to Tabu Search with recency and frequency based memory (TS2) [37], Variable Neighborhood Decomposition Search (VNDS) [37], the Simulated Annealing (SA) approach from Pop [91], and, in case of TSPlib instances, also to the Genetic Algorithm (GA) from Golden et al. [43]. While TS2 is deterministic, we provide average results over 30 runs for VNDS and VNS and over at least 10 runs for SA (due to its long running times). 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 [91], which led to significantly longer running times compared to the other algorithms. The results for the GA are adopted from Golden et al. [43].

In Table 6.2 and 6.3 we show instance names, numbers of nodes, numbers of clusters, (average) numbers of nodes per cluster, and (average) objective values of the final solutions obtained by the different algorithms. Best values are printed bold. In case of SA and VNS, we also provide corresponding standard deviations of objective values. VNDS produces very stable results as the standard deviations are always zero, except for the second instance of set "Random E. 400" where it is 0.34. For GA, we do not have any standard deviations as they are not listed in [43].

In Table 6.2 we compare our VNS to TS2, VNDS, SA, and also the GA on the TSPlib based instances. Results for the GA are adopted from Golden et al. [43], 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 mean objective values from VNS are smaller. Judging by the few results for GA, VNS finds solutions which are at least as good as those of GA. Considering VNS and TS2, we cannot draw clear conclusions. Most of the time, these two algorithms generate comparable results under the same

Table 6.2: Results on TSPlib instances with geographical clustering, $\frac{|V|}{r} = 5$, variable CPU-time.

TSPlib Instances			TS2	VNDS	SA		GA	VNS	
Name	$\|V\|$	time	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	$\overline{C(T)}$	std dev
gr137	137	150s	**329.0**	330.0	352.0	0.00	**329.0**	**329.0**	0.00
kroa150	150	150s	**9815.0**	**9815.0**	10885.6	25.63	**9815.0**	**9815.0**	0.00
d198	198	300s	7062.0	7169.0	7468.73	0.83	**7044.0**	**7044.0**	0.00
krob200	200	300s	11245.0	11353.0	12532.0	0.00	**11244.0**	**11244.0**	0.00
gr202	202	300s	**242.0**	249.0	258.0	0.00	243.0	**242.0**	0.00
ts225	225	300s	62366.0	63139.0	67195.1	34.49	62315.0	**62268.5**	0.51
pr226	226	300s	**55515.0**	**55515.0**	56286.6	40.89	**55515.0**	**55515.0**	0.00
gil262	262	300s	**942.0**	979.0	1022.0	0.00	-	942.3	1.02
pr264	264	300s	**21886.0**	22115.0	23445.8	68.27	-	21886.5	1.78
pr299	299	450s	20339.0	20578.0	22989.4	11.58	-	**20322.6**	14.67
lin318	318	450s	18521.0	18533.0	20268.0	0.00	-	**18506.8**	11.58
rd400	400	600s	**5943.0**	6056.0	6440.8	3.40	-	5943.6	9.69
fl417	417	600s	7990.0	7984.0	8076.0	0.00	-	**7982.0**	0.00
gr431	431	600s	1034.0	1036.0	1080.5	0.51	-	**1033.0**	0.18
pr439	439	600s	51852.0	52104.0	55694.1	45.88	-	**51847.9**	40.92
pcb442	442	600s	**19621.0**	19961.0	21515.1	5.15	-	19702.8	52.11

conditions. We omitted smaller TSPlib instances in Table 6.2 as the most capable algorithms TS2, GA, and VNS were all able to (almost) always provide optimal solutions as found by the exact Branch-and-Cut algorithm from [31]. The latter could solve all instances with up to 200 nodes except d198 to provable optimality in up to 5254s CPU time.

In Table 6.3 we compare our VNS to TS2, VNDS, and SA on grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances. 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 [37] used this time limit as termination criterion, so we decided to retain it. SA required 150s for small instances with 125 nodes and up to about 40000s for the largest instances with 1280 nodes.

When comparing our VNS with SA, we can observe that VNS consistently finds better solutions. Wilcoxon rank sum tests yield error probabilities of less than 1% for the assumption that the mean objective values from VNS are smaller. Also in

Table 6.3: Results on instance sets from [37] and correspondingly created new sets, 600s CPU-time (except SA). Three different instances are considered for each set.

Instances				TS2	VNDS	SA		VNS	
Set	$\|V\|$	r	$\frac{\|V\|}{r}$	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped E. 125	125	25	5	**141.1**	**141.1**	152.3	0.52	**141.1**	0.00
	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
Grouped E. 500	500	100	5	**566.7**	577.6	642.3	0.00	567.4	0.57
	500	100	5	**578.7**	584.3	663.3	1.39	585.0	1.32
	500	100	5	**581.6**	588.3	666.7	1.81	583.7	1.82
Grouped E. 600	600	20	30	85.2	87.5	93.9	0.00	**84.6**	0.11
	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
Grouped E. 1280	1280	64	20	327.2	329.2	365.1	0.46	**315.9**	1.91
	1280	64	20	322.2	322.5	364.4	0.00	**318.3**	1.78
	1280	64	20	332.1	335.5	372.0	0.00	**329.4**	1.29
Random E. 250	250	50	5	**2285.1**	2504.9	2584.3	23.82	2300.9	40.27
	250	50	5	**2183.4**	2343.3	2486.7	0.00	2201.8	23.30
	250	50	5	**2048.4**	2263.7	2305.0	16.64	2057.6	31.58
Random E. 400	400	20	20	**557.4**	725.9	665.1	3.94	615.3	10.8
	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	**587.3**	0.00
Random E. 600	600	20	30	541.6	656.1	491.8	7.83	**443.5**	0.00
	600	20	30	540.3	634.0	542.8	25.75	**537.0**	10.2
	600	20	30	627.4	636.5	469.5	2.75	**469.0**	11.9
Non-E. 200	200	20	10	**71.6**	94.7	76.9	0.21	**71.6**	0.00
	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
Non-E. 500	500	100	5	**143.7**	203.2	200.3	4.44	152.5	3.69
	500	100	5	**132.7**	187.3	194.3	1.20	148.6	4.27
	500	100	5	**162.3**	197.4	205.6	0.00	166.1	2.89
Non-E. 600	600	20	30	**14.5**	59.4	22.7	1.49	15.6	1.62
	600	20	30	17.7	23.7	22.0	0.82	**16.1**	1.24
	600	20	30	**15.1**	29.5	22.1	0.44	16.0	1.66

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 E. 500") on which VNDS performed better. In all other cases, VNS' solutions are superior with high statistical significance (error levels less than 1%). Results of VNS 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 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. We assume 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 E. 600", solutions obtained by VNS are on average 34.4% better than those of TS2. We also observe that SA, which is usually worst, is able to outperform TS2 and VNDS on some of these instances. We conclude that the neighborhood type GEEN, which is also the main component of SA, is very effective on random Euclidean instances and on instances with higher number of nodes per cluster. On non-Euclidean instances, TS2 mostly outperforms all other algorithms.

In overall, VNS and TS2 are the most powerful algorithms among all considered approaches. Out of 46 instances we have tested, VNS produces strictly better results in 20 cases, TS2 is better in 16 cases, and on 10 instances, they are equally good.

6.4.3 Contributions of Neighborhoods

In order to analyze how the different neighborhood structures of VNS contribute to the whole optimization, we logged how often each one was able to improve on a current solution and their absolute gains. Table 6.4 shows the ratios of successful improvements in contrast to how often each neighborhood structure was evaluated. These values are grouped by the different types of input instances. On the other hand, Table 6.5 shows their absolute gains, i.e. their contribution in percentage to the difference between objective values of the starting and the final solutions.

In general, each neighborhood structure 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 significant larger in case of GEEN. Considering that GSON operates on solutions which are already local optima with respect to all other neighborhoods,

Table 6.4: Individual improvement rates of NEN, GEEN, RNEN2, and GSON.

Instance Type	$\|V\|$	r	$\frac{\|V\|}{r}$	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

Table 6.5: Individual absolute gains of NEN, GEEN, RNEN2, and GSON.

Instance Type	$\|V\|$	r	$\frac{\|V\|}{r}$	NEN	GEEN	RNEN2	GSON
TSBlib based	n.a.	n.a.	5	16.58%	60.71%	15.64%	7.07%
Grouped Euclidean	125	25	5	18.02%	63.40%	13.61%	4.96%
	500	100	5	23.17%	45.19%	28.32%	3.31%
	600	20	30	13.60%	75.85%	7.29%	3.26%
	1280	64	20	16.72%	40.08%	20.45%	22.76%
Random Euclidean	250	50	5	16.90%	48.52%	16.06%	18.52%
	400	20	20	16.14%	66.75%	15.13%	1.98%
	600	20	30	21.30%	63.43%	13.65%	1.62%
Non-Euclidean	200	20	10	10.89%	48.07%	11.92%	29.13%
	500	100	5	11.74%	60.04%	24.75%	3.47%
	600	20	30	11.78%	74.80%	10.65%	2.78%

both its improvement ratios and its absolute gains are remarkable. Regarding the different instance sets, we also observe that the improvement ratio of GEEN generally increases with the size of nodes per cluster.

In addition, Table 6.6 and 6.7 show tests on switching particular neighborhood structures off. We compare results obtained by using all neighborhood structures, turning NEN and RNEN2 off, turning GEEN off, and turning GSON off. Obviously, omit-

ting NEN and RNEN2 performs worst. By switching GEEN off, we get comparable results on Group Euclidean instances and Non-Euclidean instances, but significantly worse results on Random Euclidean instances. Results on runs without GSON are taken from our previous work in [55], they are generally inferior compared to the current results.

Table 6.6: Results on TSPlib instances when switching off certain neighborhoods.

Instances	VNS		w.o. NEN		w.o. GEEN		w.o. GSON	
Name	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	7044.6	2.28	7044.3	1.64	**7044.0**	0.00
krob200	**11244.0**	0.00	11264.0	22.6	**11244.0**	0.00	**11244.0**	0.00
gr202	**242.0**	0.00	242.2	0.48	242.1	0.25	**242.0**	0.00
ts225	**62268.5**	0.51	62270.7	6.35	62269.9	4.58	62280.5	16.28
pr226	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00
gil262	**942.3**	1.02	947.0	3.63	942.9	1.36	943.2	1.63
pr264	**21886.5**	1.78	21913.0	17.1	21890.5	5.84	21890.8	5.92
pr299	**20322.6**	14.67	20422.2	44.85	20330.7	21.67	20347.4	28.09
lin318	**18506.8**	11.58	18596.1	36.9	18521.5	15.96	18511.2	9.70
rd400	**5943.6**	9.69	6067.8	48.1	5976.3	16.74	5955.0	7.57
fl417	**7982.0**	0.00	7982.3	0.47	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.18	1037.2	1.64	1033.1	0.25	**1033.0**	0.25
pr439	**51847.9**	40.92	52184.0	127.55	51893.5	65.6	51849.7	39.30
pcb442	**19702.8**	52.11	20079.2	42.39	19796.7	35.51	19729.3	50.90

Table 6.7: Results on Ghosh' instance sets when switching off certain neighborhoods.

Instances	VNS		w.o. NEN		w.o. GEEN		w.o. GSON	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped E. 125	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00	**133.8**	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00	**141.4**	0.00	**141.4**	0.00
Grouped E. 500	**567.4**	0.57	589.6	5.56	567.7	0.48	568.6	0.59
	585.0	1.32	601.8	5.33	586.5	1.18	**581.0**	1.39
	583.7	1.82	597.1	2.37	**583.3**	1.91	587.9	4.07
Grouped E. 600	**84.6**	0.11	84.8	0.27	**84.6**	0.11	84.8	0.27
	87.9	0.00	88.3	0.24	**87.9**	0.02	**87.9**	0.05
	88.5	0.00	88.7	0.12	**88.5**	0.00	**88.5**	0.00
Grouped E. 1280	315.9	1.91	327.1	4.30	**314.6**	1.10	321.8	2.41
	318.3	1.78	326.4	3.01	317.8	0.79	**316.3**	0.83
	329.4	1.29	339.9	3.87	329.6	2.16	334.3	2.13
Random E. 250	**2300.9**	40.27	2646.6	28.49	2320.0	43.08	2336.9	34.23
	2201.8	23.30	2576.2	112.15	2199.7	21.94	2304.1	47.95
	2057.6	31.58	2460.6	131.82	2061.2	26.91	**2049.8**	15.29
Random E. 400	**615.3**	10.8	703.4	53.40	621.9	14.20	625.4	14.59
	595.3	0.00	671.4	25.82	**595.3**	0.00	**595.3**	0.14
	587.3	0.00	657.4	50.38	597.5	17.15	588.8	7.40
Random E. 600	**443.5**	0.00	506.5	46.08	452.9	33.57	443.5	0.00
	537.0	10.2	685.0	21.40	545.1	18.05	**535.2**	12.20
	469.0	11.9	643.4	63.68	493.8	35.76	479.9	26.55
Non-E. 200	**71.6**	0.00	97.3	14.23	**71.6**	0.00	**71.6**	0.02
	41.0	0.00	58.5	11.18	**41.0**	0.00	**41.0**	0.00
	52.8	0.00	56.8	0.69	**52.8**	0.00	**52.8**	0.00
Non-E. 500	**152.5**	3.69	203.3	0.00	155.4	3.94	173.4	8.40
	148.6	4.27	237.9	1.28	151.2	5.41	154.6	6.55
	166.1	2.89	282.5	0.00	167.5	4.36	180.1	3.67
Non-E. 600	15.6	1.62	47.5	9.40	**15.3**	1.35	15.9	2.07
	16.1	1.24	36.0	3.67	16.3	1.24	17.6	1.75
	16.0	1.66	42.0	5.87	16.2	1.97	**15.1**	0.22

6.4.4 Adjusting the Size of GSON

The primary adjustment parameter for GSON is the size of the subtrees to be optimized via MIP. These subtrees contain at least $N_{\min}$ and at most $N_{\max}$ clusters. In Table 6.8 and 6.9 we study the influence of different values for these parameters. In general, results are ambiguous. On Random Euclidean instances, a tendency towards smaller sizes yielding better results is noticeable. On some other instances, the search process benefits from larger values as the neighborhood is searched more extensively. We decided to set $N_{\min} = 5$ and $N_{\max} = 6$ as default behavior for a balanced behavior.

Table 6.8: Results on TSPlib instances when tweaking the ILP size of GSON.

Instances	ILP size 3 – 4		ILP size 5 – 6		ILP size 7 – 8		ILP size 3 – 8	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	**7044.0**	0.00	**7044.0**	0.00	**7044.0**	0.00
krob200	**11244.0**	0.00	**11244.0**	0.00	**11244.0**	0.00	**11244.0**	0.00
gr202	**242.0**	0.00	**242.0**	0.00	**242.0**	0.18	**242.0**	0.00
ts225	**62268.2**	0.38	62268.5	0.51	62269.4	4.68	62268.3	0.47
pr226	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00
gil262	942.1	0.57	942.3	1.02	**942.0**	0.00	942.1	0.57
pr264	**21886.2**	1.28	21886.5	1.78	21887.1	3.09	21887.6	4.23
pr299	20320.9	12.98	20322.6	14.67	**20316.1**	0.55	20316.8	2.07
lin318	**18504.4**	7.25	18506.8	11.58	18508.0	8.99	18506.0	8.01
rd400	5947.8	10.25	**5943.6**	9.69	5943.9	9.99	5945.8	10.57
fl417	**7982.0**	0.00	**7982.0**	0.00	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.00	**1033.0**	0.18	**1033.0**	0.00	**1033.0**	0.18
pr439	**51837.0**	31.71	51847.9	40.92	51847.4	43.77	51841.4	41.27
pcb442	**19693.1**	49.05	19702.8	52.11	19712.4	53.58	19699.6	53.78

Table 6.9: Results on Ghosh' instance sets when tweaking the ILP size of GSON.

Instances	ILP size 3 – 4		ILP size 5 – 6		ILP size 7 – 8		ILP size 3 – 8	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped E. 125	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00	**133.8**	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00	**141.4**	0.00	**141.4**	0.00
Grouped E. 500	**567.4**	0.65	**567.4**	0.57	567.5	0.49	567.5	0.76
	585.3	1.01	585.0	1.32	**584.4**	1.50	584.8	1.32
	584.2	1.76	**583.7**	1.82	**583.7**	1.43	584.0	1.73
Grouped E. 600	**84.6**	0.00	**84.6**	0.11	84.7	0.24	**84.6**	0.11
	87.9	0.00	**87.9**	0.00	88.4	0.39	**87.9**	0.00
	88.5	0.00	**88.5**	0.00	**88.5**	0.00	**88.5**	0.00
Grouped E. 1280	**315.9**	1.75	**315.9**	1.91	316.9	2.59	317.7	2.36
	318.4	1.58	**318.3**	1.78	319.5	1.51	319.6	1.97
	329.9	1.68	**329.4**	1.29	330.8	1.37	329.5	1.05
Random E. 250	2308.6	47.04	**2300.9**	40.27	2308.0	42.85	2320.8	50.63 3
	2208.6	31.66	**2201.8**	23.30	2198.6	20.56	2212.3	33.31 1
	2055.5	34.74	2057.6	31.58	2057.2	33.67	**2050.7**	15.97
Random E. 400	**611.5**	5.20	615.3	10.8	658.8	36.55	687.6	49.36
	595.3	0.00	**595.3**	0.00	618.3	24.62	621.6	25.02
	587.3	0.00	**587.3**	0.00	598.7	23.75	623.1	36.00
Random E. 600	**443.5**	0.00	**443.5**	0.00	657.7	0.00	657.7	0.00
	530.5	7.53	537.0	10.2	579.7	42.90	562.2	20.79
	466.8	0.00	469.0	11.9	560.7	63.25	551.9	65.76
Non-E. 200	**71.6**	0.00	**71.6**	0.00	**71.6**	0.00	**71.6**	0.00
	41.0	0.00	**41.0**	0.00	**41.0**	0.00	**41.0**	0.00
	52.8	0.00	**52.8**	0.00	**52.8**	0.00	**52.8**	0.00
Non-E. 500	153.2	2.82	152.5	3.69	**152.1**	4.19	153.9	3.64
	149.1	6.54	148.6	4.27	**148.4**	6.28	148.5	3.84
	167.6	3.29	**166.1**	2.89	167.5	2.81	166.2	2.82
Non-E. 600	16.0	1.97	**15.6**	1.62	16.1	1.65	16.2	1.97
	16.7	1.42	**16.1**	1.24	16.3	1.42	17.0	1.66
	15.2	0.51	16.0	1.66	16.1	1.78	15.8	1.46

6.4.5 Using Different Starting Solutions

Table 6.10 and 6.11 show the importance of having good starting solutions for our VNS. We compare the quality of the final solutions when using MDH/IKH as initialization heuristics in contrast to starting with random solutions. The latter are constructed by choosing a random node for each cluster and connecting them via Kruskal's MST algorithm. Using MDH and IKH improves the quality of final solutions in most cases. On smaller instances, starting with a random solution leads to the same results as VNS is powerful enough and has enough time available. When facing more difficult instances, time becomes more crucial and therefore starting with a superior solution proves to be advantageous.

Table 6.10: Results on TSPlib instances when using different starting solutions.

Instances	MDH and IKH		random init	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	**7044.0**	0.00
krob200	**11244.0**	0.00	11246.0	6.32
gr202	**242.0**	0.00	**242.0**	0.00
ts225	**62268.5**	0.51	62268.8	0.42
pr226	**55515.0**	0.00	**55515.0**	0.00
gil262	**942.3**	1.02	943.3	1.49
pr264	**21886.5**	1.78	21890.0	5.54
pr299	**20322.6**	14.67	20341.6	27.26
lin318	**18506.8**	11.58	18517.2	14.57
rd400	**5943.6**	9.69	5969.0	26.52
fl417	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.18	1034.4	1.84
pr439	**51847.9**	40.92	51855.3	63.74
pcb442	**19702.8**	52.11	19735.0	56.57

Table 6.11: Results on Ghosh' instance sets when using different starting solutions.

Instances	MDH and IKH		random init	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped E. 125	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00
Grouped E. 500	**567.4**	0.57	577.2	3.85
	585.0	1.32	585.4	2.63
	583.7	1.82	585.0	3.51
Grouped E. 600	**84.6**	0.11	84.7	0.25
	87.9	0.00	88.0	0.16
	88.5	0.00	**88.5**	0.06
Grouped E. 1280	**315.9**	1.91	320.7	3.7
	318.3	1.78	320.9	4.48
	329.4	1.29	333.2	2.58
Random E. 250	**2300.9**	40.27	2363.3	40.33
	2201.8	23.30	2306.1	48.63
	2057.6	31.58	2153.1	92.65
Random E. 400	**615.3**	10.8	626.9	17.72
	595.3	0.00	**595.3**	0.00
	587.3	0.00	**587.3**	0.00
Random E. 600	**443.5**	0.00	**443.5**	0.00
	537.0	10.2	541.4	14.06
	469.0	11.9	470.7	12.36
Non-E. 200	**71.6**	0.00	**71.6**	0.00
	41.0	0.00	**41.0**	0.00
	52.8	0.00	**52.8**	0.00
Non-E. 500	**152.5**	3.69	177.3	16.54
	148.6	4.27	166.0	9.06
	166.1	2.89	187.5	10.11
Non-E. 600	**15.6**	1.62	17.8	2.93
	16.1	1.24	18.4	1.81
	16.0	1.66	16.7	2.34

6.5 Conclusions

In this chapter, we proposed a general Variable Neighborhood Search (VNS) approach for solving the Generalized Minimum Spanning Tree problem. For initializing the solution, we use 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 construction heuristic mostly yields better results.

Our Variable Neighborhood Descent combines three neighborhood types: For the Node Exchange Neighborhood, solutions are represented by the spanned 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 a complementary way by considering for a solution primarily its global connections, i.e. 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. For the Global Subtree Optimization Neighborhood, we consider subsets of clusters which are reorganized via Mixed Integer Programming and then reconnected to the remainder as well as possible.

Tests were performed on TSPlib instances, grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances. Results show that the proposed VNS algorithm produces with high probability solutions of equal or significantly better objective value, improving other metaheuristics designed previously for the GMSTP.

This holds in particular for instances with 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 is simpler. Differences between the considered algorithms are largest on random Euclidean instances. In this case, VNS produces substantially better results due to the effectiveness of the Global Edge Exchange Neighborhood.

Chapter 7

The Generalized Traveling Salesman Problem

7.1 Introduction

The Generalized Traveling Salesman Problem (GTSP) extends the classical Traveling Salesman Problem (TSP) and is defined as follows. We consider an undirected weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r,\ \bigcup_{i=1}^{r} V_i = V,\ V_i \cap V_j = \emptyset,\ i, j = 1, \ldots, r,\ i \neq j$.

A solution to the GTSP defined on G is a subgraph $S = \langle P, T \rangle$ with $P = \{p_1, p_2, \ldots, p_r\} \subseteq V$ connecting exactly one node from each cluster, i.e. $p_i \in V_i$ for all $1 \leq i \leq r$, and $T \subseteq E$ being a round trip, see Figure 7.1.

The costs of such a round trip 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. When edge costs satisfy the triangle inequality, even if we allow more than one node per cluster to be connected, an optimal solution of the GTSP always contains only one node from each cluster [74]. Obviously, the GTSP is $\mathcal{NP}$-hard since it contains the classical TSP as the special case in which each cluster consists of a single node only.

Parts of this chapter appeared in [59]

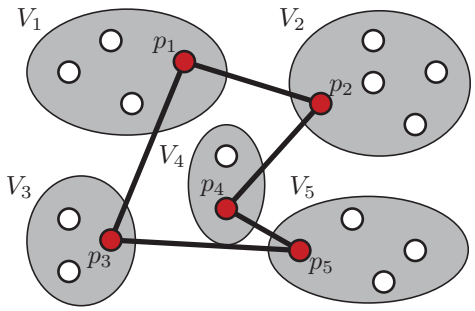


Figure 7.1: Example for a solution to the GTSP.

The GTSP finds practical application particularly in many variants of routing problems, e.g. when some good can be delivered to multiple alternative addresses of customers. Occasionally, such applications can be directly modeled as the GTSP, but more often the GTSP appears as a subproblem [72].

In this chapter, which is based on [59], we present a general Variable Neighborhood Search (VNS) approach for heuristically solving this problem. As local improvement within VNS, we use Variable Neighborhood Descent (VND) based on two different types of exponentially large neighborhoods, which can be seen as dual to each other. One neighborhood structure is the generalized 2-opt, which has been introduced by Renaud et al. [104]; for it, we propose a new incremental evaluation scheme leading to a substantial speed-up. As second neighborhood structure we investigate a new approach: the nodes to be spanned from each cluster are fixed and TSP tours are derived via the chained Lin-Kernighan algorithm.

Section 7.2 gives an overview on research done on the GTSP so far. In section 7.3 we describe the components of our VNS approach in detail, including the solution representation, initialization procedures, and especially the neighborhood structures as well as their search strategies. Experimental results are discussed section 7.4 and we conclude in section 7.5.

7.2 Previous Work

The GTSP was introduced independently by Henry-Labordere [51], Srivastava et al. [112], and Saskena [106]. Laporte et al. [74, 73] provided integer programming formulations for the symmetrical and asymmetrical GTSP, respectively. The formulation for the symmetrical case was later enhanced by Fischetti et al. [33] who

proposed several classes of facet defining inequalities and corresponding separation procedures. Based on these, they developed a branch-and-cut algorithm [34] which could solve instances with up to 442 nodes to optimality.

Several approaches exist which transform the GTSP into the classical TSP. They have been studied by Noon and Bean [88], Lien et al. [79], Dimitrijevic and Saric [20], Laporte and Semet [75], and Behzad and Modarres [6]. Unfortunately, many transformations substantially increase the numbers of nodes and edges and are therefore of limited practical value. Furthermore, some transformations even require additional constraints, thus making general algorithms for the classical TSP inapplicable. Among the more efficient approaches, Dimitrijevic and Saric [20] proposed a transformation of the GTSP into the TSP on a digraph containing twice the number of nodes compared to the original graph. This technique was further improved by Behzad and Modarres [6] where the transformed graph has the same number of nodes as the original graph. However, the transformation increases edge costs significantly, what may lead to problems in some cases.

To approach larger GTSP instances, various metaheuristics have been suggested. Renaud et al. [104] developed a complex composite heuristic whose components can be used for other (meta-)heuristics as well. They introduced generalized k-opt heuristics which are derived from Lin's classical 2-opt and 3-opt local search for the TSP [80]. Snyder and Daskin [111] describe a Genetic Algorithm (GA) that achieves relatively good results in short running times. It uses random keys to encode solutions and a parameterized uniform crossover operator including local improvement based on the 2-opt heuristic to boost solution quality. Wu et al. [121] also proposed a GA using a direct representation in which the spanned nodes from each cluster and the sequence in which they are visited in the tour are stored. This approach has further been enhanced by Huang et al. [61] who apply a so-called hybrid chromosome encoding. However, reported results are on average inferior when compared to those of the GA from [111].

7.3 Variable Neighborhood Search for the GTSP

In this section, we present our VNS approach. First, we propose the solution representation and two initialization procedures. Then, after describing our neighborhoods and the search techniques applied to them, we specify the details of our VNS framework.

7.3.1 Solution Representation and Initialization

In our VNS, we represent a solution $S = \langle P, T \rangle$ in a direct way by storing the spanned nodes of each cluster $P = \{p_1, p_2, \ldots, p_r\}$ with $p_i \in V_i$, $i = 1, \ldots, r$, and additionally the visiting order in the round trip as circular permutation $\pi = \langle \pi_1, \ldots, \pi_r \rangle$ of the cluster indices $\{1, \ldots, r\}$.

Depending on the instance type we use two different procedures to compute feasible initial solutions for the VNS. Both are extensions of well known greedy strategies for the classical TSP. The first algorithm is the (generalized) Nearest Neighbor Heuristic (NNH), and it can in principle be applied to all kinds of instances. The second procedure is specifically targeted to Euclidean instances where the clustering is based on geographical proximity. It exploits Euclidean coordinates of nodes and is called Generalized Insertion Heuristic (GIH). The following paragraphs describe these algorithms in detail.

Nearest Neighbor Heuristic for the GTSP (NNH)

Noon [89] suggested this approach, which computes a feasible solution as follows. We begin to construct a tour S_v from an arbitrarily chosen starting node $v \in V$. Iteratively, the algorithm always continues to the closest node belonging to a cluster that has not been visited yet and includes the corresponding edge. When nodes of all clusters have been reached, the tour is closed by including a final edge back to node v. This process is carried out once for each node in V as starting node, and the best tour is retained. See also Algorithm 16.

Algorithm 16: Nearest Neighbor Heuristic

```
forall v ∈ |V| do
    S_v := ∅
    W := V
    add v to S_v
    v' := v
    for i := 1, ..., r − 1 do
        remove from W all nodes belonging to the same cluster as v'
        u := node in W nearest to v'
        add u to the partial tour S_v
        v' := u
return tour S = S_{v*} with v* = argmin_{v∈V} C(S_v)
```

Generalized Insertion Heuristic for the GTSP (GIH)

This heuristic is inspired by the composite heuristic GI^3 from Renaud et al. [103]. In a first phase, it determines the set of spanned nodes P by calculating for each cluster V_i the node p_i having the lowest sum of distances to all other nodes in other clusters. After fixing these nodes, the CLOCK heuristic from [104] is performed to construct a tour containing many but not necessarily all nodes of P.

Recall that GIH only works on Euclidean instances where the nodes' coordinates are given. The CLOCK heuristic begins a partial tour S at the northernmost node from P. In case of a tie, the easternmost node among the northernmost nodes is chosen. This initial insertion is followed by four loops: In the first loop the procedure appends to S the northernmost node to the east of the last inserted node. In case of a tie, the easternmost node among these is chosen again. The process is repeated until there are no nodes to the east of the last appended node. The second, third, and forth loops work in the same way by appending to S the easternmost node to the south, the southernmost node to the west, and the westernmost node to the north of the last inserted node, respectively.

When the CLOCK heuristic terminates, there are in general some nodes from P left which are not yet included in the tour S. In contrast to the more complex GI^3 heuristic [103], we simply choose for each of these remaining nodes $p_j \in P \backslash H$ greedily the "cheapest" insertion position k so that $c(p_{\pi_{k-1}}, p_j) + c(p_j, p_{\pi_k}) - c(p_{\pi_{k-1}}, p_{\pi_k}) \leq c(p_{\pi_{i-1}}, p_j) + c(p_j, p_{\pi_i}) - c(p_{\pi_{i-1}}, p_{\pi_i})\ \forall i = 1, \ldots, |H|$ with $\pi_0 = \pi_{|H|}$.

As a final step, we try to improve the obtained feasible tour S by calling the shortest path algorithm, which will be introduced in Section 7.3.2. This procedure may replace nodes by other nodes of the same cluster, but it does not modify the visiting order π anymore. See Algorithm 17 for more details of the whole GIH.

This construction heuristic is much faster than the original GI^3, mainly because the latter uses a more sophisticated local improvement. Nevertheless, solutions obtained by GIH are typically only slightly inferior, and it usually takes just a few VNS iterations to catch up with or exceed the quality of GI^3's solutions.

While NNH has time complexity $\Theta(r \cdot |V|^2)$, GIH can be implemented in time $\Theta(|V|^2)$ and usually finds significantly better solutions to Euclidean instances with geographical clustering. However, GIH's applicability is far more limited.

Algorithm 17: Generalized Insertion Heuristic

```
for i := 1, ..., r do
    p_i := node in V_i with min. sum of costs to all other nodes in other clusters
partial tour S := CLOCK heuristic({p_1, ..., p_r})
for j := 1, ..., r do
    if p_j ∉ S then
        k := minarg_{i=1,...,|S|} (c(p_{π_{i-1}}, p_j) + c(p_j, p_{π_i}) − c(p_{π_{i-1}}, p_{π_i})),  π_0 = π_{|S|}
        insert p_j at position k in S
apply shortest path algorithm on S
return S
```

7.3.2 Neighborhood Structures

In our VNS, we use two complementary neighborhood structures. On the one hand, we approach the GTSP from the *global view* by first deciding in which order the clusters are to be visited and then computing an optimal selection of spanned nodes. On the other hand, we may start from the opposite direction and define a set of nodes for which we derive an appropriate tour.

Generalized 2-opt Neighborhood (G2-opt)

Renaud et al. [103] introduced the generalized 2-opt heuristic, which is based on the well known 2-opt heuristic for the classical TSP [80]. G2-opt is defined on a circular permutation $\pi = \langle \pi_1, \ldots, \pi_r \rangle$ indicating the visiting order of the clusters $\langle V_{\pi_1}, \ldots, V_{\pi_r} \rangle$, see Figure 7.2. Note that π also characterizes the global structure to a current solution of the GTSP, see Section 5.1.2. A particular permutation π thus represents the set of all feasible round trips $\langle p_{\pi_1}, p_{\pi_2}, \ldots, p_{\pi_r} \rangle$ with $p_{\pi_i} \in V_{\pi_i},\ i = 1, \ldots, r$, and this set is in general exponentially large with respect to the number of nodes. However, the minimum cost round trip can be determined via a shortest path algorithm in polynomial time.

Given the visiting order of clusters, we can construct a graph containing edges only between nodes of consecutive clusters and a clone of the starting cluster attached to the last cluster, as it is shown in Figure 7.3.

On this graph, we calculate shortest paths starting from each node of the starting cluster and ending at its clone. To ensure that at most one node is included from each cluster, we may simply assume the edges to be directed according to π. The overall cheapest path represents the optimal tour for this cluster order. Formally speaking, let L_{uv} denote the length of the shortest path from node $u \in V_{\pi_k}$ to node

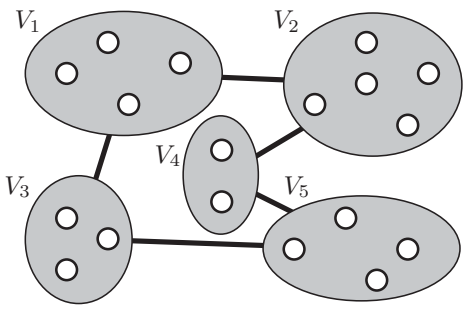


Figure 7.2: Visiting order of clusters characterized by permutation $\pi = \langle 4, 5, 3, 1, 2 \rangle$.

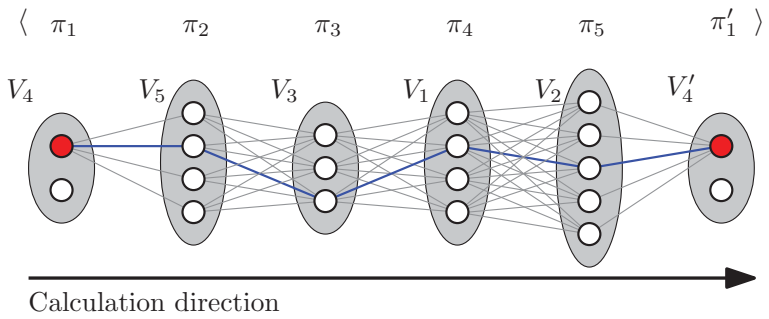


Figure 7.3: Corresponding graph to π in Figure 7.2 on which the shortest path algorithm is applied, starting at the first node of cluster V_4 and ending at its clone in cluster V'_4.

$v \in V_{\pi_l}$, $k < l$. Let L_u be the length of the shortest path containing r-edges, starting from $u \in V_{\pi_1}$, and ending at its clone in V'_{π_1}. The length L of the overall shortest tour respecting visiting order π is:

$$
\begin{array}{ll}
L = \min_{u \in V_{\pi_1}} L_u & \\
L_u = \min_{v \in V_{\pi_r}} (L_{uv} + c(v,u)) & \forall u \in V_{\pi_1} \\
L_{uv} = c(u,v) & \forall u \in V_{\pi_1},\ \forall v \in V_{\pi_2} \\
L_{uv} = \min_{w \in V_{\pi_{k-1}}} (L_{uw} + c(w,v)) & \forall u \in V_{\pi_1},\ \forall v \in V_{\pi_k},\ k = 3, \ldots, r
\end{array}
$$

To reduce the computational effort, we exploit the fact that π is rotation-invariant and choose V_{π_1} so that it is a cluster of smallest cardinality. The complexity of this dynamic programming algorithm is bounded by $O(|V_{\pi_1}| \cdot n^2/r)$.

Our generalized 2-opt neighborhood of a current solution S having cluster ordering π can now be defined as the set of all feasible round trips induced by any cluster ordering π' that differs from π by precisely one inversion I_{ij}, i.e. $\pi' = \langle \pi_1, \ldots, \pi_{i-1}, \pi_j, \ldots, \pi_i, \pi_{j+1}, \ldots, \pi_r \rangle,\ 1 \leq i < j \leq r$.

Incremental bidirectional shortest path calculation.

Instead of determining the shortest path L always from V_{π_1} (a cluster with the smallest number of nodes) to the cloned cluster V'_{π_1}, we can partition this task into three parts:

1. Perform shortest path calculations in forward direction from $u \in V_{\pi_1}$ to each node of a cluster V_{π_m} where m may be chosen arbitrarily.
2. Perform shortest path calculations in backward direction starting from $u' \in V'_{\pi_1}$ to each node of cluster $V_{\pi_{m+1}}$ where u' is the clone of node u.
3. Consider all edges in $E^m = \{(a,b) \in E \mid a \in V_{\pi_m} \wedge b \in V_{\pi_{m+1}}\}$ and the corresponding complete paths from u to u' including the above determined shortest paths to nodes in V_{π_m} and $V_{\pi_{m+1}}$, respectively. Take a $(a^*, b^*) \in E^m$ yielding an overall shortest path, i.e. $L_{ua^*} + c(a^*, b^*) + L_{b^*u'} \leq L_{ua} + c(a,b) + L_{bu'}\ \forall (a,b) \in E^m$.

This procedure, illustrated in Figure 7.4, is in practice almost equally efficient as the simple one-way dynamic programming algorithm. When considering that we want to search the general 2-opt neighborhood, however, it provides the advantage of allowing for a substantially faster incremental evaluation scheme: If π' differs from π by an inversion I_{ij} with $i \leq m \leq j$, we do not have to recalculate the distances and predecessors of the nodes in clusters $V_{\pi_1}, \ldots, V_{\pi_{i-1}}$ and $V_{\pi_{j+1}}, \ldots, V'_{\pi_1}$, assuming we have stored these values in steps 1 and 2 before.

As a matter of course, m is always chosen to lie within the inversion interval. Clusters from V_{π_i} to V_{π_j} are marked "invalid" for both calculation directions after performing the inversion. Whenever we apply the shortest path algorithm in a particular direction, the evaluation is skipped for all clusters which are still valid, and the actual computation starts at the first invalid cluster. When processing these clusters, their "invalid" flags are removed.

To fully exploit this incremental evaluation, we further enumerate the possible inversions of π in a specific way: First, all inversions of pairs of two adjacent clusters are considered from left to right, then the inversions of all triplets in the reverse direction from right to left, next all 4-cluster inversions from left to right again, etc.

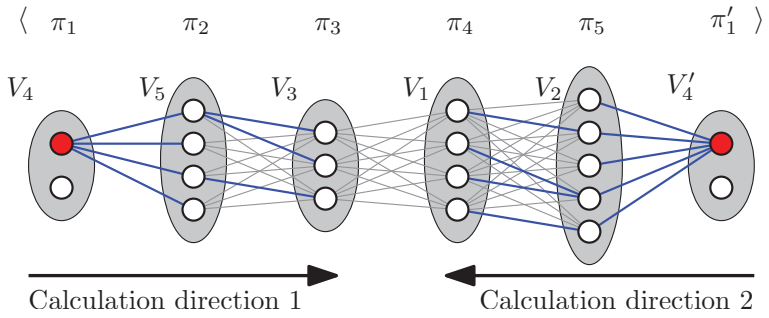


Figure 7.4: Example for a bidirectional shortest path calculation with $m = 3$.

Hereby, π_1 (and its clone in the corresponding graph for the shortest path calculation) remain fixed. See also Figure 7.5. This strategy allows the largest data-reuse and minimizes the total number of clusters for which computations are necessary. It is in particular advantageous when we use a *next improvement* strategy in the local search, since we start with inversions of smallest size yielding the largest time savings; see Algorithm 18.

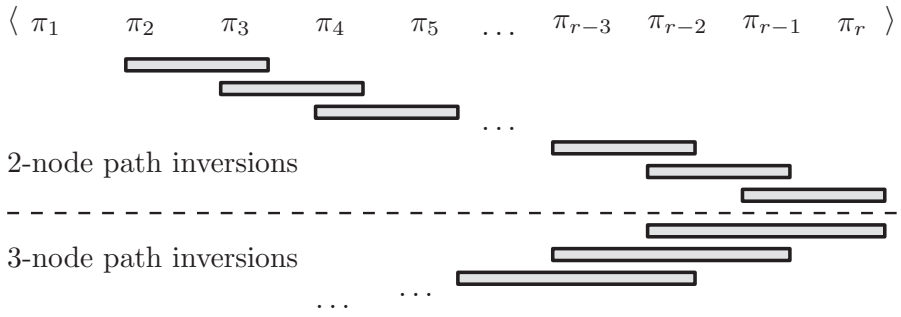


Figure 7.5: Enumeration order of the inversions on π for making best use of the incremental bidirectional shortest path calculations.

In the worst case, when we have to evaluate the whole neighborhood, $O(r^2)$ inversions must be considered. A naive complete enumeration would require time $O(r^2 \cdot |V_{\pi_1}| \cdot n^2/r) = O(|V_{\pi_1}| \cdot n^2 \cdot r)$. To be more precise, we have $(r - l + 1)$ possibilities for inversions of length l, $l = 2, \ldots, r - 2$. For each of them, the classical shortest path algorithm would have to consider all r clusters. However, with the incremental bidirectional shortest path calculation, we only have to consider $l + 1$ clusters after the first iteration. Hence, the classical algorithm evaluates $\sum_{l=2}^{r-2} r \cdot (r - l + 1) = \frac{r^3 - r^2 - 6r}{2}$ clusters while the incremental scheme only pro-

Algorithm 18: Generalized 2-opt Neighborhood (solution $S = \langle P, T \rangle$)

```
for l := 2, ..., r − 2 do
    if l is even then
        for i := 2, ..., r − l + 1 do
            π' := ⟨π_1, ..., π_{i−1}, π_{i+l−1}, ..., π_i, π_{i+l}, ..., π_r⟩
            Apply incremental bidirectional shortest path calculation on π'
            if obtained solution S' is better than original solution S then
                return solution S'
    else
        for i := r − l + 1, ..., 2 do
            π' := ⟨π_1, ..., π_{i−1}, π_{i+l−1}, ..., π_i, π_{i+l}, ..., π_r⟩
            Apply incremental bidirectional shortest path calculation on π'
            if obtained solution S' is better than original solution S then
                return solution S'
return : no better solution found, i.e. S is a local optimum w.r.t. G2-opt
```

cesses $\sum_{l=2}^{r-2}(l+1)\cdot(r-l+1) = \frac{r^3+6r^2-25r-6}{6}$ clusters for the whole neighborhood. Asymptotically, the latter is faster by factor 3.

Node Exchange Neighborhood (NEN)

With this new neighborhood structure, the search focuses on the set of spanned nodes $P = \{p_1, \ldots, p_r\}$. The node exchange neighborhood of a current solution S with spanned nodes P includes all feasible tours S' for each node set P' that can be derived from P by replacing one spanned node $p_i \in V_i$, $i \in \{1, \ldots, r\}$, by another node v of the same cluster V_i. This neighborhood therefore is induced by $\sum_{i=1}^{r}(|V_i| - 1) = O(|V|)$ different node sets resulting in a total of $O(|V| \cdot r!)$ round trips.

Unfortunately, determining the minimum cost round trip for a given node set P' is $\mathcal{NP}$-hard since this subproblem corresponds to the classical TSP. Hence, instead of calculating the optimal round trip, we use the well known Chained Lin-Kernighan (CLK) algorithm [82] implemented in the Concorde library[2] to find a good but not necessarily optimal tour S' for a certain P'.

Though the size of this TSP is relatively small ($|P'| = r$), a complete evaluation of NEN is relatively time-demanding – even when using CLK – since we have to solve

[2] www.tsp.gatech.edu/concorde.html

$O(|V|)$ different TSPs. To further speed up the neighborhood search, we restrict CLK to consider edges of the k-nearest-neighbor graph induced by P' only. For Euclidean instances and available point positions, this k-nearest-neighbor graph is efficiently derived using a KD-tree data structure. Tuning the parameter k, we can balance between speed and thoroughness of the search process. For the actual tests, we set k to 10. In contrast to G2-opt, we use a *best improvement* strategy for NEN since it yielded better results during our tests. Algorithm 19 summarizes the steps of evaluating NEN.

Algorithm 19: Node Exchange Neighborhood (solution $S = \langle P, T \rangle$)

```
for i := 1, ..., r do
    forall v ∈ V_i \ {p_i} do
        P' := P \ {p_i} ∪ {v}
        determine k-nearest-neighbor graph G^k induced by P'
        apply CLK on G^k to obtain round trip S'
        if current solution S' better than so far best then
            save S' as so far best
restore and return best solution found
```

7.3.3 Variable Neighborhood Search Framework

We use the general variable neighborhood search (VNS) scheme with embedded variable neighborhood descent (VND), see Section 3.4.

Arrangement of the neighborhoods: We alternate between G2-opt and NEN in this order. G2-opt is always considered first since its evaluation has a lower computational complexity.

Shaking in VNS: To perform shaking, we randomly exchange s spanned nodes by other nodes of the corresponding clusters and apply s random swap moves on the cluster ordering π. A swap move exchanges two positions in π. Parameter s depends on the number of clusters in the input graph and varies from 1 to $\frac{r}{7}$. We obtained the best results with these settings for s during our tests.

7.4 Computational Results

We tested the VNS on Euclidean TSPlib instances with geographical clustering, see Section 5.3. Our experiments were performed on a Pentium 4 2.6 GHz PC. In order to compute average values and standard deviations, we performed 30 runs for each instance. The VNS terminated after 10 consecutive outer iterations without finding

Table 7.1: Results on TSPlib instances with geographical clustering.

Instance	B&C		GI^3		rk-GA		hc-GA		VNS		
Name	C_{opt}	*time*	*gap*	*time*	$\overline{gap}$	$\overline{time}$	$\overline{gap}$	$\overline{time}$	$\overline{gap}$	σ_{gap}	$\overline{time}$
kroa100	9711	18.4s	**0.00%**	6.8s	**0.00%**	0.4s	-	-	**0.00%**	0.00	2.5s
krob100	10328	22.2s	**0.00%**	6.4s	**0.00%**	0.4s	-	-	**0.00%**	0.00	0.4s
rd100	3650	16.6s	**0.00%**	7.3s	**0.00%**	0.5s	-	-	**0.00%**	0.00	0.9s
eil101	249	25.6s	0.40%	5.2s	**0.00%**	0.4s	-	-	0.04%	0.12	16.3s
lin105	8213	16.4s	**0.00%**	14.4s	**0.00%**	0.5s	-	-	**0.00%**	0.00	0.6s
pr107	27898	7.4s	**0.00%**	8.7s	**0.00%**	0.4s	-	-	**0.00%**	0.00	0.5s
pr124	36605	25.9s	0.43%	12.2s	**0.00%**	0.6s	-	-	**0.00%**	0.00	26.6s
bier127	72418	23.6s	5.55%	36.1s	**0.00%**	0.4s	-	-	**0.00%**	0.00	1.4s
pr136	42570	43.0s	1.28%	12.5s	**0.00%**	0.5s	-	-	**0.00%**	0.00	48.1s
pr144	45886	8.2s	0.00%	16.3s	**0.00%**	1.0s	-	-	**0.00%**	0.00	4.0s
kroa150	11018	100.3s	0.00%	17.8s	**0.00%**	0.7s	**0.00%**	0.4s	**0.00%**	0.00	1.2s
krob150	12196	60.6s	0.00%	14.2s	**0.00%**	0.9s	**0.00%**	0.9s	**0.00%**	0.00	3.7s
pr152	51576	94.8s	0.47%	17.6s	**0.00%**	1.2s	**0.00%**	0.6s	**0.00%**	0.00	7.6s
u159	22664	146.4s	2.60%	18.5s	**0.00%**	0.8s	**0.00%**	1.0s	**0.00%**	0.00	22.6s
rat195	854	245.9s	**0.00%**	37.2s	**0.00%**	1.0s	-	-	0.01%	0.04	105.6s
d198	10557	763.1s	0.60%	60.4s	**0.00%**	1.6s	-	-	0.02%	0.05	141.3s
kroa200	13406	187.4s	**0.00%**	29.7s	**0.00%**	1.8s	0.01%	1.8s	**0.00%**	0.00	16.9s
krob200	13111	268.5s	**0.00%**	35.8s	**0.00%**	1.9s	0.06%	8.0s	**0.00%**	0.00	18.8s
ts225	68340	37875.9s	0.61%	89.0s	**0.02%**	2.1s	0.13%	19.0s	0.03%	0.07	274.4s
pr226	64007	106.9s	**0.00%**	25.5s	**0.00%**	1.5s	**0.00%**	0.6s	**0.00%**	0.00	1.7s
gil262	1013	6624.1s	5.03%	115.4s	0.79%	1.9s	**0.00%**	41.2s	0.05%	0.16	372.5s
pr264	29549	337.0s	0.36%	64.4s	**0.00%**	2.1s	**0.00%**	3.1s	0.01%	0.04	268.2s
pr299	22615	812.8s	2.23%	90.3s	0.11%	3.2s	0.10%	68.6s	**0.00%**	0.01	220.5s
lin318	20765	1671.9s	4.59%	206.8s	0.62%	3.5s	0.72%	18.3s	**0.30%**	0.61	320.1s
rd400	6361	7021.4s	1.23%	403.5s	1.18%	5.9s	2.15%	17.4s	**0.74%**	0.51	502.0s
fl417	9651	16719.4s	0.48%	427.1s	0.05%	5.3s	0.12%	19.4s	**0.00%**	0.00	92.4s
pr439	60099	5422.8s	3.52%	611.0s	0.26%	9.5s	0.76%	10.9s	**0.12%**	0.11	519.0s
pcb442	21657	58770.5s	5.91%	567.7s	1.70%	9.0s	0.94%	31.8s	**0.08%**	0.08	596.6s
Average gaps			1.26%		0.17%		0.30%		0.05%		

a new best solution.

Table 7.1 presents results of our VNS and compares them to those of Fischetti et al's exact branch-and-cut algorithm (B&C) [34], the GI^3 heuristic [104], the random key GA (rk-GA) [111], and the hybrid chromosome GA (hc-GA) [61]. Listed are for each instance its name, the numbers of nodes and clusters, the optimal solution value and run-time of B&C, and average percentage gaps of the heuristics' final objective values to the optimum solution value, as well as corresponding CPU-times. Best results are printed bold.

Since the B&C algorithm ran on a HP 9000/720, the GI^3 heuristic on a Sun Sparc Station LX, the rk-GA on a Pentium 4, 3.2 GHz PC, and the hc-GA on a 1.2 GHz PC, it is hard to compare the CPU-times directly. Nevertheless, it is obvious that in particular the rk-GA is very fast and computes high quality results within a few seconds. Comparing VNS with both GAs, VNS requires significantly more time, but it is often able to find superior solutions.

Especially on the larger instances with ≥ 299 nodes, VNS benefits from the sophisticated large neighborhood search and its average gaps of are consistently substantially smaller than those of all other considered heuristics. For 18 out of the 28 instances, VNS was even able to obtain optimal solutions in all of the 30 performed runs; its total average gap is only 0.05%, and no average gap exceeds 0.75%. From all considered heuristics, GI^3 was the weakest, with worst average results and running times in the same order of magnitude as our VNS.

In particular for the two large instances pr226 and fl417, already VND was able to directly identify the optimal solutions, i.e. merely alternating between G2-opt and NEN was sufficient to get to the global optima, and no VNS iterations were required. This documents how well these neighborhood structures complement each other.

7.5 Conclusions

In this chapter, we proposed a variable neighborhood search approach for the generalized traveling salesman problem utilizing two large neighborhood structures. They can be seen as dual to each other: While G2-opt predefines the possible cluster orderings and uses a relatively sophisticated but efficient procedure for augmenting these partial solutions with appropriate selections of nodes, the situation is vice versa in the newly proposed NEN.

Considering in particular G2-opt, the described incremental evaluation scheme turned out to be a major speed-up factor in comparison to the previously used evaluation via independent standard shortest path calculations.

It further turned out that the VNS slightly benefits from a good starting solution. Therefore, we described the more generally applicable nearest neighbor heuristic and particularly for Euclidean instances with given point positions the generalized insertion heuristic. Both are reasonably fast and provide solutions of appropriate quality.

We tested the VNS on TSPlib instances with geographical clustering consisting of up to 442 nodes. Compared to two recent genetic algorithms, the VNS performs slower, but it is able to generate remarkably better solutions, in particular for larger instances.

Future work will in particular include tests on other types of instances, e.g. with non-Euclidean distances and incomplete graphs. An incremental evaluation scheme for NEN seems to be a challenging task but might further speed up the algorithm. Promising is also the combination of these neighborhood structures with others, and to investigate their application in other types of metaheuristics.

Chapter 8

The Generalized Minimum Edge Biconnected Network Problem

8.1 Introduction

The Generalized Minimum Edge Biconnected Network Problem (GMEBCNP) is defined as follows. We consider a complete, undirected weighted graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r,\ \bigcup_{i=1}^{r} V_i = V,\ V_i \cap V_j = \emptyset\ \forall i, j = 1, \ldots, r,\ i \neq j$.

A solution to the GMEBCNP defined on G is a subgraph $S = \langle P, T \rangle,\ P = \{p_1, \ldots, p_r\} \subseteq V$ connecting exactly one node from each cluster, i.e. $p_i \in V_i,\ \forall i = 1, \ldots, r$, and containing no bridges [30, 62, 76], see Figure 8.1. A bridge is an edge which does not lie on any cycle and thus its removal would disconnect the graph. The costs of such an edge biconnected network are its total edge costs, i.e. $c(T) = \sum_{(u,v)\in T} c(u, v)$, and the objective is to identify a feasible solution with minimum costs. This problem is $\mathcal{NP}$-hard since the task of finding a minimum cost biconnected network spanning all nodes of a given graph is already $\mathcal{NP}$-hard [30, 36], which is the special case with $|V_i| = 1,\ \forall i = 1, \ldots, r$.

Parts of this chapter appeared in [77, 56]

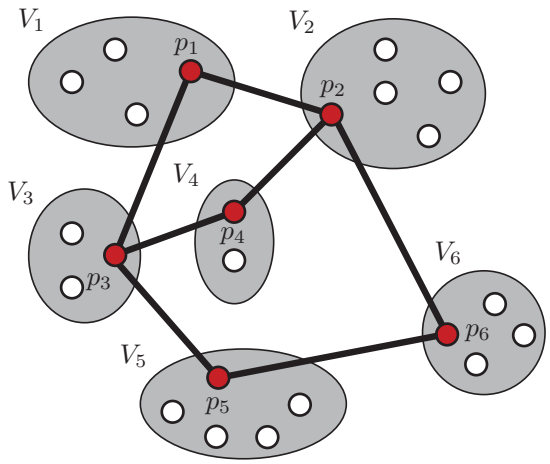


Figure 8.1: Example for a solution to the GMEBCNP.

The GMEBCNP was introduced by Huygens [62] and it arises in the design of backbones in large communication networks. For example, we can consider the possible access points of an existing local network as nodes of a cluster when designing a backbone network connecting multiple LANs. Survivability by means of a single link outage is covered via the considered edge redundancy [30].

We propose three variants of Variable Neighborhood Search (VNS) approaches for the GMEBCNP. We also propose a Mixed Integer Programming (MIP) formulation based on multi commodity flows for solving smaller instances of this problem to provable optimality.

The remainder of this chapter based on [56] is organized as follows. In Section 8.2, we give an overview on research done on the GMEBCNP and other related problems so far. Section 8.3 describes the components of our VNS approach in detail and Section 8.4 introduces the MIP formulation. Section 8.5 describes the instances we used for our computational experiments. Finally, we describe experimental results in Section 8.6 and conclude in Section 8.7.

8.2 Previous Work

Despite the importance of this problem in survivable network design, not much research has been done for this particular problem until now. Huygens [62] studied the GMEBCNP and provided integer programming formulations along with separation techniques, but no practical results on actual instances were published.

Though not identical, the GMEBCNP is related to the Generalized Minimum Spanning Tree Problem (GMSTP) [86]. As a matter of course, some concepts can be adopted from Chapter 6. There, we approached the GMSTP from contrary directions by utilizing two dual representations and associated neighborhood structures within a VNS. By fixing the spanned nodes of each cluster, optimal edges can be efficiently computed via Kruskal's MST algorithm. On the other hand, by fixing the connections between clusters, an optimal choice of spanned nodes can be determined via dynamic programming in polynomial time. Though these concepts are not directly applicable for the GMEBCNP due to higher complexity, some neighborhood structures of the current work are also based on these ideas.

Another related problem is the Generalized Traveling Salesman Problem (GTSP) [51, 106, 112], see Chapter 7. Since every solution to the GTSP is obviously edge biconnected and therefore also a solution to the GMEBCNP, its solution value can be regarded as an upper bound to the current problem. However, the GTSP is also $\mathcal{NP}$-hard and especially on large graphs, these upper bounds become rather poor as the overall costs of solutions to the GMEBCNP are substantially lower. Therefore, we will not consider the GTSP any further in this chapter.

The classical minimum edge biconnected network problem (2-ECNP or MEBCNP) has been shown to be $\mathcal{NP}$-hard by a reduction from the Hamiltonian cycle problem [36]. Khuller and Viskin [67] proposed a factor two approximation algorithm and showed that approximating the optimal solution to some additive constant is impossible in polynomial time unless P=NP. Czumaj and Lingas [12] presented more detailed results with respect to the approximability of the MEBCNP and gave a PTAS for the case of complete Euclidean graphs in $\mathbb{R}^{\mathrm{d}}$.

8.3 Variable Neighborhood Search for the GMEBCNP

In this section, we will first describe the solution representation and the initialization procedure, then discuss our neighborhood structures along with techniques to optimize the search process. Finally we assemble our VND and VNS framework.

8.3.1 Solution Representation

For each solution, we store the spanned nodes $P = \{p_1, \ldots, p_r\}$ and the global connections T^{g} of the global structure $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$, see Figure 8.2 and Section 5.1.2.

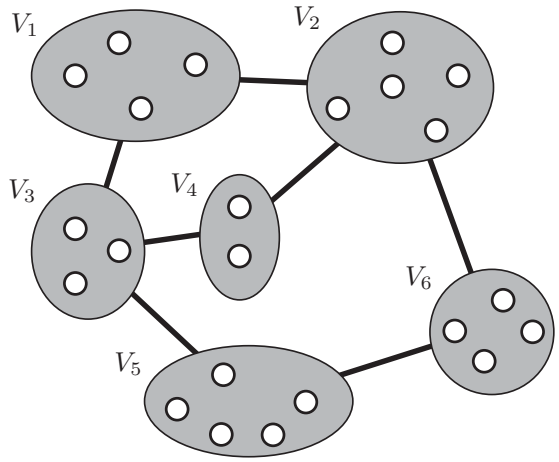


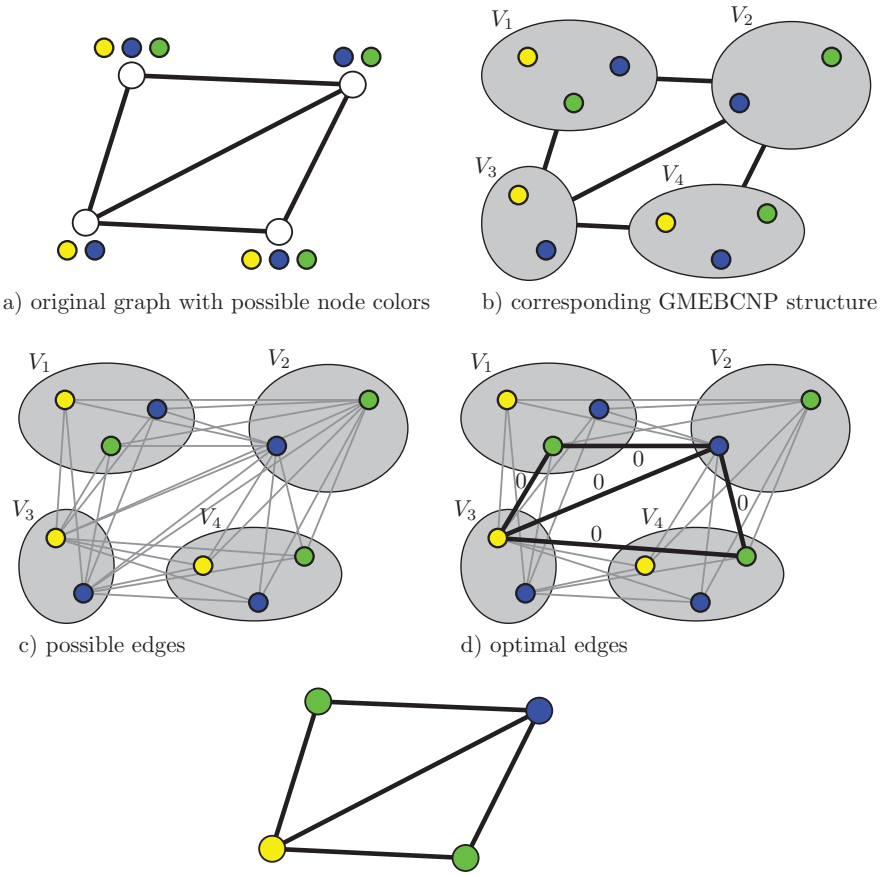
Figure 8.2: Example for the global structure of the solution in Figure 8.1.

Spanned nodes $p_1, \ldots, p_r$ alone are insufficient to represent a solution, since finding the cheapest edges for them corresponds to the classical minimum edge biconnected network problem which is $\mathcal{NP}$-hard [30, 36]. Similarly, a representation via global connections alone is also insufficient, since identifying a set of optimal nodes even when restricted to a given global structure is also $\mathcal{NP}$-hard. Since the latter is not obvious, we prove it by a reduction from the graph coloring problem:

Theorem 4 *Given an (edge biconnected) global structure $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$, $T^{\mathrm{g}} \subseteq E^{\mathrm{g}}$, it is $\mathcal{NP}$-hard to identify an optimal selection of nodes P yielding a corresponding minimum cost GMEBCNP solution.*

Proof Consider the classical $\mathcal{NP}$-hard graph coloring problem [36] on an undirected graph $H = \langle U, F \rangle$ (Figure 8.3a): To each node, one color of a restricted set of colors needs to be assigned in such a way that any pair of adjacent nodes is differently colored. We consider the input graph H as global structure S^{g}, and the clustered graph $G = \langle V, E \rangle$ is derived by the following procedure: Each node $i \in U$ becomes a cluster V_i and for each possible color c of i, we introduce a node v_i^c in cluster V_i (Figure 8.3b). For each edge $(i, j) \in F$, we create in the clustered graph edges $(v_i^c, v_j^d)\ \forall\ v_i^c \in V_i,\ \forall\ v_j^d \in V_j$ (Figure 8.3c). An edge's cost is 0 if $c \neq d$ and 1 otherwise.

If we are able to solve the problem of identifying the optimal nodes of the clusters in order to minimize the GMEBCNP's solution costs (Figure 8.3d), we also solve the original graph coloring problem on H: Suppose v_i^c is the selected node in cluster V_i,



a) original graph with possible node colors

b) corresponding GMEBCNP structure

c) possible edges

d) optimal edges

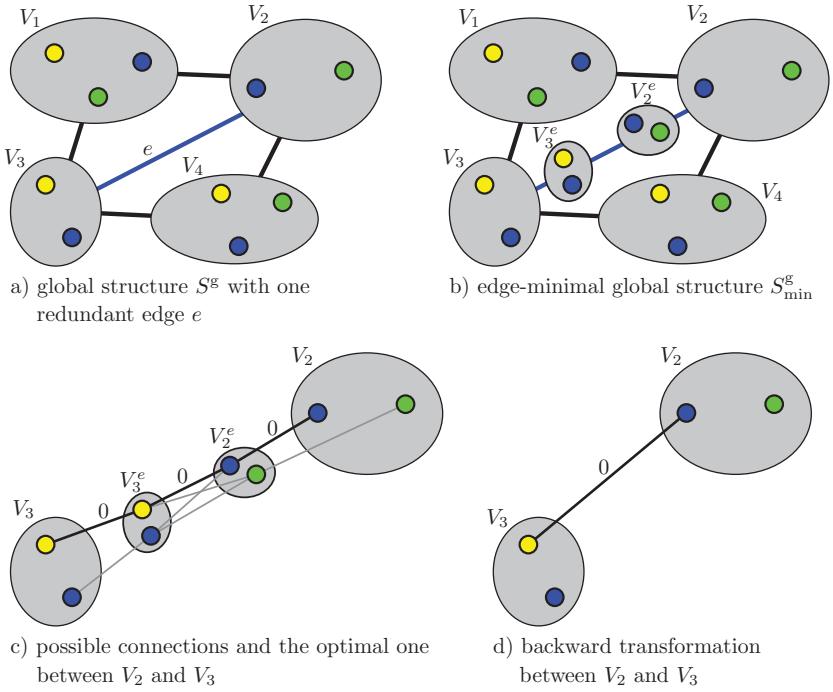
e) solution to the coloring problem

Figure 8.3: Transformation of the graph coloring problem into the problem of identifying an optimal node selection w.r.t. a given global structure.

then c becomes the color of node $i \in U$ (Figure 8.3e). The validity of the theorem thus follows from the $\mathcal{NP}$-hardness of the graph coloring problem. □

So far we considered arbitrary global structures that may contain redundant edges, i.e. edges which can be removed without violating the edge biconnectivity property. However, if the global structure meets some particular properties, optimal spanned nodes can be identified in polynomial time, e.g. for the GMSTP and GTSP. The following theorem and proof strengthens our result by showing that the $\mathcal{NP}$-hardness even holds when the global structure is edge-minimal, as it is always the case for an optimal global structure to the GMEBCNP.

Theorem 5 *Given an edge-minimal edge biconnected global structure* $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$, $T^{\mathrm{g}} \subseteq E^{\mathrm{g}}$, *it is* $\mathcal{NP}$*-hard to identify an optimal selection of nodes* P *yielding a corresponding minimum cost GMEBCNP solution.*



a) global structure S^{g} with one redundant edge e

b) edge-minimal global structure $S^{\mathrm{g}}_{\mathrm{min}}$

c) possible connections and the optimal one between V_2 and V_3

d) backward transformation between V_2 and V_3

Figure 8.4: Transformation of the graph coloring problem: Extension towards an edge-minimal global structure.

Proof If the global structure S^{g}, after the previous transformation, is not edge-minimal, T^{g} contains at least one redundant connection (Figure 8.4a). For each such redundant connection $e = (V_i, V_j) \in T^{\mathrm{g}}$, we add new artificial clusters V_i^e and V_j^e, which are exact copies of V_i and V_j, respectively. The global connection (V_i, V_j) gets replaced by (V_i, V_i^e), (V_i^e, V_j^e), and (V_j^e, V_j) (Figure 8.4b). Let $S^{\mathrm{g}}_{\mathrm{min}} = \langle V^{\mathrm{g}}_{\mathrm{min}}, T^{\mathrm{g}}_{\mathrm{min}} \rangle$ denote the resulting structure, which obviously is edge-minimal.

When adding the clusters V_i^e and V_j^e, we have to modify the edges E in the clustered graph G as well. We replace each edge $(u, v) \in E \mid u \in V_i \wedge v \in V_j$ by $(u^e, v^e) \mid u^e \in V_i^e \wedge v^e \in V_j^e$ with u^e and v^e being the new copies of u and v, respectively. Between V_i and V_i^e, we add edges (u, u^e) with costs 0 for all $u \in V_i$. The same procedure is applied for V_j and V_j^e (Figure 8.4c). Let $G' = \langle V', E', c \rangle$ denote the resulting modified graph. By determining an optimal selection of nodes on G' subject to

the global structure $T^{\mathrm{g}}_{\mathrm{min}}$, we get the optimal node set on G subject to the global structure T^{g} by simply ignoring the artificial clusters (Figure 8.4d). Thus, we also obtain the optimal solution to the original graph coloring problem on H by choosing the corresponding colors.

The backward transformation is valid because only one node (hence one color) is chosen per cluster as we solve the GMEBCNP containing exactly one node per cluster. Furthermore, the selected node of a cloned cluster is always the clone of the selected node in the original cluster due to the zero cost edges. □

8.3.2 Initialization

Our strategy for determining an initial solution for the GMEBCNP is inspired the Christofides heuristic for the traveling salesman problem [8] and therefore called *Adapted Christofides Heuristic* (ACH). Its pseudo-code is listed in Algorithms 20 and 21. We start with a solution to the Generalized Minimum Spanning Tree Problem computed via the *Improved Kruskal Heuristic* (IKH) from Golden et al. [43], see Section 6.3.1. This algorithm considers edges in increasing cost-order and adds an edge to the solution iff it does not introduce a cycle and does not connect a second node of any cluster. By fixing an initial node to be in the resulting generalized spanning tree, different solutions can be obtained. Therefore, this process is carried out $|V|$ times, once for each node to be initially fixed, and the overall cheapest spanning tree is adopted.

To augment this spanning tree to become a valid solution for the GMEBCNP, we then determine the set V_{o} of nodes with odd degree $\deg(v)$ and sort the edges induced by V_{o} and not contained in the spanning tree with respect to increasing edge costs. Next, we derive a matching T_M for the nodes V_0 by iterating through these edges and adopting any edge incident to two yet uncovered nodes until all nodes in V_{o} are covered. Note that this procedure, shown in Algorithm 21, does not necessarily generate a minimum cost matching.

Unfortunately, these steps still do not necessarily yield a solution completely satisfying the edge biconnectivity property. More precisely, ACH will fail to find a perfect matching if the last two uncovered nodes u' and v' are adjacent in the spanning tree. However, it is easy to see that $S_0 = \langle P, T_0 \cup T_{\mathrm{M}} \rangle$ will consist of at most two edge biconnected components even if no perfect matching is found, as both eventually existing components of $\langle P, T_0 \cup T_{\mathrm{M}} \setminus \{(u', v')\} \rangle$ are Eulerian in that case. Therefore Algorithm 20 adds the cheapest edge not yet part of the solution between the eventually remaining two edge biconnected components.

At the end, we remove redundant edges which might occur due to the previous step with regard to decreasing edge costs. Ties that might appear due to edges having identical costs are broken at random.

The overall time complexity of ACH is bounded by $O(|E| \log |E| + r^3)$. The most expensive operations are generating a solution to the GMSTP by IKH with complexity $O(|E| \log |E|)$ [43], and finding and removing the redundant edges which can be done in complexity $O(r^3)$.

Algorithm 20: Adapted Christofides Heuristic

$S_0 = \langle P, T_0 \rangle :=$ feasible GMST computed via *Improved Kruskal Heuristic*
$T_\mathrm{M} :=$ compute matching (S_0) *// see Algorithm 21*
$S_0 := \langle P, T_0 \cup T_\mathrm{M} \rangle$
if S_0 has two edge biconnected components **then**
  add cheapest edge $\in E \setminus T_\mathrm{o}$ between the two edge biconnected components
remove redundant edges

Algorithm 21: compute matching (GMST $S_0 = \langle P, T_0 \rangle$)

$T_\mathrm{M} := \emptyset$
$V_\mathrm{o} := \{v \in P \mid \deg(v) \text{ is odd}\}$
$E_\mathrm{o} := \{(u, v) \in E \mid u, v \in V_\mathrm{o} \wedge (u, v) \notin T_\mathrm{o}\}$
sort E_o according to increasing costs, i. e. $c(e_1) \leq \cdots \leq c(e_{|E_\mathrm{o}|})$
$i := 1$
while $V_\mathrm{o} \neq \emptyset \wedge i < |E_\mathrm{o}|$ **do**
  // current edge $e_i = (u_i, v_i)$
  if $u_i \in V_\mathrm{o} \wedge v_i \in V_\mathrm{o}$ **then**
    $T_\mathrm{M} := T_\mathrm{M} \cup \{e_i\}$
    $V_\mathrm{o} := V_\mathrm{o} \setminus \{u_i, v_i\}$
  $i := i + 1$
return T_M

8.3.3 Neighborhood Structures

We propose four different types of neighborhood structures, each of them focusing on different aspects of solutions to the GMEBCNP. For two of them, there exist simple versions and advanced versions making use of the following graph reduction technique.

Graph Reduction: Though it is generally not possible to derive an optimal set of spanned nodes in polynomial time when a global structure S^{g} is given, this task becomes feasible once the spanned nodes in a few specific clusters are fixed. The underlying concept, called *graph reduction*, is based on the observation that good solutions to the GMEBCNP usually consist of only few clusters with spanned nodes of degree greater than two, denoted as *branching clusters*, and long paths of clusters with spanned nodes of degree two connecting them, denoted as *path clusters*. Once the spanned nodes within all branching clusters are fixed, it is possible to efficiently determine for each cluster path the optimal selection of remaining nodes by computing the shortest path between the two fixed branching cluster nodes in the subgraph of G represented by the cluster path.

Formally, for any global structure $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$, we can define a *reduced global structure* $S^{\mathrm{g}}_{\mathrm{red}} = \langle V^{\mathrm{g}}_{\mathrm{red}}, T^{\mathrm{g}}_{\mathrm{red}} \rangle$. V_{red} denotes the branching clusters, i.e. $V^{\mathrm{g}}_{\mathrm{red}} = \{V_i \in V^{\mathrm{g}} \mid \deg(V_i) \geq 3\}$ with $\deg(V_i)$ being the degree of cluster V_i in S^{g}. $T^{\mathrm{g}}_{\mathrm{red}}$ consists of edges which represent strings of path clusters connecting these branching clusters, i.e. $T^{\mathrm{g}}_{\mathrm{red}} = \{(V_a, V_b) \mid (V_a, V_{k_1}), (V_{k_1}, V_{k_2}), \ldots, (V_{k_{l-1}}, V_{k_l}), (V_{k_l}, V_b) \in T^{\mathrm{g}} \wedge V_a, V_b \in V^{\mathrm{g}}_{\mathrm{red}} \wedge V_{k_i} \notin V^{\mathrm{g}}_{\mathrm{red}},\ \forall i = 1, \ldots, l\}$.

Corresponding to the reduced global structure $S^{\mathrm{g}}_{\mathrm{red}} = \langle V^{\mathrm{g}}_{\mathrm{red}}, T^{\mathrm{g}}_{\mathrm{red}} \rangle$ we can define a *reduced graph* $G_{\mathrm{red}} = \langle V_{\mathrm{red}}, E_{\mathrm{red}} \rangle$ with all nodes of branching clusters $V_{\mathrm{red}} = \{v \in V_i \mid V_i \in V^{\mathrm{g}}_{\mathrm{red}}\}$ and edges between any pair of nodes whose clusters are adjacent in the reduced global structure, i.e. $(i, j) \in E_{\mathrm{red}} \Leftrightarrow (V_i, V_j) \in T^{\mathrm{g}}_{\mathrm{red}}, \forall i \in V_i, j \in V_j$. Each such edge (i, j) corresponds to the shortest path connecting i and j in the subgraph of G represented by the reduced structure's edge (V_i, V_j), and (i, j) therefore gets assigned this shortest path's costs, see Figure 8.5.

When fixing the spanned nodes in $V^{\mathrm{g}}_{\mathrm{red}}$ we can determine the costs of the corresponding solution S with optimally chosen nodes in path clusters efficiently by using the precomputed shortest path costs stored with the reduced graph's edges. Decoding the corresponding solution, i.e. making the optimal spanned nodes within path clusters explicit, is done by choosing all nodes lying at the shortest paths corresponding to used edges from E_{red}. Detailed pseudocode for the graph reduction procedure is given in Algorithm 22.

An edge-minimal solution to the GMEBCNP, as it is obtained from our initialization procedure, may consist of $O(r)$ edges only. When computing the corresponding reduced global structure and reduced graph, each solution edge is considered exactly once, and for each edge all combinations of nodes within three clusters need to be considered. The overall worst case time complexity is thus $O(r \cdot d^3_{\max})$, with $d_{\max}$ being the maximum number of nodes within a single cluster.

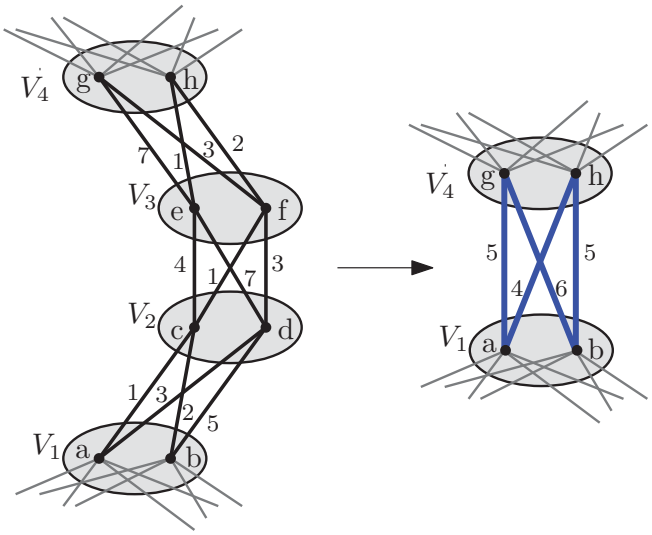


Figure 8.5: Computing the shortest paths between all node pairs of two branching clusters V_1 and V_4.

Algorithm 22: reduce (solution $S = \langle P, T \rangle$)

$S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ is the global structure 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**
  $E^{\mathrm{g}}_{\mathrm{red}} := E^{\mathrm{g}}_{\mathrm{red}} \cup \{(V_1, V_{\mathrm{n}})\}$
  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

Figure 8.6 shows an example of reducing the number of clusters to be further considered from nine to two. Note that cyclic paths in T^{g}, i.e. $V_a = V_b$, will yield loops in $T^{\mathrm{g}}_{\mathrm{red}}$, as it is the case with (V_6, V_6) in our example. Furthermore, multiple cluster

paths may exist between two branching clusters, as for V_1 and V_6, and they lead to multi-edges in the reduced graph. We can treat such multi-edges as simple edges by summing up the costs of associated single edges.

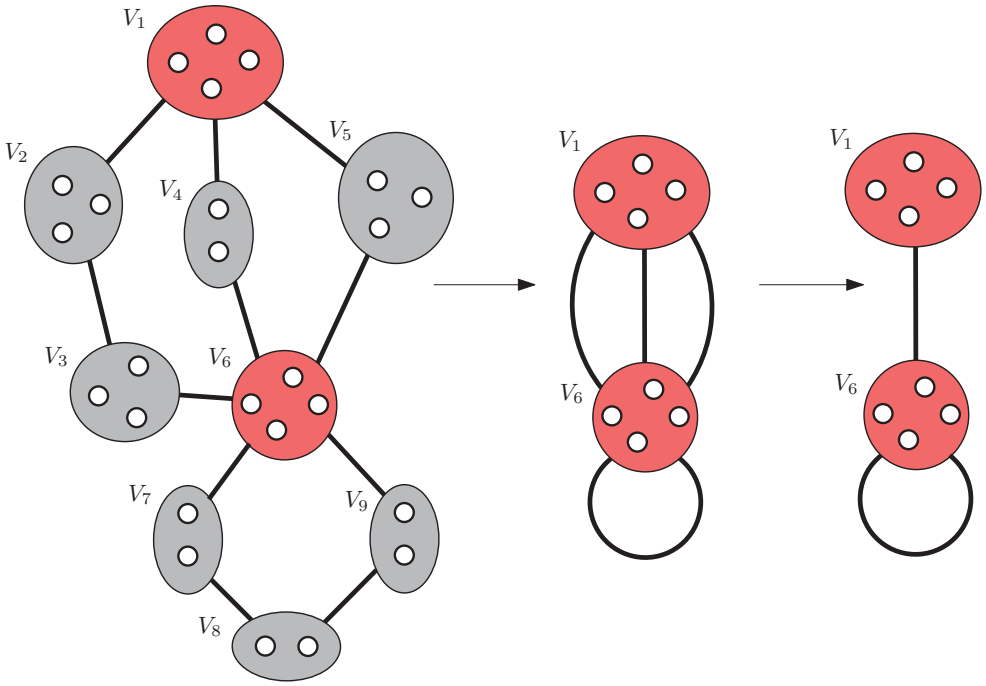


Figure 8.6: Example for graph reduction: V_1 and V_6 are branching clusters, while all others are path clusters.

Simple Node Optimization Neighborhood (SNON)

With this neighborhood structure we try to optimize a solution with respect to the spanned nodes within clusters while keeping the global structure. SNON consists of all solutions S' that differ from the current solution S by exactly one spanned node. A move within SNON (see Figure 8.7) is accomplished by changing $p_i \in V_i$ to $p'_i \in V_i, p_i \neq p'_i$, for $i \in \{1, \ldots, r\}$, removing all edges incident to p_i and adding edges from p'_i to all nodes that were incident to p_i in S, see Algorithm 23.

Since the objective value can be updated in an incremental way, the time complexity of a complete search in SNON is $O(|V| \cdot d_{\max})$.

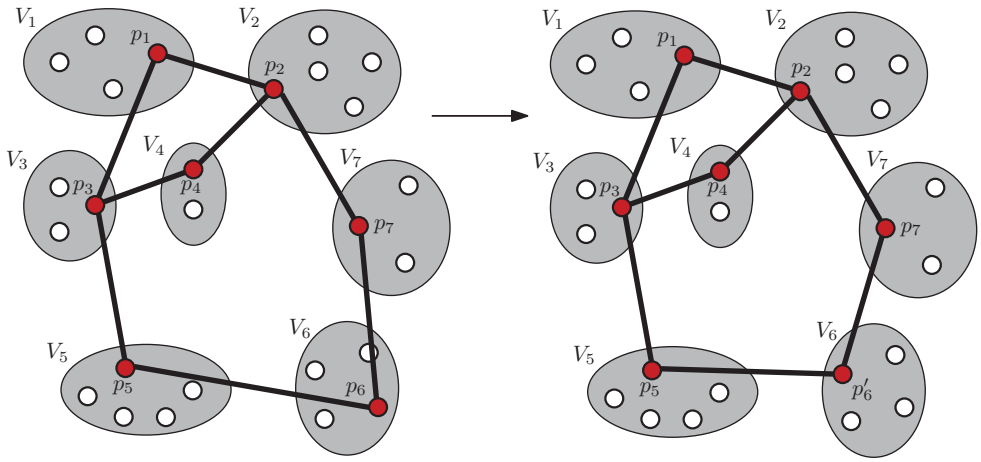


Figure 8.7: A SNON move, changing the spanned node of V_6 from p_6 to p'_6.

Algorithm 23: Simple Node Optimization (solution $S = \langle P, T \rangle$)

```
for i := 1, ..., r do
    forall v ∈ V_i \ p_i do
        change spanned node p_i 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
```

Node Optimization Neighborhood (NON)

This neighborhood structure enhances SNON by utilizing the graph reduction technique. NON consists of all solutions S' that differ from S by at most two spanned nodes within branching clusters. Again, the global structure of the solution remains unchanged. By means of the graph reduction technique, spanned nodes of path clusters are selected in an optimal way once the best neighboring solution is identified on the reduced graph, see Algorithm 24.

Carrying out graph reduction in advance adds $O(r \cdot d_{\max}^3)$ to the time complexity. Since updating the objective value for a considered neighbor can be done in $O(d_{\max})$ and $O(r^2)$ neighbors are to be considered, the overall time complexity of NON is bounded by $O(r^2 \cdot d_{\max}^2 + r \cdot d_{\max}^3)$.

Algorithm 24: Node Optimization (solution $S = \langle P, T \rangle$)

```
compute reduced structure S^g_red = ⟨V^g_red, T^g_red⟩
forall V_i, V_j ∈ V^g_red ∧ V_i ≠ V_j do
    forall u ∈ V_i ≠ p_i do
        change used node p_i of cluster V_i to u
        forall v ∈ V_j do
            change used node p_j of cluster V_j to v
            if current solution better than best then
                save current solution as best
            restore initial solution
restore best solution // fixes the spanned nodes in branching clusters
decode solution // by using precomputed shortest paths corresponding to used
edges in E_red
return solution
```

Node Re-Arrangement Neighborhood (NRAN)

With this neighborhood structure we try to optimize a solution with respect to the arrangement of nodes. A neighbor solution S' in NRAN differs from S by exactly one swap move which exchanges for two nodes a and b their sets of adjacent nodes I_a and I_b as shown in Figure 8.8. Set I_a, in respect to solution $S = \langle P, T \rangle$, is defined as $I_a = \{w \in P \mid (a, w) \in T\}$. After this swap move, $S' = \langle P, T' \rangle$ consists of $T' = T \setminus I_a \setminus I_b \cup \{(a, v) \mid v \in I_b\} \cup \{(b, u) \mid u \in I_a\}$. The pseudocode of completely searching this neighborhood is given in Algorithm 25.

Updating the objective value for a single move means to subtract the costs of the original edges and to add the costs of the new ones. Therefore, a complete evaluation of NRAN, which consists of all solutions S' differing from S by exactly one swap move, can be done in time $O(r^2 \cdot d_{\max})$.

Algorithm 25: Node Re-Arrangement Optimization (solution $S = \langle P, T \rangle$)

```
for i := 1, ..., r − 1 do
    for j := i + 1, ..., r do
        swap adjacency lists of nodes p_i and p_j
        if current solution better than best then
            save current solution as best
        restore initial solution
restore and return best solution
```

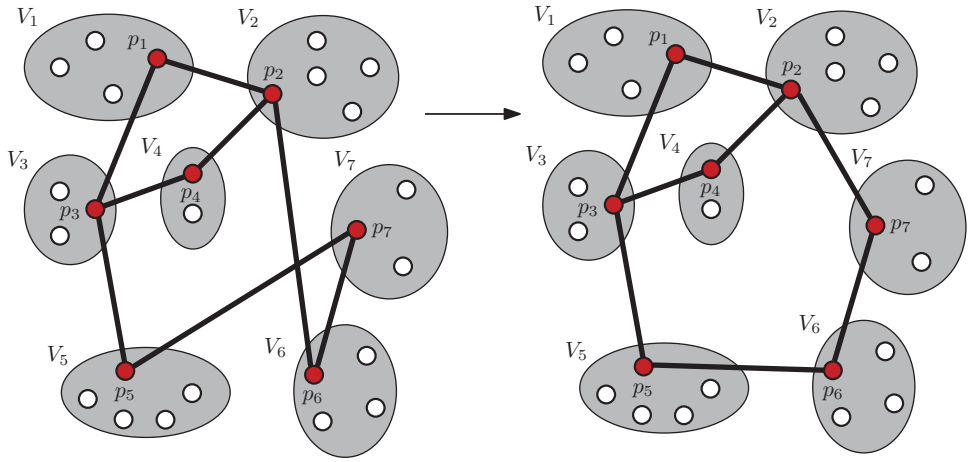


Figure 8.8: A NRAN move, swapping p_6 and p_7.

Cluster Re-Arrangement Neighborhood (CRAN)

This neighborhood structure is an extension to NRAN which again makes use of the graph reduction technique. Moving from the current solution S to a neighbor solution S' in CRAN means swapping two nodes in an analogous way as for NRAN, then computing the reduced graph, and finally determining the best nodes in all path clusters. Since applying the whole graph reduction after each move is relatively time-expensive, only incremental updates of the reduced structure and associated information are carried out whenever two nodes of path clusters are swapped, which is in practice most of the time the case. Whenever two nodes a and b of degree two on the same reduced path are swapped, only this path has to be updated; if a and b belong to different paths, only the corresponding two paths must be recomputed. However, if at least one of these nodes belongs to a branching cluster, the graph reduction procedure must be completely re-applied as the structure of the whole solution graph may change. The pseudocode is given in Algorithm 26.

The worst case time complexity of completely examining CRAN is $O(r^3 \cdot d_{\max}^3)$ when graph reduction is applied after every move. Since the complete evaluation might require too much time on larger instances, we abort the neighborhood exploration after a certain time limit is exceeded, returning the so-far best neighbor instead of following a strict best neighbor strategy.

Algorithm 26: Cluster Re-Arrangement Optimization (solution $S = \langle P, T \rangle$)

```
compute reduced structure S^g_red = ⟨V^g_red, T^g_red⟩
for i := 1, ..., r − 1 do
    for j := i + 1, ..., r do
        swap adjacency lists of nodes p_i and p_j
        if V_i or V_j is a branching cluster then
            recompute reduced solution S^g_red = ⟨V^g_red, T^g_red⟩
        else
            if V_i and V_j belong to the same reduced path P then
                update P in S^g_red
            else
                update the path containing V_i in S^g_red
                update the path containing V_j in S^g_red
        if current solution better than best then
            decode and save current solution as best
        restore initial solution and S^g_red
restore and return best solution
```

Edge Augmentation Neighborhood (EAN)

In this neighborhood structure, modifications on the edges are primarily considered. More precisely, EAN of a solution $S = \langle P, T \rangle$ consists of all solutions S' reachable from S by including a single additional edge $e \notin T$ and removing other, now redundant edges, see Figure 8.9 and Algorithm 27. Removing e itself is not allowed since this would obviously lead to the original solution S. We do not have to consider edges $e = (a, b)$ if $\deg(a) = \deg(b) = 2$ and a and b are part of the same reduced path. In these cases, adding e would lead to a graph where e is the only redundant edge.

As there are $O(r^2)$ possible moves and removing redundant edges has time complexity $O(r^3)$, the overall time complexity for evaluating EAN is bounded by $O(r^5)$. However, since good solutions are in practice usually rather sparse, we can omit the evaluation of many neighbor solutions.

Node Exchange Neighborhood (NEN)

This neighborhood structure addresses both aspects, changing the spanned nodes as well as the edges connecting them. A neighbor solution in NEN differs from the

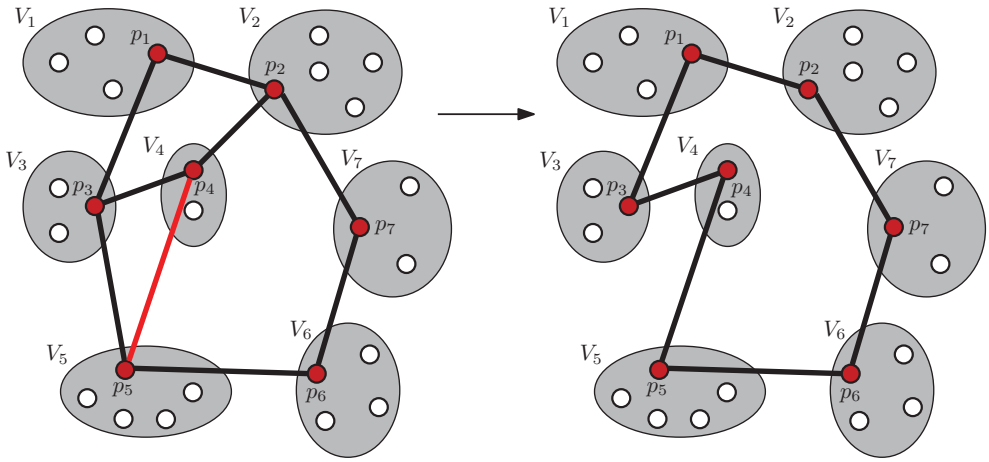
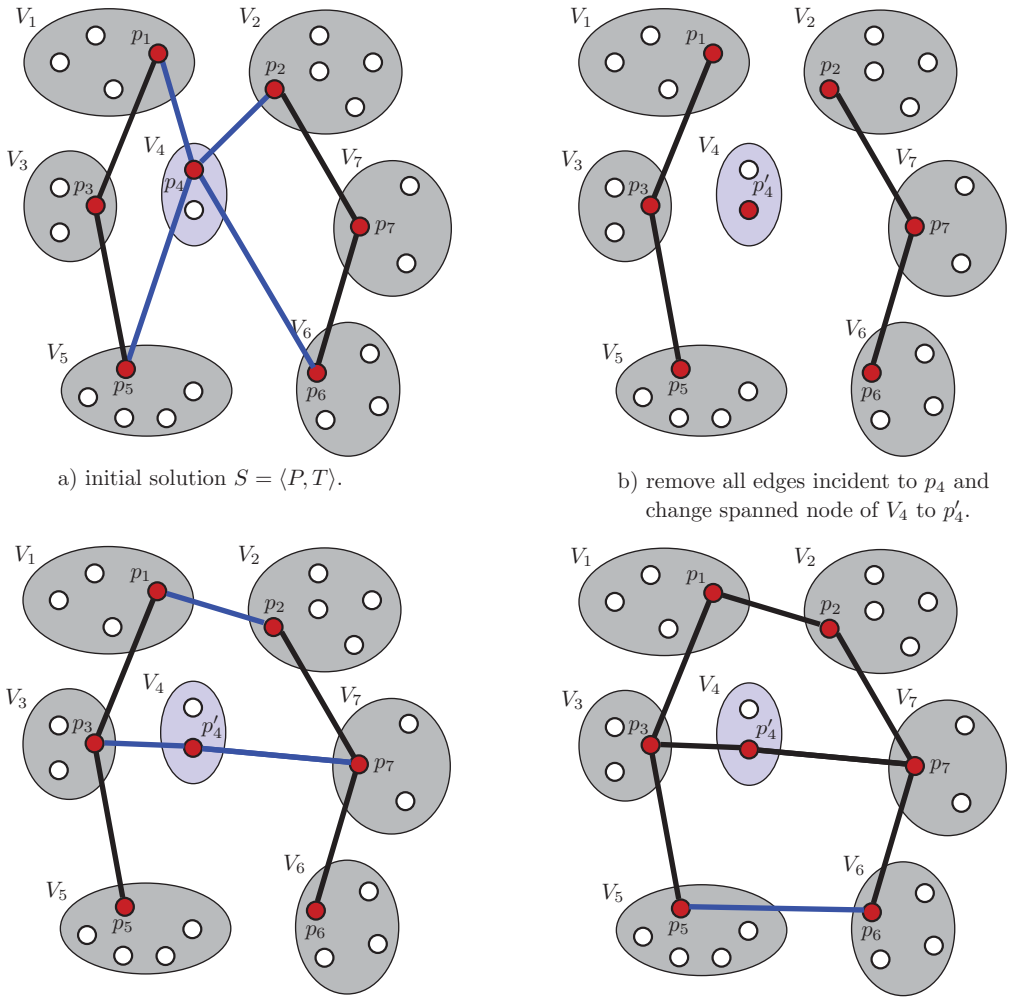


Figure 8.9: An EAN move, adding (p_4, p_5) and removing redundant edges (p_2, p_4) and (p_3, p_5).

Algorithm 27: Edge Augmentation Optimization (solution $S = \langle P, T \rangle$)

```
for i := 1, ..., r − 1 do
    for j := i + 1, ..., r do
        if (i, j) ∉ T then
            if
            deg(i) ≠ 2 ∨ deg(j) ≠ 2 ∨ i and j are not part of the same reduced path
            then
                add (i, j)
                remove redundant edges
                if current solution better than best then
                    save current solution as best
                restore initial solution
restore and return best solution
```

original solution by exactly one spanned node and an arbitrary number of edges. A single move within NEN is accomplished by first changing $p_i \in V_i$ to $p'_i \in V_i, p_i \neq p'_i$, and removing all edges incident to p_i. This leads to a graph consisting of at least two and no more than $\deg(p_i) + 1$ components. We reconnect these parts by adding the cheapest edges between any pair of these components. Once this step is completed, edge biconnectivity is restored using the advanced bridge covering strategy described below. Finally, redundant edges are removed, see Figure 8.10 and Algorithm 28.



a) initial solution $S = \langle P, T \rangle$.

b) remove all edges incident to p_4 and change spanned node of V_4 to p'_4.

c) add cheapest edges between all pairs of graph components.

d) restore biconnectivity.

Figure 8.10: A NEN move, changing the spanned node of V_4, removing all adjacent edges, and re-augmenting the graph.

The process of covering all bridges with additional edges can be expensive in practice. When disconnecting a node in a sparse graph, many bridges may arise. Therefore, we first determine all nodes with degree one and connect each of them with its cheapest partner. If only a single node with degree one exists, we connect it with the first reachable node of degree greater than two. This strategy helps to cover many bridges with only few edges. Remaining bridges are covered by simply adding the cheapest

Algorithm 28: Node Exchange Optimization (solution $S = \langle P, T \rangle$)

```
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
        remove redundant edges
        if current solution better than best then
            save current solution as best
        restore initial solution
restore and return best solution
```

edges between pairs of edge biconnected components. Even with this advanced bridge covering strategy, examining NEN still needs $O(|V| \cdot r^3)$ time. Therefore, analogous to CRAN, we stop the search of NEN after a time limit is exceeded and return the so-far best neighbor solution.

8.3.4 Variable Neighborhood Search Framework

We use the traditional general VNS scheme with different variants of VND as local improvements. In order to be able to investigate in particular the efficiency of the more complicated neighborhoods based on graph reduction (NON, CRAN), two standard VNDs which use different sets of neighborhoods are considered. Furthermore, we examine the impact of using the more sophisticated *Self-Adaptive Variable Neighborhood Descent* (SAVND) with dynamic neighborhood-ordering [58], see Section 3.4.2.

The first VND variant, VND1, is shown in Algorithm 29; it only applies the simpler neighborhood structures without graph reduction, i.e. SNON, NRAN, EAN, and NEN. This ordering has been determined taking both the computational complexity as well as preliminary test results into account.

The second VND variant, VND2, is shown in Algorithm 30 and alternates between NON, NRAN, CRAN, EAN, and NEN. It therefore also uses the more sophisticated neighborhoods having higher computational complexity due to the applied graph reduction. SNON is not considered since it is fully contained in NON and preliminary experiments with both of them did not indicate advantages.

Algorithm 29: VND1 (solution $S = \langle P, T \rangle$)

```
l := 1
repeat
    switch l do
        case 1: // SNON, see Algorithm 23
            S' := Simple Node Optimization (S)
        case 2: // NRAN, see Algorithm 25
            S' := Node Re-Arrangement Optimization (S)
        case 3: // EAN, see Algorithm 27
            S' := Edge Augmentation Optimization (S)
        case 4: // NEN, see Algorithm 28
            S' := Node Exchange Optimization (S)
    if solution S' is better than S then
        S := S'
        l := 1
    else
        l := l + 1
until l > 4
return solution S
```

Finally, the self-adaptive variable neighborhood descent (SAVND) uses the same neighborhood structures as VND2, but instead of a static order, the neighborhoods are rearranged automatically during the search process. Each neighborhood structure has associated a rating which is updated according to success probabilities and required times for evaluation. In this way, more effective neighborhood structures come to the fore and will be applied more frequently.

Shaking

Most of our neighborhood structures used in VND concentrate more on the optimization of the spanned nodes than on the global structure. In order to enhance diversity, our shaking procedure is therefore based on EAN. It augments a current solution by k randomly chosen new edges followed by a removal of other, now redundant edges. This process starts with $k = 1$ inserted edge, and as long as no improvement is achieved, k is incremented by one up to $k_{\max} = \lfloor \frac{r}{4} \rfloor$. In accordance to the used VND variant, we denote the three VNS variants VNS1, VNS2, and SAVNS, respectively.

Algorithm 30: VND2 (solution $S = \langle P, T \rangle$)

```
l := 1
repeat
    switch l do
        case 1: // NON, see Algorithm 24
            S' := Node Optimization (S)
        case 2: // NRAN, see Algorithm 25
            S' := Node Re-Arrangement Optimization (S)
        case 3: // CRAN, see Algorithm 26
            S' := Cluster Re-Arrangement Optimization (S)
        case 4: // EAN, see Algorithm 27
            S' := Edge Augmentation Optimization (S)
        case 5: // NEN, see Algorithm 28
            S' := Node Exchange Optimization (S)
    if solution S' is better than S then
        S := S'
        l := 1
    else
        l := l + 1
until l > 5
return solution S
```

8.4 A Mixed Integer Programming Formulation for GMEBCNP

To obtain proven optimal solutions for small and medium sized GMEBCNP instances, we propose a multi-commodity flow MIP formulation based on the local-global approach which was originally suggested for the GMSTP [91]. We use following decision variables.

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

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

$$y_{i,j} = \begin{cases} 1 & \text{if cluster } V_i \text{ is connected to cluster } V_j \\ & \text{in the global structure} \\ 0 & \text{otherwise} \end{cases} \qquad \forall (i,j) \in E^{\mathrm{g}}$$

$$f_{i,j}^{k} = \begin{cases} 1 & \text{if a flow } f \text{ of commodity } k \text{ exists} \\ & \text{from cluster } i \text{ to cluster } j \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \forall i,j = 1,\ldots,r \\ \forall k = 2,\ldots,r \end{array}$$

The MIP formulation consists of two parts: The multi commodity flow part operates on the global structure and is based on sending from cluster V_1, which is defined to be the root, two units of flow f to every other cluster using edge-disjoint routes. Flows dedicated to different clusters are distinguished by their commodity k. The result is stored in the binary variables $y_{i,j}$ indicating the global connections. The local-global part, originally introduced by Pop [91] for the GMSTP, relates the local variables $x_{u,v}$ and z_v with the global connections.

$$\text{minimize} \sum_{(u,v)\in E} c_{u,v}\, x_{u,v} \tag{8.1}$$

$$\text{subject to} \tag{8.2}$$

$$\sum_{i=1}^{r} f_{i,j}^{k} - \sum_{l=1}^{r} f_{j,l}^{k} = \begin{cases} -2 & \text{if } j = 1 \\ 2 & \text{if } j = k \\ 0 & \text{else} \end{cases} \qquad \forall j = 1,\ldots,r,\ \forall k = 2,\ldots,r \tag{8.3}$$

$$f_{i,j}^{k} + f_{j,i}^{k} \leq 1 \qquad \forall i,j = 1,\ldots,r,\ \forall k = 2,\ldots,r \tag{8.4}$$

$$\sum_{v\in V_k} z_v = 1 \qquad \forall k = 1,\ldots,r \tag{8.5}$$

$$\sum_{u\in V_i, v\in V_j} x_{u,v} = y_{i,j} \qquad \forall (i,j) \in E^{\mathrm{g}} \tag{8.6}$$

$$x_{u,v} \leq z_u \qquad \forall i = 1,\ldots,r,\ ,\forall u \in V_i,\ \forall v \in V \setminus V_i \tag{8.7}$$

$$y_{i,j} \geq f_{i,j}^{k} \qquad \forall i,j = 1,\ldots,r,\ i \neq j,\ \forall k = 2,\ldots,r \tag{8.8}$$

$$f_{i,j}^{k} \geq 0 \qquad \forall i,j = 1,\ldots,r,\ \forall k = 2,\ldots,r \tag{8.9}$$

$$x_{u,v} \in \{0,1\} \qquad \forall (u,v) \in E \tag{8.10}$$

$$y_{i,j} \in \{0,1\} \qquad \forall (i,j) \in E^{\mathrm{g}} \tag{8.11}$$

$$z_v \in \{0,1\} \qquad \forall v \in V \tag{8.12}$$

Constraints (8.2) ensure that two commodities k of flow f are produced in V_1, preserved by every cluster they are not dedicated for, and consumed by cluster V_k. To achieve edge biconnectivity, inequalities (8.4) forbid the transportation of two commodities dedicated for the same cluster over the same connection. To obtain a valid global structure, inequalities (8.8) ensure global connections to be included in the solution if a flow variable is active on it. Constraints (8.5) guarantee that precisely one node is selected per cluster and equations (8.6) only allow edges between nodes of clusters which are connected in the global structure. Finally, inequalities (8.7) ensure that only edges incident to selected nodes are chosen.

8.5 Test Instances

We tested our algorithms on the same instances which are used for the GMSTP and GTSP: Euclidean TSPlib instances with geographical center clustering, grouped Euclidean, random Euclidean, and non-Euclidean instance sets, see Section 5.3.

For the MIP formulation, we had to generate smaller instances in order to solve them optimaly. We derived them by reducing some original benchmark instances. Their properties are listed in Table 8.1.

Table 8.1: Small instances for comparison with the MIP approach.

Instance	$\|V\|$	$\|E\|$	r	$\frac{\|V\|}{r}$	col	row	sep	$span$
Group E. 40	40	780	8	5	4	2	10	10
Group E. 50	50	1225	10	5	5	2	10	10
Group E. 60	60	1770	12	5	6	2	10	10
Random E. 40	40	780	8	5	-	-	-	-
Random E. 50	50	1225	10	5	-	-	-	-
Random E. 60	60	1770	12	5	-	-	-	-
Non-E. 40	40	780	8	5	-	-	-	-
Non-E. 50	50	1225	10	5	-	-	-	-
Non-E. 60	60	1770	12	5	-	-	-	-

8.6 Computational Results

All experiments have been performed on a Pentium 4, 2.6 GHz PC with 1GB RAM. To test the performance of the MIP formulation, we used the general purpose MIP

solver CPLEX in version 10.0.1. In order to compute average values and standard deviations for the VNS, we performed for each algorithm variant and each instance 30 independent runs.

To allow a fair comparison among the VNS variants, we used CPU-time in dependence of the instance type and size as termination criterion. For the two complex neighborhood structures CRAN and NEN, we set the time limit for each evaluation to 5s; i.e., after 5s, if not all neighbors of the current solution could be evaluated, VND continues with the so far-best neighbor.

8.6.1 Results on Small Instances

We first consider the small instances derived in particular for testing the MIP approach. Table 8.2 shows the corresponding results. For the MIP approach, obtained optimal solution values $C(T^*)$ and the required CPU times for identifying and proving them are listed. For the three VNS variants VNS1 (including only simpler neighborhoods), VNS2 (including neighborhoods based on graph reduction), and SAVNS (VNS2 with self-adaptive ordering of neighborhoods), CPU-time was limited to one second per run. Obtained average solution values $\overline{C(T)}$, corresponding standard deviations, and success rates of how often the optimal solution was found are listed.

Table 8.2: Results on small instances. Time limit for VNS approaches is 1s per run.

	MIP		VNS1		VNS2		SAVNS	
Instance, $\|V\|$	$C(T^*)$	time	$\overline{C(T)}$	Opt.	$\overline{C(T)}$	Opt.	$\overline{C(T)}$	Opt.
Group Eucl, 40	79.4	4.6s	82.9	30/30	82.9	30/30	82.9	30/30
Group Eucl, 50	82.1	31.2s	88.2	29/30	88.1	30/30	88.1	30/30
Group Eucl, 60	91.8	539.8s	92.2	24/30	92.1	26/30	91.8	30/30
Random Eucl, 40	989.0	6.3s	996.2	28/30	992.9	29/30	989.0	30/30
Random Eucl, 50	1310.2	762.0s	1317.7	25/30	1320.8	23/30	1314.0	27/30
Random Eucl, 60	1318.0	3370.6s	1329.9	21/30	1321.6	26/30	1323.7	29/30
Non-Eucl, 40	152.8	3.8s	371.8	27/30	369.0	30/30	369.0	30/30
Non-Eucl, 50	216.2	16.1s	462.4	23/30	452.5	27/30	454.4	26/30
Non-Eucl, 60	305.3	276.9s	317.1	21/30	312.3	25/30	310.2	25/30

We can observe that the MIP approach was able so solve all instances with up to 60 nodes to proven optimality. The required CPU times are, however, relatively large and substantially increase with the number of nodes, especially for random

Euclidean instances. For the three VNS variants, these small instances turned out to be no real challenge: For most of the runs, optimal solutions could be found in fractions of a second. In particular on the group Euclidean instances, SAVNS consistently identified the optimal solution in every single run. VNS1 performed worse than VNS2 and SAVNS due to the absence of the more complex neighborhood structures.

8.6.2 Results on Larger Instances

We now turn to the larger TSPlib instances and random instances from Ghosh. Tables 8.3 and 8.4 show the results of the three VNS variants. For TSPlib instances, the allowed CPU time has been chosen roughly in dependence on the instance size as indicated in Table 8.3 (between 150s and 600s), while for random instances, each run has been terminated after 600s. For smaller random instances, shorter time limit would be enough to let VNS fully converge. However, we finally decided to use the same time limit for all these instances. As a result, final best solutions might be found rather early for small instances whereas they are obtained quite at the end of a run for large instances. The search process takes up all the time nevertheless. We list the objective values of the best solutions found during the 30 runs $C(T_{\text{best}})$, the average values $\overline{C(T)}$, and the standard deviations.

For TSPlib instances (Table 8.3), we can observe a consistent and obvious trend: Among all VNS variants, SAVNS performs best. Columns $\gamma_{A,B}$ list error probabilities of Wilcoxon tests [119] for the assumption that the difference of the solutions obtained by approach A and B are significant. In most cases, these tests yield error probabilities of less than 1% for the assumption that the mean objective values from VNS2 are superior than those of VNS1. These error probabilities become even smaller when we compare VNS1 with SAVNS.

Results are more ambiguous for the random instances in Table 8.4. Though SAVNS is still the best strategy on group Euclidean and non-Euclidean instances, its performance is significantly worse than those of VNS2 on random Euclidean instances. By analyzing the log files, we suspect that this is due to CRAN, which is very efficient on this type of instances, but also rather time consuming. In the test runs, this neighborhood structure is moved to the front and therefore examined very often. This slowed down the whole search process. Nevertheless, in almost all cases the variants utilizing the more sophisticated neighborhood structures based on graph reduction outperformed the simpler VNS1.

Table 8.3: Results on TSPlib instances with geographical clustering, variable CPU-time.

TSPlib Instances					VNS1 (I)			VNS2 (II)			SAVNS (A)			Wilcoxon tests		
Name	$\|V\|$	r	$\frac{\|V\|}{r}$	time	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$\gamma_{\text{I,II}}$	$\gamma_{\text{II,A}}$	$\gamma_{\text{I,A}}$
gr137	137	28	5	150s	440.0	440.8	2.42	440.0	**440.0**	0.00	440.0	**440.0**	0.00	0.01	NA	0.01
kroa150	150	30	5	150s	11532.0	11532.8	2.44	11532.0	11533.4	3.97	11532.0	**11532.0**	0.00	0.67	0.04	0.08
krob200	200	40	5	300s	13177.0	13309.4	79.90	13177.0	**13201.2**	22.19	13177.0	13206.4	23.79	<0.01	0.27	<0.01
ts225	225	45	5	300s	68346.0	68769.7	305.85	68346.0	68658.1	212.15	68346.0	**68563.7**	166.33	0.21	0.13	<0.01
gil262	262	53	5	300s	1078.0	1157.1	54.88	1059.0	1076.3	15.88	1059.0	**1070.8**	10.05	<0.01	0.53	<0.01
pr264	264	54	5	300s	29948.0	31639.6	1449.32	29810.0	30434.8	725.48	29810.0	**29879.3**	55.95	<0.01	<0.01	<0.01
pr299	299	60	5	450s	22853.0	23953.9	792.38	22644.0	22829.6	215.92	22644.0	**22659.1**	12.73	<0.01	0.01	<0.01
lin318	318	64	5	450s	21506.0	23101.1	840.31	21028.0	21750.4	444.44	20795.0	**21321.0**	228.67	<0.01	<0.01	<0.01
rd400	400	80	5	600s	6846.0	7275.4	237.96	6755.0	6961.7	103.21	6745.0	**6833.2**	41.99	<0.01	<0.01	<0.01
fl417	417	84	5	600s	10234.0	10636.4	286.75	10085.0	10547.9	234.15	9708.0	**9881.9**	160.77	0.10	<0.01	<0.01
gr431	431	87	5	600s	1337.0	1408.2	50.76	1303.0	1346.2	27.04	1284.0	**1312.3**	18.01	<0.01	<0.01	<0.01
pr439	439	88	5	600s	65830.0	72752.7	3857.22	60972.0	67727.8	3302.30	60642.0	**62276.9**	1710.68	<0.01	<0.01	<0.01
pcb442	442	89	5	600s	25545.0	26051.2	197.20	22439.0	23882.8	946.98	22148.0	**22612.6**	325.94	<0.01	<0.01	<0.01

Table 8.4: Results on random instance sets, three different instances are considered for each set.

TSPlib Instances				VNS1 (I)			VNS2 (II)			SAVNS (A)			Wilcoxon tests		
Name	$\|V\|$	r	$\frac{\|V\|}{r}$	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$C(T_{\text{best}})$	$\overline{C(T)}$	std dev	$\gamma_{\text{I,II}}$	$\gamma_{\text{II,A}}$	$\gamma_{\text{I,A}}$
	125	25	5	159.5	**159.5**	0.00	159.5	**159.5**	0.00	159.5	**159.5**	0.00	NA	NA	NA
Group E. 125	125	25	5	163.5	**163.5**	0.00	163.5	**163.5**	0.00	163.5	**163.5**	0.00	NA	NA	NA
	125	25	5	166.1	**166.1**	0.00	166.1	**166.1**	0.00	166.1	**166.1**	0.00	NA	NA	NA
	500	100	5	684.4	717.6	25.78	640.4	678.5	17.80	629.8	**644.8**	9.12	<0.01	<0.01	<0.01
Group E. 500	500	100	5	765.1	799.0	35.40	657.4	698.9	24.24	645.3	**663.8**	11.94	<0.01	<0.01	<0.01
	500	100	5	747.3	761.7	37.21	651.7	693.7	25.41	643.0	**655.6**	8.94	<0.01	<0.01	<0.01
	600	20	30	105.1	**105.1**	0.00	105.1	105.5	1.90	105.1	**105.1**	0.00	0.33	0.33	NA
Group E. 600	600	20	30	105.2	**105.2**	0.00	105.2	105.9	2.67	105.2	**105.2**	0.00	0.01	0.01	NA
	600	20	30	107.5	**107.5**	0.00	107.5	**107.5**	0.00	107.5	**107.5**	0.00	NA	NA	NA
	1280	64	8	399.0	436.8	36.47	364.6	385.7	11.01	342.5	**358.2**	11.54	<0.01	<0.01	<0.01
Group E. 1280	1280	64	8	376.2	404.6	24.26	354.5	390.8	14.97	345.5	**351.4**	6.33	0.05	<0.01	<0.01
	1280	64	8	379.4	433.3	35.75	377.5	406.7	17.49	357.9	**371.4**	8.19	<0.01	<0.01	<0.01
	250	50	5	3571.1	3746.6	123.22	2995.1	**3226.6**	116.9	3235.2	3449.3	145.72	<0.01	<0.01	<0.01
Random E. 250	250	50	5	3309.5	3661.8	166.90	2662.1	**2968.7**	189.44	2750.2	3299.3	161.37	<0.01	<0.01	<0.01
	250	50	5	3051.9	3403.2	270.07	2698.6	**2888.1**	98.41	2793.2	3111.9	254.96	<0.01	<0.01	<0.01
	400	20	20	943.8	1027.2	71.50	841.0	**926.9**	61.18	943.8	1028.4	48.68	<0.01	<0.01	<0.01
Random E. 400	400	20	20	813.3	1059.5	138.33	813.3	**875.7**	56.80	813.3	1043.2	109.04	<0.01	<0.01	0.40
	400	20	20	814.9	858.7	40.51	794.4	**836.1**	39.80	814.9	860.7	27.75	<0.01	<0.01	0.91
	600	20	30	599.8	725.1	103.93	599.8	**605.1**	7.54	599.8	699.2	90.40	<0.01	<0.01	0.05
Random E. 600	600	20	30	785.4	823.7	55.31	643.1	**762.0**	50.59	643.1	813.4	53.89	<0.01	<0.01	0.12
	600	20	30	695.3	778.9	60.08	596.5	**674.3**	52.30	695.3	743.7	94.24	<0.01	<0.01	<0.01
	200	20	10	180.1	244.9	34.41	172.6	224.5	23.40	179.8	**219.0**	31.03	<0.01	0.12	<0.01
Non-E. 200	200	20	10	133.8	216.7	33.83	140.2	**198.3**	31.08	179.3	215.9	19.04	0.18	0.02	0.39
	200	20	10	156.0	179.8	28.38	135.5	190.6	28.96	154.4	**175.6**	20.07	0.66	0.01	0.11
	500	100	5	902.6	1121.3	123.29	830.5	996.7	88.49	771.0	**960.2**	99.32	<0.01	0.09	<0.01
Non-E. 500	500	100	5	897.5	1008.2	128.91	860.7	1016.4	87.24	734.3	**921.2**	106.89	0.05	<0.01	0.17
	500	100	5	785.3	1025.8	109.49	844.0	990.8	82.99	710.4	**929.0**	120.78	0.09	0.01	<0.01
	600	20	30	102.9	138.6	25.11	93.0	129.9	14.19	91.6	**125.0**	20.65	0.36	0.05	0.02
Non-E. 600	600	20	30	107.3	132.0	18.98	102.3	130.2	13.62	94.6	**118.5**	19.94	0.48	<0.01	<0.01
	600	20	30	85.4	119.4	18.98	94.8	133.1	17.79	79.5	**113.4**	18.5	0.03	<0.01	0.27

8.6.3 Contributions of Neighborhood Structures

In order to analyze the individual contributions of the different neighborhood structures to the whole success, we logged how often each neighborhood structure was able to improve a current solution. We then determined the ratios of successful improvements over how often each neighborhood structure has been evaluated and normalized these values over all neighborhood structures, yielding percentage values describing their relative efficiencies. For VNS2 and all considered large test instances, these efficiencies are listed in Tables 8.5 and 8.6. For SAVNS, these values are similar since the same neighborhood structures are used.

In general, each neighborhood structure contributes substantially to the whole success. In Table 8.5, we observe that by increasing the size of the instances, the more complex neighborhood structures like EAN and NEN become more efficient while the improvement rates of the simpler ones decrease. In Table 8.6, we combined the data for each set of three identically parameterized instances for simplicity. The dependency on the instance size is less obvious, but the neighborhoods' relative efficiencies strongly depend on the structure of the instance. In particular, NRAN is less successful on non-Euclidean instances but second best for random Euclidean instances. All in all, CRAN is able to improve solutions most often. NON performs best on instances with many nodes per cluster, while EAN performs best on instances containing many relatively small clusters.

8.6.4 Impact of Self-Adaptive Variable Neighborhood Descent

Since SAVNS performs best in overall, we want to investigate the impact of SAVND on the optimization process. We expect the substantial difference to be in SAVND because the same VNS framework is used all the time.

Table 8.7 shows the results when VND2 and SAVND are applied after the initialization procedure ACH. We compare the average objective values obj of final solutions identified by those two VND variants and the average total CPU-times t needed to get there. Since both variants use the same set of neighborhood structures, the only difference is their order. Because some of the neighborhood evaluation strategies contain stochastic components, average objective values and average times over 30 runs are provided.

Looking at the final objective values identified by VND2 and SAVND, we observe that differences exist. The reason obvious; due to the different order in which the neighborhood structures are searched, different local optima are reached. However,

Table 8.5: Relative improvement rates of NON, NRAN, CRAN, EAN, and NEN for TSPlib instances.

Instance	$\|V\|$	r	$\frac{\|V\|}{r}$	NON	NRAN	CRAN	EAN	NEN
gr137	137	28	5	22.37%	20.50%	25.43%	22.29%	9.40%
kroa150	150	30	5	22.24%	17.75%	25.70%	21.94%	12.36%
krob200	200	40	5	17.87%	18.17%	23.97%	25.94%	14.05%
ts225	225	45	5	16.35%	19.82%	21.32%	25.68%	16.83%
gil262	262	53	5	15.03%	17.34%	21.56%	27.81%	18.26%
pr264	264	54	5	14.43%	20.27%	23.00%	26.49%	15.80%
pr299	299	60	5	15.07%	17.72%	21.75%	27.39%	18.08%
lin318	318	64	5	15.02%	18.42%	21.00%	27.62%	17.94%
rd400	400	80	5	13.92%	14.66%	18.40%	27.38%	25.64%
fl417	417	84	5	12.69%	21.43%	17.39%	29.69%	18.80%
gr431	431	87	5	11.39%	17.52%	19.79%	30.85%	20.45%
pr439	439	88	5	14.75%	16.52%	20.81%	27.54%	20.38%
pcb442	442	89	5	13.61%	15.21%	20.73%	28.02%	22.42%

Table 8.6: Relative improvement rates of NON, NRAN, CRAN, EAN, and NEN for random instances.

Instance	$\|V\|$	r	$\frac{\|V\|}{r}$	NON	NRAN	CRAN	EAN	NEN
Group E. 125	125	25	5	33.79%	13.56%	32.65%	12.43%	7.56%
Group E. 500	500	100	5	22.84%	12.92%	24.83%	20.36%	19.03%
Group E. 600	600	20	30	30.68%	6.40%	27.52%	4.75%	30.66%
Group E. 1280	1280	64	20	24.95%	10.92%	22.97%	14.09%	27.07%
Random E. 250	250	50	5	13.90%	24.94%	28.08%	22.04%	11.04%
Random E. 400	400	20	20	26.38%	24.94%	33.44%	8.11%	7.12%
Random E. 600	600	20	30	25.23%	27.24%	33.37%	8.13%	6.04%
Non-E. 200	200	20	10	31.76%	3.43%	35.99%	15.17%	13.65%
Non-E. 500	500	100	5	13.55%	5.10%	23.22%	33.95%	24.16%
Non-E. 600	600	20	30	35.83%	1.53%	45.57%	7.91%	9.16%

the difference is relatively small, and over all instances, no strategy yields statistically significantly better solutions than the other.

Comparing running times, the advantage of SAVND becomes very clear: It consistently requires significantly less time to get to these local optima. When used for

Table 8.7: Results on random instances, three instances per category.

Random Instances			VND2		SAVND	
Category	$\|V\|$	r	$\overline{obj}$	$\bar{t}$ [s]	$\overline{obj}$	$\bar{t}$ [s]
Grouped E. 125	125	25	178.36	5.90	180.65	3.31
Grouped E. 500	500	100	766.11	161.41	762.85	95.69
Grouped E. 600	600	20	115.49	63.34	116.26	48.10
Grouped E. 1280	1280	64	467.70	221.28	476.15	184.61
Random E. 250	250	50	4143.14	54.28	4049.57	30.67
Random E. 400	400	20	1132.70	51.51	1211.81	38.73
Random E. 600	600	20	948.83	107.85	1088.40	82.44
Non-E. 200	200	20	486.11	6.68	492.44	3.83
Non-E. 400	500	100	1141.04	66.96	1175.01	31.63
Non-E. 600	600	20	266.03	34.63	253.47	25.96

Table 8.8: Results on TSPlib instances with geographical clustering.

TSPlib Instances			VND2		SAVND	
Name	$\|V\|$	r	$\overline{obj}$	$\bar{t}$ [s]	$\overline{obj}$	$\bar{t}$ [s]
gr137	137	28	505.60	10.33	490.23	5.30
kroa150	150	30	12470.77	9.32	12562.13	6.36
d198	198	40	12330.90	37.7	12435.03	17.55
krob200	200	40	13906.47	29.49	14010.97	14.43
gr202	202	41	344.53	30.84	341.13	22.67
ts225	225	45	77418.33	18.63	77691.67	13.77
gil262	262	53	1186.63	66.71	1173.23	21.30
pr264	264	54	34691.27	70.87	35506.70	28.90
pr299	299	60	24887.30	46.73	24839.63	32.66
lin318	318	64	26285.40	41.47	26535.23	22.64
rd400	400	80	7891.80	92.65	7532.13	53.47
fl417	417	84	11042.77	89.44	11003.57	53.29
gr431	431	87	1557.83	78.95	1520.57	72.27
pr439	439	88	77427.03	116.65	78338.93	66.65
pcb442	442	89	26669.00	83.05	26881.70	43.79

VNS, this advantage is even more noticeable since more iterations can be carried out in the same timeframe.

8.7 Conclusions

In this chapter, we proposed Variable Neighborhood Search approaches as well as a Mixed Integer Programming (MIP) model for the Generalized Minimum Edge Biconnected Network Problem (GMEBCNP).

The VNS algorithms are based on four neighborhood structures addressing particular properties as spanned nodes and/or edges between them. For two of them there exist simple and more advanced variants. The latter utilize a graph reduction technique which allows to efficiently determine the optimal spanned nodes for the majority of cluster once the connections between clusters are fixed. We apply techniques to optimize the search process such as methods to omit meaningless computations for the more complex neighborhood structures and optimized evaluation strategies.

Experiments were performed on TSPlib based instances with geographical clustering, Euclidean instances with grid and random clustering, and non-Euclidean instances. The proposed multi commodity flow MIP approach is able to solve smaller GMEBCNP instances with up to 60 nodes in reasonable time to proven optimality. In comparison, the VNS variants are in most of their runs also able to identify optimal solutions for those small instances, but in substantially shorter time (fractions of a second).

Comparing the results of our VNS variants for medium and large instances, we conclude that the used neighborhood structures are effective and their combination within the VNS scales well to large instances. In particular, we observed that the graph reduction technique applied in the more sophisticated neighborhood structures of VNS2 and SAVNS is a major improvement. The self-adaptive ordering of neighborhoods, as it is done in SAVNS turned out to be helpful in the majority of the experiments.

8.8 Future Work

In the future, we want to consider further large neighborhood structures that are evaluated by means of integer linear programming. The described multi commodity flow formulation provides a basis for this. Another plan is to extend the graph reduction technique by including further shrinking strategies.

Since all of the instances we used for testing were complete graphs, we did not implement special mechanisms for handling incomplete graphs. Though it is easy to remove all unneeded variables from the MIP model, it is less straightforward to adapt the neighborhood structures for the VNS adequately. We want to investigate

possibilities like narrowing the search space or repairing solutions due to non-existing edges in the future.

Last but not least, there also exists the variant of the GMEBCNP in which *at least* one node of each cluster must be connected. Though related, many techniques in this chapter are not directly applicable to this variant, e.g. the graph reduction technique. We would like to investigate this problem in the future and adapt our proposed neighborhood structures and techniques for it.

Chapter 9

The Railway Traveling Salesman Problem

9.1 Introduction

The Railway Traveling Salesman Problem (RTSP) is defined as follows. We are given a number of cities and a timetable specifying train connections between these cities. A salesman starts his journey in a particular station, has to visit a given subset of cities denoted by B, and finally has to return to the initial location. In each of the cities $\sigma \in B$, he has to spend a minimum amount of time t_σ to complete his errands. The goal is to minimize the overall time required for the journey.

In contrast to the classical Traveling Salesman Problem (TSP), it is allowed to visit cities or railway tracks more than once. This is due to practical reasons since it makes no sense to limit the usage of some backbone stations and to enforce the salesman to take alternative, possibly slower train connections.

The RTSP has been introduced by Hadjicharalambous et al. [46] and they showed that it is $\mathcal{NP}$-hard. Furthermore, they modeled the RTSP via a so-called time-expanded digraph and provided a multi-commodity flow integer linear programming formulation to solve the problem to optimality. To increase the performance, they

Parts of this chapter appeared in [60]

introduced a reduction algorithm which decreases the size of the graph significantly. To handle larger instances, Pop et al. [92] presented an ant colony optimization algorithm to solve the problem heuristically.

The RTSP is related to the Generalized TSP (GTSP), see Chapter 7. One of the most obvious differences is that RTSP does not require all cities to be visited. A transformation to GTSP seems reasonable, since there are many successful algorithms for solving GTSP, such as a branch-and-cut [34], genetic algorithms [111, 110], other (meta)heuristics [104, 59], or even approaches that exploit transformations of the GTSP into a TSP [20, 6]. Pop et al. [94] presented a transformation to a specialized GTSP, the so-called 2-GTSP, and proposed a cutting plane approach to solve it. The 2-GTSP differs from the classical GTSP by requiring exactly two nodes in each cluster to be visited in a roundtrip.

In this chapter which is based on [60], we present two schemes to transform the RTSP into classical TSPs. Section 9.2 shows how this problem can be modelled as a graph theoretical problem and how the model can be shrinked in size via preprocessing. Section 9.3 describes a transformation of RTSP into an asymmetric TSP, while Section 9.4 proposes a transformation into a symmetric TSP. We show computational results on the performance of the symmetric transformation in Section 9.5 and finally conclude in Section 9.6.

9.2 Modelling

The Time-expanded Digraph

Hadjicharalambous et al. [46] proposed to model this problem via a time-expanded digraph [107]. This graph $G = \langle V, E \rangle$, indicated by Figure 9.1, is defined as follows. Vertex set V is partitioned into $\sigma_1, \ldots, \sigma_m$ clusters, representing the train stations. The railway timetable contains a list of train entries in terms of 5-tuples $T_i = (z, \sigma^{\mathrm{d}}, \sigma^{\mathrm{a}}, t(d), t(a))$, each representing the corresponding train ID z, departure station σ^{d}, arrival station σ^{a}, departure time $t(d)$ and arrival time $t(a)$. For each 5-tuple, a departure-node d with time $t(d)$ is added to σ^{d} and a arrival-node a with time $t(a)$ is added to σ^{a}. Departure and arrival times are integers in the interval $[0, 1439]$ representing time in minutes after midnight.

All departure-nodes of a station are ordered according to their times. Let $d_1, \ldots, d_l$ be the departure-nodes of σ in that order, then there are stay edges $(d_1, d_2), \ldots, (d_{l-1}, d_l), (d_l, d_1)$ connecting them, meaning that the salesman simply waits at the station. Among these edges, (d_l, d_1) implies that he stays overnight to wait for the first train on the next day. It is assumed that train schedules do not

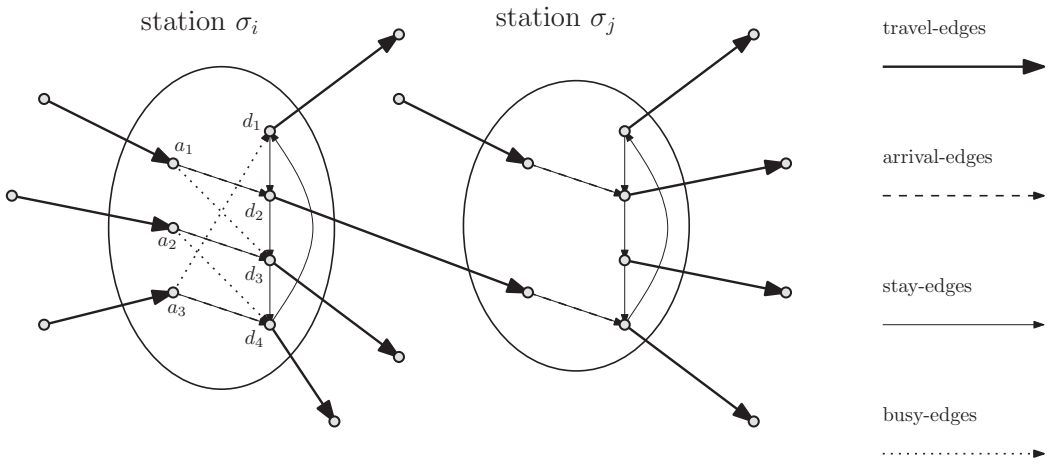


Figure 9.1: Example for two stations in the time-expanded digraph, $\sigma_i \in B$ and $\sigma_j \notin B$

change from day to day. If they do change, the model has to be expanded so that each station σ contains all depart and arrival-nodes for the whole week (assuming that schedules do not change from week to week). For practical reasons, arrival-nodes $a_1, \ldots, a_k$ are also ordered according to their times, but there are no edges among themselves.

Arrival-edges connect each arrival-node with their immediately next departure-node (w.r.t. their time values) of the same station. Travel-edges connect depart-nodes with their corresponding arrival-nodes according to the timetable. Finally, for all stations $\sigma \in B$ which the salesman has to visit, there are busy-edges which connect arrival-nodes with their next possible departure-nodes where he could leave σ after spending the required amount of time. The cost for each edge $c(u, v),\ (u, v) \in E$ is the cycle-difference $(1440 + t(v) - t(u)) \mod 1440$. This assumes that a day-to-day model is used and no train requires more than one day to get from one station to the next stop.

Graph Size Reduction

Hadjicharalambous et al. [46] presented a preprocessing algorithm to reduce the size of the time-expanded graph. For each station $\sigma \in B$, a new sink-node s_σ is added and all arrival-nodes in σ are connected to s_σ with zero cost edges. Then, for all departure-nodes in $d \in \sigma$, shortest paths to sink-nodes in all other stations in B are computed. For each path, an edge between d and the last arrival-node of that

path is added to G. The edge costs equal the costs of the corresponding shortest path. Then, all sink-nodes and their incoming edges are removed again, as well as all stations not in B along with their nodes. Arrival-nodes which are not used by any of the shortest paths and all arrival-edges are removed, too.

After the size reduction procedure, G only consists of stations $\sigma \in B$, their arrival and departure-nodes, the edges connecting them, and the newly added shortest path edges.

9.3 Transformation to asymmetric TSP

We propose a scheme to transform the reduced RTSP into an asymmetric TSP defined on a graph $G' = \langle V', E' \rangle$ by applying similar ideas as Behzad et al. [6], who presented a transformation for the GTSP to TSP. For each station $\sigma \in B$ in the original reduced graph, we apply the following procedure to obtain the new station $\sigma' \subset V'$.

Let $a'_1, \ldots, a'_k \in \sigma'$ be exact copies of arrival-nodes $a_1, \ldots, a_k \in \sigma$, ordered according to their times. We connect them to a cycle by zero cost edges $(a'_1, a'_2), \ldots, (a'_{k-1}, a'_k)$, (a'_k, a'_1). The same procedure is applied on the departure-nodes $d'_1, \ldots, d'_l$, which get connected by zero cost edges $(d'_1, d'_2), \ldots, (d'_{l-1}, d'_l)$, (d'_l, d'_1). Let $pred(v)$ denote the cyclic predecessor of node v according to these cycles. We connect every arrival-node a'_i, $i = 1, \ldots, k$ with every departure-node d'_j, $j = 1, \ldots, l$. For the initial station, $c(a'_i, d'_j)$ is simply set to $t(d'_j) - t(d'_1)$ which does not depend on $t(a'_i)$. The reason is that the salesman has to end his tour at the initial station, but not at the initial time. So it only makes a difference when he starts the journey, thus $c(a'_i, d'_j) = t(d'_j) - t(d'_1)$ where $t(d'_1)$ is the earliest possible depart time from the initial station. For all other stations in B, the costs are set to $c(a'_i, d'_j) = (1440 + t(d'_j) - t(pred(a'_i)) - t_\sigma) \mod 1440 + t_\sigma$. Note that while all original stay edges are not present, their costs are included in these new edges. For example, in Figure 9.2, going from a_1 to d_4 means to take the busy-edge (a_1, d_3) and the stay-edge (d_3, d_4). The costs are $t(d_4) - t(a_1)$, which equal the costs of the new edge (a'_3, d'_4).

Finally, we have to adapt the outgoing edges from d'_j, $j = 1, \ldots, l$ as well. For each original travel-edge (d_j, v), we introduce an edge $(pred(d'_j), v)$. The costs are $c(pred(d'_j), v) = c(d_j, v) + M$ where M must be a sufficiently large number, e.g. $\sum_{(u,v)\in E} c(u, v)$, to prevent more than one travel-edge to be included in the optimal solution.

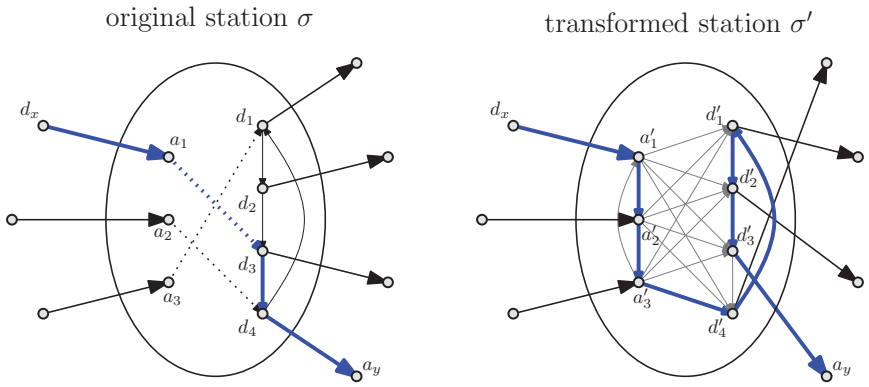


Figure 9.2: Transforming σ into σ' in the asymmetric TSP. The bold path marks the travel route from d_x to a_y in the original σ and in the transformed σ'.

Figure 9.2 illustrates this transformation procedure and shows an example how a route is adapted in the new graph. Unfortunately, the resulting graph is directed, thus the TSP is asymmetric.

9.4 Transformation to symmetric TSP

Since the previous transformation results in an asymmetric TSP, we propose an alternative procedure to generate a symmetric TSP defined on the graph $G' = \langle V', E' \rangle$. This approach follows the same basic ideas for the previous transformation. Unfortunately, the number of nodes needs to be doubled during this process.

Let again $a'_1, \ldots, a'_k \in \sigma'$ be exact copies of arrival-nodes $a_1, \ldots, a_k \in \sigma$. We duplicate them one more time, obtaining $a''_1, \ldots, a''_k$. Then, we connect these nodes by edges $(a'_2, a''_1), (a'_3, a''_2), \ldots, (a'_k, a''_{k-1}), (a'_1, a''_k)$ with zero costs and edges $(a'_i, a''_i),\ i = 1, \ldots, k$ with high costs M to a cycle. High costs on edges (a'_i, a''_i) cause as few of them to be present in the solution as possible. For example, in Figure 9.3, when a travel-edge leads to a_1, there are two possible routes to go through $\{a'_1, \ldots, a'_3\} \cup \{a''_1, \ldots, a''_3\}$, one ending at a''_3 and one ending at a''_1. The latter one is cheaper since edge (a'_1, a''_1) with high costs is spared.

The same procedure is applied to nodes $d'_1, \ldots, d'_l$ and their duplicates $d''_1, \ldots, d''_l$. Then, we connect all $a''_i,\ i = 1, \ldots, k$ with all $d'_j,\ j = 1, \ldots, l$. The costs of edges (a''_i, d'_j) are set to $(1440 + t(d'_j) - t(a''_i) - t_\sigma) \mod 1440 + t_\sigma$. Like in the previous transformation, costs of original stay edges are implicitly included in these edges.

Finally, for each original travel-edge (d_j, v), we add a new edge (d''_j, v) with costs $c(d''_j, v) = c(d_j, v) + M'$. Since M is already used by edges (a'_i, a''_i) and (d'_j, d''_j) inside σ', we have to choose an even larger value for M' to ensure that only one travel-edge is included in the solution on G'. We set $M' = M \cdot |B|$ for this matter.

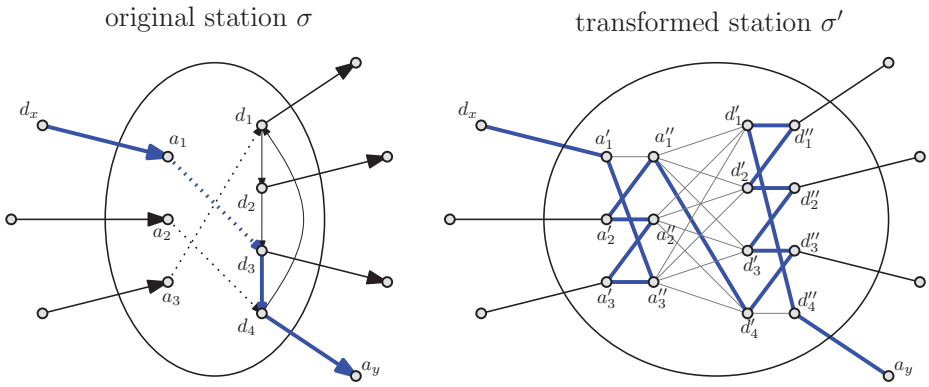


Figure 9.3: Transforming σ into σ' in the symmetric TSP. The bold path marks the travel route from d_x to a_y in the original σ and in the transformed σ'.

9.5 Computational Results

To test our TSP transformation schemes, we use instances based on two railway timetables containing real-world data of train schedules in the Netherlands. They were provided by the authors of [46, 92]. Instance 3000_3 contains 23 cities and represent trains of a local region. Instance ic_times contains 27 larger cities in the Netherlands which are connected by Intercity trains. The requirements for the traveling salesman were generated by us at random, as the authors could not find theirs anymore. They are created by letting the salesman either visit 5 or 10 cities, and stay times were chosen between 10 and 240 minutes at each location. Since [46, 92] also used randomly generated data for the salesman with the same number of cities to be visited, the results should be at least roughly comparable. Stay times were not mentioned though.

The time-expanded digraph based on both timetables contain more than 2000 nodes and around 4000 edges (which depends on the traveling person). After reducing the graph, we carry out the transformation procedure to generate a symmetric TSP instance and solve it with the Branch-and-Cut (B&C) algorithm of the Concorde

library[2]. Our experiments were performed on an Intel Core 2 Quad 2.4 GHz PC with 4GB memory and B&C uses ILOG CPLEX 9.0 as solver.

Table 9.1: Results on timetable 3000_3, containing 23 cities and 1095 train entries.

Index	$\|B\|$	nodes	$\overline{\text{time}}$	std dev
1	5	348	1.14s	0.35
2	5	514	1.02s	0.12
3	5	514	0.98s	0.07
4	5	688	7.14s	3.42
5	5	852	2.65s	0.45
6	5	762	2.41s	0.34
7	5	1312	56.81s	22.89
8	5	782	4.03s	1.20
9	5	854	2.84s	0.73
10	5	456	0.84s	0.11
11	10	1786	15.48s	6.57
12	10	1562	8.67s	4.04
13	10	1960	36.29s	13.39
14	10	1260	5.17s	1.07
15	10	1410	7.89s	3.03
16	10	1478	219.67s	267.30
17	10	2276	52.95s	13.69
18	10	1748	24.64s	13.40
19	10	2252	20.77s	4.88
20	10	1342	6.20s	2.98

Table 9.1 and 9.2 show the performance of Concorde branch-and-cut (B&C) on the transformed instances. We performed 30 runs for each instance, since B&C uses random seeds to decide its branching order, thus resulting in variable run-times. Listed are the instance's indices, numbers of cities the salesman must visit, numbers of nodes in the TSP after graph reduction and transformation are applied, the average run-times needed for B&C to find and prove the optimal solutions, and standard deviations of run-times.

For instances 12, 14, and 18 in Table 9.2, all 30 runs were terminated after 10000s where B&C still could not prove optimality. In all these cases, however, the remaining gap is less than 0.1%.

[2] www.tsp.gatech.edu/concorde.html

Table 9.2: Results on timetable ic_times, containing 27 cities and 1129 train entries.

Index	$\|B\|$	nodes	$\overline{\text{time}}$	std dev
1	5	422	3.35s	1.35
2	5	542	5.50s	1.93
3	5	712	6.19s	1.36
4	5	364	0.52s	0.04
5	5	1178	30.26s	9.38
6	5	450	0.77s	0.06
7	5	516	3.47s	1.09
8	5	1140	10.16s	3.57
9	5	324	29.68s	16.17
10	5	568	8.50s	3.29
11	10	1786	1907.47s	990.37
12	10	1562	–	0.00
13	10	1960	45.68s	27.39
14	10	1260	–	0.00
15	10	1410	71.29s	48.41
16	10	1478	1471.14s	781.79
17	10	2276	239.52s	72.94
18	10	1748	–	0.00
19	10	2252	1071.12s	377.63
20	10	1342	346.39s	134.51

Analyzing the results, we observe large differences in run-times between the instances. Among instances with the same size for B, trips can contain quite different number of nodes. This is due to the random selection for stations in B. Looking at data from the railway timetables, some stations only contain a few arrival and departure-nodes while others have a full lineup. This imbalance appears to a greater extent for timetable "ic_times", in which some stations contain as few as 2 nodes and others are as large as 254 nodes. This could be the reason why these instances seems to be harder to solve for Concorde B&C.

Table 9.3 shows the performance of direct approaches for the RTSP: The multi-commodity flow Integer Linear Programming (ILP) formulation solved via GLPSOL (GNU Linear Programming Kit LP/MIP Solver) version 4.6 reported in [46] and the Ant Colony Optimization (ACO) approach [92]. While the ILP operated on reduced graphs, ACO used original graphs.

Table 9.3: Reported results using ILP [46] and ACO [92]

Instance	$\|B\|$	time ILP	time ACO
3000_3	5	319.00s	18.68s
3000_3	10	9111.90s	677.36s
ic_times	5	29.10s	16.60s
ic_times	10	6942.60s	374.28s

Comparing these results with those obtained by Concorde B&C, the latter seems to perform better. However, we have to keep in mind that different data were generated for the salesman, different integer linear programming solvers and different hardware were used.

9.6 Conclusions

In this chapter, we proposed two transformation schemes for the Railway Traveling Salesman Problem (RTSP), resulting in the asymmetric TSP and the symmetric TSP. Though the number of nodes has to be doubled in the symmetric TSP transformation, the branch-and-cut algorithm of the Concorde library could solve most of the tested instances to provable optimality in very reasonable time.

For larger RTSP instances, the transformation to the asymmetric or symmetric TSP is still meaningful when using a faster state-of-the-art TSP heuristic, such as the chained Lin-Kernighan algorithm [2].

In future work, we will evaluate the approach on more and in particular larger instances and also test the performance of the direct integer linear programming formulation for RTSP using ILOG CPLEX for a fairer comparison. Other direct (meta)heuristic approaches also seem appealing, especially when the number of cities which the salesman has to visit becomes too high for exact approaches.

For the symmetric TSP transformation, we still have some implementation-dependent problems with the big M and M' added to the edges costs. Trying to solve instances where the salesman visits 15 cities results in integer overflows and the branch-and-cut algorithm can only operate with integer values. Therefore, we would like to enhance the transformation procedure to overcome this difficulty.

We also have not made practical experiments exploiting the transformation to the asymmetric TSP yet. Since the number of nodes is only half of the symmetric case

and only one big M is required for the transformation, this seems to be a very reasonable alternative.

Chapter 10

Conclusions

This thesis considered three Generalized Network Design Problems (GNDPs) in detail: the Generalized Minimum Spanning Tree Problem (GMSTP), the Generalized Traveling Salesman Problem (GTSP), the Generalized Minimum Edge Biconnected Network Problem (GMEBCNP), and the Railway Traveling Salesman Problem (RTSP).

Hybrid metaheuristics were used in order to attack these $\mathcal{NP}$-hard combinatorial optimization problems. The fundamental strategy was to design complementary neighborhood structures which augment each other well. Most neighborhood structures are large in the sense that they contain exponentially many candidate solutions, but we used efficient algorithms to identify a best or nearly best neighbor. Variable Neighborhood Search (VNS) approaches based on these neighborhood structures were used as frameworks to handle these problems.

The Generalized Minimum Spanning Tree Problem

We developed a (VNS) approach for the GMSTP using three neighborhood structures of different types. Two of them use complementary search strategies. They are exponentially large, but polynomial-time algorithms are used to identify the best neighbor. The third neighborhood structure uses techniques from Integer Linear Programming to (IP) in order to improve the structure of subareas in candidate solutions.

Computational results show that this approach is able to generating high quality solutions in acceptable time. Whereas state-of-the-art exact approaches are able to identify optimal solutions for instances with about 200 nodes within more than one hour, VNS only requires 10 minutes to generate near optimal solutions for instances with up to 1280 nodes. It was also able to obtain significantly better solutions on certain types of instances compared to other metaheuristic approaches which only rely on neighborhood structures of a single type.

The Generalized Traveling Salesman Problem

We proposed a VNS algorithm for the GTSP that uses two neighborhood structures which can be seen as dual to each other. The first one has already been successfully applied to this problem in the literature, but we presented a novel efficient evaluation technique to further reduce computational effort. The second neighborhood uses the well known Chained Lin-Kernighan algorithm as sub-procedure to evaluate candidate solutions.

Although our VNS is more time consuming than recent approaches based on genetic algorithms, it is able to generate solutions of much higher quality. Measured on Euclidean TSPlib instances containing 100 to 442 nodes, the average gap in the objective values between our results and the optimal ones is only 0.05%.

The Generalized Minimum Edge Biconnected Network Problem

This problem extends the GMSTP by requiring additional fault tolerance on the solution graph in terms of edge biconnectivity. We developed different VNS variants based on four neighborhood structure types that address different aspects of the solutions. For the more complex ones, we also proposed advanced evaluation techniques to reduce evaluation effort or to avoid searching in areas where definitely no improving solutions exist.

Computational results show that all neighborhood structures contribute significantly to the search process. Since it was hard to determine the order in which the neighborhood structures are applied in VNS, we presented a new variant called self-adaptive VNS. By rearranging the order the neighborhoods dynamically during the optimization process according to the success of each neighborhood, we could obtain better solutions compared to classical VNS approaches.

The Railway Traveling Salesman Problem

As a related problem to the GTSP, we considered the RTSP. Based on modelling and preprocessing techniques that were proposed in the literature, we described two transformations that reformulate the RTSP as classical asymmetric and symmetric TSPs, respectively.

For testing the performance, we solved the resulting symmetric TSP instances using the branch-and-cut (B&C) algorithm from the Condore library. Compared to other direct methods for the RTSP, we observe that our transformation approach benefits from the strengths of the Concorde B&C solver and optimal solutions could be obtained in less overall time.

Looking at the results obtained for the considered GNDPs, we conclude that our general strategy worked out very well. By combining diverse solution representations with complementary search techniques, it is possible to reach areas in the search space that are inaccessible when only using neighborhood structures of a single type.

It is also crucial to use efficient search strategies to identify the best solutions in large neighborhoods that contain exponentially many candidates. Wherever applicable, incremental evaluation schemes are important to further reduce the computational effort for searching through such neighborhoods.

Bibliography

[1] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[2] D. Applegate, W. Cook, and A. Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15, issue 1:82–92, 2003.

[3] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York, 1997.

[4] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[5] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6:126–140, 1994.

[6] A. Behzad and M. Modarres. A new efficient transformation of the generalized traveling salesman problem into traveling salesman problem. In *Proceedings of the 15th International Conference of Systems Engineering*, pages 6–8, 2002.

[7] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[8] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. In J. F. Traub, editor, *Symposium on New Directions and Recent Results in Algorithms and Complexity*, page 441, New York, NY, 1976. Academic Press.

[9] R. K. Congram. *Polynomially Searchable Exponential Neighbourhoods for Sequencing Problems in Combinatorial Optimisation*. PhD thesis, University of Southampton, Faculty of Mathematical Studies, UK, 2000.

[10] C. Cotta and J. Troya. Embedding branch and bound within evolutionary algorithms. *Applied Intelligence*, 18(2):137–153, 2003.

[11] P. I. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *PATAT '00: Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, pages 176–190, London, UK, 2001. Springer-Verlag.

[12] A. Czumaj and A. Lingas. On approximability of the minimum-cost k-connected spanning subgraph problem. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 281–290. Society for Industrial and Applied Mathematics, 1999.

[13] T. S. D. Whitley and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings on the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.

[14] E. Danna, E. Rothberg, and C. Le Pape. Integrating mixed integer programming and local search: A case study on job-shop scheduling problems. In *Fifth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'2003)*, 2003.

[15] C. Darwin. *The Origin of Species*. John Murray, 1859.

[16] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 162–164, 1985.

[17] R. Dawkins. *The selfish gene*. Oxford University Press, Oxford, 1976.

[18] D. de Werra and A. Hertz. Tabu-search techniques a tutorial and an application to neural networks. *Operations Research Spektrum*, 11:131–141, 1989.

[19] I. Devarenne, H. Mabed1, and A. Caminada. Adaptive tabu tenure computation in local search. *Evolutionary Computation in Combinatorial Optimization*, 4972:1–12, 2008.

[20] V. Dimitrijevic and Z. Saric. An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs. *Information Science*, 102(1-4):105–110, 1997.

[21] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

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

[23] C. W. Duin, A. Volgenanta, and S. Voss. Solving group steiner problems as steiner problems. In *European Journal of Operational Research*, volume 154, issue 1, pages 323–329, 2004.

[24] I. Dumitrescu and T. Stützle. Combinations of local search and exact algorithms. In G. R. Raidl et al., editors, *Applications of Evolutionary Computing: EvoWorkshops 2003*, volume 2611 of *LNCS*, pages 212–224. Springer, 2003.

[25] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, Berlin Heidelberg, 2003.

[26] O. Ergun and J. B. Orlin. Dynamic programming methodologies in very large scale neighborhood search applied to the traveling salesman problem. Technical Report Working Paper 4463-03, MIT Sloan School of Management, Massachusetts, 2003.

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

[28] 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.

[29] 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.

[30] C. Feremans, M. Labbe, and G. Laporte. Generalized network design problems. *European Journal of Operational Research*, 148(1):1–13, 2003.

[31] 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.

[32] M. Fischetti and A. Lodi. Local Branching. *Mathematical Programming Series B*, 98:23–47, 2003.

[33] M. Fischetti, J. J. Salazar, and P. Toth. The symmetric generalized traveling salesman polytope. *Networks*, 26:113–123, 1995.

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

[35] L. J. Fogel, A. J. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, UK, 1966.

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

[37] 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.

[38] F. Glover. Future paths for integer programming and links to artiïcial intelligence. *Decision Sciences*, 8:156–166, 1977.

[39] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Norwell, MA, 2003.

[40] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, MA, 1997.

[41] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[42] D. Goldberg and R. Lingle. Alleles, loci, and the travelling salesman problem. In J. J. Grefenstette, editor, *Proceedings of the First Int. Conf. on Genetic Algorithms*, pages 154–159. Lawrence Erlbaum, 1985.

[43] 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.

[44] M. Gruber and G. R. Raidl. (Meta-)heuristic separation of jump cuts for the bounded diameter minimum spanning tree problem. In P. Hansen et al., editors, *Proceedings of Matheuristics 2008: Second International Workshop on Model Based Metaheuristics*, Bertinoro, Italy, 2008.

[45] M. Gruber and G. R. Raidl. (Meta-)heuristic separation of jump cuts in a branch&cut approach for the bounded diameter minimum spanning tree problem. Technical Report TR 186–1–08–02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria, 2008. submitted to a special issue on Matheuristics of Operations Research/Computer Science Interface Series, Springer.

[46] G. Hadjicharalambous, P. Pop, E. Pyrga, G. Tsaggouris, and C. Zaroliagis. The railway traveling salesman problem. *Algorithmic Methods for Railway Optimization*, 4359:264–275, 2007.

[47] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In S. Voss, S. Martello, I. Osman, and C. Roucariol, editors, *Meta-*

heuristics: advances and trends in local search paradigms for optimization, pages 433–438. Kluwer Academic Publishers, Boston MA, 1999.

[48] P. Hansen and N. Mladenovic. An introduction to variable neighborhood search. In S. Voss et al., editors, *Meta-heuristics, Advances and trends in local search paradigms for optimization*, pages 433–458. Kluwer Academic Publishers, 1999.

[49] 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.

[50] 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.

[51] Henry-Labordere. The record balancing problem: A dynamic programming solution of a generalized traveling salesman problem. *RAIRO Operations Research*, B2:43–49, 1969.

[52] A. Hertz and D. de Werra. The tabu-search metaheuristic: how we used it. *Annals of mathematics and artificial intelligence*, 1:54–60, 1990.

[53] J. Holland. *Adaptation In Natural and Artificial Systems*. University of Michigan Press, 1975.

[54] H. Hoos and T. Stützle. *Stochastic Local Search – Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2004.

[55] B. Hu, M. Leitner, and G. R. Raidl. Computing generalized minimum spanning trees with variable neighborhood search. In P. Hansen, N. Mladenović, 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.

[56] B. Hu, M. Leitner, and G. R. Raidl. The generalized minimum edge biconnected network problem: Efficient neighborhood structures for variable neighborhood search. Technical Report TR 186–1–07–02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2007. submitted to Networks.

[57] B. Hu, M. Leitner, and G. R. Raidl. Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. *Journal of Heuristics*, 14(5):473–499, 2008.

[58] B. Hu and G. R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In C. Cotta, A. J. Fernandez, and J. E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*, Malaga, Spain, 2006.

[59] B. Hu and G. R. Raidl. Effective neighborhood structures for the generalized traveling salesman problem. In J. van Hemert and C. Cotta, editors, *Evolutionary Computation in Combinatorial Optimisation – EvoCOP 2008*, volume 4972 of *LNCS*, pages 36–47, Naples, Italy, 2008. Springer.

[60] B. Hu and G. R. Raidl. Solving the railway traveling salesman problem via a transformation into the classical traveling salesman problem. In F. Xhafa, F. Herrera, A. Abraham, M. Köppen, and J. M. Benitez, editors, *8th International Conference on Hybrid Intelligent Systems – HIS 2008*, pages 73–77, Barcelona, Spain, 2008.

[61] H. Huang, X. Yang, Z. Hao, C. Wu, Y. Liang, and X. Zhao. Hybrid chromosome genetic algorithm for generalized traveling salesman problems. *Advances in Natural Computation*, 3612/2005:137–140, 2005.

[62] D. Huygens. Version generalisee du probleme de conception de reseau 2-arete-connexe. Master's thesis, Universite Libre de Bruxelles, 2002.

[63] E. Ihler, G. Reich, and P. Widmayer. Class steiner trees and vlsi-design. *Discrete Appl. Math.*, 90(1-3):173–194, 1999.

[64] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[65] G. Kendall, E. Soubeiga, and P. Cowling. Choice function and random hyperheuristics. In *Proceedings of the fourth Asia-Pacific Conference on Simulated Evolution And Learning, SEAL*, pages 667–671. Springer, 2002.

[66] L. Khachiyan. A polynomial algorithm in linear programming (english translation). *Soviet Mathematics Doklady*, 20:191–194, 1979.

[67] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. *Journal of the ACM (JACM)*, 41:214–235, 1994.

[68] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[69] A. Koster, S. V. Hoesel, and A. Kolen. The partial constraint satisfaction problem: Facets and lifting theorems. *Operations Research Letters*, 23:89–97, 1998.

[70] K. Kostikas and C. Fragakis. Genetic programming applied to mixed integer programming. In M. Keijzer et al., editors, *Genetic Programming - EuroGP 2004*, volume 3003 of *LNCS*, pages 113–124. Springer, 2004.

[71] J. B. Kruskal. On the shortest spanning subtree and the traveling salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, 1956.

[72] G. Laporte, A. Asef-Vaziri, and C. Sriskandarajah. Some applications of the generalized travelling salesman problem. *Journal of the Operational Research Society*, 47(12):1461–1467, 1996.

[73] G. Laporte, H. Mercure, and Y. Nobert. Generalized traveling salesman problem through n sets of nodes: The asymmetric case. *Discrete Applied Mathematics*, 18:185–197, 1987.

[74] G. Laporte and Y. Nobert. Generalized traveling salesman problem through n sets of nodes: An integer programming approach. *INFOR*, 21, issue 1:61–75, 1983.

[75] G. Laporte and F. Semet. Computational evaluation of a transformation procedure for the symmetric generalized traveling salesman problem. *INFOR*, 37(2):114–120, 1999.

[76] M. Leitner. Solving two generalized network design problems with exact and heuristic methods. Master's thesis, Vienna University of Technology, 2006.

[77] M. Leitner, B. Hu, and G. R. Raidl. Variable neighborhood search for the generalized minimum edge biconnected network problem. In B. Fortz, editor, *Proceedings of the International Network Optimization Conference 2007*, pages 69/1–6, Spa, Belgium, 2007.

[78] W. J. Li, J. Tsao, and O. Ulular. The shortest path with at most l nodes in each of the series/parallel clusters. *Networks*, 26:263–271, 1995.

[79] Y. N. Lien, E. Ma, and B. W. S. Wah. Transformation of the generalized traveling salesman problem into the standard traveling salesman problem. *Information Sciences*, 74(1–2):177–189, 1993.

[80] S. Lin. Computer solutions of the traveling salesman problem. *Bell Systems Computer Journal*, 44:2245–2269, 1965.

[81] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In *Handbook of Metaheuristics* [39], pages 321–353.

[82] O. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 5:299–326, 1991.

[83] R. Montemanni and D. H. Smith. A tabu search algorithm with a dynamic tabu list for the frequency assignment problem. Technical report, University of Glamorgan, UK, 2001.

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

[85] P. Moscato. Memetic algorithms: A short introduction. In D. Corne et al., editors, *New Ideas in Optimization*, pages 219–234. McGraw Hill, 1999.

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

[87] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[88] C. Noon and J. C. Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR*, 31(1):39–44, 1993.

[89] C. E. Noon. *The Generalized Traveling Salesman Problem*. PhD thesis, University of Michigan, 1988.

[90] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

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

[92] P. C. Pop, C.-M. Pintea, and C. P. Sitar. An ant-based heuristic for the railway traveling salesman problem. In *EvoWorkshops*, volume 4448, pages 702–711. Springer, 2007.

[93] 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.

[94] P. C. Pop, C. D. Zaroliagis, and G. Hadjicharalambous. A cutting plane approach to solve the railway traveling salesman problem. *Studia Universitatis Mathematica*, 53(1):63–72, 2008.

[95] R. C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[96] J. Puchinger and G. R. Raidl. An evolutionary algorithm for column generation in integer programming: an effective approach for 2D bin packing. In X. Y. et. al, editor, *Parallel Problem Solving from Nature – PPSN VIII*, volume 3242 of *LNCS*, pages 642–651. Springer, 2004.

[97] J. Puchinger and G. R. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *Proceedings of the First International Work-Conference on the Interplay Between Natural and Artificial Computation*, volume 3562 of *LNCS*, pages 41–53. Springer, 2005.

[98] J. Puchinger and G. R. Raidl. Relaxation guided variable neighborhood search. In *Proceedings of the XVIII Mini EURO Conference on VNS*, Tenerife, Spain, 2005.

[99] J. Puchinger and G. R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research*, 183(3):1304–1327, 2007.

[100] J. Puchinger and G. R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research, Feature Issue on Cutting and Packing*, to appear 2005.

[101] I. Rechenberg. *Evolutionsstrategie, Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Verlag, 1973.

[102] 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.

[103] J. Renaud and F. F. Boctor. An efficient composite heuristic for the symmetric generalized traveling salesman problem. *European Journal of Operational Research*, 108:571–584, 1998.

[104] J. Renaud, F. F. Boctor, and G. Laporte. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, 8, issue 2:134–143, 1996.

[105] J. J. Salazar. A note on the generalized steiner tree polytope. *Discrete Applied Mathematics*, 100(1-2):137–144, 2000.

[106] J. P. Saskena. Mathematical model of scheduling clients through welfare agencies. *Journal of the Canadian Operational Research Society*, 8:185–200, 1970.

[107] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5(12):571–584, 2000.

[108] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[109] A. Segev. The node-weighted steiner tree problem. *Networks*, 17:185–200, 1987.

[110] J. Silberholz and B. Golden. The generalized traveling salesman problem: A new genetic algorithm approach. *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, 37(4):165–181, 2005.

[111] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. Technical Report 04T-018, Dept. of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, USA, 2004.

[112] Srivastava, S. S. S. Kumar, R. C. Garg, and P. Sen. Generalized traveling salesman problem through n sets of nodes. *CORS Journal*, 7:97–101, 1969.

[113] G. Syswerda. Schedule optimization using genetic algorithms. pages 332–349. Int. Thomson Computer Press, 1991.

[114] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[115] E. Taillard and S. Voss. POPMUSIC: Partial optimization metaheuristic under special intensification conditions. In C. Ribeiro and P. Hansen, editors, *Essays and surveys in metaheuristics*, pages 613–629, 2001.

[116] P. Thompson and H. Psaraftis. Cycle transfer algorithm for multivehicle routing and scheduling problems. *Operations Research*, 41:935–946, 1993.

[117] V. V. Vazirani. *Approximation algorithms*. Springer, 2001.

[118] S. Voss, S. Martello, I. Osman, and C. Roucariol. Kluwer Academic Publishers, Boston MA, 1999.

[119] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1(6):80–83, 1945.

[120] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

[121] C. Wu, Y. Liang, H. P. Lee, and C. Lu. Generalized chromosome genetic algorithm for generalized traveling salesman problems and its applications for machining. *Physical Review E*, 70, issue 1, 2004.

[122] B. Yang and P. Gillard. The class steiner minimal tree problem: a lower bound and test problem generation. *Acta Informatica*, 37(3):193–211, 2000.

Curriculum Vitae

Personal Information

- Name: Bin Hu
- Date of birth: October 23, 1980
- Place of birth: Shanghai, China

Education

- since 10/2004: PhD student at Vienna University of Technology. Main research: "Hybrid Metaheuristics for Generalized Network Design Problems", supervised by Günther Raidl
- 10/1999 – 04/2004: Computer Science studies at Vienna University of Technology with graduation to "Diplom Ingenieur" (MSc). Diploma thesis: "Human Guided Car Sequencing for the Automobile Industry", supervised by Gunnar Klau
- 09/1993 – 06/1999: Comprehensive school in Vienna, Austria
- 09/1991 – 06/1993: Secondary school in Grieskirchen, Austria
- 09/1987 – 06/1991: Primary school in Grieskirchen, Austria
- 09/1986 – 12/1986: Primary school in Shanghai, China

Work Experience

- since 03/2005: Research and teaching assistant, Algorithms and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology
- 03/2005 – 01/2006: Employed in the FWF project *Combining Memetic Algorithms with Branch and Cut and Price for Some Network Design Problem* under grant P16263-N04, Algorithms and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology
- 10/2000 – 01/2005: Tutor ("Studienassistent") for Introduction to Programming, Institute of Pattern Recognition and Automation
- 10/2000 – 06/2004: Tutor ("Studienassistent") for Algorithms and Data Structures 1 and 2, Institute of Computer Graphics and Algorithms

Publications

Refereed Journal Articles

- Bin Hu, Markus Leitner, and Günther R. Raidl. Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. *Journal of Heuristics*, 14(5): 473–499, 2008
- Bin Hu, Markus Leitner, and Günther R. Raidl. The generalized minimum edge biconnected network problem: Efficient neighborhood structures for variable neighborhood search. *Networks*. Accepted for publication 2008.

Refereed Conference Papers

- Bin Hu and Günther R. Raidl. Solving the railway traveling salesman problem via a transformation into the classical traveling salesman problem. *Hybrid Intelligent Systems – HIS 2008*, Barcelona, Spain, 2008.
- Bin Hu and Günther R. Raidl. Effective neighborhood structures for the generalized traveling salesman problem. *Evolutionary Computation in Combinatorial Optimisation – EvoCOP 2008*, volume 4972 of LNCS, pages 36–47, Naples, Italy, Springer, 2008.
- Markus Leitner, Bin Hu, and Günther R. Raidl. Variable neighborhood search for the generalized minimum edge biconnected network problem. *Proceedings of the International Network Optimization Conference – INOC 2007*, pages 69/1–6, Spa, Belgium, 2007.

- Bin Hu and Günther R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*, Malaga, Spain, 2006.
- Bin Hu, Markus Leitner, and Günther R. Raidl. Computing generalized minimum spanning trees with variable neighborhood search. *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search*, Tenerife, Spain, 2005.

Master Thesis

- Bin Hu. Interaktive Reihenfolgeplanung für die Automobilindustrie. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, April 2004. Supervised by G. W. Klau.