D I S S E R T A T I O N

Exact and Memetic Algorithms for Two Network Design Problems

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

Univ.-Prof. Dr. Petra Mutzel
Institut für Computergraphik und Algorithmen - E186
Technische Universität Wien

und
a.o. Univ.-Prof. Dr. Ulrich Pferschy
Institut für Statistik und Operations Research
Universität Graz

eingereicht an der Technische Universität Wien
Fakultät für Informatik

von

Mag. Ivana Ljubić
Matrikelnummer 0027118
Ospelgasse 17/7/1, 1200 Wien

Wien, am 23.11.2004

Ivana Ljubić

Abstract

This thesis focuses on two combinatorial optimization problems (COPs) that belong to the class of NP-hard network design problems: The first one, *vertex biconnectivity augmentation* (V2AUG), appears in the design of survivable communication or electricity networks. In this problem we search for the set of connections of minimal total cost which, when added to an existing network, makes it survivable against failures of any single node. The second problem, *the prize-collecting Steiner tree problem* (PCST), describes a natural trade-off between maximizing the sum of profits over all selected customers and minimizing the implementation costs, e.g. when designing a fiber optic or a district heating network.

The available techniques for COPs 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 for every instance of a COP. Due to sometimes exponential running times or memory requirements of exact algorithms we sometimes sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a limited time and therefore use heuristic algorithms. This thesis provides tools that can solve given network design problems of respectable size to provable optimality. For fairly large instances, these tools obtain suboptimal, high quality solutions of practical relevance and provide optimality gaps as a measure of their quality.

As a heuristic tool, we choose *memetic algorithms* (MAs), a symbiosis of evolutionary and neighborhood search algorithms. Over the last few years, memetic algorithms have shown their great capabilities in finding high quality solutions to difficult global optimization tasks. The exact approaches considered in the scope of this thesis are *branch-and-cut* (BC) and *branch-and-cut-and-price* (BCP) algorithms. Nowadays these methods are the most effective exact algorithms for plenty of integer and mixed-integer programming problems.

The memetic algorithms that we propose for V2AUG and the PCST, comprise new solution representation techniques, search operators, constraint handling techniques, local-improvement strategies, and heuristic biasing methods. Our exact algorithms are based on the state-of-the-art in polyhedral combinatorics. They rely on sophisticated separation algorithms or advanced column generation methods. In this thesis, we also investigate some possibilities of combining promising variants of exact algorithms and MAs, like incorporating exact algorithms that solve some special cases within MAs, biasing primal heuristics or guiding column generation using MA results.

For solving V2AUG, we first propose running a deterministic preprocessing algorithm that reduces the search space. Based on the generation of the so-called block-cut graph data structure, we provide new tests for reducing the instance size. We then propose a memetic algorithm in which all candidate solutions are locally optimal with respect to their number of augmentation edges. Locality, heritability and biasing of variation operators play very important roles in the design of our MA. Empirical results show that the approach scales well to instances of large size. Our results are significantly better than those obtained by three previously published heuristics. To be able to estimate the quality of obtained MA solutions, we develop a branch-and-cut algorithm that relies on a connectivity-based ILP formulation with the separation procedure that runs in polynomial time. Our computational experiments show that the branch-and-cut algorithm is an efficient tool for solving small and randomly generated instances to optimality. For solving larger benchmark instances we extended the proposed branch-and-

cut algorithm with a column generation procedure (also called pricing). Our results indicate that the incorporation of pricing represents the only practical way to solve very large instances to proven optimality. For the largest instances we tested, we initialize upper bounds with the best MA solutions, in order to improve the overall performance of the BCP algorithm and in order to reduce the optimality gaps.

In the second part of the thesis, we concentrate on the prize-collecting Steiner tree problem. After running a preprocessing procedure for PCST, we propose running a memetic algorithm in which all individuals of the population represent local optima with respect to their subtrees. This is ensured by applying a linear-time local improvement algorithm that solves the PCST on trees to optimality. A clustering procedure that groups the subsets of vertices enhances our problem-dependent variation operators. Extensive experiments on benchmark instances from the literature show that the MA compares favorably to previously published results. While the solution values are almost always the same as in previously published results, substantial reductions of running times are achieved.

Our next contribution is the formulation of an integer linear program on a directed graph model based on connectivity inequalities. As for V2AUG, the main advantage of this model is the efficient separation of violated inequalities by a polynomial time algorithm. Moreover, we introduce new asymmetry constraints that reject multiple consideration of the same solution. Our new approach manages to solve all benchmark instances from the literature to optimality, including eight for which the optimum has not been known previously. Compared to a recent exact algorithm our new method is faster by more than two orders of magnitude. For these instances, the ILP approach is also significantly faster than the memetic algorithm itself. Furthermore, we introduce a new class of larger randomly generated instances and reach optimal results for all of them. We test modified real-world instances obtained from the German company NetCologne (used for the augmentation of existing fiber optic networks). Even these large scale instances are successfully solved to provable optimality in less than 12 hours, which is still considered to be a reasonable running time for off-line network design problems.

Acknowledgments

First of all I want to thank my advisor Prof. Petra Mutzel for all the patience, motivation, and the time she dedicated to me. Petra introduced me into combinatorial optimization and polyhedral combinatorics, she involved me into the organization of various events and gave me the opportunity to travel to workshops and conferences all over the world. I am also very grateful to Prof. Ulrich Pferschy, whose comments and advice in all matters connected to this thesis are invaluable. During his visiting professorship in Vienna, we had many discussions during which I learned a lot about algorithms and about modeling of real-world problems, in general. I want to express sincere appreciation to Petra and Uli for their most valuable advice, criticism, encouragement and support.

I also want to thank Jozef Kratica and Prof. Günther Raidl who introduced me into the field of evolutionary algorithms and whose work and ideas strongly influenced this thesis. As an adviser of my master thesis several years ago, Jozef drew my interests to the field of evolutionary algorithms. Working together with Günther during my first years in Vienna, in the framework of a project supported by the Austrian Science Fund, I learned much of what I know about memetic algorithms. Günther is a co-author on three papers in which the main results of this thesis are published.

I also owe gratitude to all of my colleagues from the Algorithms and Data Structures Group of the Vienna University of Technology. The group seminars helped me to clarify my thoughts and gave me many valuable ideas. I really enjoyed fruitful discussions, in particular with Gunnar Klau and René Weiskircher, who are co-authors on two papers related to the prize-collecting Steiner tree problem (PCST), that led to some very important results of this thesis. Thanks to René for his contribution in implementing the primal heuristic for the PCST. It was pleasure and fun to advice practical works and diploma thesis of Andreas Moser, Philipp Neuner and Sandor Kersting. Their work contributed to the computational studies of this thesis. Andy, Gunnar, Philipp and René also helped a lot in making the four years of studying and teaching an enjoyable experience. I want to thank Martin Gruber and Philipp Neuner for their quick response whenever something went wrong with our computer systems.

Many thanks to Prof. Michael Jünger for providing the implementation of the minimum-cut algorithm and the framework for the sparse and reserve graphs pricing. Thanks to Prof. Matteo Fischetti for helpful and enlightening discussions related to the PCST. Thanks to An Zhu for providing the generator of vertex biconnectivity augmentation benchmark instances.

Thanks to Gunnar Klau, Jozef Kratica, Dragoslav Ljubić, Jakob Puchinger, Günther Raidl and René Weiskircher, who proofread parts of my thesis and made many valuable suggestions.

Further, I like to thank the Austrian Academy of Sciences for their financial support in the framework of the Doctoral Scholarship Program (DOC), to the Austrian Science Fund and also to the IEEE Computational Intelligence Society for their Student Summer Research Program support.

Very special thanks to my family: to my parents and my sister, to my husband and my children, who always supported me in all my decisions I made so far.

To Nani and Čedi,
who deserve my excuse
for all the time I spent playing with algorithms
instead of playing with them.

Contents

Chapter 1

Introduction

> *The genes are the master programmers, and they are programming for their lives. They are judged according to the success of their programs in copying with all the hazards that life throws at their survival machines, and the judge is the ruthless judge of the court of survival.*
>
> Richard Dawkins, "The Selfish Gene"

Network design problems occur frequently in various practical areas like e.g. in the design of communication networks, in the development of electronic circuits, in the design of fiber optic networks or in the development of district heating or water supply systems. One of the well-known network design problems is the minimum spanning tree problem (MST), in which all vertices of the network need to be connected at minimum cost. Other well-known examples for network design problems are the traveling salesman problem (TSP, finding a shortest tour visiting all vertices of a given network exactly once), or the minimum Steiner tree problem (connecting a given subset of vertices at minimum cost). All these problems are *combinatorial optimization problems* (COPs) – they search for values of discrete variables such that an optimal solution with respect to a given objective function is identified subject to some specific constraints emanating from a combinatorial structure. Although for some of the problems, like finding the MST, efficient algorithms are known, most of the COPs of practical interest are known to be *NP-hard* [63][1]. But also simple problems for which efficient polynomial algorithms are known, often become hard after adding new constraints. For example, the minimum spanning tree problem becomes NP-hard if only a limited number of edges may enter/leave each vertex [21].

The available techniques for COPs 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 for every instance of a COP. Due to sometimes exponential running times or memory requirements of exact algorithms, we are forced to use heuristic algorithms when

[1] No algorithm with a worst-case running time bounded by a polynomial in the size of the input is known for any NP-hard problem, and it is strongly believed that no such algorithm exists.

instance size exceeds a certain threshold value. Heuristics sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a limited time.

Some well known exact methods are branch-and-bound [17, pp. 485–490], [1], dynamic programming [32, pp. 323–356], Lagrangian relaxation based methods [129, pp. 323–337], and *cutting-plane* techniques based on linear programming [17, pp. 480–484]. In recent years enormous progress has been made in solving NP-hard problems with *integer (linear) programming* (ILP). Remarkable improvements have been reported for solving particular problems, like the traveling salesman problem [7], by ILP methods.

The ILP approaches considered in the scope of this thesis are *branch-and-cut* (BC) [114] and *branch-and-cut-and-price* (BCP) [92, 103] algorithms. These methods have been implemented in many mixed-integer optimizers such as ILOG CPLEX, XPRESS-MP, ABACUS, COIN, and nowadays they are the most effective exact algorithms for plenty of integer and mixed-integer programming problems (see, for example, [29, 146, 19, 96]).

In general, for problem instances of moderate size, ILP techniques are often able to yield provably optimal solutions. However, due to the NP-hard nature of the considered problems, computation time and memory requirements may increase exponentially with instance size. Hence, the ILP optimization often need to be be stopped prematurely. Since linear programming variables can take fractional values and the problems discussed above involve discrete quantities, making a decision halfway between *yes* and *no* does not make sense in a real-world decision context. Thus, prematurely terminated ILP techniques often yield only to fractional bounds without finding any feasible (for practice relevant) solution.

For large instances of NP-hard problems, the only possible way to get feasible solutions is to trade optimality for the running time and to tackle these instances with a heuristic which gives no guarantee of finding an optimum solution. Consequently, an enormous effort has been made in developing algorithms that find *nearly optimal* solutions in a reasonable amount of computing time [8]. These heuristics for combinatorial optimization problems can be separated into problem-specific algorithms and more or less problem-independent methodologies. Examples of modern problem-independent techniques are neighborhood search algorithms such as local search, variable-neighborhood search [75], tabu search [77], or simulated annealing [3], and biologically inspired methods like evolutionary algorithms (EAs) [120], scatter search [104], ant colony optimization [39], and artificial neural networks [133].

This thesis is focused on a particular class of metaheuristics: *memetic algorithms* [124]. The first use of the term memetic algorithms in the computing literature has appeared in 1989 in P. Moscato's paper [123]. While evolutionary algorithms are based on a crude simplification of natural evolution, memetic algorithm rely on the rules of socio-cultural evolution. About relationships between *genes* and *memes*, Cliff Joslyn and Valentin Turchin wrote[2]:

> *In biological evolution survival means essentially survival of the genes, not so much survival of the individuals. With the exception of species extinction, we may say that genes are effectively immortal: it does not matter that an individual dies, as long as his genes persist in its offspring.*

[2]Principia Cybernetica Web, `http://pespmc1.vub.ac.be/`

> *In socio-cultural evolution, the role of genes is played by memes, embodied in individual brains or social organizations, or stored in books, computers and other knowledge media. Thus the creative core of human individual is the engine of memetic evolution. In memetic evolution, memes must be immortal. While the mortality of multicellular organisms is necessary for biological evolution, it is no longer necessary for memetic evolution.*

From the computer science point of view, memetic algorithms incorporate some kind of domain knowledge into EAs to make them competitive to other problem specific optimization techniques. Mostly seen as hybrids of neighborhood search algorithms with evolutionary algorithms, memetic algorithms exploit the symbiotic effects of this combination. Neighborhood search algorithms are well-suited for the *exploitation* of the search space, while the evolutionary framework enables effective diversification (*exploration*). Over the last few years, memetic algorithms have shown their great capabilities in finding high quality solutions to difficult global optimization tasks [34, 20, 6].

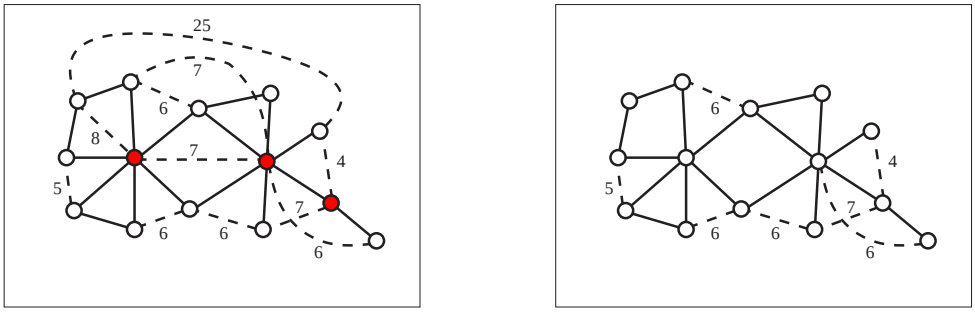
Summary of Obtained Results

Specific advantages of metaheuristics are that they can examine a large number of possible solutions in relatively short computation time and in many cases they are found to be the best performing algorithms for large practical problems [153, 35]. On the other hand, (meta)heuristics cannot prove optimality and they do not give tight quality guarantees for approximate solutions. The purpose of this thesis is to provide tools that can solve given network design problems to provable optimality, or, if this is not possible, to obtain suboptimal, high quality solutions and to provide optimality gaps as a measure of their quality.

We concentrate on two NP-hard network-design problems that can be modeled using integer linear programming: *minimum vertex-biconnectivity augmentation* (V2AUG) and *the prize-collecting Steiner tree problem* (PCST). For V2AUG and PCST we develop and investigate memetic algorithms and branch-and-cut methods, but we also explore some synergetic effects of their combination.

The memetic algorithms (MAs) that we propose for V2AUG and the PCST comprise new solution representation techniques, search operators, constraint handling techniques, local-improvement strategies, and heuristic biasing methods. Our exact algorithms are based on the state-of-the-art in polyhedral combinatorics. They rely on sophisticated separation algorithms or advanced column generation methods. In this thesis, we also investigate some possibilities of combining promising variants of exact algorithms and MAs, like incorporating exact algorithms that solve some special cases within MAs, biasing primal heuristics or guiding column generation using MA results.

The main results of this thesis related to V2AUG are published in [108]. Preliminary results appeared in [93]. We also developed a memetic algorithm for edge biconnectivity augmentation and published our results in [144]. Preliminary results appeared in [107].



(a) (b)

Figure 1.1: Vertex biconnectivity augmentation example: (a) An instance of the vertex biconnectivity augmentation problem – bold lines belong to the existing network (E_0), while dashed lines represent possible augmentation edges ($E \setminus \{E_0\}$). Shaded vertices are articulation points of the existing network; (b) A feasible solution of the problem, with augmentation costs 40.

In [97, 109] we published our most important results related to the prize-collecting Steiner tree problem. In [98] we consider a related problem, the so-called *fractional* prize-collecting Steiner tree problem.

Vertex Biconnectivity Augmentation

After some recent electrical power blackouts in the USA and in some European countries, it has become obvious that the survivability of networks plays an important role in the design of electrical power supplies. Redundant connections need to be established in the network to provide alternative routes in case of a temporary break down of one or more vertices. The simplest break down form appears when a failure of a single vertex disconnects the network. Such a vertex is called *articulation point*, and a network without articulation points is said to be *biconnected*. For every pair of vertices of a biconnected network, there exist at least two vertex-disjoint paths between them. The minimum-cost vertex biconnectivity augmentation problem consists of augmenting an already existing network $G_0 = (V, E_0)$ with edges from $A \subset E \setminus \{E_0\}$ of minimal total cost such that the network $G_A = (V, E_0 \cup A)$ is biconnected. This problem, which also arises in the design of communication and transportation networks, has been introduced by Eswaran and Tarjan [46] who have shown that it is NP-hard. Figure 1.1 illustrates an example.

Within this thesis, we first propose a deterministic preprocessing algorithm for reducing the search space. The algorithm follows the idea already given in [46] of generating a block-cut graph. We propose new preprocessing tests that shrink, fix or discard certain augmentation- or tree-edges. One of these tests, the so-called *edge elimination* represents an extension of a dynamic programming algorithm given by Frederickson and Jájá [56]. Although a theoretical upper bound for the computational costs of preprocessing is relatively high ($O(|V|^2|E|)$), our

computational results indicate that the algorithm is in practice very fast, even if large problem instances are considered.

We then propose a memetic algorithm for V2AUG with the following features: Our local improvement procedure guarantees local optimality with respect to the number of augmentation edges of any candidate solution. The proposed recombination, respectively mutation are specially designed to provide strong heritability and locality. We use biasing of initialization and recombination to make the inclusion of the low-cost edges more likely. Finally, we bias the mutation operator to remove more expensive edges with higher probability.

We also propose supporting data structures established during preprocessing that allow efficient implementations of initialization, recombination, mutation, and local improvement. Empirical results show that the approach scales well to instances of large size and calculates solutions that are usually significantly better than those of the other three heuristics known from the literature [95, 161, 106]. However, at this stage, we still do not know how far away they are from the optimal ones.

To be able to estimate the quality of obtained MA solutions, we develop a branch-and-cut algorithm that provides optimal values or, in case of exhausted computational resources, lower bounds that can be used to determine optimality gaps for MA solutions. The branch-and-cut algorithm relies on an integer programming formulation for the *survivable network design problem* (a generalization of V2AUG) given by Stoer [152]. Biconnectivity of a network is described through degree-constraints and an exponential number of biconnectivity-constraints. We initialize the root vertex of the branch-and-bound tree with simple degree constraints. Separation of violated vertex-biconnectivity constraints can be done exactly by applying the polynomial-time algorithm for finding the minimum-weight cut of a graph. Small and randomly generated problem instances can be solved exactly by using only the branch-and-cut method. For these instances, the exact approach is even faster than the proposed MA.

For solving larger instances to optimality, we investigate the incorporation of column generation into the branch-and-cut algorithm. For detection of inactive variables that should be priced in, we use the reserve graph technique proposed by Jünger et al. [89]. We also use special data structures for the fast calculation of reduced costs. We also show that the well-designed primal heuristics based on MA's initialization operator and biased by the last LP solution can further improve the quality of our algorithm. Our BCP algorithm relies on the MA, since it uses its high-quality solutions as starting solutions and initial bounds. Our computational results indicate that, using pricing, we can significantly improve the algorithm's performance. For instances of small and moderate size, finding high-quality upper bounds by means of the MA can slightly slow down the optimization. However, for large instances, it is advantageous to combine both approaches, in order to obtain small optimality gaps. Using a sophisticated separation procedure and a local improvement method as primal heuristics, we found optimal solutions for some complete graphs with more than 400 vertices.

Finally, we investigate the performance of the BCP algorithm if, instead of nearest neighbor graphs, MA solutions are used within pricing based on the reserve graph technique. The obtained results show that both approaches have similar performance and that none of them is significantly better than the other one in terms of running time. Our attempt to combine

memetic (or evolutionary, in general) with exact algorithms is part of pioneering work in this direction (see also [36, 105, 113, 151, 141, 49, 58], to mention some of them). They all together lead us to a better understanding of both, evolutionary and exact approaches. Finally, the pioneering work should help us to instantiate better interactions between these, so far independent, heterogenous streams.

The Prize-Collecting Steiner Tree Problem

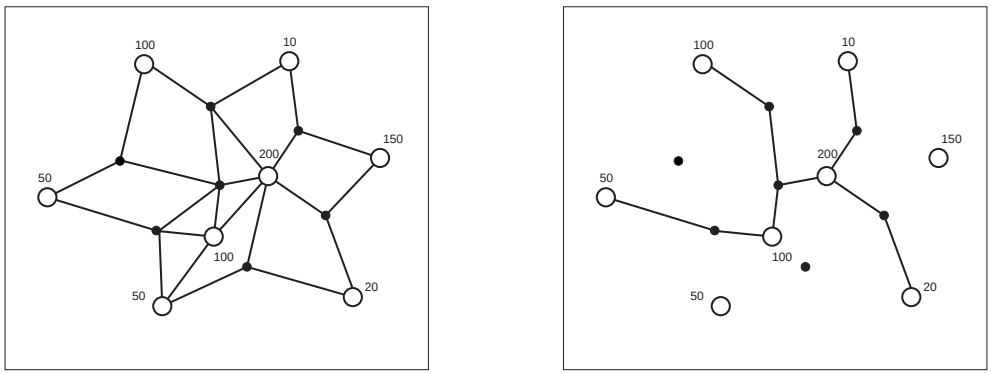
The recent deregulation of public utilities such as electricity and gas in Austria has shaken up the classical business model of energy companies and opened up the way towards new opportunities. Of particular interest in this field is the planning and expansion of district heating networks. This area of energy distribution is characterized by extremely high investment costs but also by an unusually loyal customer base and limited competition. Moreover, the required reduction of greenhouse emissions forces many energy companies to seek ways of improving their ecological balance sheet. A very attractive possibility to meet this goal is the use of biomass for heat generation. The combination of these two factors has made the planning of heating networks one of the major challenges for companies in this field [74].

In a typical planning scenario the input is a set of potential customers with known or estimated heat demands (represented by discounted future profits), and a potential network for laying the pipelines (which is usually identical to the street network of the district or town). Costs of the network are dominated by labor and right-of-way charges for laying the pipes and the costs for building the heating plant.

A similar problem appears in the design or augmentation of fiber optic networks: The wide expansion of fiber optic access networks (last mile) requires enormous financial resources. The according costs are mainly determined by the underground work (cable laying). Based on this fact, information about the relation between the investment volume and the corresponding return on investment represents a crucial competitive factor for new network or network-augmentation projects. The main research topic in this area is the optimization of cable laying routes for networks or network augmentation projects within urban areas.

Typically, a set of new households with estimated profits needs to be attached to an existing fiber optic network. The fiber may be laid down through the streets – in this case the costs of lying the fiber directly correspond to streets' length, but may vary depending on the importance or function of each particular street. The fiber can also be laid through public properties, in which case special costs need to be considered.

Essentially, in both network design problems mentioned above, the decision process faced by a profit oriented company consists of two parts: First, a subset of particular profitable customers has to be selected from a total set of all potential customers. Secondly, a network has to be designed to connect all selected customers in a feasible way – Figure 1.2 illustrates an example. The natural trade-off between maximizing the sum of profits over all selected customers and minimizing the cost of the network leads to a prize-collecting objective function. Given a network with prizes associated with its vertices and weights associated with its edges, the prize-collecting Steiner tree problem consists of finding a subtree of this network which



(a) (b)

Figure 1.2: The prize-collecting Steiner tree problem: (a) A network with customer and non-customer vertices (hollowed and bold circles, respectively). We suppose that all connections have a cost of 20; (b) A feasible solution of the problem.

minimizes the sum of the weights of its edges plus the prizes of the vertices not spanned by that tree. If there is a vertex that must be contained in the solution, we speak of *rooted* PCST.

Before applying optimization algorithms to PCST, we propose running a preprocessing procedure which is adopted from the work of Duin and Volgenant [41] for the related *node weighted Steiner tree problem*. The procedure requires $O(|E|^2|V| + |E||V|^2 \log |V|)$ time in the worst case, in which the input graph could be reduced to a single vertex. However, in practice, the running time is much lower which is documented in our results on benchmark instances from the literature.

We develop an efficient memetic approach based on a dynamic programming subroutine for the problem on trees that runs in linear time (see also [160, 84]). Furthermore, the algorithm uses efficient edge-set encoding and comprises efficient problem-dependent variation operators that all run in $O(|V| \log |V| + |E|)$ time. In the design of district heating or fiber optic networks, it is often the case that in small settlements the customers are grouped together, and that it either pays off to take all of them at once, or not to take any of them. By employing *clustering* as a grouping procedure within variation operators, we group subsets of vertices together and insert or delete them at once. For this purpose we use an algorithm proposed by Mehlhorn [115].

Our computational results document that the MA is competitive against the heuristic approach proposed by Canuto et al. [23] in terms of running time and quality of solutions. The average gap and its standard deviation indicate a stable performance and the reliability of our memetic algorithm.

To solve PCST instances to optimality within reasonable running times we choose a branch-and-cut approach. For the unrooted PCST, we insert an artificial root vertex and connect it to all customers. We propose the transformation of the original PCST problem into the so-called *Steiner arborescence problem*. We extend the ILP formulation given by Fischetti [51]

with new *asymmetry constraints*, and also with the *flow-balance* constraints proposed by Koch and Martin in [99]. The formulation is based on connectivity constraints that are separated by finding minimum-weight cuts between the root and every selected customer vertex. This separation algorithm runs in polynomial time. While the choice of the ILP model is essential for the success of our method, it should also be pointed out that solving the basic ILP model by a default algorithm is by no means sufficient to reach reasonable results. Indeed, our experiments show that a satisfying performance can be achieved only by appropriate initialization and strengthening of the original ILP formulation and in particular by a careful analysis of the separation procedure.

Using our ILP approach, we manage to solve to optimality (even without the usual preprocessing) all instances from the literature in a few seconds thereby deriving new optimal solution values and new certificates of optimality for a number of previously addressed problem instances. For these instances, the ILP approach is also significantly faster than the memetic algorithm itself.

We also tested real-world instances arising in the design of fiber optic networks and we created a number of new large instances constructed from Steiner tree instances. For solving all of them within reasonable running time, the preprocessing proves to be an indispensable tool which allowed us to find the optimum.

Finally, we propose a column generation algorithm as a lower bounding procedure to solve the multi-commodity flow (MCF) formulation of the PCST. As for V2AUG, we proposed to use best MA results within pricing in order to improve the algorithm's performance. Our comparison against two other pricing strategies shows that our new algorithm represents an advantageous approach.

Guide to the Thesis

Chapter 2 provides some basic terms and definitions from the areas of graph theory, memetic algorithms and exact ILP approaches. Moreover, we present generic evolutionary and branch-and-bound algorithms to solve combinatorial optimization problems.

We study vertex biconnectivity augmentation in Chapter 3. An overview on former approaches to V2AUG and related problems is given in Section 3.1. Within Section 3.2 we describe an efficient preprocessing procedure based on the derivation of a more compact *block-cut graph* from the problem's original graph. Section 3.3 is devoted to a memetic algorithm which searches for a low-cost solution on the reduced block-cut graph. The best solution found is finally mapped back to a solution for the original V2AUG instance. We provide an exhaustive experimental comparison of the new approach against other algorithms for V2AUG. In Section 3.4, we propose a branch-and-cut-and-price (BCP) algorithm that searches for optimum solutions on the block-cut graph. We first describe a simple branch-and-cut algorithm based on the minimum-cut ILP formulation of the problem. To enhance its performance, we propose the incorporation of the column generation method based on the sparse and reserve graph technique. In Section 3.5, we investigate possible ways how to use the knowledge about the problem obtained from running the MA, to improve the performance of the branch-and-

cut-and-price approach. We consider setting upper bounds by MA, biasing primal heuristic and guiding column generation using MA results. Conclusions are drawn in Section 3.6.

In Chapter 4, the problem of choosing a subset of potential customers and connecting them within a sub-network in order to maximize the profit is modeled as the prize-collecting Steiner tree problem. In Section 4.1 we give a short overview of previous work on PCST and some of its relatives. Preprocessing, which helps to significantly reduce the size of many instances, is treated in Section 4.2. In Section 4.3, we propose a MA used for finding approximate solutions for the prize-collecting Steiner tree problem. Extensive computational results are also provided. Different ILP models for PCST are presented and discussed in Section 4.4. In Section 4.4.3 we introduce our cut-based ILP model. In Section 4.5 we describe how to solve the cut-based ILP model in an efficient branch-and-cut framework. Extensive computational experiments are reported in Section 4.5.4. They include results on the cut-based formulation, but also some results obtained for the column generation approach applied to the multi-commodity flow formulation described in Section 4.4. We conclude this chapter with Section 4.6 where we discuss our results.

Finally, in Chapter 5 we draw some conclusions and present a few ideas for future research. We also provide definitions of some new problems arising in the design of fiber optic or district heating networks that represent natural extensions of V2AUG and PCST.

Chapter 2

Preliminaries

In this chapter we provide basic terms and definitions of graph theory needed to introduce two network design problems we are dealing with. Furthermore, principal concepts of evolutionary computation and memetic algorithms are introduced; for a comprehensive introduction to these fields, we refer to [120, 52, 11, 12]. Finally, we describe some basic ideas of integer linear programming, such as cutting planes, column generation and their incorporation within a branch-and-bound framework [160, 17].

2.1 Notation and Definitions

Given a finite set $\mathcal{I}$ of feasible solutions and a function $c : \mathcal{I} \mapsto \mathbb{R}$ (the *objective function*), a *combinatorial optimization problem* (COP) consists of finding an element I^* with

$$c(I^*) = \min\{c(I) \mid I \in \mathcal{I}\} \ .$$

Throughout this thesis, without loss of generality, we concentrate on minimization problems, since each maximization problem $\max\{c(I) \mid I \in \mathcal{I}\}$ can be trivially transformed into it.

The two combinatorial optimization problems we are concentrating on in the framework of this thesis, belong to the class of *subset selection problems* which are defined as follows:

Definition 1. [Subset Selection Problem]
Given are a finite set E, a set $\mathcal{I} \subseteq 2^E$ of subsets of E (the feasible solutions) and a function $c : E \mapsto \mathbb{R}$. For each set $F \subseteq E$ let $c(F) = \sum_{e \in F} c(e)$. A subset selection problem $(E, \mathcal{I}, c)$ consists of finding a subset $I^* \subseteq E$ with

$$c(I^*) = \min\{c(I) \mid I \in \mathcal{I}\} \ .$$

Most subset selection problems can also be modeled as integer linear optimization problems.

2.1.1 Linear Optimization

The goal of an *integer linear programming* (ILP) is to find an integer solution vector $x^* \in \mathbb{Z}^n$ such that:

$$c^T x^* = \min\{c^T x \mid Ax \geq b,\ x \in \mathbb{Z}^n\} \ , \tag{2.1}$$

where a matrix $A \in \mathbb{R}^{(m,n)}$ and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ are given. If all variables x_i, $1 \leq i \leq n$ are from the set $\{0,1\}$ only, we speak of the *zero-one linear programming* (0–1-ILP)[1].

The variables $x_1, \ldots, x_n$ are called *decision variables*, and a vector x satisfying all the constraints

$$a_i^T x \geq b_i, \quad i = 1, \ldots, m$$

expressed compactly in the form $Ax \geq b$, is called a *feasible solution* or *feasible vector*. The set of all feasible solutions is called the *feasible set* or the *feasible region*. A feasible solution x^* that minimizes the objective function (that is $c^T x^* \leq c^T x$, for all feasible x) is called an *optimal solution*, and the value $c^T x^*$ is called the *optimal cost*.

Many important graph problems can be stated as 0–1-ILP problems [126, 1, 2, 96]. In general, the ILP and also the 0–1-ILP are known to be NP-hard [63]. The computational difficulty arises mainly due to the integrality constraints $x_i \in \mathbb{Z}$ (respectively $x_i \in \{0,1\}$). If we relax these constraints to $x_i \in \mathbb{R}$ (respectively $0 \leq x_i \leq 1$), a linear program (LP) called the *LP-relaxation* of the ILP is obtained. This LP can usually be solved efficiently by means of e.g. the simplex algorithm. Although in general the solution of the LP-relaxation does not directly allow for deriving the solution of the ILP, it may significantly help in finding it.

In this thesis, we will also consider the *dual* of an LP. With every linear program (P) (*primal linear program*) of the form

$$c^T x^* = \min\{c^T x \mid Ax \geq b,\ x \geq 0\} \ , \tag{2.2}$$

we associate a *dual linear program* (D) which consists of finding a vector $y^* \in \mathbb{R}^n$ such that:

$$b^T y^* = \max\{b^T y \mid A^T y \leq c,\ y \geq 0\} \ . \tag{2.3}$$

An important relation between the primal and the dual linear program is given by the following two theorems.

Theorem 1. [Weak Duality]
If x is a feasible solution to the primal problem (P) and y is a feasible solution to the dual problem (D), then

$$b^T y \leq c^T x \ .$$

The weak duality theorem gives rise to the following corollaries:

- If the optimal costs of P are $-\infty$, then the dual problem is infeasible.
- If the optimal costs of D are $+\infty$, then the primal problem is infeasible.

[1] In the *mixed integer linear programming* (MIP) , we consider not only integer but also real-valued variables. Our goal is to find a vector $(x^*, z^*) \in \mathbb{Z}^{n-k} \times \mathbb{R}^k$ such that

$$c_x^T x^* + c_z^T z^* = \min\{c_x^T x + c_z^T z \mid A_x x + A_z z \geq b,\ x \in \mathbb{Z}^{n-k}, z \in \mathbb{R}^k\} \ .$$

Theorem 2. [Strong Duality]
If a linear programming problem has an optimal solution, so does its dual, and the respective optimal costs are equal.

The *complementary slackness* conditions further describe the relation between primal and dual optimal solutions. They are presented within the next theorem.

Theorem 3. [Complementary Slackness]
Let x and y be feasible solutions to the primal and the dual problem, respectively. The vectors x and y are optimal solutions for the two respective problems if and only if:

$$\begin{aligned} y_i(a_i^T x - b_i) &= 0, \quad \forall i \ , \\ (c_j - y^T A_j)x_j &= 0, \quad \forall j \ , \end{aligned}$$

where A_j denotes the j-th column of the matrix A.

For a feasible solution x of a primal problem (P), a constraint $a_i^T x \geq b_i$ is called *active* at x if $a_i^T x = b_i$. The first complementary slackness condition asserts that the corresponding dual variable y_i is zero unless the constraint is active.

2.1.2 Linear Programming vs. Integer Combinatorial Optimization

In what follows, we describe the polyhedral ties between linear programming and integer combinatorial optimization. For $d_1, d_2, \ldots, d_k \in \mathbb{R}^n$ and a vector $\lambda \in \mathbb{R}^k$, the sum

$$d = \sum_{i=1}^{k} \lambda_i d_i$$

is called the *linear combination* of points $d_1, d_2, \ldots, d_k$. Additionally, if:

- $\lambda_i \geq 0, \forall i$, we speak of *conic combination*, and
- $\sum_{i=1}^{k} \lambda_i = 1$, we speak of *affine combination*, and
- $\sum_{i=1}^{k} \lambda_i = 1, \lambda_i \geq 0, \forall i$, we are dealing with a *convex combination* of points $d_1, d_2, \ldots, d_n$.

Given a finite set of points $S \subset \mathbb{R}^n, S \neq \emptyset$, a *convex* (*affine*, *conic*) *hull* of S, notated as $\text{conv}(S)$ ($\text{aff}(S)$, $\text{cone}(S)$), is defined as the set of all points in $\mathbb{R}^n$ which can be represented as a convex (affine, conic) combination of points from S.

$S \subset \mathbb{R}^n$ is an *affine subspace* of $\mathbb{R}^n$ if and only if there exists a matrix $A \in \mathbb{R}^{m \times n}$, a vector $b \in \mathbb{R}^m$, such that $S = \{x \in \mathbb{R}^n \mid Ax = b\}$.

Hyperplanes and *half-spaces* play an important role in linear programming. Let a be a non-zero vector in $\mathbb{R}^n$, and let b be a scalar. The set

$$\{x \in \mathbb{R}^n \mid a^T x = b\}$$

is called a *hyperplane*, while the set

$$\{x \in \mathbb{R}^n \mid a^T x \geq b\}$$

is called a *half-space*.

A set of vectors $S = \{x_1, x_2, \ldots, x_k\} \subset \mathbb{R}^n$ is called *affine independent* if

$$\sum_{i=1}^{k} \lambda_i x_i = 0 \ \wedge \ \sum_{i=1}^{k} \lambda_i = 0 \Rightarrow \lambda_i = 0, \forall i = 1, \ldots, k \ .$$

The *affine rank* of a set $S \subset \mathbb{R}^n$ is defined as follows:

$$\text{affrank}(S) = \max\{|T| \mid T \subset S \text{ is affine independent}\} \ .$$

The *dimension* of a set $S \subset \mathbb{R}^n$ is then

$$\dim(S) = \text{affrank}(S) - 1 \ .$$

Definition 2. [Polyhedron]
A polyhedron *is a set that can be described in the form* $\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax \geq b\}$*, where* A *is a matrix from* $\mathbb{R}^{m \times n}$*, and* b *is a vector from* $\mathbb{R}^m$*. The polyhedron* $\mathcal{P}$ *is bounded, if there exists* $w \in R$ *such that* $\mathcal{P} \subset \{x \in \mathbb{R}^n \mid -w \leq x_i \leq w, \forall i = 1, \ldots, n\}$*. A bounded polyhedron is called a* polytope*.*

A classical result in polyhedral theory is the theorem of Minkowski and Weyl (see, for example, [148]), saying that each polyhedron $\mathcal{P} \in \mathbb{R}^n$ can be written as $\mathcal{P} = \text{conv}(X) + \text{cone}(Y)$, where $X \subset \mathbb{R}^n$ and $Y \subset \mathbb{R}^n$ are finite sets of points. In other words, polyhedra are sums of convex and conic hulls of finite subsets in $\mathbb{R}^n$. Thus, there always exist two representations of a polyhedron:

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax \geq b\} = \text{conv}(X) + \text{cone}(Y) \ .$$

Being a bounded polyhedron, each polytope can be presented as a convex hull of a finite subset of points $X \subset \mathbb{R}^n$:

$$\mathcal{P} = \text{conv}(X) \ .$$

Consider now a subset selection problem $(E, \mathcal{I}, c)$ with associated linear objective function c. Given a finite set E and a subset $I \subset E$, the *incidence vector* $h_I \in \mathbb{R}^E$ is given by:

$$h_I(e) = \begin{cases} 1, & \text{if } e \in I \\ 0, & \text{otherwise} \ . \end{cases}$$

With $(E, \mathcal{I}, c)$, we associate the polytope

$$\mathcal{P}_{\mathcal{I}} = \text{conv}\{h_I \mid I \in \mathcal{I}\} \ ,$$

i.e. the convex hull of the *incidence vectors* of all feasible sets $I \in \mathcal{I}$. Note that polytope $\mathcal{P}$ does not depend on the cost function $c : E \mapsto \mathbb{R}$, but if we associate a vector $c \in \mathbb{R}^E$ to it, we can solve the original problem $(E, \mathcal{I}, c)$ by solving

$$\min\{c^T x \mid x \in \mathcal{P}_{\mathcal{I}}\} \ .$$

Using a finite set of inequalities, a so-called *linear description* of the polytope $\mathcal{P}$ [148]:

$$\mathcal{P}_{\mathcal{I}} = \{x \in \mathbb{R}^E \mid Ax \geq b\} \ ,$$

we transform the starting combinatorial optimization problem into the linear program given by:

$$\min\{c^T x \mid x \in \mathbb{R}^E, Ax \geq b\} \ .$$

In order to solve $(E, \mathcal{I}, c)$ over a polytope, we need a formulation which may involve large numbers of variables or constraints (size of matrix A and vector b) increasing exponentially with the problem's size. For NP-hard optimization problems, complete linear description of the underlying polytope can not be found. In practice however, by using methods of *polyhedral combinatorics* (see Section 2.6), we are able to solve some of the COP instances even when dealing only with a small subset of these inequalities. The most important role play the facet defining inequalities, defined as follows.

Definition 3. [Valid Inequalities]
Given a polytope $\mathcal{P}_{\mathcal{I}} = \{x \in \mathbb{R}^n \mid Ax \leq b\}$, an inequality $f^T x \leq f_0$ is called valid *for $\mathcal{P}_{\mathcal{I}}$, if $f^T \overline{x} \leq f_0$ holds for all $\overline{x} \in \{x \in \mathbb{R}^n \mid Ax \leq b\}$.*

Definition 4. [Facet Defining Inequality]
If $f^T x \leq f_0$ is a valid inequality with respect to the polytope $\mathcal{P}_{\mathcal{I}} \subset \mathbb{R}^n$ and the intersection of the $(n-1)$-dimensional affine subspace $H = \{x \mid f^T x = f_0\}$ with $\mathcal{P}_{\mathcal{I}}$ is neither empty nor equals $\mathcal{P}_{\mathcal{I}}$, then $F = \mathcal{P}_{\mathcal{I}} \cap H$ is called a face *of $\mathcal{P}_{\mathcal{I}}$ defined by the valid inequality $f^T x \leq f_0$. Let $s = \dim(\mathcal{P}_{\mathcal{I}})$ be dimension of the polytope $\mathcal{P}_{\mathcal{I}}$. The $(s-1)$-dimensional faces are called* facets *of $\mathcal{P}_{\mathcal{I}}$. If $F = \mathcal{P}_{\mathcal{I}} \cap \{x \mid f^T x = f_0\}$ is a facet of $\mathcal{P}_{\mathcal{I}}$, the inequality $f^T x \leq f_0$ is called* facet defining inequality *for $\mathcal{P}_{\mathcal{I}}$.*

2.1.3 Cuts and Flows

Throughout this work, we concentrate on *simple graphs*, i.e. on graphs without parallel edges or self-loops. If there exists an edge $e = \{i, j\}$ (denoted also with $e = (i, j)$) between two vertices i and j, these two vertices are called *adjacent*, and e is *incident* to i and j. With $n = |V|$ and $m = |E|$ we will denote the number of vertices and edges of G, respectively. In a *directed graph* $G = (V, A)$, we have directed edges, called *arcs*; (i, j) describes an edge leading from vertex i (the so-called *source*) to vertex j (the so-called *target*).

In a *weighted* graph $G = (V, E, c)$, an *edge-weight function* $c : E \mapsto \mathbb{R}$ is associated to the set of edges. Sometimes we write $c(i, j)$ also for undirected graphs, when it is clear from context that we are dealing with the cost of an undirected edge $c(\{i, j\})$.

Given the undirected graph $G = (V, E)$ and a subset $W \subset V$, the edge set

$$\delta(W) = \{\{i, j\} \in E \mid i \in W, j \in V \setminus W\}$$

is called the *undirected cut* induced by W. We write $\delta_G(W)$ to make clear – in case of possible ambiguities – with respect to which graph the cut induced by W is considered.

Similarly, in a directed graph, we denote with

$$\delta^-(W) = \{(j,i) \in A \mid i \in W, j \in V \setminus W\}$$

and

$$\delta^+(W) = \{(i,j) \in A \mid i \in W, j \in V \setminus W\}$$

the *ingoing* and *outgoing cuts* induced by W, respectively.

The *degree* of a vertex v, in notation $\deg(v)$, is the cardinality of $\delta(v) = \delta(\{v\})$. Similarly, we define the *ingoing* and *outgoing* degrees $\deg_{in}(v)$ and $\deg_{out}(v)$ of v as the cardinalities of $\delta^-(v)$ and $\delta^+(v)$, respectively. We denote by $V - v = V \setminus \{v\}$ and $E - e = E \setminus \{e\}$ the subsets obtained by removing one vertex or one edge from the set of vertices or edges. $G - v$ denotes the graph $(V - v, E - \delta(v))$ and $(V - v, E - \delta^+(v) - \delta^-(v))$ in the undirected and directed case, respectively.

The most interesting value about a cut is its *weight* (or *capacity*): the total capacity of all the edges in the cut. We denote it as

$$c(\delta(W)) = \sum_{e \in \delta(W)} c(e) \ .$$

A *flow* is a mathematical formulation of how fluids, or electrical circuits can move from special selected vertices, called *sources*, to so-called *targets* (or *sinks*), without violating capacity constraints. Here, the entity we are most interested in, is the *value of the flow*, i.e. the total amount of flow that reaches the sinks. Without loss of generality, throughout this thesis we are going to concentrate on the single source-single target flow values.

One of the fundamental results in combinatorial optimization is the duality between the flow value and the cut capacity in networks.

Theorem 4. Min-cut Max-flow [Ford & Fulkerson [54]]
The value of the maximum flow in the undirected weighted graph $G = (V, E, c)$ is equal to its minimum cut capacity.

A straightforward algorithm for finding the minimum weight cut of a graph $G = (V, E, c)$ with n vertices and m edges is the computation of the minimum s-t-cuts between an arbitrarily fixed vertex s and each other vertex $t \in V \setminus \{s\}$. From these $n - 1$ cuts, one with minimum weight represents the global minimum cut. Gomory and Hu proposed a more elaborate algorithm which employs a vertex shrinking operation so that the $n - 1$ minimum s-t-cuts have to be computed in smaller graphs. The worst-case running time of their algorithm is $O(n^2 m)$. Nagamochi and Ibaraki [128] showed how to find a minimum cut without using maximum flow calculations. The algorithm runs in $O(nm + n^2 \log n)$ time. Hao and Orlin [76] used the flow approach by showing that a clever modification of the Gomory-Hu algorithm implemented with a push-relabel maximum flow algorithm runs in time asymptotically equal to the time needed to compute one s-t-flow: $O(nm \log(\frac{n^2}{m}))$. Jünger et al. [90] provided a brief overview of the most important algorithms for the minimum capacity cut problem. They compared these methods both with problem instances from the literature and with problem instances originating from the solution of the traveling salesman problem by branch-and-cut.

2.1.4 Graph Connectivity

We will use the following definitions arising from graph connectivity theory. For any pair of distinct vertices $s, t \in V$, an undirected (directed) $[s,t]$-*path* P is a sequence of vertices and edges (arcs) $(v_0, e_1, v_1, e_1, \ldots, v_{l-1}, e_l, v_l)$, $((v_0, a_1, v_1, a_1, \ldots, v_{l-1}, a_l, v_l))$, where each edge (arc) e_i (a_i) is incident to the vertices v_{i-1} and v_i $(i = 1, \ldots, l)$, where $v_0 = s$ and $v_l = t$, and where no edge or vertex appears more than once in P. We call the vertices $v_i, i = 1, \ldots, l-1$ *inner vertices* of the path P, while v_0 and v_l are its *end vertices*.

If for any two vertices $i, j \in V$ of a graph $G = (V, E)$ an $[i,j]$-path exists, the graph is said to be *connected*, otherwise it is *disconnected*. A maximal connected subgraph of G is a *component* of G.

If G is connected, and $G - W$ is disconnected, where W is a set of vertices or a set of edges, than we say that W *separates* G.

Definition 5. [k-Connectivity]
A graph G is vertex (edge) k-connected *($k \geq 2$), if it has at least $k+2$ vertices and no set of $k-1$ vertices (edges) separates it. The maximal value of k for which a connected graph G is k-connected is the* connectivity *of G. For $k = 2$, graph G is called* biconnected.

If $G - e$ has more connected components than G, we call edge e a *bridge*. Similarly, if W is a vertex set such that $G \setminus W$ has more connected components than G, set W is called *articulation set*. If $W = \{v\}$, the vertex v is called *articulation* or *cut vertex*.

By $G[W]$, we denote a subgraph of G induced by W, i.e. $G[W] = (W, E[W])$, where $E[W] = \{\{i,j\} \in E \mid i, j \in W\}$.

A collection $P_1, P_2, \ldots, P_k$ of $[s,t]$-paths is called *edge-disjoint* if no edge appears in more than one path and is called *vertex-disjoint* if no vertex (other than s and t) appears in more than one path. A *cycle* is the union of two vertex-disjoint $[s,t]$-paths.

The following theorem represents a fundamental result in the theory of graph connectivity:

Theorem 5. [Menger's theorem]
A graph $G = (V, E)$ is k-edge-connected *(k-*vertex-connected*) if, for each pair s, t of distinct vertices, G contains at least k edge-disjoint (vertex-disjoint) $[s,t]$-paths.*

Note: While vertex k-connectivity implies edge k-connectivity, the reverse does not hold in general. We always assume vertex-connectivity, when others is not specified.

In what follows, we provide some further definitions we need.

A *forest* is an undirected cycle-free graph. A *tree* is a connected forest. An *arborescence* is a directed tree in which no two arcs are directed into the same vertex. The *root* of an arborescence is the unique vertex that has no arcs directed into it. A *branching* is defined as a directed forest in which each tree is an arborescence. A *spanning tree* (*spanning arborescence*) is a tree (arborescence) that includes every vertex in the graph.

A *minimum outgoing spanning arborescence* (MOSA) of a weighted directed graph $G = (V, E, c)$, $(c : E \mapsto \mathbb{R}^+)$ with a fixed root $r \in V$ is a spanning arborescence $T = (V, E_T)$ of G that minimizes $c(T) = \sum_{a \in E_T} c(a)$.

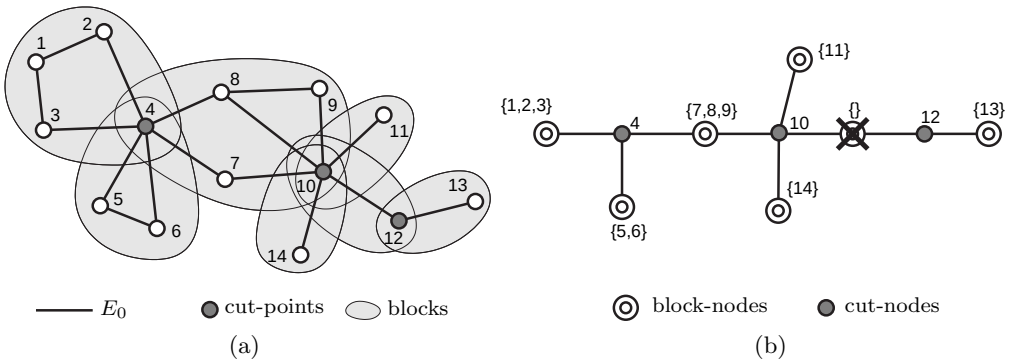


Figure 2.1: (a) A connected, but not vertex-biconnected graph $G = (V, E)$ and (b) the corresponding block-cut tree $T = (V_T, E_T)$.

Using the algorithm described in [60], the MOSA of a connected graph G can be found efficiently in $O(|V| \log |V|)$ time.

Definition 6. [Least Common Ancestor]
Given an arborescence $T = (V, E_T)$ with the root r, the least common ancestor of a pair of vertices $u, v \in V$, $u, v \neq r$ in notation $\mathrm{lca}(u, v)$ *is the first vertex that $[u, r]$- and $[v, r]$-paths have in common.*

2.1.5 The Block-Cut Graph

All maximal subgraphs of a graph G that are vertex-biconnected, i.e. the *vertex-biconnected components*, are referred to as *blocks*. If graph G is vertex-biconnected, the whole graph represents one block. Otherwise, any two blocks of G share at most a single vertex, and this vertex is a cut-point; its removal would disconnect G into at least two components.

A *block-cut tree* $T = (V_T, E_T)$ with vertex set V_T and edge set E_T is an undirected tree that reflects the relations between blocks and cut-points of graph G in a simpler way [46]. Figure 2.1b illustrates this. Two types of vertices form V_T: cut-vertices and block-vertices. Each cut-point in G is represented by a corresponding cut-vertex in V_T, each maximal vertex-biconnected block in G by a unique block-vertex in V_T.

A cut-vertex $v_c \in V_T$ and a block-vertex $v_b \in V_T$ are connected by an undirected edge (v_c, v_b) in E_T if and only if the cut-point corresponding to v_c in G is part of the block represented by v_b. Thus, cut-vertices and block-vertices always alternate along any path in T. The resulting structure is always a tree, since a cycle would form a larger vertex-biconnected component, and thus, the block-vertices would not represent maximal biconnected components.

A block-vertex is associated with all vertices of the represented block in G excluding cut-points. If the represented block consists of cut-points only, the block-vertex is not associated with any vertex from V. Thus, each vertex from V is associated with exactly one vertex from V_T, but not vice-versa.

In contrast to the previous definition of the block-cut tree according to [46], we apply here the following simplification: Block-vertices representing blocks that consist of exactly two cut-points are redundant in our approach and are therefore removed; a new edge directly connecting the two adjacent cut-vertices is included instead. In Figure 2.1b, the block-vertex labeled "{}" is an example.

The computational effort for deriving the block-cut graph is linear in the number of edges and the number of vertices of the original graph G. Indeed, for a connected graph G, when using a modified depth-first search algorithm (see, for example, [32, pp. 552–557]), all maximal biconnected subgraphs can be found in $O(|E|)$ time since each edge needs to be considered only once.

2.2 Evolutionary Algorithms

```
t ← 0;
Initialization(P(0));
Evaluation(P(0));
while not termination criterion do
    P' ← Selection(P(t));
    Recombination(P');
    Mutation(P');
    Evaluation(P');
    P(t + 1) ← Replacement(P(t), P');
    t ← t + 1;
end
```

Algorithm 1: Generic evolutionary algorithm.

Natural evolution, from the information science point of view, can be regarded as a huge information processing system. Each organism carries its genetic information referred to as the *genotype*, which can be thought of as the construction plan of an organism. The organism's traits, which are developed while the organism grows up, constitute the *phenotype*. If the organism reproduces before it dies, the genetic information will be passed on to the next generation. Thus, the organisms can be regarded as the "mortal survival machines of the potentially immortal genetic information" [116]. Natural evolution implicitly causes the adaptation of life forms to their environment since only the fittest have a chance to reproduce ("survival of the fittest").

Since the early 60's, several computer scientists independently studied new algorithms based on the idea of solving engineering problems by simulating natural evolution processes. Although these imitations represent crude simplifications of biological reality, these mimicked search processes of natural evolution yielded robust optimization algorithms. In the '90s, an increasing interaction among the researchers of *genetic algorithms* [79], *genetic programming* [101], *classifier systems* and *evolutionary strategies* took place. The boundaries between these methods

were broken down to some extent and *evolutionary algorithms* (EAs) have been developed that combine advantages of all these approaches.

A general template of an EA is shown in Algorithm 1. Evolutionary algorithms are based on the collective learning process within a population of *individuals*, each of which represents a search point I in the space $\mathcal{I}$ of potential solutions to a given problem. Within the population, several copies of the same individual may appear. We assume that the current population contains μ individuals, i.e. $P(t) = \{I_1, \ldots, I_\mu\}$. Generations are indexed by $t \in \mathbb{N}$, i.e. $P(t)$ denotes the t-th generation. The *initialization* procedure generates, usually at random, the population $P(0)$, the origin of the evolutionary search.

The following four consecutive steps make an iteration of the evolutionary search: the evaluation of the offspring (through its fitness), the parent selection, the variation, and the generational replacement. An objective function evaluates search points, and the *selection* process favors those individuals with better objective values to reproduce more often than worse individuals. The *variation* mechanism allows the mixing of parental information (*crossover* or *recombination*) and introduction of innovation into the population (*mutation*). In the sequel, we shortly describe basic concepts of these steps. For a general introduction into evolutionary algorithms we refer to [12, 120, 52, 11].

The process is stopped when some termination criterion is met, e.g.: the algorithm could not improve on the overall best solution during a certain number of generations; a given time limit is reached; or a satisfactory solution is found.

2.2.1 Encoding

The first step in designing an EA for a particular problem is to devise a suitable representation scheme, i.e. the *encoding*. By means of an encoding technique, we map the candidate solutions from the so-called *phenotype space*, into the so-called *genotype space*.

Each point in the search space (i.e. candidate solution) is represented by a *chromosome* where all parameters that describe the solution are stored in the encoded form. The chromosomes consist of *genes*: each gene takes its value from a finite set of possible values.

The most traditional encoding in genetic algorithms is *binary encoding*, in which a solution is represented as a binary string. In case of subset selection problems, this string may correspond to the characteristic vector of a solution.

There also exists a spectrum of enhanced problem-dependent encoding techniques, like the *ordinal representation* (proposed for the traveling salesman problem in [69]), the *permutation based encoding* (proposed for the knapsack problem in [78]), or the relatively general *weight-coding* [139]. Raidl and Julstrom [143] proposed representing spanning trees in EAs for network design problems directly as sets of their edges, the so-called *edge-set encoding*.

2.2.2 Fitness Evaluation

Evolutionary algorithms can be seen as optimization algorithms trying to maximize a *fitness function* defined on the search space. The fitness function is usually given by the objective function of the underlying problem, thus, for each problem, the fitness function needs to be

defined individually. For combinatorial optimization problems variation operators may not always produce feasible solutions. It is then necessary to run a *repair algorithm* [120] before the fitness of a solution is evaluated. An alternative approach for handling infeasible solutions is *penalization* [12], where the fitness function is modified by adding a penalty term to the objective function.

2.2.3 Selection

If the selection pressure of an EA is to high, good individuals are selected too often for mating (the so-called *super-individuals*), the diversity of population decreases and the EA usually converges to a local optimum. If the selection pressure is too low, good individuals are almost never favored, the whole approach degenerates to a random search and the EA converges very slow or does not converge at all.

There are two options in EAs to control selection pressure: *parent selection* and *replacement scheme*. In the parent selection, a set P' of parent individuals is selected from the current generation $P(t)$. The role of the parent selection is to favor individuals with high fitness in order to focus the evolutionary search on the promising parts of the search space.

The *fitness-proportional* selection is related to traditional genetic algorithms [79, 67]. The probability of selecting an individual I_i from the population $P(t)$ is given by:

$$p(I_i) = \frac{f(I_i)}{\sum_{I_j \in P(t)} f(I_j)} \ ,$$

where $f(I_j)$ denotes the fitness of the solution I_j. Realization of this selection is usually done by *roulette wheel sampling* [67]: the selection can be seen as spinning a roulette wheel, on which each solution has a slot sized in proportion to its fitness.

In *k-tournament selection* ($k \geq 3$), independent k-tournaments are performed in the following way: k individuals are randomly drawn from $P(t)$ and the best drawn individual deterministically wins the tournament. The tournament selection can be performed *with* or *without replacement*, i.e. drawn individuals can be returned back into the selection pool of potential parents, or can be selected at most once, respectively.

2.2.4 Replacement

The second possibility to induce selection pressure is by using the replacement scheme.

Generational replacement [79, 67] represents the simplest scheme, which has been commonly used in traditional genetic algorithms together with fitness-proportional selection. According to this scheme, the whole population $P(t)$ is replaced by the offspring population P'. This simplest technique does not consider fitness, and therefore does not induce selection pressure.

Steady-state replacement [155] tries to overcome the drawback of the generational replacement. The number of children produced by variation is smaller than the number of parents, thus, additional strategy is needed which decides about the parents to be replaced. By using

elitism strategy, the best individuals always survive to the next generation. *Duplicate elimination* method assures that the children identical to a parent are not included in the new generation.

2.2.5 Variation

While parent selection and replacement can be defined without knowledge of the underlying problem, mutation and recombination strongly depend on the encoding of the candidate solutions. For different kind of problems and representations, different variation operators have been proposed. A survey on the most successful variation operators can be found in [12]. In the following, classical operators for binary encodings usually used in genetic algorithms will be described.

Crossover A crossover operator should be designed with the aim to provide highest possible *heritability*, i.e. an offspring should have as many common properties to its parents as possible.

The traditional *one-point crossover* operator [79] works by cutting two parental bit strings at a randomly selected cutting point p. The head of the first (second) is then connected to the tail of the second (first) chromosome. There are also generalizations of the one-point crossover: *two-point* and *multi-point crossover* proposed by Goldberg [67] and *uniform crossover* proposed by Syswerda [154].

For the advanced evolutionary algorithms, when problem knowledge is incorporated into the crossover operator, we rather call it *recombination* to stress the difference between advanced and standard crossover operators which are usually related to binary encoding.

Mutation Mutation is typically applied to an offspring generated by crossover before the evaluation of the fitness. However, in genetic algorithms, mutation operators are often used as "background operators" to add a source of diversity aimed to prevent a premature convergence.

The *standard bit-flip mutation*, employed in classical genetic algorithms [79, 67], flips independently all bits of a string I with a certain small probability p_{mut}. The parameter choice $p_{\text{mut}} = 1/|I|$ ($|I|$ is the length of the bit string I) is frequently used [11].

2.2.6 Hybrid Evolutionary Algorithms

For many combinatorial optimization tasks, it has been shown that it is essential to incorporate some form of domain knowledge into EAs to yield effective optimization tools. *Hybrid evolutionary algorithms* usually represent an incorporation of local-search or greedy heuristic or repair heuristic (when variation operators produce infeasible solution), into traditional evolutionary algorithms. They can be divided into two groups [12]:

- Algorithms that exploit the *Baldwin effect* are based on the following idea: Before the fitness of a solution is evaluated, the heuristic is applied. The fitness is evaluated after the improvement/reparation, but the changes made by the heuristic are not saved in the

individual. That way, learned traits during lifetime of the parents are not inherited by their offspring.

- Opposite to the former algorithms, in algorithms based on *Lamarckian evolution*, the acquired traits of an organism influence its genetic code. Although the theory of Lamarck (1809) has been shown wrong in biology with the publication of Darwin's work, Lamarckian approach has its advantages in optimization: in most hybrid algorithms, the individuals are altered by the variation operators using improvement/repair heuristics.

2.3 Local Search

Data : A feasible solution I of the problem $(\mathcal{I}, c)$.
Result: A locally optimal solution I with respect to neighborhood N.
repeat
  generate neighboring solution $I' \in \mathcal{N}(\mathcal{I})$;
  if $c(I') < c(I)$ **then**
    $I \leftarrow I'$;
  end
until $\forall I' \in \mathcal{N}(I)\ c(I) \leq c(I')$;

Algorithm 2: Generic local-search algorithm.

Local search (LS) is the basis of many improvement heuristics for combinatorial optimization problems. It is a simple iterative method for searching a *neighborhood* of a current solution I. Algorithm 2 shows an example of a generic local search algorithm. Suppose that a problem instance is defined by the pair $(\mathcal{I}, c)$, where $\mathcal{I}$ denotes the discrete search space, and c represents the objective function. A neighborhood $\mathcal{N}$ for the problem instance $(\mathcal{I}, c)$ is given by a mapping $\mathcal{N} : \mathcal{I} \mapsto 2^{\mathcal{I}}$. $\mathcal{N}(I)$ contains all the solutions that can be reached from I by a single *move*. A move here is an operator which transforms one solution into another with small modifications. A solution I^* is called a *local minimum* of c with respect to the neighborhood $\mathcal{N}$ iff:

$$c(I^*) \leq c(I),\ \forall I \in \mathcal{N}(I^*)\ .$$

Thus, local search represents a procedure that minimizes the objective function c in a number of successive steps in each of which the current solution I is being replaced by a solution I' such that: $c(I') < c(I),\ I' \in \mathcal{N}(I)$.

There are different ways to conduct local search [158, 83, 75]. For example, *best improvement* performs in a greedy way: the current solution is always replaced with the best solution in the whole neighborhood. On the other side, *first improvement* accepts a better solution whenever it is found.

The time complexity of a certain local improvement procedure strongly depends on the size of the neighborhood and on the complexity of a single move. The larger the neighborhood, the

longer it takes to search it; however better local optima will be reached. The disadvantages of local search are obvious. As a neighborhood search based algorithm, it highly depends on the starting solution. Furthermore, the resulting solution is only locally optimal.

An unfavorable starting solution may lead to a local optimum with high objective value and thus high distance to the optimum with respect to both: the objective function and the neighborhood distance (i.e. the number of moves needed to reach the optimal solution). Population-based algorithms, like EAs, may overcome this drawback, if a large number of starting solutions is well distributed over the whole search space. The search is then focused in parallel on several promising regions of the search space.

The most-popular metaheuristic methods based on the local search are: *multi-start local search*, *iterated local search* [110], *variable neighborhood search* [75], *simulated annealing* [3], *tabu search* [77], and *GRASP* [150].

2.4 Memetic Algorithms

Memetic algorithms (MAs) can be seen as "evolutionary algorithms which intend to exploit all available knowledge of the underlying problem" [124] available in the form of greedy heuristics, approximation algorithms, local search, specialized recombination operators, or some other ways. Note that this incorporation of the domain knowledge is not an option – it represents a fundamental feature of MAs. In the following, the combination of EAs with local improvement or local search will be addressed, since this symbiosis has been shown to be very successful ([117, 102, 80]). That way, the exploration abilities of the evolutionary algorithm are complemented with the exploitation capabilities of local search procedures.

The similarities and the differences between hybrid evolutionary algorithms and MAs can be found argued in [124].

The memetic algorithm's basic structure used throughout this thesis is shown in Algorithm 3. In this scheme, all candidate solutions in the population always represent local optima, which is ensured by applying local improvement after a solution's creation and after application of the evolutionary variation operators. In each generation, with probability p_{cross}, one offspring is generated by applying a recombination operator on a pair of selected parents. Mutation is applied with probability p_{mut}. In the general case, unlike in standard EAs, recombination and mutation operators can be performed independently of each other. In our implementation, the population typically contains only different solutions and a newly created one replaces the worst from the population [142].

The following important aspects, among others, must be considered when designing MAs:

- Since local improvement is iteratively applied, it must be fast; the neighborhood must be chosen carefully – a neighborhood of a size larger than $O(n^2)$ (if n is the input size) is often too time-consuming.

- Problem specific heuristics that are used to create the initial population, but also the variation operators and local improvement, have to provide enough *diversity*, i.e. in general the created solutions should not be identical or too similar.

```
t ← 0;
Initialization(P(0));
for solution I ∈ P(0) do
    Local-Improvement(I);
end
repeat
    select two parents I_1, I_2 ∈ P(t);
    if random > p_cross then
        I ←Recombination(I_1, I_2);
        Local-Improvement(I);
    end
    if random > p_mut then
        Mutation(I);
        Local-Improvement(I);
    end
    if I ∉ P(t) then
        replace a solution in P(t) with I;
        t ← t + 1;
    end
until no new best solution found in the last Ω iterations;
```

Algorithm 3: A memetic algorithm representing a symbiosis of a steady-state evolutionary algorithm (with duplicate elimination) with a local improvement method.

- Recombination and mutation operators must be efficient and should be able to create only feasible solutions complying with the constraints of the target problem.
- Suitable balance between *exploration* and *exploitation* must exist: The search should on one side cover the whole search space, but on the other side should focus on the surroundings of already identified high quality solutions.

2.5 Fitness Landscapes

Fitness landscapes have been shown to be an important concept not only in the development of theory of evolution, but also in understanding and predicting the performance of evolutionary algorithms for particular combinatorial optimization problems [116, 119, 50].

A fitness landscape is defined by:

- the set of all possible solutions $\mathcal{I}$,
- an objective function that assigns to each $I \in \mathcal{I}$ a fitness value $f(I)$, and
- a *distance* measure $d(I, I')$ which is usually defined in correspondence to the used neighborhood as the minimum number of moves needed to transform I into I'.

Hence, the search space of an optimization problem can be viewed as a fitness landscape in which each point represents a candidate solution and where the height of the point in the landscape is determined by its solution quality (the fitness of the solution). Intuitively, the fitness landscape can be imagined as a mountain region with peaks, craters, valleys and hills. In that context, a local-search based algorithm can be thought of as navigating through the fitness landscape in order to find its highest peak.

The effectiveness of any local-search based method or memetic algorithm highly depends on the structure of the corresponding fitness landscape. The important *global* properties of the fitness landscape are: the ruggedness of the landscape, the distribution of valleys and local optima in the search space, the overall number of local optima and the topologies of the basins of attraction [50]. Analyzing these parameters, we can predict the efficiency of an optimization to some extent.

Fitness distance correlation has been proposed by Jones and Forrest [85] as a useful tool for analyzing the corelation between solution fitness and the distance between solutions to the nearest globally optimal solution. Given a set of candidate solutions with fitness values $\mathcal{F} = \{f_1, f_2, \ldots, f_\mu\}$ and the corresponding distances $\mathcal{D} = \{d_1, d_2, \ldots, d_\mu\}$ to the closest global optimum, the fitness-distance correlation coefficient is given as:

$$\rho(\mathcal{F}, \mathcal{D}) = \frac{f_{\mathcal{FD}}}{\sigma_{\mathcal{F}} \cdot \sigma_{\mathcal{D}}} \ ,$$

where $f_{\mathcal{FD}}$ is the covariance:

$$f_{\mathcal{FD}} = \frac{1}{\mu} \sum_{i=1}^{\mu} (f_i - \overline{f})(d_i - \overline{d}) \ .$$

$\overline{d}$ and $\overline{f}$ represent the average distance and the average fitness, respectively, and $\sigma_{\mathcal{D}}$ ($\sigma_{\mathcal{F}}$) are the corresponding variances.

If fitness increases when the distance to the optimum becomes smaller, then search is expected to be easy for local-search based algorithms. A value of $\rho = 1.0$ ($\rho = -1$, for maximization problems) indicates that there is a strong correlation between the objective values of MA solutions and their distance to the nearest optimum, i.e. the search should be "easy". On the other side, a value of $\rho = -1.0$ ($\rho = 1$, maximization) indicates that the random search may be more efficient then a local-search based method.

2.6 Exact Optimization Methods Based on Linear Programming

In the following, we briefly review existing approaches to solve ILPs with the aid of linear programming. For an exhaustive introduction to the theory of linear and integer programming we refer to [148, 129, 1].

2.6.1 Cutting Plane Algorithm

The cutting plane approach has been proposed in 1954 by Dantzig, Fulkerson, and Johnson [37], and since then, many of successful applications for COPs have been reported. A nice bibliography on cutting plane methods can be found in [89].

As an example for the cutting plane approach, within this section we consider the *symmetric traveling salesperson problem* (TSP) and its ILP formulation.

Given a weighted undirected graph $G = (V, E, c)$ with n vertices and m edges, the symmetric TSP is to find a tour visiting each vertex exactly once and having minimum total cost. The problem can be modeled in the following way:

$$\begin{aligned} & \min c^T x \\ \text{s.t.} \quad & x(\delta(v)) = 2 \quad \text{for all } v \in V \\ & x(\delta(W)) \geq 2 \quad \text{for all } W \subset V, W \neq \emptyset, V \\ & 0 \leq x \leq 1 \\ & x \text{ integer} \end{aligned}$$

Here, $x(F) = \sum_{e \in F} x(e)$ for $F \subset E$, while $\delta(W)$ represents the undirected cut induced by $W \subset V$. For each edge $e \in E$, value x_e is set to one if the edge belongs to the solution, otherwise it is set to zero. The first group of equations, the so-called *degree constraints*, force the degree of each vertex in V to be exactly two. What follows, are the inequalities that forbid subtours, the so-called *subtour elimination constraints* (SECs). This ILP formulation represents a typical example in which the number of constraints is, in general, too large to be explicitly represented on a computer. In the *cutting plane* approach [1, 2, 73], the integrality constraints for x are relaxed to $0 \leq x \leq 1$ (or to $x \in \mathbb{R}^m$, in general case) and only a small initial subset of constraints is chosen. This linear program is then solved, and the cutting

plane algorithm tries to find constraints that are not satisfied by the obtained solution (and are therefore not already included in the LP) but are valid for the problem. These violated constraints are called *cuts* or *cutting planes*.

If there exist such valid inequalities that can cut off the current LP-solution (the, so-called, *violated inequalities*), one or more of them are added to the LP and it is resolved. If cutting plane algorithm does not find violated constraints anymore, and if the obtained solution is feasible and integer, then it is also optimal. However, the solution of the final LP need not to be feasible for the starting ILP, in which case its objective value represents a *lower bound* of the original ILP. Algorithm 4 shows a generic cutting plane algorithm for a 0-1 integer linear program. Such lower bound can be used to measure the quality of a known feasible solution (obtained by means of a heuristic). To get an optimal solution, we can then switch to a branch-and-bound approach (see the next Section).

However, the problem of finding violated constraints (the so-called *separation problem*) is not always solvable in a polynomial time. In this case, *heuristic separation algorithms* can be used. Then, if no violated constraints can be found anymore, there is no guarantee that the solution found so far is optimal or feasible for the original problem. Nevertheless, this solution can provide valuable lower bounds for the actual ILP optimum [92].

initialize the constraint system (A', b') with a small subset of (A, b);
repeat
  find optimum solution of $c^T\overline{x} = \min\{c^T x \mid A'x \leq b',\ 0 \leq x \leq 1\}$;
  if $\overline{x}$ *is not feasible for the starting LP* **then**
    generate a cutting plane $(f, f_0), f \in \mathbb{R}^n, f_0 \in \mathbb{R}$ with
      $f^T\overline{x} > f_0$, and
      $f^T x \leq f_0,\ \forall x$ such that $Ax \leq b, x_i \in \{0,1\}, \forall i$;
    add the inequality $f^T x \leq f_0$ to the system (A', b');
  end
until $\overline{x}$ *is feasible for the starting LP*;

Algorithm 4: Generic cutting plane algorithm for a 0-1 ILP.

There exist general cutting plane algorithms (like those based on *Gomory cuts* [129, pp. 367–371]), but for larger instances of COPs their applicability seems to be limited. Thus, cutting planes designed for a particular problem need to be taken into consideration. A closer investigation of the polytope $\mathcal{P}_\mathcal{I}$ associated to a COP $(E, \mathcal{I}, c)$ (see Section 2.1.2) usually provides useful informations.

Facet defining inequalities are not dominated by any other valid inequalities, and thus, they represent the best cuts. However, finding facet defining inequalities for a particular COP is, in general, not an easy task, and often the corresponding separation problems are NP-hard themselves.

2.6.2 LP-based Branch-and-Bound

When applying a cutting plane algorithm in general we end up in the situation where the current solution x^* is not feasible for the original integer linear program, but we are unable to identify further inequalities violated by x^*. At that point we can employ another basic technique for solving hard ILPs: *branch-and-bound*. This is a divide-and-conquer approach that solves an ILP by splitting it recursively into smaller subproblems until a subproblem can be trivially solved or discarded. The algorithm maintains a *global upper bound*, UB_g (the objective value of the best solution known so far; usually initialized by some heuristic), and for every (sub)problem, a *local lower bound*, LB. The latter is computed by solving the LP-relaxation of the subproblem. We distinguish the following cases:

- If the solution of the relaxation happens to be feasible for the original problem and its objective function is lower than the current upper bound, the algorithm updates the upper bound and memorizes the solution. Additionally, the current vertex is *fathomed*, since no optimal solution of subproblems corresponding to descendants of that vertex can be better than the current solution.

- In case the local lower bound is greater than or equal to the global upper bound, the algorithm fathoms the current vertex for the same reasons as above.

- If the local lower bound is lower than the global upper bound and the optimal solution of the LP-relaxation is not feasible for the original problem, the algorithm selects one of the fractional variables x_i and branches over it by creating two new subproblems. The simplest *branching strategy* in a 0-1 ILP splits the current problem into two subproblems with x_i permanently set to 0 and 1, respectively[2].

Algorithm 5 summarizes the steps of the generic branch-and-bound algorithm. For more details we refer to e.g. [17]. Unfortunately, for most of the NP-hard problems, LP-relaxations do not provide local lower bounds that are strong enough for pruning the search tree sufficiently enough in order to solve large instances. To enhance the performance of branch-and-bound, we incorporate the cutting plane or column generation algorithm into it.

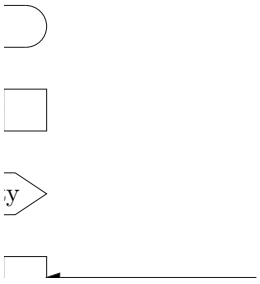
2.6.3 Branch-and-Cut

The *branch-and-cut* algorithm extends LP-based branch-and-bound by combining it with the cutting plane algorithm. At every vertex of the branch-and-bound tree, the algorithm searches for cutting planes, trying to improve the bound of the LP-relaxation. For many ILPs, much better local lower bounds can be derived in this way, resulting in substantially smaller numbers of necessary subproblems in the branch-and-bound trees.

In practice, branch-and-cut has been highly successful on various, sometimes even large NP-hard combinatorial optimization problems [131, 86, 87, 127, 125, 29].

[2]If we are dealing with a general ILP, the branching is usually done by splitting the constraint $l \leq x_i \leq u$ into two $l \leq x_i < m$ and $m \leq x_i \leq u$ and resolving the corresponding subproblems.

Data : Integer linear program:
ILP = $\min\{c^T x \mid Ax \leq b,\ x \in \{0,1\}^n\}$
Result: Optimal solution for ILP.



Algorithm 5: Generic branch-and-bound algorithm.

2.6.4 Column Generation

If the starting LP – the so-called *master problem* – contains too many (e.g. exponentially many) variables so that it cannot be solved explicitly, one may start with a small subset of variables. Suppose that the indices of these variables are stored in J, $J \subset \{1, 2, \ldots, n\}$ (J is also called the set of *active*, or *basic*, variables). We first compute an optimal solution of the so-called *restricted master problem*:

$$\min\{c_J^T x_J \mid A_J x_J \leq b_J, x_J \in \mathbb{R}_+^{|J|}\} \ ,$$

Note that if a column i is not present in the matrix A_J, the variable x_i is automatically set to zero. We then need to check if the addition of an inactive variable might improve the objective value. This can be done by computing the *reduced costs* of inactive variables. These reduced costs of a non-basic variable j with associated column $a_j \in \mathbb{R}^m$ and objective function coefficient $c_j \in \mathbb{R}$, are defined as:

$$r_j = c_j - y^T a_j \ ,$$

where y represents the solution of the corresponding basic dual LP problem. If all non-basic variables "price out correctly", i.e. if there is no variable with negative reduced costs (as far as minimization is concerned), the optimal solution of the restricted problem is also optimal for the corresponding master problem. If there is a variable with negative reduced costs, we add it to the restricted problem, re-optimize and iterate. This process of dynamically identifying

variables with negative reduced costs can be also thought of as a process of generating constraints for the dual of the LP-relaxation. Algorithm 6 shows the generic column generation algorithm.

Select a small subset J of the variables $\{1, \ldots, n\}$;
repeat
  Obtain an optimum basic solution $\overline{x}_J$ of the LP:
    $c_J^T \overline{x}_J = \min\{c_J^T x_J \mid A_J x_J \leq b,\ x_J \in \mathbb{R}^{|J|}\}$;
  Determine the values of the dual variables $\overline{y}$;
  if $\exists i,\ i \in \{1, \ldots, n\} \setminus J$ *such that* $r_i = c_i - \overline{y}^T a_i < 0$ **then**
    Add column i into J;
  end
until $r_i = c_i - \overline{y}^T a_i \geq 0,\ \forall i \in \{1, \ldots, n\} \setminus J$;

Algorithm 6: Generic column generation algorithm.

The problem of finding a non-basic variable that does not price out correctly, or proving that there are no such variables, is also called *pricing*. Usually, when the LP relaxation of an NP-hard problem contains an exponential number of variables, the corresponding pricing problem which searches for a column with minimal reduced costs, can be NP-hard as well. In that case, *a heuristic pricing algorithm* may find a variable with negative reduced costs; if it does not, however, there is no guarantee that all non-basic variables price out correctly. For a thorough review of the column generation approach, we refer to [17, 157], for example.

Sparse and Reserve Graph Pricing The pricing algorithm shown in Algorithm 6 inserts one non-basic variable per iteration. However, it is also possible to add a subset of non-basic variables with negative reduced costs at once.

Suppose we are dealing with a group of graph problems with one-to-one correspondence between the integer variables in the ILP formulation of the problem and the edges of the original graph. Examples of such problems are the traveling salesman problem (TSP), the Steiner problem in graphs, and also the prize-collecting Steiner tree problem. The size of feasible solutions in such problems is usually bounded by n (the number of vertices), while the number of variables can be $\binom{n}{2}$ in the worst case, when the graph is complete. In that case, the computation can be accelerated significantly if only a suitable, promising subset of edges is initially selected and appropriately augmented during the column generation phase. This small set of active variables corresponds to a subgraph of the original graph (the so-called *sparse graph*), and the whole technique is called the *sparse graph technique* [70].

The question of how to initialize a sparse graph plays an important role in the design of a column generation algorithm. For some problems, e.g. the traveling salesman problem, a good choice for a sparse graph is the k-nearest neighbor graph [70]. If there is no guarantee that the sparse graph contains a feasible solution, it should be augmented by the edges of a solution computed by a heuristic. Padberg and Rinaldi [131] suggested to create heuristically a series of feasible solutions and initialize the sparse graph with all involved edges. Reinelt [145] proposed

to use a Delaunay graph to initialize the sparse graph.

In addition to the sparse graph, Jünger et al. [89] proposed to use the so-called *reserve graph*. Edges of the reserve graph represent a set of additional promising edges, which do not belong to the sparse graph and should be considered for inclusion in the LP before any other edges. For example, if the 5-nearest neighbor graph is chosen as sparse graph, then the difference to the 10-nearest neighbor graph can be a suitable reserve graph. In the pricing phase, edges from the reserve graph are checked with higher priority than the other edges, thus the running time of the algorithm can be improved, if the reserve graph contains well-chosen edges. In one pricing iteration, all edges from the reserve graph with negative reduced costs are added to the restricted problem and the LP is resolved. If all edges from the reserve graph price out correctly, *complete pricing* is performed: from the set of all inactive edges, those with negative reduced costs are determined and added to the LP at once.

Branch-and-Price The *branch-and-price* algorithm is an LP-based branch-and-bound algorithm in which the variables are dynamically generated. Instead of solving a separation problem at each vertex of the enumeration tree as in branch-and-cut, we need to solve the problem of finding a column, i.e., a feasible non-active variable that can improve the objective value of the current solution. In [15] a thorough review of this method is provided.

2.6.5 Branch-and-Cut-and-Price

When both variables and constraints are generated dynamically during an LP-based branch-and-bound algorithm, the technique is called *branch-and-cut-and-price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of constraints and that of variables. However, the difficulty of this combination is that the pricing problem is modified by the addition of cutting planes. The coefficient of a new column in a row added to the original problem has to be considered in the pricing problem.

In addition to the standard branch-and-cut steps, pricing must be performed before the search tree is pruned at some subproblem. Its purpose is to check if the LP-solution computed in the sparse graph is valid on the complete graph, i.e. if all inactive variables "price out" correctly. If this is not the case, inactive variables with negative reduced costs are added to the sparse graph and the new LP is solved. Otherwise, the local lower bound, and probably the global lower bound can be updated.

The combination of constraint and column generation has been successfully applied to combinatorial optimization problems that involve a large number of variables, yet for which the number of variables in feasible solutions is relatively small [103].

Chapter 3

Vertex Biconnectivity Augmentation

There are two main aspects for robust communication networks: reliability and survivability. Reliability is the probability that a network functions according to a specification. Survivability is the ability of a network to perform according to a specification after some failure. After the recent electrical power blackouts in the USA and some European countries, it has become clear that the survivability of networks play an important role in the design of electrical power supply. In many other applications it is also not acceptable that the failure of a single service vertex—be it a computer, router, or other device—leads to a disconnection of other vertices. Survivability is thus extremely important in modern telecommunication networks, in particular in backbones. Redundant connections need to be established to provide alternative routes in case of a temporary break of any one vertex.

This kind of robustness of a network is described in graph theory by means of *vertex connectivity*. A k-connected network, $k \geq 2$, is said to be survivable. With today's technology the probability of a failure can typically be kept small, but dropouts are still potentially harmful. The probability of a second failure before a first one is repaired is often neglected; k-connected networks with $k \geq 3$ are usually considered not worth the additional costs. Therefore, throughout this thesis we focus on the most common case of reliable networks which are 2-connected networks.

In the *weighted vertex biconnectivity augmentation problem for graphs* (V2AUG), a connected but not vertex biconnected network is given. Thus, the removal of cut-points separates the network into unconnected components. We say that we *cover a cut-point* when we add some links to ensure that the removal of this vertex no longer disconnects the network. The global aim is to identify a set of additional links with minimum total costs in order to cover all cut-points. Besides designing survivable networks, vertex biconnectivity augmentation has important applications in the development of reliable database systems [45] and in improving statistical data security [72].

Formally, the problem is defined as follows.

Definition 7. [Weighted Vertex-Biconnectivity Augmentation, V2AUG]
Let $G = (V, E, c)$ be a vertex biconnected, undirected graph with vertex set V and edge set E representing all possible connections. Each edge $e \in E$ has associated costs $c(e) > 0$. A connected, spanning, but not vertex biconnected subgraph $G_0 = (V, E_0)$, with $E_0 \subset E$ represents an existing network, and $E_a = E \setminus E_0$ is the set of edges that may be used for augmentation. The objective is to determine a subset of these candidate edges $E_s \subseteq E_a$ so that the augmented graph $G_s = (V, E_0 \cup E_s)$ is vertex biconnected and

$$c(E_s) = \sum_{e \in E_s} c(e) \tag{3.1}$$

is minimal.

Figure 3.3a shows an example of a graph G with its set of given edges E_0 and possible augmentation edges E_a. In the sequel we will refer to this problem briefly as *vertex biconnectivity augmentation*. If in Definition 7 it is requested that the augmented graph $G_s = (V, E_0 \cup E_s)$ is edge biconnected (instead of vertex biconnected), we speak of *weighted edge biconnectivity*

augmentation (E2AUG), or shortly *edge biconnectivity augmentation*. In the sequel, we will also sometimes refer to weighted graphs as $G = (V, E)$, when it is clear from context which edge-cost function is involved.

This chapter is organized as follows: An overview on former approaches to V2AUG and related problems is given in Section 3.1. Within Section 3.2 we describe an efficient preprocessing based on the derivation of a more compact *block-cut graph* G_A from the problem's original graph. Techniques are introduced, which may shrink the block-cut graph substantially by fixing or discarding certain augmentation edges in safe ways. Furthermore, additional preprocessing steps that remove some block-vertices, and compress paths are proposed. Finally, we describe some further data structures created during preprocessing that allow the following optimization to be implemented in a more efficient way. Section 3.3 is devoted to a memetic algorithm which searches for a low-cost solution on the reduced block-cut graph. The best solution found is finally mapped back to a solution of the original V2AUG instance. We provide an exhaustive experimental comparison of the new approach against other algorithms for V2AUG.

In Section 3.4, we propose a branch-and-cut-and-price (BCP) algorithm which searches for optimum solutions on the block-cut graph. We first describe a simple branch-and-cut algorithm based on the minimum-cut ILP formulation of the problem. To enhance its performance, we propose the incorporation of the column generation method based on the sparse and reserve graph techniques. We also propose the insertion of multiple violated cuts within the separation phase, and a local improvement procedure as primal heuristics. Using the latest approach, we found optimal solutions for some complete graphs with more than 400 vertices.

In Section 3.5, we investigate possible ways to use the knowledge about the problem obtained after running the MA, to improve the performance of the branch-and-cut-and-price approach. We consider setting lower bounds by MA, biasing primal heuristic and guiding column generation using MA results. Conclusions are drawn in Section 3.6.

The main results of this chapter related to the memetic algorithm are published in [108]. Preliminary results appeared also in [93]. We also developed a memetic algorithm for edge biconnectivity augmentation and published our results in [144]. Preliminary results appeared in [107].

3.1 Previous Work

Eswaran and Tarjan [46] were the first to investigate V2AUG. Using a reduction from the Hamiltonian circuit problem, they proved that the decision problem associated with V2AUG is NP-complete. In [159], Watanabe and Nakamura proved that minimum-cost augmentation for edge or vertex k-connectivity is NP-hard, for any $k \geq 2$.

Special Cases Solvable in Polynomial Time

Regarding V2AUG, an exact polynomial-time algorithm could be found for the special case when G has unit edge costs [81]. The algorithm runs in $O(|V|)$ time. It is also based on the

construction of the block-cut tree. A parallel implementation runs in $O(\log |V|)$ time using $O(|V| + |E|)$ processors on an EREW PRAM.

Optimal edge biconnectivity augmentation of a Hamiltonian path is possible in $O(|V| \log |V|)$ time [61]. If the graph $G_0 = (V, E_0)$ represents a *depth-first search tree*[1] of G, optimal edge biconnectivity augmentation can be done in $O(M \cdot \alpha(M, n))$ where α is the inverse Ackermann function and $M = |E| \cdot \alpha(|E|, |V|)$ [62]. Provan and Burk [136] showed that it is possible to optimally augment any subtree $T = (V_T, E_T)$ of G in $O(n^2k^2)$ time, if the graph G is planar, and $|V_T| = k$.

Approximation Algorithms

Frederickson and Jàjà [56] described an approximation algorithm for undirected graphs to achieve vertex biconnectivity. The algorithm runs in $O(|V|^2)$ time and for the general case finds a solution within a factor 2 of the optimum[2], with the assumption that the graph G_0 is connected. If G_0 is not necessarily connected, the approximation factor increases to 3, however, we do not consider this case here. The algorithm includes a preprocessing step that transforms the fixed graph G_0 into the corresponding block-cut tree, superimposes the augmentation edges, and performs the basic reductions (see Section 3.2). The augmented block-cut graph is further extended to a complete graph such that there is an augmentation edge for each pair of vertices $\{u, v\} \notin E_T$. All these augmentation edges get new "reduced" costs according to the following definition and maintain back-references to the original augmentation edges where the costs come from:

$$c'(u, v) = \min_{\{x,y\} \in E_A} \left(\{ c(x, y) \mid u, v \text{ are on the } \{x, y\}\text{-path in } T \} \cup \{\infty\} \right). \tag{3.2}$$

In the main part of the algorithm, the block-cut tree T is directed toward an arbitrarily chosen leaf r, the root vertex, yielding an ingoing arborescence. Each directed tree-edge is assigned zero costs. Each cut-vertex is substituted by a star-shaped structure including new dummy-vertices in order to guarantee that strongly connecting the block-cut tree implies vertex biconnectivity of the underlying fixed graph G_0. Two different types of augmentation edges are distinguished: A *back-edge* connects a vertex with one of its descendants in the directed block-cut tree; all other augmentation edges are called *cross-edges*. Back-edges are directed from the vertex nearer to the root toward the vertex farther away; cross-edges are replaced by pairs of reversely directed edges. Finally, on this directed block-cut graph a minimum outgoing spanning arborescence is derived and the solution's edge-set E_s is obtained.

The approximation algorithm described above has been improved by Khuller and Thurimella [95]. The main difference and the advantage of their algorithm is that the extension of the block-cut graph to a complete graph is omitted. Instead, each cross-edge (u, v) is replaced

[1] A rooted spanning tree T of a graph G is a depth-first search tree if for each non-tree edge (u, v) either u is an ancestor of v in T, or v is an ancestor of u in T.

[2] Let P be a minimization problem, and let A be an algorithm that, for any instance s of P, returns a feasible solution $A(s)$. A is a k-approximation algorithm for P if, for any instance s, $A(s) \leq k \cdot \mathrm{OPT}(s)$, where $\mathrm{OPT}(s)$ denotes the optimal solution for s.

by two reversely directed cross-edges (u, v) and (v, u) and two back-edges $(\text{lca}(u, v), u)$ and $(\text{lca}(u, v), v)$, i.e. back edges from the least common ancestor of u and v, to u, respectively v. Further, no dummy vertices are included, but each augmentation edge (v_c, v) going out of some cut-point v_c is replaced by an edge (v_b, v), where v_b is the vertex adjacent to v_c on the undirected path from v_c to v in T. The algorithm exhibits a time complexity of only $O(|E| + |V| \log |V|)$, but has still the approximation factor 2. Empirical results of this algorithm can be found in Section 3.3.6.

An iterative approach based on Khuller and Thurimella's algorithm has been proposed by Zhu et al. in [162]; for more details, see also [161]. In each step, a *drop*-heuristic measures the gain of each augmentation edge as if it would be included into a final solution. This is achieved by calling the minimum outgoing spanning arborescence algorithm for each edge once with its cost set to zero and once with its original cost. We choose an edge with the highest gain and set its cost permanently to zero. The process is repeated until the obtained arborescence has zero total costs. Furthermore, the whole algorithm is applied with each leaf of the block-cut tree becoming once the root, and the overall cheapest solution is the final one. Although the theoretical approximation factor remains 2, practical results are usually much better than those obtained by applying Khuller and Thurimella's algorithm; our empirical comparison in Section 3.3.6 also supports this. However, time requirements are raised substantially.

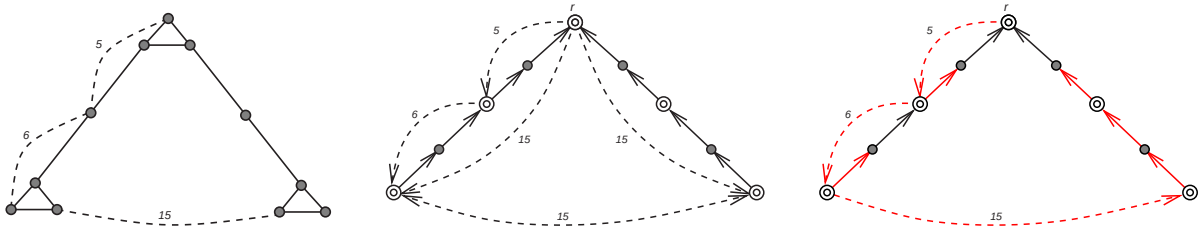
Let $G_s = (V, E_0 \cup E_s)$ be a feasible solution. An edge $e \in E_s$ is said to be *redundant* if its removal does not violate the biconnectivity-property of G_s. Figure 3.1 shows that due to the computation of the minimum outgoing spanning arborescence, Khuller and Thurimella's algorithm produces redundant edges. Although the transformations in Frederickson and Jàjà's algorithm are different, the same behavior can be observed. Due to the computation of *drop*-values within the algorithm of Zhu et al., such redundant edges can be avoided.

Recently, Kortsarz et al. [100] proved that it is APX-hard[3] to solve V2AUG even in the case where every pair of vertices forms an augmenting edge of cost 1 or 2. Furthermore, the authors showed that minimum-cost vertex connectivity augmentation from k to $k+1$ is APX-hard, for any $k \geq 2$, even for graphs with uniform costs.

Metaheuristics

A straight-forward genetic algorithm for V2AUG has been proposed in [106]. This algorithm is based on a binary encoding in which each bit corresponds to an edge in E_a. Standard uniform crossover and bit-flip mutation are applied. Infeasible solutions are repaired in Lamarckian way by a greedy algorithm which temporarily removes cut-points one by one and searches for the cheapest augmentation edges that reconnect the separated components. The major disadvantage of this genetic algorithm is its high computational effort, which mainly comes from the repair strategy having a worst-case running time of $O(|V|\,|E_a| \log |V|)$ per candidate solution.

[3] A problem $\mathcal{P}$ is APX-hard, if there exists some fixed $\varepsilon > 0$ such that it is NP-hard to approximate the problem within ratio $1 + \varepsilon$.



(a) (b) (c)

Figure 3.1: Due to the application of the minimum outgoing spanning arborescence algorithm at the end of Khuller and Thurimella's algorithm, the final solution may contain obviously redundant edges. (a) Given graph $G = (V, E_0 \cup E_a)$. Solid edges represent graph $G_0 = (V, E_0)$, dashed edges represent E_a. (b) Directed block-cut graph with a randomly chosen root r, obtained after the transformations of Khuller and Thurimella's algorithm. (c) Minimum outgoing spanning arborescence. Its weight is 26; the corresponding V2AUG solution's costs are 26. The optimal solution's costs are 15, while the corresponding arborescence costs are 30.

Related Work

Eswaran and Tarjan [46] proved that for strong connectivity augmentation on a directed graph as well as for edge biconnectivity augmentation on an undirected graph, minimum-cost augmentation is NP-hard. The problems remain NP-hard even if graph G_0 is connected. Frederickson and Jàjà [56] have shown that the problem of strongly connecting a weakly connected directed graph remains NP-hard. In their proof, they made a reduction from the 3-dimensional matching problem.

Some relations between V2AUG and the traveling salesman problem are pointed out in [57]. In [56, 95, 161], the approximation algorithms described above address edge biconnectivity augmentation as well as strong connectivity augmentation. In [144], an effective evolutionary algorithm for E2AUG is given, which scales well to large problem instances and outperforms several previous heuristics. A compact edge-set encoding and special initialization and variation operators that include a local improvement heuristic are applied.

Based on this algorithm for E2AUG, a memetic algorithm for V2AUG, presented in the sequel, has been developed. Major differences to the evolutionary algorithm for E2AUG lie in the underlying data structures (e.g., the now necessary block-cut graph), in the preprocessing, in the recombination and mutation operators, in the local improvement algorithm, and in the way how this local improvement is integrated in the evolutionary algorithm. While it is relatively easy to check and eventually establish the cover of a single fixed edge, this is significantly harder to achieve for a cut-vertex, especially in an efficient way: A critical fixed edge can always be covered by a single augmentation edge, and it is obvious which augmentation edges are able to cover the critical edge. On the other side, in general, a combination of multiple augmentation edges is necessary to completely cover a cut-vertex.

"Augmenting Empty Graphs". The problems of finding the minimum-cost edge- or vertex-k-connected spanning subgraph, $k \geq 2$, of a graph (k-ECSS, or k-VCSS, respectively) are also known to be NP-hard. The first approximation algorithm with an approximation factor of $2 + 2(k-1)/n$, for any $k \geq 2$, has been provided in [94]. However, this algorithm works only on complete weighted graphs where the triangle inequality is satisfied. Cheriyan et al. [25] developed an improved approximation algorithm for the vertex k-connectivity case and gave a survey on former approaches. To our knowledge, meta-heuristics have not yet been applied to this problem.

In [31], a so-called *bootstrap heuristic* for the construction of a minimum-weight edge biconnected subgraph has been developed. Solutions are obtained by means of bootstrapping a lower bounding procedure based on linear programming relaxations of the problem.

Smallest Augmentation. Khuller and Raghavachari [94] provided an 1.85-approximation algorithm for finding the smallest k-ECSS with respect to the number of edges. The algorithm has been improved in Cheriyan and Thurimella [24], where an approximation algorithm with factor $1+2/(k+1)$ has been proposed. Regarding k-VCSS, the same paper provides a $1+1/k$-approximation algorithm.

Planar Augmentation The planar augmentation problem is the problem of adding a minimum number of edges to a given planar graph such that the resulting graph is biconnected and still planar. In the context of graph drawing, Fialko and Mutzel [48] developed a factor $\frac{5}{3}$ approximation algorithm for planar augmentation. A polyhedral approach to planar augmentation and related problems is proposed in [126].

Augmenting Multi-Graphs. Another class of related problems is the augmentation of multi-graphs with the smallest number of unweighted edges so that the resulting graph becomes edge- or vertex-k-connected. In particular the edge k-connectivity case turned out to be an easier problem: Watanabe and Nakamura [159] described a polynomial time algorithm which solves the general edge k-connectivity problem to optimality. Gabow [59] improved the running time of this algorithm to $O(m + k^2 n \log n)$. In case of vertex k-connectivity, exact polynomial time algorithms are known for $k \in \{2, 3, 4\}$; it is still an open question whether the problem is NP-hard for $k \geq 5$. A recent study on this topic can be found in [82].

Survivable Network Design Problem. The more general problem of designing a minimum-cost network with individually specified connectivity requirements for each vertex—the so-called *survivable network design problem*—has been attacked by Stoer in [152], using a polyhedral approach. By means of cutting-plane techniques the algorithm is able to find optimal or near-optimal solutions for instances of small and moderate size. Monma and Shallcross [122] considered a variant of this problem in which the connectivity requirements of each vertex are limited to $\{0, 1, 2\}$ and suggested heuristics for constructing feasible solutions and locally improving them. Note that V2AUG and E2AUG represent special cases of this problem.

Recently, Fortz [55] studied a new kind of survivable network design problem with *bounded rings*. It includes an additional constraint limiting the maximum length of cycles for which no shortcuts exist. The author provided a study of the underlying polyhedron and proposed several classes of facet-defining inequalities used in a branch-and-cut algorithm. Several heuristics are also proposed in order to solve real-world instances of larger size.

3.2 Preprocessing

To attack problems of larger size and to allow an efficient optimization, we propose a general optimization framework whose outline is given in Figure 3.2.

Out of the original graph G, a more compact block-cut graph G_A is deterministically derived. Then, a new enhanced preprocessing is applied, which may shrink the block-cut graph substantially by fixing or discarding certain augmentation- or tree-edges in safe ways. Some further data structures allowing the following optimization to be implemented in a more efficient way are also created during preprocessing. The core of the whole system are the following two algorithms, both described in detail in the sequel:

- a new memetic algorithm (MA) that uses problem specific variation operators and strongly interacts with a local improvement procedure, and
- a branch-and-cut-and-price (BCP) algorithm that searches for good lower bounds and, if possible, for optimal solutions.

Both algorithms search for the solutions on the reduced block-cut graph. These solutions are finally mapped back to the corresponding solutions of the original V2AUG instance.

The following subsections describe the preprocessing mechanisms in detail.

3.2.1 Superimposing Edges

After the block-cut tree T has been derived from graph G_0, all augmentation edges in E_a are *superimposed* on T forming a new edge-set E_A: For each edge $(u, v) \in E_a$, a corresponding edge (u', v') is created with $u', v' \in V_T$ being the vertices that are associated with u, respectively v; edge costs are adopted, i.e. $c(u', v') = c(u, v)$. The so-called (augmented) block-cut graph $G_A = (V_T, E_T \cup E_A)$ may be a multi-graph containing self-loops and multiple edges between two vertices. However, applying the following safe reductions yields a simple graph:

1. Self-loops $(u, u) \in E_A$, as, e.g., edge e'_1 in Figure 3.3b, are discarded. They can never help in establishing biconnectivity.
2. Each augmentation edge that connects the same vertices as an edge from E_T is discarded, since such an edge can also never help in establishing biconnectivity. See edge e'_2 in Figure 3.3b.
3. Augmentation edges connecting two cut-vertices that are adjacent to the same block-vertex in T are also discarded because of the same reason; see edge e'_3 in Figure 3.3b.

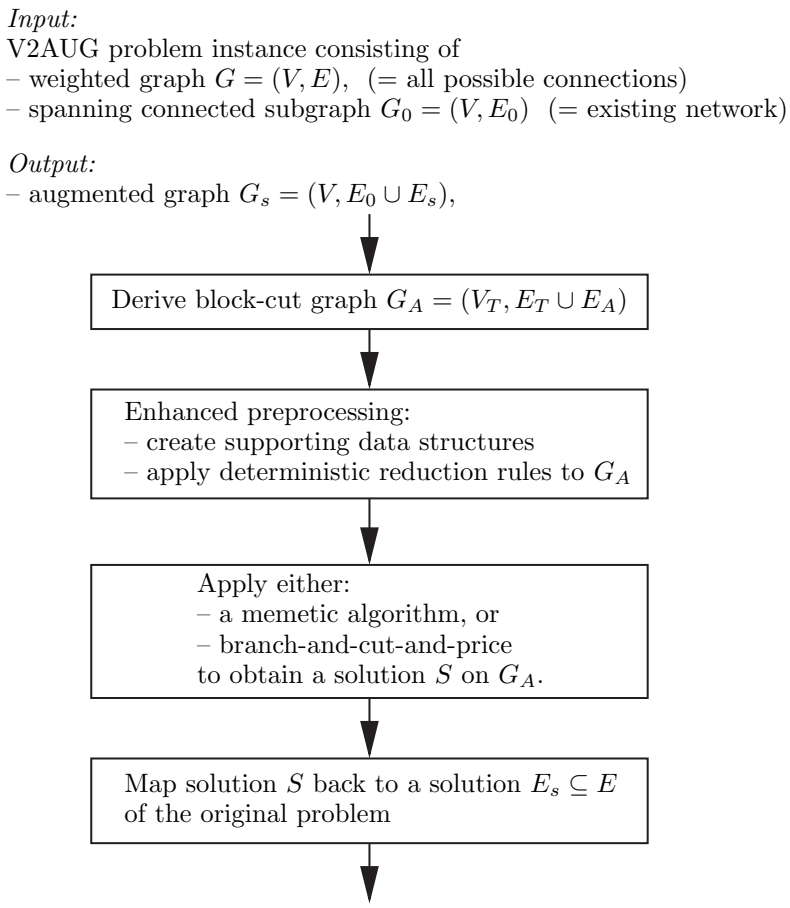


Figure 3.2: Basic structure of the proposed approach.

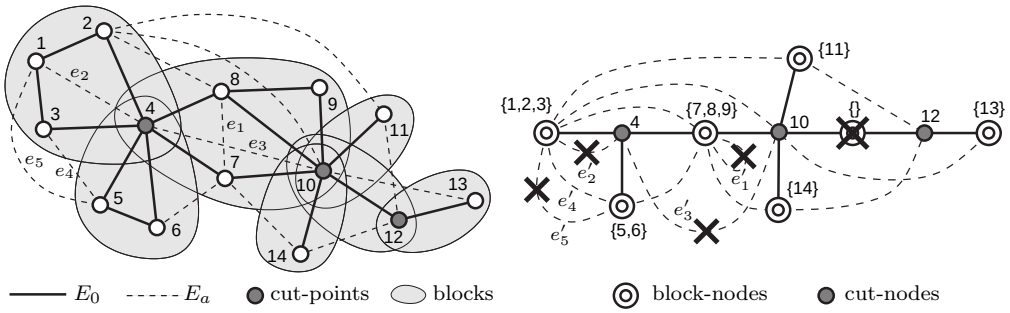


Figure 3.3: (a) A base graph $G = (V, E_0 \cup E_a)$ with its fixed edges E_0 and optional augmentation edges E_a and (b) the corresponding block-cut tree $T = (V_T, E_T)$ with the superimposed augmentation edges E_A, generating the block-cut graph $G_A = (V_T, E_T \cup E_A)$.

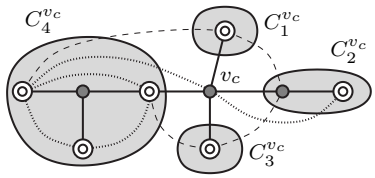


Figure 3.4: The cut-components $C_1^{v_c}$, $C_2^{v_c}$, $C_3^{v_c}$, and $C_4^{v_c}$ of a cut-vertex v_c. Dashed edges form edge-set $\Gamma(v_c)$. Dotted augmentation edges do not contribute in covering v_c.

4. From multiple augmentation edges connecting the same vertices from V_T, only one with minimum weight is retained; see edge e'_4 in Figure 3.3b when assuming $c(e'_4) < c(e'_5)$. The more expensive edges may never appear in an optimum solution.

In order to be finally able to derive the original edges $E_s \subseteq E_a$ corresponding to a solution $S \subseteq E_A$ identified on the block-cut graph, it is necessary to maintain a back-mapping from E_A to E_a.

3.2.2 When is a Cut-Vertex Covered?

A block-cut tree's edge $e \in E_T$ is said to be *covered* by an augmentation edge $e_A = (u, v) \in E_A$ if and only if e is part of the tree path connecting u with v. In order to completely cover a cut-vertex $v_c \in V_T$, all its incident tree-edges need to be covered, but this is in general not sufficient.

If v_c and its incident edges are removed from T, the tree falls apart into l connected components $C_1^{v_c}, \ldots, C_l^{v_c}$, where l is the degree of v_c in T; we call them *cut-components* of v_c; Figure 3.4 illustrates this. To completely cover v_c, at least $l-1$ augmentation edges are needed such that all cut-components $C_1^{v_c}, \ldots, C_l^{v_c}$ are united into one connected graph.

We say that an augmentation edge $e_A = (u, v) \in E_A$ *contributes in covering the cut-vertex*

v_c, if it connects two cut-components $C_i^{v_c}$ and $C_j^{v_c}$. Note that two tree-edges incident to v_c are covered by e_A.

For any cut-vertex $v_c \in V_T$, let $\Gamma(v_c) \subseteq E_A$ be the set of augmentation edges that contribute in covering v_c. Furthermore, for each $e_A \in E_A$, let $\Psi(e_A) \subseteq V_T$ be the set of cut-vertices to whose covering e_A contributes, i.e., $\Psi(e_A) = \{v_c \in V_T \mid e_A \in \Gamma(v_c)\}$. Preprocessing explicitly computes and stores the sets $\Gamma(v_c)$ for all cut-vertices and the sets $\Psi(e_A)$ for all augmentation edges as supporting data structures. This is done by first performing a depth-first search on T and storing for each vertex its depth and a reference to its parent vertex, in order to be able to efficiently determine the tree-path between any pair of vertices. Then, the computation of all sets $\Gamma(v_c)$ and $\Psi(e_A)$ can be performed in $O(|E_A|\,|V_T|)$ time. The space required for these data structures is bounded above by $O(|E_A|\,|V_T|)$.

Let us consider the following special case in which the graph G is complete, and the fixed graph G_0 represents a balanced binary tree:

Definition 8. [Perfect Binary Tree]
A perfect binary tree *of height $h \geq 0$ is a binary tree $T = \{r, T_L, T_R\}$, with the following properties:*

- *If $h = 0$, then $T_L = \emptyset$ and $T_R = \emptyset$,*
- *If $h > 0$, then both T_L and T_R are perfect binary trees of height $h - 1$.*

The vertex r is called the root vertex of the tree.

It follows directly from the definition that a perfect binary tree of height h has exactly $n = 2^{h+1} - 1$ vertices.

Augmenting a Perfect Binary Tree. Suppose we are given a complete graph $G = (V, E)$ with a fixed perfect binary tree $G_0 = (V, E_0)$ of height h.

We want to answer the following question: What are the memory complexities of Γ and Ψ data structures in this special case? After the transformation of G_0 into the block-cut tree T, we obtain a perfect binary block-cut tree with block-vertices representing leaves, while the rest of the vertices are cut-vertices with one-to-one correspondence to the inner vertices of G_0[4]. Thus, the total number of vertices in T is also $n = 2^{h+1} - 1$. Let us first compute the average number of edges that contribute in covering of a certain cut-vertex. The root vertex r has degree two, i.e. its removal disconnects the graph into two connected components, each having $(n-1)/2 = 2^h - 1$ vertices. Hence, the size of $\Gamma(r)$ is the number of edges representing the maximal edge-cut between two subtrees:

$$|\Gamma(r)| = (2^h - 1)^2 \ .$$

Let the root have depth zero, and let each leaf have depth h. Consider now the removal of a cut-vertex v_c^k in depth k, where $1 \leq k \leq h - 1$. Each of these cut-vertices has degree three,

[4]Empty block-vertices can be taken out of consideration. For an explanation, see the next section.

and the tree T will fall apart into three trees: Two of them are perfect binary subtrees T_1 and T_2 of height $h-k-1$, i.e:

$$|V[T_1]| = |V[T_2]| = 2^{h-k} - 1 \ .$$

The total number of vertices in the third three T_3 is:

$$|V[T_3]| = |V[T]| - |V[T_1]| - |V[T_2]| - |\{v_c^k\}| \ ,$$

i.e.

$$|V[T_3]| = 2^{h+1} - 1 - 2 \cdot (2^{h-k} - 1) - 1 = 2^{h+1} - 2^{h-k+1} \ .$$

The number of edges that contribute in covering of the cut-vertex v_c^k is then:

$$|\Gamma(v_c^k)| = |E[T_1 : T_2]| + |E[T_2 : T_3]| + |E[T_1 : T_3]| \ ,$$

where $E[T_1 : T_2] = \{(u, v) \mid u, v \in V_T, u \in V[T_1], v \in V[T_2]\}$.

$$|\Gamma(v_c^k)| = (2^{h-k} - 1)^2 + 2(2^{h+1} - 2^{h-k+1})(2^{h-k} - 1) =$$

$$= (2^{h-k} - 1)(2^{h-k} - 1 + 2^{h+2} - 2^{h-k+2}) \leq (2^{h-k} - 1)2^{h-k}(1 + 2^{k+2} - 2^2) \ .$$

We can finally say:

$$|\Gamma(v_c^k)| \approx 2^{2h-k+2} \ .$$

For the cut vertex $v_c \in V_T$, the average size of $\Gamma(v_c)$ is then:

$$|\overline{\Gamma}(v_c)| = \frac{1}{2^h - 1}(|\Gamma(r)| + \sum_{k=1}^{h-1} 2^k |\Gamma(v_c^k)|) \approx \frac{(2^h - 1)^2}{2^h - 1} + \sum_{k=1}^{h-1} 2^k \frac{2^{2h-k+2}}{2^h - 1} \approx$$

$$\approx 2^h - 1 + (h-1)2^{h+2} \approx h2^h \ .$$

In other words:

$$|\overline{\Gamma}(v_c)| = \Theta(n \log n) \ .$$

Therefore, the memory space needed to store the whole Γ data-structure for this special case is $O(n^2 \log n)$.

On the other hand, it is easy to see that the average length of a path in T is:

$$|\overline{\Psi}(e_A)| = \Theta(\log n) \ ,$$

which implies that the average space needed to store data-structure Ψ is $O(n^2 \log n)$.

How Much Space do Γ and Ψ Need on Average in the General Case? If we suppose that the tree structure is not degenerated, i.e. that the tree's diameter is $O(\log |V_T|)$, then the same as above holds, i.e. each augmentation edge $e_A \in E_A$ covers on average $O(\log |V_T|)$ vertices. In total, Ψ needs $O(|E_A| \log |V_T|)$ space.

The average space needed to store Γ is $O(|E_A| \log |V_T|)$ as well. One observes that the Γ and Ψ data-structures are "dual" to each other.

For each entry $e \in \Gamma(v_c)$, preprocessing also stores references to the two tree-edges being incident to v_c and covered by e; we denote them by $e_{T_1}^{v_c}(e)$ and $e_{T_2}^{v_c}(e)$. They directly correspond to the two cut components edge e can connect.

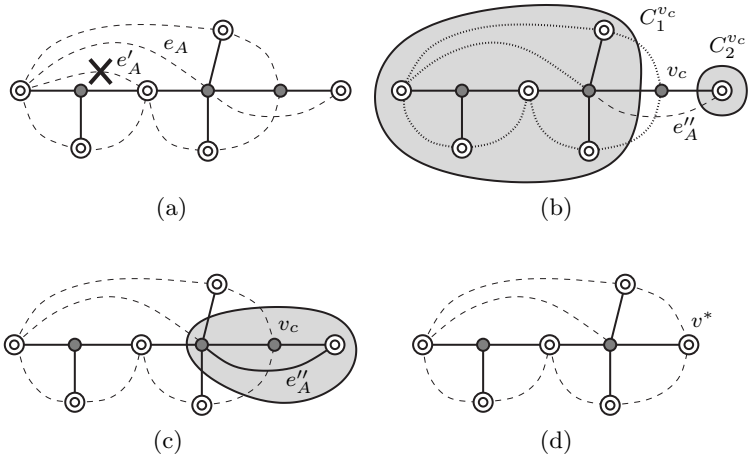


Figure 3.5: Reducing the block-cut graph: (a) Assuming $c(e_A) \leq c(e'_A)$, edge e'_A can be eliminated. (b) e''_A is the only edge able to connect the cut-components $C_1^{v_c}$ and $C_2^{v_c}$ of v_c and is therefore fixed. This introduces a new biconnected component in T (c), which is shrunk into the single new block-vertex v^* (d).

Check whether a Cut-Vertex is Covered. In the memetic algorithm, it is necessary to efficiently check if a certain cut-vertex is covered by a subset of augmentation edges $S \subseteq E_A$. With the precomputed Γ and Ψ and the aid of a temporary union-find data structure with weight balancing and path compression [5, pp. 183–189], this check can be performed in $O(|S| \cdot \alpha(|V_T|, |S|))$ time. In most cases the degree of the cut-vertex v_c is less than four. Then, not even a union-find data structure is needed, since it is sufficient to check whether each of the tree-edges incident to v_c is covered by some augmentation edge being not incident to v_c.

3.2.3 Reducing the Block-Cut Graph

In addition to the simple reductions (superimposition) of the block-cut graph described above and used in the literature so far [56, 95], we apply the following more sophisticated rules which are partly adopted from the preprocessing for E2AUG we published in [107]. These rules are safe in the sense that they never prevent the following optimization from finding an optimal solution.

Edge Elimination

If there are two edges $e_A, e'_A \in E_A$, $c(e_A) \leq c(e'_A)$, and e_A covers all those tree-edges that are covered by e'_A (in addition to others), e'_A is obsolete and can be discarded; see Figure 3.5a. All such edges can be identified in $O(|V_T|^2)$ time as a byproduct of a dynamic programming algorithm for computing the reduced costs given by equality. (3.2) that have been used in the approximation algorithm proposed in [56].

Fixing of Edges

An edge $e_A \in E_A$ must be included in any feasible solution to V2AUG if it represents the only possibility to connect a cut-component $C_i^{v_c}$ of a cut-vertex v_c to any other cut-component of v_c. In more detail, we consider for each cut-vertex v_c its set $\Gamma(v_c)$ and look for those edges being the only ones able to cover one of the tree-edges incident to v_c. Such augmentation edges are fixed by moving them from E_A to E_T; see edge e''_A in Figures 3.5b and 3.5c. The corresponding original augmentation edges from E_a are permanently marked to be included in any future solution. The whole procedure runs in $O(|V_T|\,|E_A|)$ time.

Note that the fixing can also be seen in the following way: In the block-cut graph we have detected that the minimal edge-cut in the block-cut graph is 2. Two edges generate this cut: one is a tree edge $e_0 \in E_0$, and the other one is an augmentation edge which has to be fixed, since it represented the only possibility to connect the two components obtained after removing e_0. The correctness of this test follows due to the fact that the block-cut graph has to be edge biconnected, i.e. the removal of any tree-edge must not disconnect it.

Shrinking

By fixing an edge, a cycle is introduced in T. This cycle forms a new vertex biconnected component that can be shrunk into a single new block-vertex v^* as shown in Figure 3.5d. Let $Z \subseteq V_T$ be the set of vertices forming the cycle. The following rules are applied:

1. Each block-vertex $v_b \in Z$ is re-mapped to v^*. All tree-edges between v_b and a vertex $u \notin Z$ are re-mapped to (v^*, u). Each cut-vertex $v_c \in Z$ having degree two is now completely covered and therefore handled in the same way. All the edges connecting vertices in Z are removed.

2. The remaining cut-vertices $v_c \in Z$ are not re-mapped to v^*. Instead, their membership to the new block is expressed via new edges (v^*, v_c).

3. All augmentation edges incident to one of the vertices in Z are superimposed anew on the modified block-cut tree according to the rules of Section 3.2.1.

The running time of this procedure is $O(|V_T| + |E_A|)$ due to the fact that each vertex or edge need to be considered at most once.

Block-Vertex Elimination

Block-vertices $v_b \in V_T$ having degree two in G_A and whose adjacent vertices v_1 and v_2 are cut-vertices, can be removed from the block-cut tree and replaced with a new edge (v_1, v_2).

Due to this elimination, the size of Ψ data-structure may be reduced. However, this procedure is usually only efficient on sparse graphs, where non-empty block-vertices exist. Figure 3.6a shows an example.

This simple procedure runs in $O(|V_T|)$ time, since each block-vertex has to be considered exactly once.

Path Compression

This procedure reduces the size of the input graph by compressing paths $P = \{v_1, e_1, \ldots, v_{l-1}, e_{l-1}, v_l\}$, $l \geq 4$, of the block-cut tree that satisfy the following properties:

- Vertices v_i $(i = 2, \ldots, l-1)$ are not incident to any augmentation edge, and
- v_1 and v_l are incident to at least one augmentation edge.

Such a path P can be compressed into the path $P' = \{v_1, e'_1, v^*_c, e'_2, v_l\}$ of length two, where the newly generated cut vertex v^*_c represents all inner vertices of P. An example of this path compression is shown in Figure 3.6.

Note that each such path P must contain at least one inner cut-vertex. Indeed, due to the elimination of empty block-vertices or the inner block-vertices of degree two, it often happens that two cut-vertices are adjacent. On the other side, it may never happen that two block-vertices become adjacent, since we start with the structure where the block-vertices and the cut-vertices alternate along any path in T, and we consequently remove only block vertices.

This procedure can be applied iteratively, as long as such paths exist.

Using a modified depth-first search algorithm, the identification of a path that can be compressed can be done in $O(|V_T|)$ time. Updating data-structures Ψ and Γ can take $O(|E_A| \cdot |V_T|)$ time. Theoretically, the compression might be done $O(|V_T|)$ times, which yields $O(|V_T|^2|E_A|)$ as the total worst-time complexity of this procedure.

The Outline of the Preprocessing Algorithm

Block-vertex elimination and path compression depend on the density of a graph and do not influence the edge elimination and the fixing of edges. Thus, we can divide the preprocessing into two phases and apply them independently to each other. In the first phase, the number of augmentation edges and tree edges is reduced due to edge elimination, fixing of edges, and shrinking procedures. In the second part, only the tree-structure is changed, due to path compression and block-vertices elimination procedures.

We propose a preprocessing algorithm whose outline is given in Figure 3.7. We start with edge elimination, after which possible fixing of edges is done. As soon as some edges are fixed, the shrinking is applied. After shrinking all cycles in T, the modifications are also reflected to the supporting data structures Γ and Ψ. Owing to the reductions, more edges may become available for elimination or fixing. Therefore, these reduction steps are repeated until no further shrinking is possible.

In the second preprocessing phase, we apply path compression and block-vertices elimination. Path compression is applied iteratively, as long as there are paths to be compressed. In each iteration, the corresponding Ψ and Γ data-structures need to be updated. In the final step, some possible block-vertices are eliminated.

Edge elimination, fixing of edges, and shrinking may theoretically iteratively be applied up to $O(|V_T|)$ times, which gives $O(|V_T|^2|E_A|)$ as upper bound of the first preprocessing phase. This is at the same time worst-time complexity of the whole preprocessing procedure.

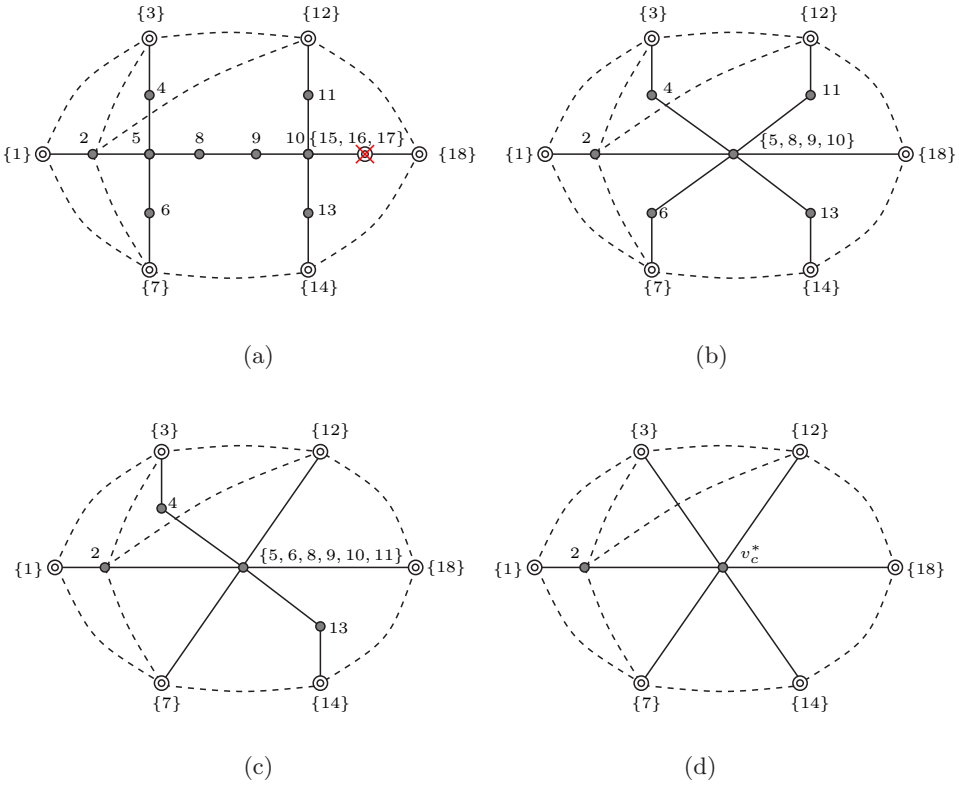


Figure 3.6: Block-vertex elimination and path compression: (a) The inner block-vertex $\{15, 16, 17\}$ has degree two in the block-cut graph, and thus, can be removed. (b) The tree path between 2 and $\{18\}$ can be compressed into a path of length two, with a new cut-vertex in the middle. (c) The tree path between $\{3\}$ and $\{14\}$ can be also compressed. (d) Finally, we obtain the cut-vertex vertex v_c^* which maps to the set $\{4, 5, 6, 8, 9, 10, 11, 12\}$ of original vertices in the block-cut tree. Thus, starting from a block-cut tree with 9 cut-vertices, we ended with a block-cut tree having only 2 cut-vertices.

Input: V2AUG problem instance consisting of
– weighted block-cut graph $G_A = (V_T, E_T \cup E_A)$, (= all possible connections)
– block-cut tree $T = (V_T, E_T)$ (= existing network)

Output: Reduced V2AUG instance

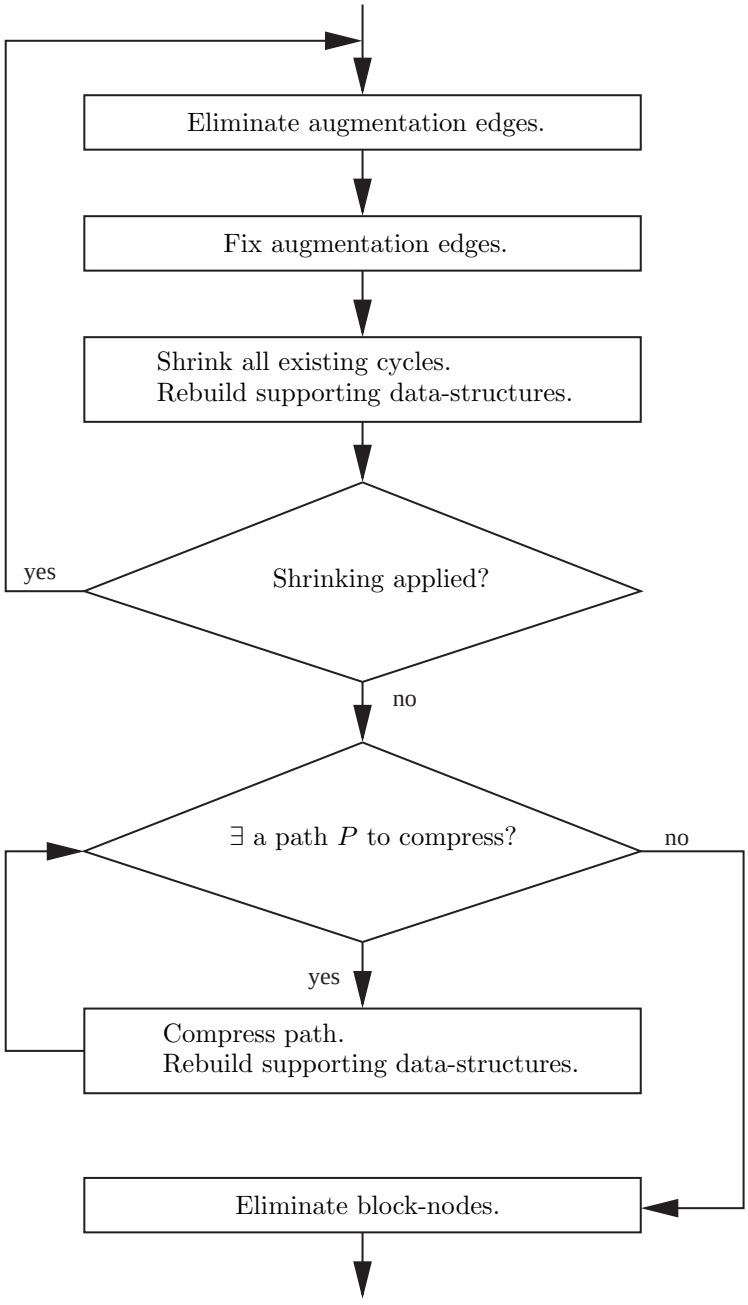


Figure 3.7: Basic structure of the proposed deterministic preprocessing.

Note that the worst case running time may be achieved only in extreme situations, where the starting graph can be solved to optimality, only by applying the preprocessing procedures. The expected total effort of preprocessing is lower. This is also documented in the empirical results of the next section.

3.2.4 Impacts of Preprocessing

Within this section, we show the impacts of the proposed preprocessing procedure. We first describe the set of instances on which our new memetic algorithm and the branch-and-cut-and-price approach are tested. Then we present the properties of reduced instances that are taken as input for testing these two V2AUG algorithms.

Benchmark Instances

To test the memetic algorithm, to compare it with previous approaches, and to test the branch-and-cut-and-price method, problem instances of different size and structure were used. Since shrinking can always trivially reduce the problem of augmenting a general connected graph G_0 to the problem of augmenting a tree, we consider here only instances in which the fixed graph G_0 is a spanning tree. The used test instances were adopted from the following two sources.

- Random instances created by means of Zhu's generator[5]:

  Table 3.1 shows the characteristics of 27 instance-groups `A1` to `R2`, each consisting of 30 different instances. We call them *random instances*, since they were randomly created by a program from Zhu [161]: Starting from $|V|$ vertices, edges are created between each pair of vertices $u, v \in V$, $u \neq v$, with the probabilities listed in column *dens*, the density of the graph. If the resulting graph is not biconnected, the creation is restarted. A random spanning tree is then determined on the graph yielding the set of fixed edges E_0. All other edges form set E_a and get assigned randomly chosen integer costs from the intervals listed in column $cost(e)$.

  Note that instances with the same names `A1` to `R2` and the same characteristics have already been used in previous works [162, 106, 144], however with only one representative instance per group instead of 30. Column $|E_a|$ of Table 3.1 lists the average numbers of augmentation edges and column CP(G_0) the average numbers of cut-points.

- Instances derived from Reinelt's TSP-library (TSPLIB)[6]:

  The larger instances listed in Table 3.2 are adopted from real-world traveling salesman problems. `pr226`, `lin318`, `pr439`, and `pcb442` are of Euclidean type, meaning that vertices represent points in the Euclidean plane, edges exist between any two vertices, and edge costs are the Euclidean distances of the corresponding points rounded up to the nearest integer value. The largest instance `pa561` is not of Euclidean type; it is a complete graph with edge costs directly given by a matrix – the triangle inequality is satisfied.

[5] Available at `www.ads.tuwien.ac.at/research/NetworkDesign/Augmentation`.
[6] Available at `www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95`.

Table 3.1: Characteristics of instance-groups created with Zhu's generator and average results of the preprocessing.

Group	$\|V\|$	*dens*	$c(e) \in$	$\|E_a\|$	$\mathrm{CP}(G_0)$	$\|V_T\|$	$\|E_A\|$	$\mathrm{CP}(T)$	t_{pre} [s]	$\mathrm{CP}(G_0)/\mathrm{CP}(T)$	$\|E_a\|/\|E_A\|$
A1	20	0.16	[1..190]	18	11	8	5	4	<0.1	3.0	3.4
A2	30	0.10	[1..435]	29	16	10	8	5	<0.1	3.2	3.8
A3	40	0.08	[1..780]	37	22	9	6	4	<0.1	4.9	6.0
A4	30	0.12	[1..435]	32	16	10	8	5	<0.1	3.4	4.2
A5	40	0.10	[1..780]	46	23	20	18	10	<0.1	2.2	2.5
B1	60	0.05	[1..1770]	54	35	14	10	7	<0.1	4.7	5.4
B2	20	0.50	[1..190]	81	10	20	44	10	<0.1	1.0	1.8
B3	50	0.06	[1..1225]	45	29	14	11	7	<0.1	4.1	4.1
B4	50	0.08	[1..1225]	61	27	26	26	14	<0.1	2.0	2.3
B5	60	0.07	[1..1770]	74	33	29	29	16	<0.1	2.1	2.6
B6	70	0.06	[1..2415]	96	39	45	48	24	0.1	1.6	2.0
C1	80	0.06	[1..3160]	113	43	50	55	26	0.1	1.6	2.1
C2	90	0.05	[1..4005]	125	50	53	55	29	0.1	1.7	2.3
C3	100	0.05	[1..4950]	153	54	63	73	34	0.1	1.6	2.1
C4	30	0.50	[1..435]	191	15	30	107	15	<0.1	1.0	1.8
D1	70	0.15	[1..2415]	279	37	70	196	37	<0.1	1.0	1.4
D2	40	0.50	[1..780]	349	20	40	187	20	<0.1	1.0	1.9
D3	90	0.15	[1..4005]	507	46	90	366	46	0.1	1.0	1.4
D4	80	0.15	[1..3160]	396	41	80	284	41	0.1	1.0	1.4
D5	100	0.15	[1..4950]	648	52	100	464	52	0.1	1.0	1.4
M1	70	0.15	[10..1000]	284	37	70	207	36	<0.1	1.0	1.4
M2	80	0.15	[10..1000]	388	42	80	278	42	<0.1	1.0	1.4
M3	90	0.15	[10..1000]	492	46	90	352	46	0.1	1.0	1.4
N1	100	0.25	[11..50]	1124	50	100	705	50	0.1	1.0	1.6
N2	110	0.25	[11..50]	1384	56	110	874	56	0.1	1.0	1.6
R1	200	0.50	[1..100]	9734	117	200	3888	117	4.3	1.0	2.5
R2	200	0.50	[5..100]	9744	118	200	3852	118	3.5	1.0	2.5

Table 3.2: Instances derived from the TSPLIB and results of the preprocessing.

Instance	$\|V\|$	Type	$\|E_a\|$	$\mathrm{CP}(G_0)$	$\|V_T\|$	$\|E_A\|$	$\mathrm{CP}(T)$	t_{pre} [s]	$\mathrm{CP}(G_0)/\mathrm{CP}(T)$	$\|E_a\|/\|E_A\|$
pr226-dt	226	Euclidean	361	192	226	256	192	1.4	1.0	1.4
pr226-sp	226	Euclidean	3987	187	226	2550	187	1.7	1.0	1.6
pr226	226	Euclidean	25200	187	226	7089	187	11.0	1.0	3.6
lin318-dt	318	Euclidean	623	248	318	434	248	9.9	1.0	1.4
lin318-sp	318	Euclidean	5495	246	318	1874	246	12.6	1.0	2.9
lin318	318	Euclidean	50086	246	318	9473	246	95.6	1.0	5.3
pr439-dt	439	Euclidean	859	358	439	490	358	30.1	1.0	1.8
pr439-sp	439	Euclidean	11183	366	439	3026	366	42.7	1.0	3.7
pr439	439	Euclidean	95703	366	439	18700	366	280.2	1.0	5.1
pcb442-dt	442	Euclidean	845	347	374	385	347	148.6	1.2	2.2
pcb442-sp	442	Euclidean	10528	345	442	2557	345	53.7	1.0	4.1
pcb442	442	Euclidean	97020	345	442	19824	345	299.5	1.0	4.9
pa561-sp	561	matrix	18504	406	561	5175	406	157.2	1.0	3.6
pa561	561	matrix	156520	406	561	40601	406	417.6	1.0	3.9

Since all these instances represent complete base graphs G, and incomplete graphs are of particular interest, too, additional sparse instances `pr226-sp`, `lin318-sp`, `pr439-sp`, `pcb442-sp`, and `pa561-sp` were derived from the original TSPLIB-graphs by considering for each vertex the edges to its $\lceil |V| \cdot 10\% \rceil$ nearest neighbors only, i.e. the 10%-*nearest-neighbor graphs*. In case of instance `pr226-sp`, the 10%-nearest-neighbor graph turned out to be not biconnected, and the 15%-nearest-neighbor graph was used instead.

For the Euclidean instances we further calculated the Delaunay triangulation yielding additional sparse instances `pr226-dt`, `lin318-dt`, `pr439-dt`, and `pcb442-dt`.

In all these cases, minimum spanning trees were chosen as fixed graphs G_0.

Impacts of Preprocessing

In their last six columns, Tables 3.1 and 3.2 show the results of the first preprocessing phase: the numbers of vertices $|V_T|$, augmentation edges $|E_A|$, and cut-vertices $\mathrm{CP}(T)$ of the block-cut-graphs, the CPU-times t_{pre} for preprocessing (in seconds) and the savings factors $\mathrm{CP}(G_0)/\mathrm{CP}(T)$ and $|E_a|/|E_A|$. In the case of random instances, these values are average values over the 30 instances per group. All experiments described in this section were performed on a Pentium-III/800MHz PC.

These results document that the fixing of augmentation edges, which enables a shrinking of the block-cut-graph and therefore a reduction of cut-vertices, is highly effective in sparser graphs like those of groups `A1` to `C3`. In these cases, the numbers of cut-vertices could often be reduced to less than one half. As a consequence, also the numbers of augmentation edges could be substantially reduced. 16% of the instances from groups `A1` to `B4` could even be

Table 3.3: The structure of the block-cut tree for the instance-groups created with Zhu's generator.

Group	$Path_{tot}$	$\#Path$	$Path_{avg}$	$Path_{\neq 0}$ [%]	*Block-vrtx*	*All-vrtx*	*Tree-size*
A1	3.8	1.2	3.2	33.3	0.1	2.4	7.7
A2	4.2	1.3	3.3	36.7	0.1	3.2	10.5
A3	5.5	1.6	3.3	36.7	0.0	3.3	9.0
A4	6.0	1.8	3.3	33.3	0.2	2.6	9.6
A5	4.8	1.5	3.2	33.3	0.1	4.0	19.9
B1	5.8	1.8	3.2	36.7	0.0	4.7	13.8
B2	4.2	1.3	3.2	30.0	0.0	1.5	19.9
B3	3.9	1.2	3.3	50.0	0.0	3.8	13.9
B4	5.4	1.7	3.1	46.7	0.1	4.9	26.2
B5	3.9	1.3	3.1	60.0	0.1	5.2	29.3
B6	3.7	1.2	3.2	36.7	0.1	6.6	45.2
C1	5.2	1.7	3.0	56.7	0.1	6.6	49.8
C2	5.3	1.7	3.2	60.0	0.1	8.1	52.6
C3	4.4	1.4	3.1	50.0	0.2	8.3	62.8
C4	3.3	1.0	3.3	20.0	0.0	1.1	30.0
D1	4.7	1.6	3.0	60.0	0.0	2.5	69.7
D2	4.2	1.4	3.0	40.0	0.0	1.8	40.0
D3	3.5	1.1	3.1	26.7	0.0	1.6	90.0
D4	3.7	1.2	3.1	56.7	0.0	2.1	79.8
D5	3.0	1.0	3.0	16.7	0.0	1.3	100.0
M1	4.1	1.3	3.2	33.3	0.0	1.9	69.7
M2	4.0	1.3	3.1	36.7	0.0	1.9	80.0
M3	3.8	1.2	3.1	43.3	0.0	2.1	90.0
N1	3.6	1.2	3.1	43.3	0.0	2.1	100.0
N2	4.2	1.4	3.0	33.3	0.0	1.9	110.0
R1	4.3	1.3	3.3	43.3	0.0	2.8	160.0
R2	4.5	1.3	3.4	40.0	0.0	3.2	166.7

Table 3.4: The structure of the block-cut tree for the instances derived from the TSPLIB.

Group	$Path_{tot}$	$\#Path$	$Path_{avg}$	*All-vrtx*	*Tree-size*
pr226-dt	33	5	6.6	41	226
pr226-sp	0	0	0.0	1	226
pr226	0	0	0.0	0	226
lin318-dt	17	5	3.4	36	318
lin318-sp	0	0	0.0	0	318
lin318	0	0	0.0	0	318
pr439-dt	83	20	4.2	82	439
pr439-sp	8	2	4.0	10	439
pr439	0	0	0.0	3	439
pcb442-dt	71	18	3.9	75	374
pcb442-sp	0	0	0.0	6	442
pcb442	0	0	0.0	6	442
pa561-sp	3	1	3.0	9	561
pa561	6	2	3.0	14	561

completely solved by preprocessing since it was able to reduce each block-cut graph to a single block-vertex.

On denser problem instances, no edges could be fixed, thus, the numbers of cut-vertices in the block-cut graphs are identical to the numbers of cut-points in the original graphs. However, edge-elimination was in these cases highly effective. On average over all instances, the number of augmentation edges that need to be considered for further optimization could be reduced to about a quarter of the edges in E_a.

After the first phase of the preprocessing has been applied, we study the structure of the obtained block-cut tree, in order to avoid an ineffective application of the second preprocessing phase. As an illustration, Tables 3.3 and 3.4 show statistics about the structures of the underlying graphs. We sum up the path lengths and present the following three values: $Path_{tot}$ represents the total length of all paths that can be compressed; $\#Path$ shows the total number of such paths in the block-cut tree; $Path_{avg}$ is the average length of a path that allows compression ($Path_{tot}/\#Path$). These values are averaged only over the instances with non zero path length – the total percentage of such instances is given in $Path_{\neq 0}$ [%] column. We also provide the total number of non-empty block-vertices without adjacent edges (*Block-vrtx*) and the total number of vertices not adjacent to any augmentation edge (*All-vrtx*). The last column (*Tree-size*) shows the size of the underlying block-cut tree after the application of the first preprocessing phase. On the set of Zhu's instances, Table 3.3 shows average values over the 30 instances per group. For each single TSPLIB instance the corresponding values are presented in Table 3.4.

The analysis of the block-cut tree structure after the first preprocessing phase shows that, in all benchmark instances obtained from Zhu's generator, the average length of the paths that can be compressed is close to 3. In the worst case it does not exceed 3.4. However, $Path_{\neq 0}$ [%]

values indicate that not all of instances from a group contain such paths, but usually 1/3 of them. Regarding the instances derived from the TSP-library, with the only exception of the Delaunay graphs, paths that may be compressed exist in very few cases. Their average length does not exceed 7. Furthermore, the number of non-empty block-vertices without adjacent edges (*Block-vrtx*) is zero for all these instances.

Note that only the paths of length larger than two are of interest, and that each such path may be substituted by a new one of length two. Thus, the total number of saved vertices during the path compression can be approximately calculated as:

$$\#Path \times (Path_{avg} - 2) \ .$$

This means that in case of Delaunay graphs the total savings on the number of vertices may be in the best case about 10% (instances `pr226-dt` and `pr439-dt`).

Finally, we can conclude that the savings of the relatively time-consuming second preprocessing phase are not high enough to be applied to our benchmark instances. The preprocessed V2AUG instances provided in the rest of the thesis are therefore treated with the first preprocessing phase only.

3.3 A Memetic Algorithm for V2AUG

We propose a memetic algorithm for V2AUG which is based on a straight-forward steady-state evolutionary algorithm as shown in the last chapter in Algorithm 3. Within this section, we describe in detail the core of our MA implementation: solution representation technique, the local improvement strategy, and initialization, recombination, and mutation procedures.

3.3.1 Representation of Solutions

Many evolutionary algorithms for combinatorial optimization problems represent candidate solutions by vectors of fixed length and apply classical operators as k-point or uniform crossover and position-wise mutation. In Ljubić and Kratica [106] this concept has been followed within a genetic algorithm for V2AUG: solution were represented by a vector of $|E_a|$ Booleans indicating which augmentation edges are included in the solution.

The main disadvantage of this approach is that created candidate solutions need not to be feasible; an expensive repair strategy, which also reduces the variation operators' locality and heritability is necessary. Furthermore, the memory effort for storing a solution is $O(|E_a|)$.

In our memetic algorithm, a candidate solution is represented by directly storing references to all the augmentation edges of $S \subseteq E_A$ in the form of a hash-table. In this way, only $O(|S|) = O(|V|)$ space is needed, since $|S| < |V|$ in any solution that is locally optimal with respect to the number of edges (in fact, $|S| \ll |V_T|$ in most larger instances). Using a hash-table allows an edge to be added, deleted, or checked for existence in expected constant time[7]. More about the advantages of these so-called *edge-set encoding* can be found in [143].

[7] We used LEDA library implementation, `http://www.algorithmic-solutions.com/enleda.htm`, for most of the underlying data structures.

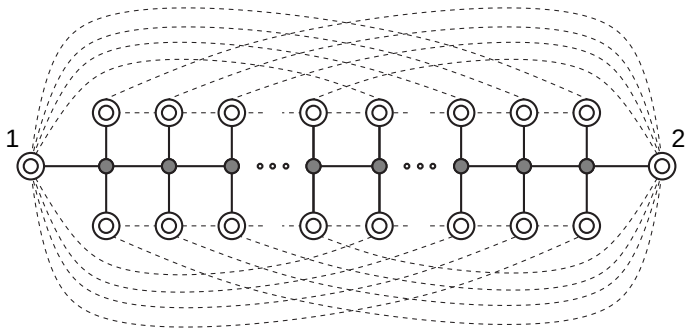


Figure 3.8: Worst case example for local improvement.

3.3.2 Local Improvement

A feasible candidate solution S is said to be *locally optimal* with respect to the number of edges, if the removal of any edge $e \in S$ violates the biconnectivity-property of graph $G_s = (V, E_0 \cup E_s)$, where $E_s \subseteq E_a$ is the set of original augmentation edges corresponding to S united with the edges fixed during preprocessing. The local improvement operator shown in Algorithm 7 and described in the following makes a given feasible solution locally optimal by removing redundant edges. Recall that an edge $e \in S$ is said to be redundant if its removal does not violate the biconnectivity property of G_s. Our local improvement operator is specifically designed to perform efficiently on sparse solutions where $|S| \in O(|V_T|)$, since the solutions created by initialization, recombination, and mutation usually do not have many redundant edges.

As first step, the algorithm identifies so-called *obviously essential edges* that must remain in S. An edge $e \in S$ is obviously essential if it is the only one from S to be able to connect a certain cut-component $C^i_{v_c}$ of a cut-vertex v_c to any other of v_c's cut-components—compare the fixing of edges during preprocessing. Such obviously essential edges from S are determined efficiently by finding each tree-edge e_T incident to a cut-vertex v_c and covered only once by an edge $e \in S$ that is not incident to v_c; e is then obviously essential. The worst-case time complexity of this part of the algorithm, when implemented as shown in the pseudo-code, is $O(|S|\,|V_T|)$. However, since $|\Psi(e)|$ is in $O(\log |V_T|)$ in the expected case, the average running-time is $O(|V_T| + |S| \log |V_T|)$.

The remaining not obviously essential edges from S, in the pseudo-code denoted by set R, are then processed one-by-one in decreasing-costs order. Each edge $e \in R$ is temporarily removed from S, and the cut-vertices in whose covering e contributes, i.e. all $v_c \in \Psi(e)$, are checked if they remain covered (see Sect. 3.2.2). If any of them is now uncovered, e is not redundant and therefore included in S again.

In the worst case, the total computational effort of this local improvement procedure is $O(|S|^2 \cdot |V_T| \cdot \alpha(|V_T|, |S|))$ per call. The example in Figure 3.8 illustrates this: Assuming that each block-vertex in the shown block-cut graph represents a single vertex in the original graph G_0, there are $|V_T| = (|V| - 2)/3 \in O(|V|)$ cut-vertices having all degree four. No augmentation edge is obviously essential. $O(|S|) = O(|V_T|)$ augmentation edges incident to

Data : An augmentation set of edges $S \subset E_A$ representing a feasible solution.
Result: A modified set S, representing a locally optimal solution with respect to the number of edges.

$R \leftarrow S$; // set of edges to be considered for removal;
// look for obviously essential edges:
for $e_T \in E_T$ **do**
    $covered[e_T] \leftarrow \emptyset$;
end
for $e \in S$ **do**
    for $v_c \in \Psi(e)$ **do**
        $covered[e_{T_1}^{v_c}(e)] \leftarrow covered[e_{T_1}^{v_c}(e)] \cup \{e\}$;
        $covered[e_{T_2}^{v_c}(e)] \leftarrow covered[e_{T_2}^{v_c}(e)] \cup \{e\}$;
    end
end
// remove obviously essential edges from R:
for $e_T \in E_T$ **do**
    if $|covered[e_T]| = 1$ **then**
        $R \leftarrow R \setminus covered[e_T]$;
    end
end
// check remaining edges in R:
for $e \in R$ *in decreasing cost-order* **do**
    $S \leftarrow S \setminus \{e\}$;
    if $\exists v_c \in \Psi(e)$*:* v_c *is uncovered in* G_s **then**
        $S \leftarrow S \cup \{e\}$;
    end
end
return S;

Algorithm 7: The local improvement procedure.

block-vertices 1 and 2 contribute in the covering of each cut-vertex. On the other side, each of these augmentation edges contributes in the covering of $O(|V_T|)$ cut-vertices. Since the time for checking whether a single cut-vertex remains covered when a certain augmentation edge is removed is $O(|S| \cdot \alpha(|V_T|, |S|))$, it takes $O(|V_T| \cdot |S| \cdot \alpha(|V_T|, |S|))$ time to completely check an augmentation edge for redundancy, and the overall effort is $O(|S|^2 \cdot |V_T| \cdot \alpha(|V_T|, |S|)) = O(|V_T|^3 \cdot \alpha(|V_T|, |S|))$ per solution.

However, since $|\Psi(e)| \in O(\log |V_T|)$ on average, the average time for checking one edge from R for redundancy is $O(|S| \log |V_T|)$, and the average total time for one complete local improvement is $O(|V_T| + |S| \log |V_T| + |R| \cdot |S| \log |V_T| \cdot \alpha(|V_T|, |S|))$. In case of the memetic algorithm's candidate solutions, usually most edges are obviously essential; thus, $|R|$ is generally small.

Another possibility for checking an edge $e \in S$ for redundancy is to temporarily remove it and to check whether the augmented graph $G'_s = (V, E_0 \cup E'_s)$, where $E'_s \subset E_a$ is the set of original augmentation edges corresponding to $S \setminus \{e\}$, remains biconnected. Using the algorithm from Tarjan [156], the biconnectivity-check can be performed in time $O(|V| + |S|)$. However, experimental results have shown that in the memetic algorithm, this alternative redundancy-check is particularly on larger problem instances significantly slower than the originally proposed one. The explanation lies in the fact that in locally optimal solutions, $|S|$ is typically substantially smaller than $|V_T|$, since several cut-points can often be covered by a single augmentation edge. Since we apply local improvement only to candidate solutions obtained from the initialization, recombination, or mutation procedures, and these solutions do not usually have many redundant edges, $|S| \ll |V_T|$ also holds in most of our cases. On the other hand, when considering local improvement without the memetic algorithm framework and the number of augmentation edges $|S|$ may be large, the redundancy-check using Tarjan's algorithm would presumably be more efficient.

3.3.3 Initialization

A solution of the initial population is created by starting with an empty edge-set S. Iteratively, an edge is randomly selected from E_A and included in S if it is not redundant. This process is repeated until all cut-vertices are completely covered, thus, until the augmented graph G_s becomes biconnected.

Intuitively, cheaper edges appear in optimum solutions more likely than expensive edges. Therefore, the selection of edges for inclusion is biased toward cheaper edges according to a scheme originally proposed in Raidl [140] for the selection of edges to be included by mutation in candidate solutions to the degree-constrained minimum spanning tree problem: During preprocessing, the edges in E_A are sorted according to their costs. In this way, each edge has a rank, with ties broken randomly. A rank, thus an edge, is selected by sampling the random variable

$$rank = \lfloor |\mathcal{N}(0, s)| \, |V_T| \rfloor \bmod |E_A| + 1 \,, \tag{3.3}$$

where $\mathcal{N}(0, s)$ is a normally distributed random variable with zero mean and standard deviation s, a strategy parameter controlling the strength of the scheme's bias toward cheap edges.

The check, whether an edge e_A actually helps in covering a cut-vertex—thus, if e_A is not redundant—can be performed efficiently in nearly constant amortized time $\alpha(|V_T|, |E_A|)$, when union-find data structures are maintained for all cut-vertices of degree greater than three. Compare the check whether a set of augmentation edges covers a cut-vertex described in Sect. 3.2.2. Thus, the running time of the initialization is $O(|E_A| \cdot |V_T| \cdot \alpha(|V_T|, |E_A|))$ in the worst-case and $O(|E_A| \cdot \log |V_T| \cdot \alpha(|V_T|, |E_A|))$ on average. A solution created in this way is not necessarily locally optimal since the inclusion of an edge may make previously included edges redundant. Therefore, the memetic algorithm applies local improvement also to each initial solution.

Algorithm 8 shows the pseudo-code of the initialization procedure.

Data : The weighted block-cut graph $G_A = (V_T, E_T \cup E_A)$.
Result: A feasible solution represented by its set of augmentation edges $S \subset E_A$.
$S \leftarrow \emptyset$;
repeat
  select $e \in E_A$ by sampling random *rank* variable;
  if $\exists v_c \in \Psi(e)$: *edge e helps in covering* v_c **then**
    $S \leftarrow S \cup \{e\}$;
  end
until *all cut-vertices* $v_c \in V_T$ *are covered*;
return S;

Algorithm 8: Initialization.

3.3.4 Recombination

The recombination operator was designed with the aim to provide highest possible heritability, i.e. an offspring should consist of edges from its two parental solutions only. In the first step, edges common in both parents S_1 and S_2 are always adopted: $S \leftarrow S_1 \cap S_2$. Then, while not all cut-vertices are completely covered, an edge is selected from the set of remaining parental edges $(S_1 \cup S_2) \setminus S$ and included in the offspring S if it is not redundant. To emphasize the inclusion of low-cost edges again, they are selected via binary tournaments with replacement.

Algorithm 9 provides the recombination's pseudo-code. The computational effort of the whole recombination procedure is $O((|S_1| + |S_2|) \cdot |V_T| \cdot \alpha(|V_T|, |S_1| + |S_2|))$ in the worst case and $O((|S_1| + |S_2|) \cdot \log |V_T| \cdot \alpha(|V_T|, |S_1| + |S_2|))$ on average.

3.3.5 Edge-Delete Mutation

The aim of mutation is to introduce new edges not appearing in the population into candidate solutions. Algorithm 10 shows the mutation procedure in pseudo-code. From the candidate solution S, an edge e is selected and removed. That way, one or more cut-vertices from $\Psi(e)$ become uncovered. These uncovered cut-vertices are identified and processed in random order:

Data : Two parental solutions given by their sets of edges S_1 and S_2.
Result: A new feasible solution represented by its set of augmentation edges $S \subset E_A$.
$S \leftarrow S_1 \cap S_2$;
$R \leftarrow (S_1 \cup S_2) \setminus S$;
repeat
  select $e \in R$ via binary tournament selection;
  $R \leftarrow R \setminus \{e\}$;
  if $\exists v_c \in \Psi(e)$: *edge e helps in covering* v_c **then**
    // edge e is not redundant
    $S \leftarrow S \cup \{e\}$;
  end
until *all cut-vertices* $v_c \in V_T$ *are covered*;
return S;

Algorithm 9: Recombination.

Data : A feasible solution given by its set of augmentation edges $S \subset E_A$.
Result: A mutated solution S.
select $e \in S$ via binary tournament selection;
$S \leftarrow S \setminus \{e\}$;
for *uncovered* $v_c \in \Psi(e)$ *in random order* **do**
  $F \leftarrow \Gamma(v_c)$;
  while v_c *is uncovered* **do**
    pick $e' \in F$ randomly; $F \leftarrow F \setminus \{e'\}$;
    if e' *helps in covering* v_c **then**
      $S \leftarrow S \cup \{e'\}$;
    end
  end
end
return S;

Algorithm 10: Edge-delete mutation.

For each such cut-vertex v_c, the edges from $\Gamma(v_c)$ are considered in random order and included in S if they help in covering of v_c, i.e., if they connect two yet unconnected cut-components of v_c.

The selection of the edge to be removed is biased toward more expensive edges by performing a binary tournament with replacement on S. The new edges to be included in S for reestablishing biconnectivity are chosen in an unbiased way to not reduce the population's diversity too much.

As initialization and recombination, this procedure does not guarantee to yield a locally optimal solution. Therefore, the memetic algorithm applies local improvement also after mutation. An upper bound for the worst-case computational effort of mutation is $O(|V_T| \cdot |E_A| \cdot \alpha(|V_T|, |E_A|))$. However, mutation is substantially faster in practice, and the time needed for local improvement dominates the time for mutation.

3.3.6 Empirical Results

The following setup was used for the memetic algorithm as it proved to be robust for many different classes of instances in preliminary tests: Population size $|P| = 800$; group size for tournament selection $k = 5$; parameter for biasing initialization to include cheaper edges $s = 2.5$; crossover probability $p_{\text{cro}} = 1$; mutation probability $p_{\text{mut}} = 0.7$. Each run was terminated when no new best solution could be identified during the last $\Omega = 10\,000$ iterations. On all the instances we considered, this criterion allowed the MA to converge so that only minor improvements in the quality of final solutions can be expected when prolonging the runs. Thus, the main goal was to find high-quality solutions, and running times were considered only secondary.

All experiments described in this section were performed on a Pentium-III/800MHz PC.

We compare the memetic algorithm (MA) to the heuristics from Khuller and Thurimella [95] (KT), [162] Zhu et al.(ZKR), and the genetic algorithm from Ljubić and Kratica [106] (LK). These previous heuristics were implemented and applied as described in these works. Thus, the new enhanced preprocessing of the MA was not used by them.

Solutions obtained using KT or ZKR approach directly depend on the choice of the leaf vertex that becomes a root of the minimum outgoing spanning arborescence (see Section 3.1). Therefore, for each instance, KT has been run with each leaf-vertex of the block-cut tree becoming once the root of the arborescence, and the best solution obtained in this way is regarded as KT's final solution to the instance.

ZKR could only be applied to the smaller random instances of groups `A1` to `N2` because of its high computational effort. (The total CPU-time of a run was limited to 20 000 seconds.) For instances of groups `M1` to `N2`, only 10% of all leaves were subsequently tried as root of the block-cut tree, while for the other instances, all leaves were considered.

The setup of LK was the same as described in [106] except of the termination criterion, which was changed to be similar to that of the MA in order to ensure convergence: A run was terminated when no new best solution could be identified during the last 100 000 evaluations. (Note, however, that the number of evaluations of the MA and LK may not directly be compared

due to the different computational complexities of the algorithms.)

Table 3.5 shows average results of the four approaches on the random instances. Each heuristic was run once on each of the 30 instances of each group. We were able to solve all these instances also to guaranteed optimality by the branch-and-cut-and-price approach described in the next section.

Column $|E_s^*|$ lists average numbers of edges in these optimal solutions. The qualities of the solutions E_s obtained by the algorithms are reported as percentage gaps defined as:

$$\%\text{-}gap = \frac{cost(E_s) - c(E_s^*)}{c(E_s^*)} \cdot 100\%. \tag{3.4}$$

$c(E_s^*)$ represent optimal or best-known values. Standard deviations of average gaps (σ) are also presented in the table. For LK and MA, average CPU-times and numbers of evaluated solutions until the best solutions were found (t, respectively *evals*), and success rates (sr), i.e. the percentage of instances for which optimal solutions could be found, are reported in addition. CPU-times include preprocessing: in case of KT, ZKR, and LK the derivation of the block-cut graph according to Sect. 2.1.5, in case of MA additionally the more sophisticated reductions and the creation of supporting data structures—in particular Γ and Ψ—according to Sect. 3.2.

Results show that MA clearly outperformed the other heuristics in most cases. It could find optimal solutions to all instances of groups `A1` to `D4` and `M1` to `M3`. On the remaining random instances, MA was able to identify high-quality solutions with an average gap of only 0.33%. KT yielded in all cases the worst results. Among ZKR and LK, ZKR could usually identify slightly better solutions. Quality differences become most apparent in groups `M1` to `R2`.

Regarding the running times, KT was usually fastest (about 2 to 3 times faster than the times reported for MA), followed by MA. LK was usually much slower, in particular on the larger instances. ZKR needed in any case the most time. With over 15 000 seconds CPU-time for instances of group `N2`, ZKR is definitely only suitable for small instances.

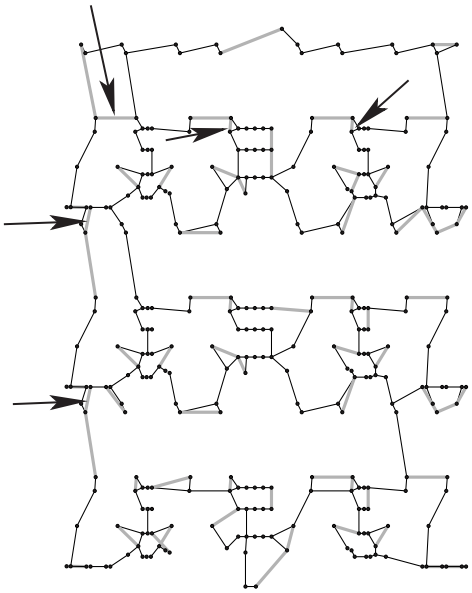
Table 3.6 shows results for the larger TSPLIB-derived instances. Optimum solutions could be found by branch-and-cut only up to instance `pcb442-sp`. Total costs of these optimum solutions—or if unknown best-known solution values—and the numbers of edges in those solutions are listed in columns $c(E_s^*)$ and $|E_s^*|$, respectively.

On these TSPLIB-derived instances, ZKR never terminated within the allowed maximum time of 20 000 seconds, and LK could obtain meaningful results on the eight sparse Euclidean instances only. Because of the stochastic nature of LK and MA, these heuristics were performed 30 times on each considered instance and Table 3.6 prints average results for them.

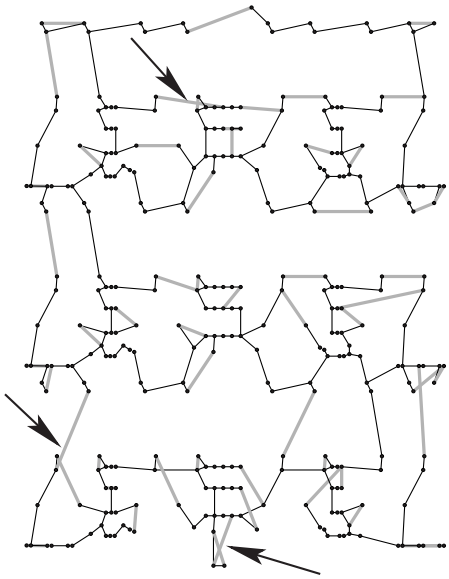
In contrast to ZKR and LK, MA scales well to the larger instances. Its CPU-time increases only moderately with the problem size due to the relatively low computational complexities of local improvement, recombination, and mutation. Because of the data structures created during preprocessing, MA required up to 420MB main memory for the largest instance `pa561` with $|E_a| = 156\,520$ augmentation edges. MA's solutions are of high quality again: On average, the gap was only 0.65%, and optimum or best-known solutions could be found several times. KT followed far behind with gaps between 19.6% and 32.6%; LK's results were even worse: its

Table 3.5: Results on random instances obtained by the heuristics from Khuller and Thurimella (KT) and Zhu et al. (ZKR), the genetic algorithm from Ljubić and Kratica (LK), and the memetic algorithm (MA).

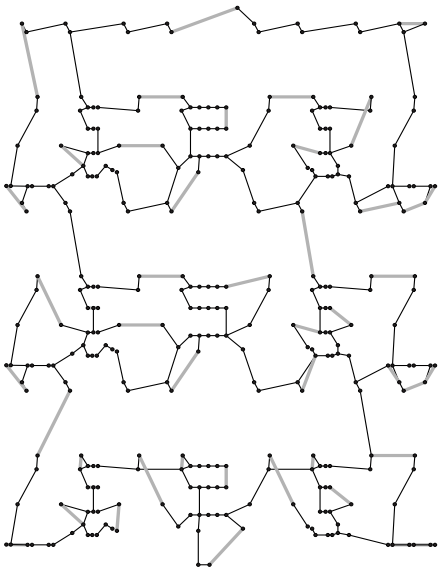
Grp.	$\lvert E_s^* \rvert$	KT			ZKR			LK					MA				
		%-*gap*	σ	t [s]	%-*gap*	σ	t [s]	%-*gap*	σ	t [s]	*evals*	*sr* [%]	%-*gap*	σ	t [s]	*evals*	*sr* [%]
A1	6	1.2	2.2	0.1	0.0	0.0	0.8	0.5	1.6	<0.1	623	90.0	0.0	0.0	<0.1	586	100.0
A2	9	4.6	6.0	0.2	0.0	0.0	4.3	0.3	1.8	<0.1	2116	96.7	0.0	0.0	<0.1	666	100.0
A3	12	3.9	4.0	0.2	<0.1	0.1	14.2	0.0	0.0	0.1	4041	100.0	0.0	0.0	<0.1	506	100.0
A4	10	5.1	5.3	0.2	0.2	0.6	5.5	0.1	0.3	0.1	3038	93.3	0.0	0.0	<0.1	453	100.0
A5	12	7.7	6.4	0.2	<0.1	0.2	18.5	0.0	0.0	0.1	2640	100.0	0.0	0.0	<0.1	776	100.0
B1	18	4.4	4.2	0.4	0.1	0.5	73.5	<0.1	0.1	0.4	3788	96.7	0.0	0.0	<0.1	613	100.0
B2	7	4.5	6.7	0.1	0.0	0.0	6.1	<0.1	0.1	0.1	6103	96.7	0.0	0.0	0.1	887	100.0
B3	16	4.9	5.2	0.3	0.0	0.0	33.1	0.0	0.0	0.3	5951	100.0	0.0	0.0	<0.1	696	100.0
B4	16	6.2	5.6	0.3	0.1	0.4	64.4	0.2	0.8	0.4	5035	93.3	0.0	0.0	0.1	776	100.0
B5	19	4.9	4.2	0.4	0.1	0.3	131.9	0.2	0.9	0.7	8178	90.0	0.0	0.0	0.1	875	100.0
B6	22	7.5	4.4	0.7	<0.1	0.1	328.0	0.1	0.5	2.2	20615	86.7	0.0	0.0	0.3	1022	100.0
C1	26	6.8	4.5	0.9	<0.1	0.1	687.2	0.4	0.9	3.5	17681	80.0	0.0	0.0	0.4	1073	100.0
C2	29	7.6	4.2	1.1	0.2	0.6	1121.2	0.9	1.5	4.4	20166	60.0	0.0	0.0	0.5	1176	100.0
C3	32	8.9	5.6	1.6	0.1	0.2	2331.1	0.5	1.0	7.6	25078	53.3	0.0	0.0	0.6	1160	100.0
C4	11	4.6	7.3	0.3	<0.1	0.2	71.3	0.5	1.4	0.7	17303	80.0	0.0	0.0	0.4	1409	100.0
D1	23	7.6	4.5	1.1	0.2	0.5	1805.5	0.9	1.5	6.0	53353	43.3	0.0	0.0	1.7	1915	100.0
D2	14	5.0	5.0	0.6	0.1	0.2	403.0	0.4	1.1	2.4	31420	86.7	0.0	0.0	0.7	1838	100.0
D3	31	6.9	4.0	2.4	0.2	0.4	11046.5	1.1	1.2	15.9	51278	20.0	0.0	0.0	3.9	3016	100.0
D4	27	9.4	5.2	1.8	0.1	0.2	5208.8	1.6	2.0	11.2	49910	36.7	0.0	0.0	2.9	2686	100.0
D5	34	9.7	4.8	3.5	0.1	3.2	21762.5	1.8	2.2	25.5	59190	26.7	<0.1	0.1	5.9	4167	96.7
M1	23	8.5	4.9	1.2	0.4	0.9	237.0	1.7	2.1	6.7	49498	30.0	0.0	0.0	1.7	1984	100.0
M2	27	8.3	3.6	1.7	0.6	0.9	503.7	2.2	1.7	21.9	82105	10.0	0.0	0.0	2.9	2719	100.0
M3	30	9.8	4.5	2.4	0.3	0.6	1178.4	1.4	1.4	33.4	87141	26.7	0.0	0.0	4.3	3354	100.0
N1	27	34.4	6.5	5.2	3.2	1.7	14993.3	13.2	4.3	107.0	147146	0.0	0.2	0.3	9.5	8387	60.0
N2	29	36.5	5.7	7.0	4.1	2.3	16006.1	16.4	4.3	147.5	147030	0.0	0.4	0.5	13.7	11694	43.3
R1	57	16.3	3.8	64.3	–	–	–	10.7	4.4	4896.1	228723	0.0	0.1	0.2	39.8	9766	63.3
R2	47	31.9	4.2	91.6	–	–	–	20.2	4.7	5232.4	309073	0.0	0.4	0.4	58.5	21877	13.3



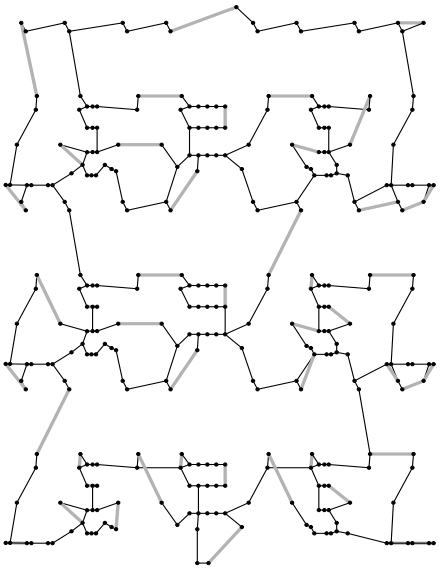
(a) KT: %-$gap = 29.3$, $|E_s| = 69$



(b) LK: %-$gap = 35.8$, $|E_s| = 58$



(c) MA: %-$gap = 0.09$, $|E_s| = 45$



(d) OPT: $|E_s| = 46$

Figure 3.9: Exemplary solutions to problem instance `lin318-sp` found by (a) Khuller and Thurimella's heuristic, (b) the genetic algorithm from Ljubić and Kratica, (c) the memetic algorithm and (d) the branch-and-cut algorithm (see next section). Solution edges are shown in gray. In (a), arrows mark obviously redundant edges. In (b), arrows mark obviously suboptimal crossing augmentation edges.

Table 3.6: Results on the TSPLIB-derived instances obtained by the heuristics from Khuller and Thurimella (KT) and Zhu et al. (ZKR), the genetic algorithm from Ljubić and Kratica (LK), and the memetic algorithm (MA). Values marked by '*' are the best-known feasible solutions.

Instance	$c(E_s^*)$	$\lvert E_s^* \rvert$	KT		LK			MA				
			%-*gap*	t [s]	%-*gap*	σ	t [s]	%-*gap*	σ	t [s]	*evals*	*sr* [%]
`pr226-dt`	25152	25	19.6	1.4	26.6	8.8	47.8	0.0	0.0	3.4	2188	100.0
`pr226-sp`	22824	24	22.5	11.7	27.3	4.8	640.2	0.1	0.6	17.6	9800	96.7
`pr226`	22824	24	22.0	138.9	–	–	–	2.6	1.2	33.2	13073	16.7
`lin318-dt`	12013	45	28.1	5.0	20.9	2.3	246.7	<0.1	<0.1	21.9	10051	3.3
`lin318-sp`	11797	46	29.3	33.2	41.1	3.8	2633.7	0.3	0.3	40.8	27130	6.7
`lin318`	11797	46	29.3	620.4	–	–	–	1.0	0.5	128.9	23391	0.0
`pr439-dt`	28310	52	20.6	8.3	25.1	6.0	491.6	0.0	0.0	43.3	7026	100.0
`pr439-sp`	26800	48	21.2	71.0	40.5	4.3	13471.2	1.1	0.7	79.5	12164	20.0
`pr439`	26800	48	21.2	1498.5	–	–	–	2.5	1.1	408.1	22301	0.0
`pcb442-dt`	10328	60	31.4	12.3	21.4	3.1	320.5	< 0.1	0.1	233.1	8472	83.3
`pcb442-sp`	10460	60	32.6	106.5	33.8	5.2	18429.2	0.3	0.2	91.9	16902	6.7
`pcb442`	10460	60	30.7	2030.8	–	–	–	0.3	0.2	366.3	21493	0.0
`pa561-sp`	782*	101*	31.1	303.0	–	–	–	0.3	0.1	250.0	20868	3.3
`pa561`	784*	101*	30.4	5482.7	–	–	–	0.4	0.4	599.8	34710	13.3

average gaps are all larger than 20.9%.

Statistical t-tests were performed and indicate that the quality differences between MA's solutions and those of the other approaches are significant at a 0.1% error-level on each instance. This also holds for the results on random instances shown in Table 3.5, except in those cases were also ZKR was able to identify always optimal solutions.

Figure 3.9 shows three exemplary solutions to the Euclidean problem instance `lin318-sp` found by KT, LK, and MA. Obviously redundant edges, as they are contained in the solution of KT, can never appear in a solution of MA due to its local improvement procedure.

The preprocessing proposed in Section 3.2 can also be adapted to work with KT, ZKR, and LK. Tests we performed indicate that in particular the total running times of these approaches are reduced significantly in this way. However, the quality of obtained solutions was not substantially higher. On average over all random and TSPLIB-derived instances where the individual approaches terminated within the allowed time of 20 000 seconds, the total times were reduced by the factors 0.62 in case of KT, 0.81 in case of ZKR, and 0.27 in case of LK. The %-gaps were reduced on average by the factors 0.92 in case of KT, 0.96 in case of ZKR, and 0.91 in case of LK. On several instances of groups `A1` to `C4`, the combinations of our preprocessing with KT, ZKR, and LK were able to identify optimal solutions as the MA did. Nevertheless, on the larger and more complicated instances, these approaches were still not competitive with the MA.

Table 3.7: Fitness distance correlation coefficient ρ, average distance $\overline{d_{\text{opt}}}$ of locally improved random solutions to the optimum, average distance $\overline{d_{\text{loc}}}$ between locally improved random solutions, and average probabilities (in percent) PD_{cross} and PD_{mut} that recombination, respectively mutation, followed by local improvement produces a candidate solution being identical to a parent.

Instance	ρ	$\overline{d_{\text{opt}}}$	$\overline{d_{\text{loc}}}$	PD_{cross} [%]	PD_{mut} [%]
`N1` (1. instance)	0.57	53.4	62.7	15.3	14.1
`N2` (1. instance)	0.54	58.2	69.4	13.8	14.3
`R1` (1. instance)	0.51	107.5	116.9	13.0	3.4
`R2` (1. instance)	0.55	99.9	118.2	12.5	4.5
`pr226-sp`	0.53	66.6	71.5	17.0	12.6
`pr226`	0.52	66.8	71.0	18.4	2.0
`lin318-sp`	0.65	97.7	115.2	15.5	14.5
`lin318`	0.63	95.8	115.8	15.4	2.3
`pr439-sp`	0.71	113.9	129.5	14.0	10.2
`pr439`	0.68	119.4	125.8	13.4	1.3

3.3.7 Fitness-Distance Correlation Analysis

To further investigate the difficulty of the problem instances and the effects of local improvement, we performed fitness-distance correlation analysis according to [85] and [119, 116]. For each problem instance, 10 000 candidate solutions were created randomly and locally improved as in the initialization of the MA. These solutions were evaluated and their distances to the optimum solution in the search space were calculated. As distance metric, the size of the symmetric difference of the corresponding edge sets was used. Figure 3.10 shows fitness-distance plots for the first instance of group `R2` and instance `pr439`. The plots for the other instances have similar structure. Each point in these plots represents one locally optimal solution; the global optima are located at the lower left corners (point 0/0). In addition, Table 3.7 shows fitness-distance correlation coefficients ρ for ten of the largest instances with known global optima.

In all considered instances, the fitness is clearly correlated with the distance to the optimum ($0.51 \leq \rho \leq 0.71$), which is a general indication that an evolutionary algorithm might work efficiently on these instances. Furthermore, all local optima are plotted near to each other and have a relatively large distance to the optimum. This shows that simply creating random solutions and locally improving them by our method is not effective for its own. It does not imply that the local optima are also grouped together in the search space and the global optima are located far away from them. Table 3.7 also shows average distances of locally improved random solutions to the optima ($\overline{d_{\text{opt}}}$) and average distances between locally improved random solutions ($\overline{d_{\text{loc}}}$). Since $\overline{d_{\text{opt}}}$ is significantly smaller than $\overline{d_{\text{loc}}}$ for each instance, we can argue that the global optimum lies more or less in the center of the space of all locally optimal solutions. In [118], problems with such a characteristic are said to have a *big valley* structure,

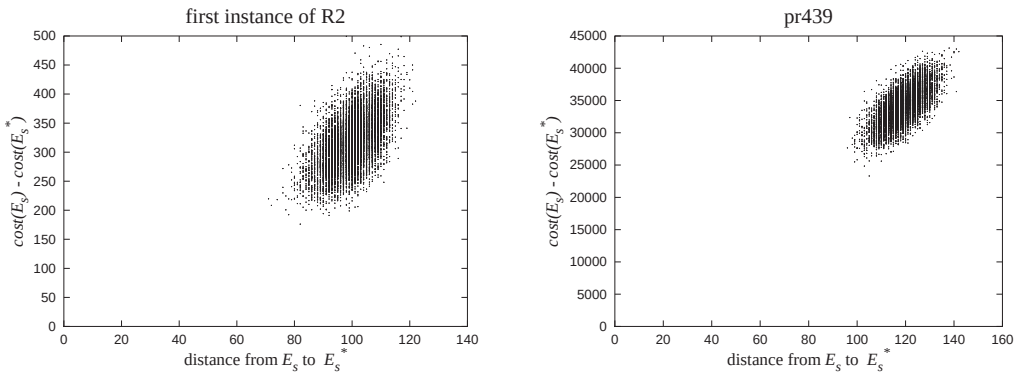


Figure 3.10: Fitness-distance plots for the first instance of group `R2` and instance `pr439`.

and recombination operators preserving properties common to both parents can be expected to work well.

3.3.8 Performance Analysis of Variation Operators

In the last two columns, Table 3.7 lists average probabilities PD_{cross} and PD_{mut} with which recombination, respectively mutation, followed by local improvement produces a candidate solution being identical to (one of) its parent solution(s). These probabilities were measured over complete runs of the MA. High values would indicate that the investigated variation operator does not work efficiently and it might be omitted without decreasing the overall effectiveness of the search significantly. In our case, these probabilities are always smaller than 18.5%. Thus, the variation operators in combination with local improvement successfully create new solutions in more than four out of five cases. In particular on dense base graphs such as the complete Euclidean problem instances, the probability of mutation leading to the same local optimum is very small: $PD_{\text{mut}} \leq 2.3\%$.

Table 3.8 further illustrates the importance of using both, recombination and mutation, and that it is not necessary to apply local improvement immediately after each variation operator. Shown are results for the following three variants of the MA: In MA-CLML, recombination and mutation are used, and local improvement is performed after each operator. In MA-CL, new candidate solutions are created only by recombination followed by local improvement. MA-ML applies always only mutation followed by local improvement. All strategy parameters were set identical as in the previous experiments with the only exception that in MA-ML, the probability of applying mutation was $p_{\text{mut}} = 1$. The performance values of these variants can therefore directly be compared to those of the original MA in Table 3.6.

MA-CL converged fastest, but the obtained solutions were in nearly all cases substantially poorer than those of the original MA. This points out the particular importance of mutation. MA-ML, on the other side, generally needed much more evaluations and also more time to converge. In particular on dense problem instances, MA-ML's solutions are far worse than those of the original MA.

Table 3.8: Performance of different MA-variants: applying local improvement after recombination and after mutation (MA-CLML), using only recombination followed by local improvement (MA-CL), and using only mutation followed by local improvement (MA-ML).

Grp./Inst.	MA-CLML					MA-CL					MA-ML				
	%-*gap*	σ	t [s]	*evals*	*sr*	%-*gap*	σ	t [s]	*evals*	*sr*	%-*gap*	σ	t [s]	*evals*	*sr*
N1	0.3	0.3	8.1	8391	46.7	0.6	0.6	7.2	6318	33.3	0.9	0.9	21.9	31825	20
N2	0.5	0.5	11.5	12023	36.7	1.3	1.3	7.1	5561	13.3	1.2	1.2	30	40819	6.7
R1	0.1	0.1	106.4	22558	43.3	0.4	0.4	22.7	4120	30	16	16.3	233.6	106740	0.0
R2	0.7	0.8	76	27587	3.3	1.3	1.4	22.8	5677	0.0	8.8	8.9	148.3	79119	0.0
pr226-dt	0.0	0.0	3.0	1856	100.0	<0.1	<0.1	2.8	1765	96.7	0.0	0.0	6.7	14002	100.0
pr226-sp	0.0	0.0	12.5	11777	100.0	22.9	4.4	13.8	5278	0.0	0.2	0.3	29.0	52534	46.7
pr226	2.4	1.4	30.7	24320	16.7	24.0	3.9	27.6	5367	0.0	8.2	2.7	43.8	56114	0.0
lin318-dt	<0.1	<0.1	18.7	8460	0.0	0.3	0.4	15.2	5536	0.0	0.1	0.1	34.6	42332	0.0
lin318-sp	0.3	0.3	41.0	35853	0.0	3.3	1.0	21.3	6771	0.0	1.6	1.1	54.0	73078	0.0
lin318	1.5	0.7	129.9	33955	0.0	3.5	1.6	111.0	6179	0.0	15.4	3.9	156.5	92059	0.0
pr439-dt	0.0	0.0	41.7	7011	100.0	1.0	1.2	37.2	4414	16.7	<0.1	0.1	70.0	45179	40.0
pr439-sp	1.0	0.7	72.3	18073	23.3	5.4	1.8	68.7	5507	0.0	2.3	1.1	130.7	84829	0.0
pr439	2.0	0.8	376.1	44083	0.0	40.9	3.1	383.0	6373	0.0	19.9	5.7	457.0	98467	0.0
pcb442-dt	<0.1	0.1	232.4	7702	73.3	0.5	0.4	224.6	3754	0.0	<0.1	0.1	251.8	33994	76.7
pcb442-sp	0.3	0.2	95.2	24473	6.7	1.7	0.8	68.1	4805	0.0	1.6	0.9	187.9	106494	0.0
pcb442	0.7	0.4	365.3	28673	0.0	1.4	0.6	334.8	5120	0.0	33.5	7.5	464.0	106346	0.0
pa561-sp	0.4	0.2	275.4	30987	3.3	3.7	0.5	195.1	5802	0.0	5.9	1.3	462.2	129287	0.0
pa561	2.2	0.5	584.7	31361	0.0	3.8	0.6	493.9	5369	0.0	34.5	5.9	764.8	106649	0.0

Performance values of MA and MA-CLML are similar for nearly all instances. Only on the single instance `pa561`, MA yielded substantially better solutions than MA-CLML. No statistically significant differences can be observed for MA and MA-CLML in their numbers of needed evaluations and running times. We conclude that the question whether local improvement should be applied once per candidate solution or once after each variation operator is of minor importance.

3.4 A Branch-and-Cut-and-Price Algorithm for the V2AUG

In the work of Grötschel et al. [71] and Stoer [152], it turns out that the most effective ILP formulations of two-connected network design problems use an exponential number of constraints. Hence, the use of the cutting plane framework is a natural approach for solving them.

Within this section we propose an exact algorithm for V2AUG based on an LP-based branch-and-bound algorithm with incorporated cutting plane and column generation methods. Our exact algorithm searches for the optimal solution S on the reduced block-cut graph obtained after the preprocessing described is Section 3.2 has been applied. In its last stage, the algorithm maps the edges from S back to the subset E_s of edges of the original graph G.

In the following discussion, we consider a minimum-cut based formulation of the problem. Section 3.4.2 describes the branch-and-cut algorithm, and highlights the initialization and separation phases of the algorithm. In Section 3.4.3, we present an enhanced branch-and-cut-and-price algorithm. The pricing of edges and the application of sparse and reserve graph techniques are described in detail. Finally, we present our computational results in Section 3.4.4.

3.4.1 Minimum-Cut Based Problem Formulation

The design of a minimum-cost vertex biconnected network from scratch, using the edges of a graph $G = (V, E, c)$ (which can be seen as an augmentation of the empty graph), can be formulated as the following integer linear program [152, 55]:

$$\min \quad \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{3.5}$$

$$\text{subject to} \tag{3.6}$$

$$x(\delta_G(W)) \geq 2, \qquad \forall\, W \subset V, W \neq \emptyset \tag{3.7}$$

$$x(\delta_{G-v}(W)) \geq 1, \qquad \forall\, W \subset V \setminus v, W \neq \emptyset, \forall v \in V \tag{3.8}$$

$$0 \leq x_{ij} \leq 1, \qquad \forall (i,j) \in E \tag{3.9}$$

$$x_{ij} \quad \text{integer}, \qquad \forall (i,j) \in E. \tag{3.10}$$

Vector x is the characteristic vector of a feasible solution given by some subset $S \subset E$.

The cuts (3.7) are called *edge-connectivity requirements* – they assure that the obtained solution is edge biconnected. The inequalities (3.8), so-called *vertex-connectivity requirements*, assure that the removal of any vertex $v, v \in V$ must not disconnect the graph, i.e. they assure vertex biconnectivity of the optimal solution.

Christofides and Whitlock [30] proposed a branch-and-cut algorithm to solve this formulation exactly. The authors initialized the LP formulation of the problem with degree constraints that assure that the degree of each vertex $v \in V$ is at least two. Violated edge-connectivity inequalities are then continuously inserted during the separation phase, after which the branch-and-cut algorithm is applied. The authors proposed to add possibly violated

vertex-connectivity requirements, each time an integer solution satisfying all edge-connectivity constraints is found.

Similarly to the formulation of Christofides and Whitlock, we represent feasible solutions by incidence vectors (x) of subsets in $|E_A|$. We establish a one-to-one correspondence between the variables x_i and augmentation edges $e_A \in E_A$.

In the sequel, V_C, $V_C \subset V_T$, will be the set of all cut-vertices in the block-cut tree $T = (V_T, E_T)$. Following the definition of biconnectivity, an edge set S that augments the block-cut tree T represents a valid vertex biconnected solution, if and only if all cut-vertices in T are covered. In other words, removal of any cut-vertex $v \in V_C$ leaves the augmented graph $G - v = (V_T \setminus \{v\}, E_T \cup S \setminus \delta(v))$ connected. This can be described using the following inequalities:

$$x(\delta_{G-v}(W)) \geq 1, \forall W \subset V_T - v, W \neq \emptyset, \forall v \in V_C \tag{3.11}$$

An integer programming formulation of V2AUG is given by

$$\min\{ \sum_{ij \in E_A} c_{ij} x_{ij} \mid x \in \mathcal{R} \cap Z_+^{|E_A|} \}, \tag{3.12}$$

where $\mathcal{R}$ represents a polyhedral region $\mathcal{R}$ defined as:

3.4.2 The Branch-and-Cut Algorithm

Within this section we propose a straight-forward branch-and-cut algorithm for the vertex biconnectivity augmentation of the block-cut tree. The most important steps of the algorithm, i.e. initialization, separation, branching and enumeration, are described in the sequel. Finally, we also provide some implementation details.

We use the terms edge and variable of the integer programming formulation interchangeably, since they are in a one-to-one correspondence in our formulation. We also consider the notions cutting planes (or cuts) and inequalities (or constraints) as equivalent.

Initialization. The linear programming relaxation of the problem (3.12) is given by:

$$\min\{ \sum_{(i,j) \in E_A} c_{ij} x_{ij} \mid x \in \mathcal{R} \} \tag{3.13}$$

Since there are exponentially many connectivity constraints in this formulation, one may exclude some or all of these inequalities in the initial stage of the optimization. This is usually done by relaxing $\mathcal{R}$ into a polyhedral region $\mathcal{R}_1 \supset \mathcal{R}$. An adequate initialization of the set of constraints is attained, for example, by choosing the so-called *degree inequalities* that correspond to the requirements (3.15) for $|W| = 1$. Hence, the corresponding polyhedral region $\mathcal{R}_1$ is:

$$\mathcal{R}_1 = \{x_{ij} \mid (i,j) \in E_A, \quad x(\delta(v)) \geq 2, \forall v \in V_T, \quad 0 \leq x_{ij} \leq 1\}.$$

A valid LP lower bound for (3.12) can be found as:

$$\min\{ \sum_{(i,j) \in E_A} c_{ij} x_{ij} \mid x \in \mathcal{R}_1 \}. \tag{3.14}$$

For an optimal solution x of (3.14), further connectivity constraints may be introduced as cutting planes. In this process the following separation problem must be solved: Find a connectivity constraint that is violated by x, or determine that no such inequality exists. If no violated connectivity constraints exist, the optimality of (3.13) is verified. Otherwise, violated constraints are added to $\mathcal{R}_1$ and the corresponding LP is resolved until the optimality of (3.13) is proved.

Separation. Given a solution to the last LP, we perform separation in two stages. The block-cut graph needs to be *edge biconnected*, thus, in the first separation stage we impose the *edge-connectivity requirements* by adding violated constraints.

Recall, an augmented block-cut graph $(V_T, E_T \cup S)$ is edge biconnected if, for each pair of vertices u and v from V_T, there are at least two edge disjoint paths connecting them. In other words, this graph is edge biconnected, if the maximum flow between u and v is at least two (where x_i values are considered as edge capacities):

$$x(\delta(W)) \geq 2, \forall W \subset V_T, W \neq \emptyset. \tag{3.15}$$

The violated edge connectivity inequalities can be found in polynomial time by computing the minimum weight cut in the support graph $G_x = (V_T, E_T \cup E_A, c')$. The edge weights are defined as:

$$c'(e) = \begin{cases} 1, & \text{if } e \in E_T \\ x(e), & \text{otherwise} \end{cases},$$

where $x(e)$ represents the fractional value of the corresponding variable in the current LP. In each iteration, we evaluate the minimum weight cut of the support graph and if its value is less than two, we insert the violated constraint. For the computation of minimum cuts, we use an efficient algorithm proposed by Padberg and Rinaldi [130] and implemented by Michael Jünger [90]. The details of this implementation and an exhaustive comparison of a variety of minimum weight cut algorithms are given in [90]. We suggest resolving the LP, each time a violated edge-connectivity cut is inserted into the system.

The second separation phase is executed only if the previous stage did not generate any inequality. Thus, when all edge-connectivity constraints are satisfied, we check if there are some uncovered cut-vertices. For this purpose, for each cut-vertex $v \in V_C$, we reduce the support graph G_x by eliminating v from it. In other words, we search for the minimum weight cut in the graph $(V_T \setminus \{v\}, E_T \cup E_A \setminus \delta(v), c')$. If the cut we found is less than one, the corresponding constraint is inserted into the system.

We consider the following two variants of inserting the violated vertex-connectivity requirements into the system:

1. Each time a violated vertex-connectivity constraint is found and inserted into the system, the LP is resolved. We refer to it later as BC-S, or

2. Only when all violated constraints regarding uncovered cut-vertices are inserted, the LP is resolved and a new fractional solution x is found. We refer to it as BC-M.

Branching and Enumeration Rules. We apply the most widespread branching method – branching on a single variable. Some fractional variable ij with value $x_{ij}, (i,j) \in E_A$ (whose value must be integer in any feasible solution) is chosen as the branching variable, and two new branch-and-cut nodes are created. We used the so-called `CloseHalfExpensive` strategy, also described in [157]. This strategy works as follows:

Suppose x is the fractional solution of the currently solved linear program. Define L and H in the following way:

$$L = \min\{0.5, \max\{x_{ij} \mid x_{ij} \leq 0.5, (i,j) \in E_A\},$$

$$H = \max\{0.5, \min\{x_{ij} \mid x_{ij} \geq 0.5, (i,j) \in E_A\}.$$

Let finally

$$C = \{(i,j) \in E_A \mid 0.75L \leq x_{ij} \leq H + 0.25(1-H)\}$$

be the set of fractional variables "close" to 0.5. We select from C the variable x_{ij} with the maximum absolute cost, i.e. with the maximum objective value coefficient (note that our edge weights are positive).

Best First Search strategy has been used as the default enumeration strategy: from the set of open subproblems the "most promising" one is selected. In our case, the node with the minimal local lower bound is said to be the most promising one.

Initializing Upper Bounds. Usage of good upper bounds plays an important role in the design of branch-and-bound based algorithms. The better the upper bound, the more nodes in the branch-and-bound tree can be fathomed. We used the memetic algorithm proposed above for the initialization of upper bounds.

Implementation Details. We used the software ABACUS developed by Thienel [157, 91] as a generic implementation of the branch-and-cut approach. For solving LP relaxations, we used the commercial package CPLEX (version 7.1).

For the initialization of upper bounds the same MA setting as described in Section 3.3 was applied. When solving small and medium-size instances we observed that there is a trade-off between the MA's running time and the running time needed to prove optimality. Thus, we used weaker termination criteria and smaller populations: the population size was 100 and the MA was terminated when no new best solution could be identified during the last $\Omega = 1\,000$ generations. The last two parameters are set according to the preliminary tests in which they proved to be robust in solving different classes of smaller instances with the branch-and-cut method. Because of MA's non-deterministic nature, we ran it with a fixed seed value.

In ABACUS, all inequalities and variables are stored in pools. A constraint (variable) usually belongs to the set of active constraints (variables) of several subproblems that still have to be processed. The advantage of pools is that in the sets of active constraints (variables) only the pointers to the corresponding inequalities (variables) need to be stored, while the constraints (variables) themselves are stored in one central place. For the degree constraints it is advantageous to stay active in any case, thus, they need to be treated differently from

the connectivity constraints. Hence, in our implementation we used three pools: one pool for variables, one pool for the vertex- and edge-connectivity constraints, and one pool for the degree constraints.

We also used *tailing-off* strategy [43]: If during the last $It = 10$ iterations in the bounding part, *lpval* did not increase by more than $k = 0.0025\%$, the cutting plane part is aborted and new subproblems are created.

Solving Zhu's Instances to Optimality

The pure cutting plane method outlined so far performed only in the root node provides lower bounds to the optimal solutions. For most of Zhu's instances these lower bounds were already optimal. However, for large instances branching was necessary. We started with the initialization and separation at the root node. When no new violated constraints can be generated, we applied branching on a binary variable x_{ij} following specific branching rules described above.

Table 3.9 shows the results of our branch-and-cut algorithm, on the set of Zhu's instances. The results are averaged over 30 different instances of the same group. Column OPT represents the averaged optimal values. The next two columns provide results of the memetic algorithm described in Section 3.3: the average percentage gap (%-*gap*) and the average running time in seconds (t [s]). The last four columns are devoted to the branch-and-cut algorithm: the average running time in seconds (t [s]), the average number of generated subproblems (SP), the average number of generated levels in the branch-and-cut tree (*Levels*), and the average number of solved linear programs (LPs). All experiments were performed on a Pentium-III/800MHz machine.

The results show that the branch-and-cut algorithm is significantly faster than the MA when small and easy (randomly generated) instances are considered. Solving of medium-size and large instances is addressed within next sections. Recall that the MA was the fastest heuristic approach as far as Zhu's instances are concerned, and that Zhu's algorithm itself didn't terminate for the `R`-group within the allowed maximum time of 20 000 seconds. In the branch-and-cut algorithm, all instances (with the exception of `N-` and `R`-group) have been solved using on average slightly more than one subproblem and generating slightly more than one level in the branch-and-bound tree. This means that, in most cases, the cutting plane method performed in the root node solved the underlying problem to optimality.

Solving TSPLIB Instances to Optimality

For each TSPLIB instance described in Section 3.2.4, we generated additional graphs in the following way: G is the graph containing all vertices of the TSP-instance and edges for each vertex to $k\%$ of its nearest neighbors, where k is the number shown in parentheses in Table 3.10. Edge costs are always the Euclidean distances rounded to nearest integer values. From G, a minimum spanning tree of the corresponding "sparse" instance (`-sp`) is fixed as G_0.

For solving instances derived from the TSPLIB, we used a Pentium IV/2.8GHz PC with 2 GB RAM.

Table 3.9: Branch-and-cut algorithm solved all Zhu's instances to optimality.

		MA		Branch-and-Cut			
Group	*OPT*	%-*gap*	t [s]	t [s]	SP	Levels	LPs
A1	511.50	0.00	0.00	0.00	1.17	1.07	1.37
A2	1764.77	0.00	0.00	0.00	1.07	1.03	1.33
A3	4055.47	0.00	0.01	0.01	1.00	1.00	1.10
A4	1948.07	0.00	0.01	0.01	1.27	1.13	1.47
A5	3753.87	0.00	0.02	0.01	1.27	1.13	1.47
B1	13426.03	0.00	0.03	0.02	1.10	1.03	1.37
B2	163.77	0.00	0.12	0.00	1.60	1.30	2.27
B3	8311.93	0.00	0.02	0.02	1.13	1.07	1.30
B4	7131.37	0.00	0.08	0.02	1.53	1.27	2.00
B5	12460.57	0.00	0.12	0.03	1.27	1.13	1.70
B6	19849.73	0.00	0.32	0.05	1.13	1.07	1.47
C1	27085.03	0.00	0.41	0.07	1.27	1.13	1.57
C2	40478.83	0.00	0.49	0.09	1.13	1.07	1.33
C3	52441.30	0.00	0.62	0.12	1.07	1.03	1.33
C4	341.50	0.00	0.38	0.01	1.07	1.03	1.53
D1	7339.93	0.00	1.68	0.37	1.27	1.13	1.50
D2	762.70	0.00	0.75	0.05	1.33	1.17	1.90
D3	12773.33	0.00	3.91	1.64	1.27	1.13	1.57
D4	9886.33	0.00	2.87	1.19	1.27	1.13	1.43
D5	13489.10	0.02	5.90	2.13	1.40	1.20	1.80
M1	3492.33	0.00	1.70	0.46	1.40	1.20	2.00
M2	3266.33	0.00	2.86	1.04	1.27	1.13	1.77
M3	3433.33	0.00	4.31	1.62	1.13	1.07	1.37
N1	389.93	0.17	9.50	2.83	19.07	3.50	18.77
N2	413.63	0.39	13.72	2.95	7.00	2.53	7.63
R1	128.93	0.08	39.78	19.31	2.20	1.50	2.47
R2	331.54	0.42	58.52	16.84	3.60	1.63	3.50

Table 3.10: Branch-and-cut algorithm applied on instances derived from the TSP-library.

Instance	OPT	*Pr. Bound*	BC-S			BC-E0			BC-M		
			t [s]	SP	LP	t [s]	SP	LP	t [s]	SP	LP
pr226-dt	25152	25152	1.0	5	11	1.1	5	12	1.2	5	9
pr226-sp	22824	23524	3.5	3	33	4.2	3	40	2.4	3	5
pr226 (20)	22824	23524	4.0	3	34	9.5	11	153	2.8	3	4
pr226 (30)	22824	23924	5.5	5	40	6.7	5	59	4.4	5	11
pr226 (40)	22824	23724	6.1	5	40	7.0	5	50	4.8	5	9
pr226 (50)	22824	23429	6.9	5	39	7.9	5	49	5.8	5	11
pr226 (60)	22824	24353	8.6	5	46	9.9	5	56	7.1	5	11
pr226 (70)	22824	23624	9.4	5	40	10.2	5	49	8.3	5	12
pr226 (80)	22824	24853	10.3	7	49	12.1	7	57	8.7	7	12
pr226 (90)	22824	24558	11.0	5	47	12.2	5	57	9.4	5	10
pr226	22824	23524	10.6	5	39	11.0	5	49	9.1	5	10
pr226-avg			7.0	4.8	38.0	8.3	5.5	57.4	**5.8**	4.8	9.5
lin318-dt	12013	12178	4.0	3	13	5.0	3	21	3.8	3	5
lin318-sp	11797	12397	15.8	73	72	30.1	107	350	15.6	73	53
lin318 (20)	11797	12105	21.4	73	72	38.8	107	350	20.9	73	53
lin318 (30)	11797	12382	28.7	73	77	46.3	107	350	27.7	73	53
lin318 (40)	11797	12139	36.5	73	72	56.3	109	340	34.6	73	53
lin318 (50)	11797	12264	40.7	81	80	62.4	113	344	41.2	81	56
lin318 (60)	11797	12466	52.4	73	75	74.3	107	351	52.6	73	53
lin318 (70)	11797	13099	60.5	73	75	64.9	75	166	60.5	73	53
lin318 (80)	11797	12162	70.1	73	71	101.2	109	339	69.2	73	53
lin318 (90)	11797	12248	78.7	85	85	103.9	113	336	78.3	85	62
lin318	11797	12332	68.7	81	80	91.4	103	360	67.7	81	56
lin318-avg			43.4	69.2	70.2	61.3	95.7	300.6	**42.9**	69.2	50.0
pr439-dt	28310	28310	13.8	15	34	69.5	23	305	13.6	15	19
pr439-sp	26800	27907	33.3	41	71	45.0	41	100	28.8	41	37
pr439 (20)	26800	27886	46.8	45	73	66.5	45	141	42.9	45	38
pr439 (30)	26800	28439	75.2	45	76	91.9	45	141	71.3	45	41
pr439 (40)	26800	27943	86.3	45	76	104.2	45	141	80.8	45	37
pr439 (50)	26800	30577	108.1	45	80	127.0	45	144	104.9	45	41
pr439 (60)	26800	28620	143.1	45	75	162.1	45	144	138.9	45	39
pr439 (70)	26800	28537	166.7	45	75	184.8	45	142	161.5	45	40
pr439 (80)	26800	29397	183.1	45	75	202.6	45	141	178.7	45	41
pr439 (90)	26800	30130	195.3	45	76	223.8	45	141	189.7	45	40
pr439	26800	30885	164.1	45	76	183.8	45	141	158.1	45	39
pr439-avg			110.5	41.9	71.5	132.8	42.6	152.8	**106.3**	41.9	37.5
pcb442-dt	10328	10344	78.0	97	104	78.8	97	104	78.2	97	99
pcb442-sp	10460	10766	91.2	253	199	94.5	253	212	96.1	253	195
pcb442 (20)	10460	10871	136.8	253	199	144.3	253	212	141.8	253	195
pcb442 (30)	10460	11240	181.4	253	199	191.2	253	212	187.1	253	195
pcb442 (40)	10460	10601	226.0	253	199	228.1	253	212	231.1	253	195
pcb442 (50)	10460	11633	283.5	253	200	297.4	253	212	287.5	253	191
pcb442 (60)	10460	11062	341.8	253	199	357.1	253	212	349.2	253	195
pcb442 (70)	10460	11242	408.5	253	199	410.2	253	212	414.8	253	195
pcb442 (80)	10460	11742	442.9	253	199	462.4	253	212	448.8	253	195
pcb442 (90)	10460	11336	474.2	253	199	493.0	253	212	481.7	253	195
pcb442	10460	12076	465.3	253	199	456.9	253	212	457.5	253	191
pcb442-avg			**284.5**	238.8	190.5	292.2	238.8	202.2	288.5	238.8	185.5

Table 3.10 represents results of three different scenarios for running the branch-and-cut algorithm. All three scenarios found optimal solutions for the considered instances–their values are given in OPT column. Upper bounds obtained by the MA are given in the *Pr. Bound* column. In BC-S strategy, we used edge-connectivity cuts in the separation phase, but we inserted only one cut per iteration. In BC-E0 scenario, only vertex-connectivity cuts are used during the separation. Finally, in BC-M, as in BC-S, we detected both, violated edge- and vertex-connectivity constraints during the separation phase, but we allowed insertion of multiple cuts per iteration. For each of these strategies, t $[s]$ represents the running time in seconds (including the time needed for preprocessing); SP shows the number of generated subproblems; and LP provides the number of solved LPs.

The results clearly show the importance of edge-connectivity constraints in the separation phase. We can see that, allowing the insertion of edge-cuts, we can reduce the total number of inserted cuts by a factor of four (see, for example group `lin318` of instances). The results also indicate that using multiple instead of single cuts, we can reduce the number of solved LPs by a factor of six (`lin318` group). On the other side, the results clearly document the trade-off between the number of solved LPs and their size. For the `pr439` group of instances, for example, for BC-S we needed to solve almost two times more LPs than for BC-M. On the other side, BC-S was only 10% slower than BC-M, which means that the corresponding LPs of BC-M were significantly larger than those of BC-S.

3.4.3 The Branch-and-Cut-and-Price Algorithm

In this section we propose an enhancement of the previous algorithm. This will be achieved by embedding a column generation method into each node of the branch-and-bound tree, thus performing the so-called branch-and-cut-and-price method. We also propose a primal heuristic based on the local improvement method embedded in the MA. In the following, we highlight computation of upper bounds, initialization and column generation procedures. Finally, we also consider some important aspects of an efficient implementation.

Initialization. We start the optimization with processing the root node of the enumeration tree. Besides the initial set of the degree constraints, we also have to select a subset of columns to set up the restricted master problem. Since we need dual variables to formulate the pricing problem, the restricted problem must be feasible. As the sparse graph we choose k_1-nearest neighbor graph ($k_1 \in \mathbb{N}$) of the reduced block-cut graph. The value for k_1 is properly chosen so that the block-cut tree together with the corresponding k_1-nearest neighbors of each vertex represents a feasible problem instance (see next section).

Computation of Upper Bounds. The primal upper bound of the branch-and-cut algorithm described above can be improved only if the LP-solution is integer feasible. However, it can be observed that this happens rather rarely. Therefore a sophisticated heuristic (the so-called *primal heuristic*) must be applied at each node of the branch-and-bound tree, before a branching step is applied, in order to generate new, better feasible solutions.

The fractional LP-solutions occurring in the lower bound computations may give good hints for the structure of optimum or near optimum feasible solutions [43]. Our heuristic is designed in the following way: Starting from the fractional solution x of the last LP, we generate a support block-cut graph using the edges of the block-cut tree T and the edges (i.e. variables) from the set E_{prHeur}, where

$$E_{prHeur} = \{e_{ij} \in E_A \mid x_{ij} \geq p_{prHeur}\}.$$

$p_{prHeur} \in [0, 1]$ is a fractional parameter which controls the influence of the LP solution on the generation of feasible solutions.

If the support graph is not biconnected, we first make it feasible by adding an additional subset of augmentation edges. Iteratively, non-redundant edges are randomly selected from $E_A \setminus E_{prHeur}$ and included in the support graph. This process is repeated until all cut-vertices are completely covered. The selection of edges for inclusion is biased toward cheaper edges according to a scheme used within the initialization operator of the MA (see Section 3.3.3).

Finally, from such obtained feasible solution, we eliminate redundant edges by using the local improvement procedure which has been embedded within the MA, with one exception: to better exploit the LP-solution, we forbid elimination of those edges with $x_{ij} = 1$.

Column Generation. Column generation is necessary before a branch-and-cut node can be fathomed. Its purpose is to check whether the LP-solution computed on the sparse graph is valid for the complete block-cut graph G_A. In other words, using this procedure we have to check whether all inactive variables "price out" correctly. If this is not the case, inactive variables with negative reduced costs are added to the reduced subproblem and the linear program is resolved.

Pricing of inactive edges is done using the reserve graph strategy, i.e. only if all the edges of the reserve graph price out correctly, reduced costs of the rest of inactive variables have to be checked (the so-called complete pricing is necessary). The reserve graph is chosen as the difference between the k_2-nearest neighbor graph and the sparse graph, where $k_2 > k_1, k_2 \in \mathbb{N}$.

Using an idea of Padberg and Rinaldi [131, 157], we also fix some inactive variables by their reduced costs. If our current branch-and-cut node is the root of the remaining branch-and-cut tree, we search for inactive variables x_i such that:

$$c(LP) + r_i > UB_g.$$

Here $c(LP)$ denotes the last computed lower bound, UB_g represents the global upper bound, and r_i represents the reduced costs of the variable x_i. Such variables x_i can be discarded forever. The other variables are inserted in a list that maintains possible candidates that can be priced in later iterations. In the early steps of computation, the number of inactive edges that cannot be discarded may be longer. Due to a possible memory overconsumption, only a partial list of these edges is stored (building the so-called *candidate graph*), together with a pointer where the systematic enumeration has to be resumed (see also [89]).

Further Implementation Details. The mechanisms described above are implemented using the so called EDGERESERVOIR data-structure. The implementation is adopted from Stefan Thienel [157, pp. 142] where it has been used for solving the traveling salesman problem. Furthermore, as also suggested in [157, 91], we use the following implementation technique to enhance the calculation of the reduced costs of an inactive variable: Since all constraints we are dealing with are given in the form $x(\delta(W)) \geq const$, a necessary condition that the coefficient of an edge e is nonzero for this constraint is that exactly one end-node of e is contained in W. Hence, for both ends u and v of edge e, we store a list of constraints where they are involved. We initialize the reduced costs with the objective function coefficient, and compare the constraint lists of u and v. Whenever the two lists disagree in a constraint from the current reduced costs value, we subtract the value of the dual variable multiplied by the corresponding coefficient. Alternatively, we could calculate the reduced costs in a straightforward way by multiplying the vector of dual variables with the corresponding column in the current constraint matrix. However, using these refinements, we can significantly speed up the total time spent on pricing.

When dealing with a special constraint class, we are faced with the problem of the trade-off between performance and memory usage [157]. Memory efficient storage of constraints may become a bottleneck when the coefficients of a variable for several constraints have to be computed. This usually happens when a variable needs to be added to the LP-solver (during the pricing phase). Using more memory in these cases we are able to perform these operations more efficiently.

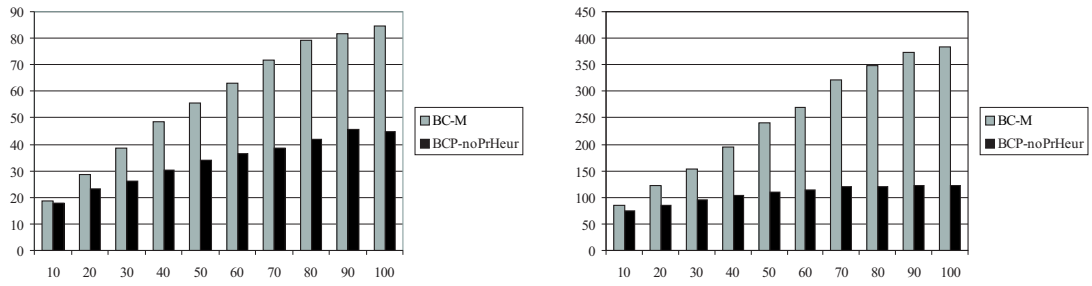
Therefore, we switch between the compressed format and the expanded format of a constraint. Before the variables with negative reduced costs are searched, we generate the expanded format of all constraints, in order to avoid too many time consuming coefficient computations. The constraints are afterwards compressed again.

For an inequality of a type $x(\delta(W)) \geq const$, we always store either the set W itself, or its complement $V \setminus W$, whichever is smaller. Without loss of generality, we assume $|W| < |V \setminus W|$. The nodes of set W are stored in the compressed format, for example using a simple set data structure where only $O(|W|)$ space is needed for each constraint. The computation of the coefficient of an edge (i, j) requires $O(|W|)$ time, since in the worst case all nodes of the set W need to be passed through. In the expanded format, we use an additional boolean array indexed by the set of nodes V_T, whose element with index v is set to true if and only if $v \in W$. Now, we can determine the coefficient of an edge in constant time, while the memory requirement of each constraint has been increased from $O(|W|)$ to $O|V_T|$).

3.4.4 Computational Experiments

In this section, we analyze the performance of the proposed branch-and-cut-and-price approach. In particular, we investigate impacts of incorporating pricing and primal heuristics and compare the performance of two different MA settings that are used for the initialization of upper bounds.

For initialization of the sparse and reserve graph, we used $k_1 = 5$ and $k_2 = 10$, as proposed



(a) (b)

Figure 3.11: Comparing running times of the branch-and-cut algorithm (BC-M) and the branch-and-cut-and-price algorithm (BCP-noPrHeur). To investigate the role of pricing, no primal heuristic is used, and the preprocessing times are taken out of consideration. Instances of (a) `pr439` and (b) `pcb442` group.

in [89] for solving the traveling salesman problem. To assure the feasibility of the sparse graph, for the instances of the `pr226` group we set $k_1 = 8, k_2 = 12$ and for the `pr439` group $k_1 = 6, k_2 = 10$. We allowed insertion of multiple node-connectivity cuts during the separation phase. For the settings, where the primal heuristic is used, we set $p_{prHeur} = 0.5$. As before, for all computational experiments presented in this section we used a Pentium IV/2.8GHz with 2 GB RAM. We have used the same tailing-off strategy as in the branch-and-cut algorithm.

The results depicted on Figure 3.11 show the running times of the branch-and-cut algorithm, with and without pricing, BC-M and BCP-noPrHeur, respectively. To be able to investigate the role of the pricing in improving the BC-M performance, the primal heuristic was switched off in both of the settings. Furthermore, we subtracted the preprocessing times, thus comparing only the computational effort of "pure" exact algorithms. The obtained results show that the incorporation of pricing into the branch-and-cut framework for the V2AUG problem significantly speeds up the computation. For both, pcb442 and `pr439` groups, the overall running time can be on average reduced for about 50%. Furthermore, one observes that the density of an instance does not substantially influence the running time, if the pricing is used. On the other hand, without pricing, the overall running time may be up to three times longer (see the complete graph of group `pcb442`, for example).

Table 3.11 shows the results of the BC-M, the branch-and-cut algorithm described in the previous section, and of the proposed branch-and-cut-and-price algorithm with the following MA settings:

- In BCP-0.5, the population size was 100 and each MA run was terminated when no new best solution could be identified during the last $\Omega = 1\,000$ iterations.
- In BCP-Full-MA, the convergence criteria for the MA was stronger (the same as described

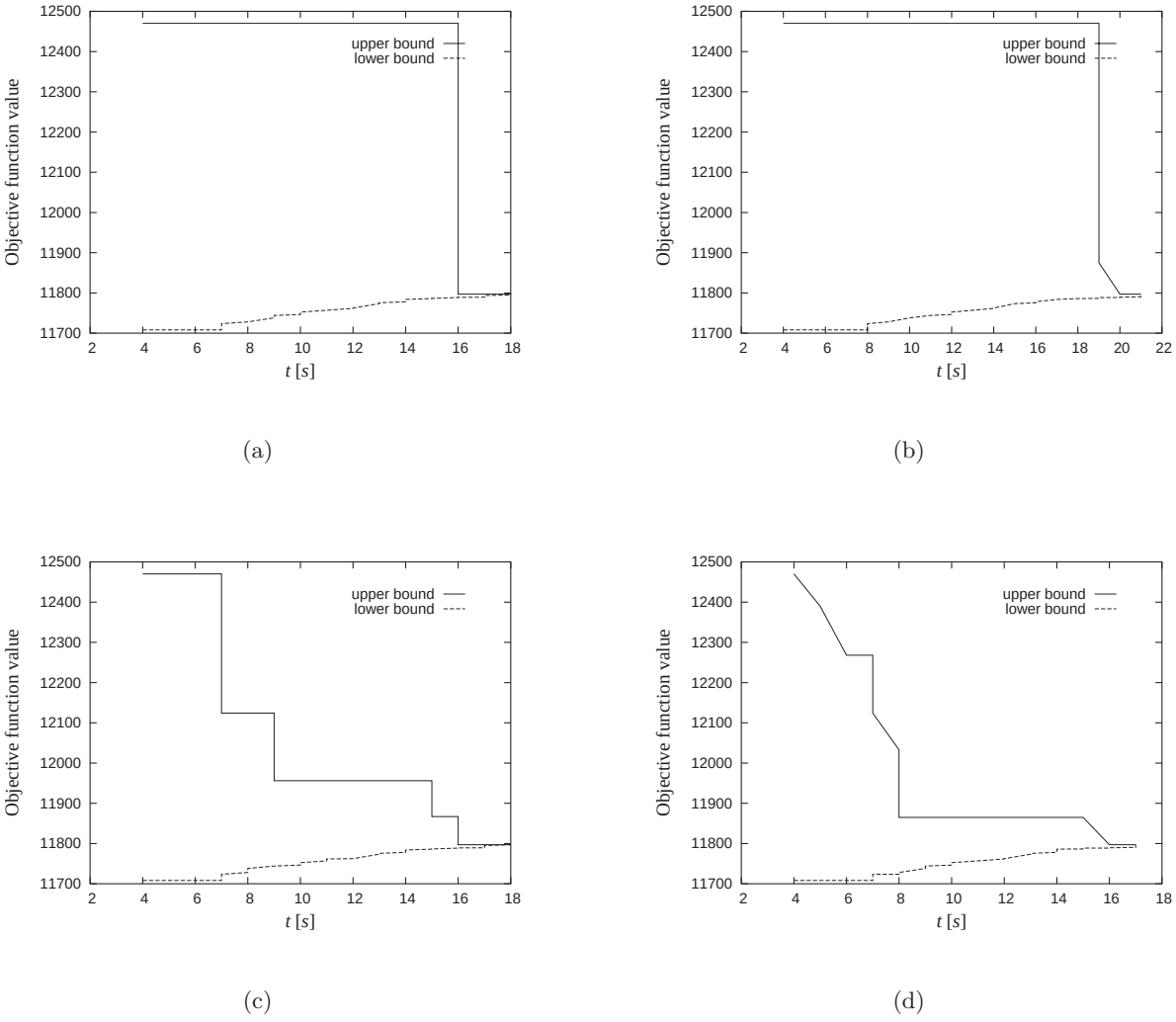


Figure 3.12: Gap versus time plot for `lin318 (70)` instance. Due to the application of the primal heuristic, good feasible solutions can be found earlier. (a) BCP performance without primal heuristic. BCP performance with (b) $p_{prHeur} = 0.0$; (c) $p_{prHeur} = 0.2$; (d) $p_{prHeur} = 0.5$;

in Section 3.3.6), thus $\Omega = 10\,000$ while the population size was set to 800.

For each of these strategies, in Table 3.11, the running time in seconds (t [s]), the total number of generated subproblems (SP) and the total number of solved LPs (LP) are given. Additionally, for BCP-0.5 and BCP-Full-MA, in t_{best} [s] column, we provide the time when the optimal solution has been found. Table 3.11 shows that the pricing together with primal heuristic substantially improves the performance of the branch-and-cut algorithm. Not only that the overall running time can be on average reduced for about 50% (for the group `pcb442`, for example) but also the size of the branch-and-bound tree can be reduced for about 25%.

Figure 3.12 depicts one example which shows the advantage of using a primal heuristic. The performance of the BCP algorithm is better if only promising variables (i.e. those whose LP-values $x_{ij} \geq 0.5$) are used within the upper bounding procedure. If all possible augmentation edges (i.e. $x_{ij} \geq 0$) are used within the upper bounding procedure, there is no significant difference between the performance of the BCP algorithm with or without primal heuristic.

One also observes that there is a certain trade-off between the time needed to run the MA (in order to initialize upper bounds) and the overall BCP running time. Thus, the results are much better if the "strong" convergence and diversity of the MA are *not* requested, i.e. if Ω is set to 1 000, and the population size to 100. The results also indicate that this trade-off diminishes when the problem size becomes larger.

Besides graphs derived from `pa561` TSPLIB instance, we considered two additional groups of larger instances: `d1291` and `d2103`, belonging to TSPLIB as well. After applying preprocessing described in Section 3.2, these graphs contain between 5237 and 22707 augmentation edges (`d1291` group), and between 5680 and 33360 augmentation edges (`d2103` group). For all these instances, except for `d2103 (2)`, our branch-and-cut-and-price algorithm terminated abnormally because of memory overconsumption. Thus, for these instances we did not succeed in finding provably optimal solutions. Therefore, we measured the following optimality gap:

$$gap_g = \frac{UB_g - LB_g}{LB_g} \times 100\%,$$

where UB_g represents the costs of a known augmentation (obtained either within the MA, or during the branching), and LB_g is a global lower bound. The *optimality* gap_g expresses that the solution with costs UB_g is at most gap_g% more expensive than the optimal solution.

In Table 3.12, we consider the following three different settings: BCP-0.5 and BCP-Full-MA are the settings described above, while BCP-noPrHeur represents the branch-and-cut-and-price algorithm without primal heuristic. In UB and UB_{full} columns, we show the upper bounds obtained after running the memetic algorithm with $\Omega = 1\,000$, population size 100; and $\Omega = 10\,000$, population size 800, respectively. For each of these settings, we provide the total running time in seconds (t [s], including preprocessing and MA running time) and the optimality gap (gap_g).

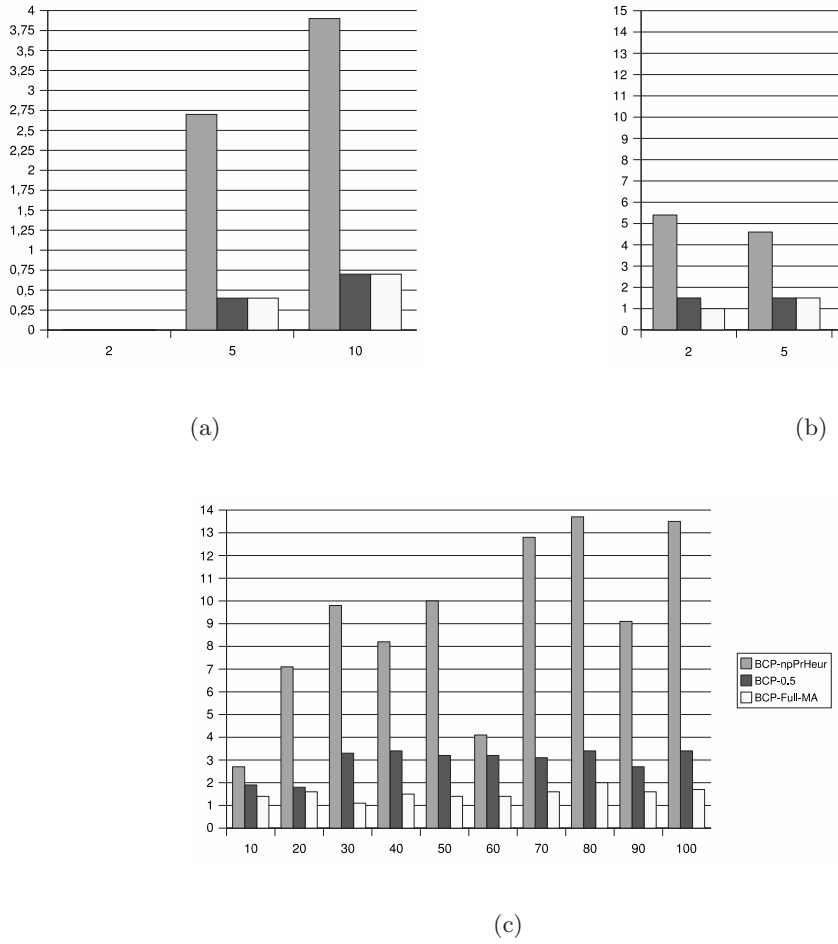
Table 3.12 documents that for large instances it is recommendable to run the MA to obtain as good solutions as possible ($\Omega = 10\,000$) and to keep higher diversity (population size of 800). While for small and medium-size instances the computation of high-quality upper bounds can slow-down the optimization, for larger instances it helps significantly in reducing the gap

Table 3.11: Comparing the branch-and-cut algorithm (BC-M) with the branch-and-cut-and-price algorithm with primal heuristic whose $p_{prHeur} = 0.5$ (BCP-0.5) and the branch-and-cut-and-price algorithm where the upper bounds are initialized with the MA whose population size is 800 and $\Omega = 10\,000$ (BCP-Full-MA). The best running times are highlighted.

Instance	BC-M			BCP-0.5				BCP-Full-MA			
	t [s]	SP	LP	t [s]	SP	LP	t_{best} [s]	t [s]	SP	LP	t_{best} [s]
pr226-dt	**1.2**	5	9	1.3	5	9	0.0	3.5	5	9	2.0
pr226-sp	**2.4**	3	5	2.6	5	10	1.0	9.3	5	10	8.0
pr226 (20)	**2.8**	3	4	**2.8**	3	9	2.0	9.4	3	9	8.0
pr226 (30)	**4.4**	5	11	4.6	7	18	3.0	10.8	3	14	9.0
pr226 (40)	4.8	5	9	**4.4**	5	13	3.0	12.8	5	12	10.0
pr226 (50)	5.8	5	11	**4.9**	5	14	3.0	13.7	7	14	10.0
pr226 (60)	7.1	5	11	**6.5**	5	13	4.0	14.1	7	16	11.0
pr226 (70)	8.3	5	12	**7.8**	5	13	4.0	14.9	5	19	11.0
pr226 (80)	8.7	7	12	**7.6**	5	10	4.0	14.7	5	13	11.0
pr226 (90)	9.4	5	10	**8.1**	3	13	5.0	16.1	5	11	12.0
pr226	9.1	5	10	**7.8**	3	11	5.0	16.6	5	12	13.0
pr226-avg	5.8	4.8	9.5	**5.3**	4.6	12.1	3.1	12.4	5.0	12.6	9.5
lin318-dt	**3.8**	3	5	4.0	3	6	1.0	9.7	3	6	7.0
lin318-sp	15.6	73	53	**14.4**	59	66	10.0	23.0	59	65	19.0
lin318 (20)	20.9	73	53	**15.6**	59	65	11.0	22.8	59	65	18.0
lin318 (30)	27.7	73	53	**19.5**	59	66	11.0	29.2	59	65	21.0
lin318 (40)	34.6	73	53	**24.1**	59	65	12.0	39.7	59	65	28.0
lin318 (50)	41.2	81	56	**25.1**	59	65	13.0	35.4	59	65	23.0
lin318 (60)	52.6	73	53	**36.2**	59	66	14.0	48.2	59	65	26.0
lin318 (70)	60.5	73	53	**40.9**	59	66	15.0	54.1	59	65	28.0
lin318 (80)	69.2	73	53	**47.8**	59	68	16.0	54.3	59	65	23.0
lin318 (90)	78.3	85	62	**50.8**	59	68	16.0	60.8	59	65	26.0
lin318	67.7	81	56	**41.3**	59	68	16.0	51.6	59	65	26.0
lin318-avg	42.9	69.2	50.0	**29.1**	53.9	60.8	12.3	39.0	53.9	59.6	22.3
pr439-dt	**13.6**	15	19	13.9	15	19	1.0	22.2	15	19	9.0
pr439-sp	28.8	41	37	**24.8**	27	31	13.0	35.3	27	31	24.0
pr439 (20)	42.9	45	38	**33.3**	31	32	17.0	52.9	31	32	37.0
pr439 (30)	71.3	45	41	**54.6**	31	33	19.0	82.9	31	34	48.0
pr439 (40)	80.8	45	37	**57.7**	31	32	23.0	86.6	31	34	51.0
pr439 (50)	104.9	45	41	**79.3**	31	33	27.0	107.4	31	35	55.0
pr439 (60)	138.9	45	39	**107.4**	31	35	28.0	134.0	31	35	55.0
pr439 (70)	161.5	45	40	**122.4**	31	34	30.0	150.6	31	32	58.0
pr439 (80)	178.7	45	41	**136.1**	31	36	33.0	165.6	31	32	63.0
pr439 (90)	189.7	45	40	**148.6**	31	33	37.0	177.3	31	33	66.0
pr439	158.1	45	39	**113.4**	31	35	37.0	142.9	31	35	66.0
pr439-avg	106.3	41.9	37.5	**81.0**	29.2	32.1	24.1	105.2	29.2	32.0	48.4
pcb442-dt	**78.2**	97	99	78.8	97	101	18.0	89.1	89	100	28.0
pcb442-sp	96.1	253	195	**74.1**	195	201	58.0	94.0	195	202	78.0
pcb442 (20)	141.8	253	195	104.8	253	198	80.0	**100.4**	195	201	77.0
pcb442 (30)	187.1	253	195	**118.0**	195	201	76.0	130.7	195	201	90.0
pcb442 (40)	231.1	253	195	**128.2**	195	201	82.0	141.4	195	200	96.0
pcb442 (50)	287.5	253	191	**142.9**	195	200	86.0	158.4	195	199	102.0
pcb442 (60)	349.2	253	195	**178.4**	195	201	89.0	197.7	195	200	109.0
pcb442 (70)	414.8	253	195	**211.8**	253	198	107.0	224.7	195	201	121.0
pcb442 (80)	448.8	253	195	**206.1**	195	201	94.0	221.9	195	201	110.0
pcb442 (90)	481.7	253	195	**215.7**	199	196	96.0	240.2	195	201	122.0
pcb442	457.5	253	191	**182.4**	199	196	96.0	199.8	199	196	114.0
pcb442-avg	288.5	238.8	185.5	**149.2**	197.4	190.4	80.2	163.5	185.7	191.1	95.2

Table 3.12: Comparing three BCP settings for large instances derived from TSPLIB: BCP algorithm without primal heuristic (BCP-noPrHeur), BCP with primal heuristic whose $p_{prHeur} = 0.5$ (BCP-0.5), and BCP with $p_{prHeur} = 0.5$ with upper bounds obtained from running the MA with $\Omega = 10\,000$ and population size 800 (BCP-Full-MA). For BCP-noPrHeur and BCP-0.5, the MA's population size was 100, and $\Omega = 1\,000$. The best optimality gap values are highlighted.

Instance	UB	BCP-noPrHeur		BCP-0.5		UB_{full}	BCP-Full-MA	
		t [s]	gap_g	t [s]	gap_g		t [s]	gap_g
pa561-sp	794	580.8	2.7	596.1	1.9	784	593.2	**1.4**
pa561 (20)	828	682.3	7.1	681.5	1.8	786	694.0	**1.6**
pa561 (30)	849	750.5	9.8	754.8	3.3	782	763.8	**1.1**
pa561 (40)	837	934.6	8.2	933.6	3.4	785	774.1	**1.5**
pa561 (50)	851	1094.4	10.0	1067.1	3.2	785	852.2	**1.4**
pa561 (60)	805	1018.4	4.1	1208.7	3.2	784	929.0	**1.4**
pa561 (70)	872	1771.4	12.8	1377.7	3.1	785	938.8	**1.6**
pa561 (80)	879	2226.3	13.7	1401.5	3.4	788	1031.2	**2.0**
pa561 (90)	844	2243.6	9.1	1380.0	2.7	785	1042.0	**1.6**
pa561	878	1754.3	13.5	1690.4	3.4	786	842.6	**1.7**
d1291 (2)	12298	1443.4	5.4	1450.7	1.5	11788	1493.2	**1.0**
d1291 (5)	12210	1571.5	4.6	1583.4	**1.5**	11855	1624.6	**1.5**
d1291 (10)	12634	1772.0	8.3	1768.2	**1.7**	11924	1810.7	**1.7**
d1291 (30)	13357	3499.8	14.7	3305.1	**1.9**	11997	3385.3	**1.9**
d2103 (2)	7633	3766.6	**0.0**	3723.5	**0.0**	7490	3749.7	**0.0**
d2103 (5)	7610	4637.4	2.7	4653.9	**0.4**	7521	4692.1	**0.4**
d2103 (10)	7691	5478.1	3.9	5468.6	**0.7**	7503	5529.3	**0.7**



(a) (b)

(c)

Figure 3.13: Comparison of optimality gaps for three different BCP settings: BCP-noPrHeur, BCP-0.5 and BCP-Full-MA. Instances of (a) `d2103`, (b) `d1291` and (c) `pa561` group.

between the global lower bound and the best-known feasible solution. Figure 3.13 depicts the optimality gap for BCP-noPrHeur, BCP-0.5 and BCP-Full-MA on the set of largest instances we tested. One observes that the BCP-Full-MA approach produces optimality gap less than 2%, the gap of the BCP-0.5 approach lies beyond 4%, while the BCP algorithm without primal heuristic produces the worst solutions, with optimality gaps with up to 14%.

3.5 Pricing with MA Solutions

In the previous section we used a pricing technique based on sparse and reserve graphs, and initialized with nearest-neighbor graphs, as originally proposed in [89]. In the sequel, we are going to investigate the performance of the branch-and-cut-and-price algorithm if, instead of using nearest-neighbor graphs, the MA solutions are used.

We propose the following strategy:

(1) Initialize the restricted master problem with the last MA's population obtained after no new best solution could be identified during the last Ω iterations. Thus, we initialize the master problem with all the edges that belong to at least one solution from this population.

(2) Initialize the reserve graph with all edges contained in the solutions of the first MA's population, which do not belong to the sparse graph already. Note that such initialization does not include purely random edges, but the edges obtained using the sophisticated problem-dependant initialization and local improvement procedures described in Section 3.3.

The advantages of this approach are:

- The sparse graph is always feasible;
- The number of edges of the sparse graph is small, which is particularly important for those instances where the LP-solver represents the bottleneck of the implementation.
- As we have seen in the last section, the best MA's solution provide excellent upper bounds for the branch-and-cut-and-price algorithm, especially for larger instances. Thus, there are no additional computational costs needed for the initialization of sparse and reserve graphs.
- Edges of the sparse graph belong to the optimal solution with a high probability.
- It can be observed from problems like TSP that the reduced costs of edges not belonging to the reserve graph are seldom positive if the reserve graph is appropriately chosen. Since the last MA's population contains several locally optimal and promising solutions, the probability that some of the edges from the last population belong to the optimal solution is high.

Table 3.13: Comparing the BCP algorithm relying on nearest-neighbor pricing (BCP-0.5) and on pricing with MA solutions (BCP-SR-MA). The best running times are highlighted.

Instance	BCP-0.5							BCP-SR-MA						
	$\|Sp.\|$	$\|Res.\|$	$\notin$ SR	AV	LP	t_{best} $[s]$	t $[s]$	$\|Sp.\|$	$\|Res.\|$	$\notin$ SR	AV	LP	t_{best} $[s]$	t $[s]$
pr226dt	255	1	0	0	9	0.0	**1.3**	36	131	0	18	15	1.0	1.6
pr226m15	1120	428	368	500	10	1.0	**2.6**	57	782	54	282	17	2.0	2.8
pr226m20	1153	451	383	509	9	2.0	**2.8**	49	779	55	283	18	2.0	3.1
pr226m30	1162	477	417	541	18	3.0	**4.6**	61	778	81	340	28	3.0	4.8
pr226m40	1162	494	528	640	13	3.0	4.4	51	751	66	254	20	3.0	**4.3**
pr226m50	1163	497	662	781	14	3.0	**4.9**	55	761	44	273	23	4.0	5.7
pr226m60	1157	502	964	1084	13	4.0	6.5	48	766	78	297	18	4.0	**6.1**
pr226m70	1163	497	1280	1394	13	4.0	7.8	46	795	100	313	23	4.0	**7.3**
pr226m80	1163	497	1277	1392	10	4.0	**7.6**	52	767	86	367	22	5.0	7.9
pr226m90	1163	497	1345	1465	13	5.0	**8.1**	51	780	58	244	21	5.0	8.4
pr226m0	1163	497	1277	1393	11	5.0	7.8	50	777	96	311	19	4.0	**7.6**
pr226-avg	1074.9	439.8	772.8	881.7	12.1	3.1	**5.3**	50.5	715.2	65.3	271.1	20.4	3.4	5.4
lin318dt	426	8	0	1	6	1.0	**4.0**	71	264	0	53	8	1.0	4.2
lin318m10	973	614	5	29	66	11.0	14.4	78	1091	70	267	84	10.0	**13.3**
lin318m20	1007	791	10	30	65	12.0	**15.6**	65	1204	43	240	74	12.0	16.1
lin318m30	1018	866	10	30	66	13.0	**19.5**	82	1192	57	246	74	13.0	19.9
lin318m40	1022	900	10	30	65	13.0	24.1	74	1210	64	277	85	12.0	**22.5**
lin318m50	1023	925	10	30	65	14.0	25.1	80	1214	64	256	80	14.0	**24.7**
lin318m60	1022	923	10	30	66	16.0	36.2	69	1201	49	258	74	14.0	**34.3**
lin318m70	1022	925	10	30	66	16.0	40.9	77	1220	71	255	72	14.0	**38.2**
lin318m80	1022	930	13	33	68	18.0	47.8	70	1231	68	269	94	17.0	**47.5**
lin318m90	1022	927	13	33	68	18.0	50.8	70	1203	68	251	84	17.0	**49.7**
lin318m0	1021	927	13	33	68	18.0	**41.3**	78	1210	54	302	90	18.0	41.5
lin318-avg	961.6	794.2	9.5	28.1	60.8	13.6	29.1	74.0	1112.7	55.3	243.1	74.5	12.9	**28.4**
pr439dt	490	0	0	0	19	5.0	**13.9**	72	262	6	104	36	9.0	17.6
pr439m10	1536	688	22	38	31	14.0	**24.8**	82	1456	26	250	63	18.0	28.3
pr439m20	1635	901	23	42	32	19.0	**33.3**	76	1380	108	381	56	20.0	34.3
pr439m30	1663	989	23	44	33	21.0	**54.6**	75	1379	93	376	75	29.0	61.9
pr439m40	1680	1026	23	42	32	25.0	**57.7**	73	1389	85	312	54	26.0	58.5
pr439m50	1680	1032	21	42	33	29.0	**79.3**	78	1355	97	355	63	31.0	81.0
pr439m60	1680	1033	23	43	35	31.0	**107.4**	77	1380	64	271	64	34.0	110.0
pr439m70	1679	1032	30	50	34	32.0	**122.4**	67	1398	100	320	60	35.0	124.7
pr439m80	1678	1034	23	42	36	36.0	**136.1**	70	1377	219	458	61	40.0	140.0
pr439m90	1678	1032	21	42	33	40.0	**148.6**	71	1401	438	690	70	43.0	151.1
pr439m0	1678	1029	17	36	35	40.0	**113.4**	80	1400	106	398	73	44.0	117.7
pr439-avg	1552.5	890.5	20.5	38.3	32.1	26.5	**81.0**	74.6	1288.8	122.0	355.9	61.4	29.9	84.1
pcb442dt	383	2	0	0	101	18.0	**78.8**	66	200	0	69	128	22.0	83.0
pcb442m10	1324	742	1	22	201	61.0	74.1	76	1554	62	205	232	57.0	**69.8**
pcb442m20	1437	1163	0	12	198	86.0	104.8	96	1616	0	151	231	60.0	**78.8**
pcb442m30	1468	1247	1	22	201	83.0	118.0	86	1653	54	216	205	56.0	**91.5**
pcb442m40	1476	1269	1	22	201	91.0	128.2	99	1639	49	215	295	89.0	**126.1**
pcb442m50	1477	1278	1	22	200	96.0	**142.9**	85	1622	0	316	398	133.0	179.8
pcb442m60	1483	1275	1	22	201	99.0	178.4	106	1615	0	226	315	96.0	**175.7**
pcb442m70	1482	1279	0	12	198	118.0	**211.8**	79	1645	0	315	503	154.0	247.5
pcb442m80	1483	1276	1	22	201	104.0	**206.1**	93	1625	0	267	540	152.0	253.6
pcb442m90	1485	1277	0	14	196	107.0	215.7	85	1644	0	243	335	93.0	**201.5**
pcb442m0	1484	1274	0	14	196	108.0	**182.4**	78	1672	0	241	328	112.0	186.5
pcb442-avg	1362.0	1098.4	0.5	16.7	190.4	88.3	**149.2**	86.3	1498.6	15.0	224.0	319.1	93.1	154.0

Table 3.14: Results on larger instances: Comparing the BCP-Full-MA algorithm (nearest-neighbor graph pricing), with the BCP-SR-Full-MA (pricing with MA solutions). The best obtained optimality gaps are highlighted.

Instance	BCP-Full-MA						BCP-SR-Full-MA					
	$\|Sp.\|$	$\|Res.\|$	$\notin$ SR	AV	t [s]	gap_g	$\|Sp.\|$	$\|Res.\|$	$\notin$ SR	AV	t [s]	gap_g
pa561-sp	1725	1348	41	82	593.2	**1.4**	167	3316	353	538	542.4	**1.4**
pa561 (20)	1759	1570	41	82	694.0	1.6	160	3234	23	351	600.8	**1.4**
pa561 (30)	1768	1593	37	78	763.8	**1.1**	160	3259	255	438	617.7	**1.1**
pa561 (40)	1762	1603	37	71	774.1	**1.5**	154	3275	20	487	685.1	**1.5**
pa561 (50)	1763	1601	37	79	852.2	**1.4**	153	3269	21	494	703.8	1.5
pa561 (60)	1764	1597	37	70	929.0	**1.4**	152	3273	21	395	767.6	**1.4**
pa561 (70)	1761	1598	33	64	938.8	1.6	143	3275	2	367	850.3	**1.4**
pa561 (80)	1758	1595	38	70	1031.2	2.0	123	3271	19	427	892.2	**1.6**
pa561 (90)	1753	1595	35	63	1042.0	**1.6**	135	3293	26	573	902.8	**1.6**
pa561	1747	1589	31	62	842.6	**1.7**	131	3274	1	267	783.8	**1.7**
d1291 (2)	3446	1551	6	141	1493.2	**1.0**	182	4209	99	516	1396.7	1.1
d1291 (5)	3940	3387	4	123	1624.6	1.5	167	7480	94	475	1475.1	**1.3**
d1291 (10)	4022	3758	2	125	1810.7	1.7	172	7505	115	489	1655.1	**1.5**
d1291 (30)	4070	3906	7	113	3385.3	**1.9**	169	7460	26	544	3267.7	2.0
d2103 (2)	4430	914	2	20	3749.7	**0.0**	129	2834	8	101	3701.3	**0.0**
d2103 (5)	6492	4129	16	58	4692.1	**0.4**	131	4264	25	157	4545.7	0.7
d2103 (10)	7373	6144	17	57	5529.3	**0.7**	152	3945	25	150	5372.8	0.8

Table 3.13 compares the performance of the BCP-0.5 algorithm with settings as given in section above, and the new BCP algorithm based on pricing with MA solutions (denoted by BCP-SR-MA). The size of the sparse and reserve graph are given in $|Sp.|$ and $|Res.|$, respectively. The number of variables that have been priced in, but that did belong neither to sparse nor reserve graph, is shown in the $\notin$ SR column. The total number of added variables (added from both, reserve and complete pricing) is given in the AV column. The total number of solved LPs, the time needed to detect the optimal solution, and the total time needed to prove the optimality are given as LP, t_{best} $[s]$ and t $[s]$, respectively.

The results show that the sparse graph in BCP-SR-MA case (i.e. the number of edges from the last MA population) can be on average more than 20 times smaller than the corresponding k_1-nearest-neighbor graph. On the other hand, the BCP-SR-MA reserve graph can be on average more than 50% larger. The advantage of a small sparse graph is that a single LP-iteration can be solved very fast. The disadvantage can be seen by comparing values given in the $\notin$ SR and AV columns: (a) The number of edges that need to be priced into the restricted master problem is significantly larger in the BCP-SR-MA case, (with the only exception of `pr226` group); (b) The total number of solved LPs can be up to 100% larger in the BCP-SR-MA case. One observes that the average number of edges that are priced in ($\notin$ SR and AV columns) for `pr226` group significantly deviates from the average values of the other groups, when BCP-0.5 is considered. This can be explained by the specific geometric structure of this instance in which the vertices are clustered within small groups distant from each other.

Finally, we investigate pricing with MA solutions on the set of largest instances we have, the `pa561`, `d1291` and `d2103` groups. Table 3.14 compares the BCP algorithm with the straightforward nearest-neighbor graph based pricing, BCP-Full-MA, and the BCP algorithm based on pricing with MA solutions, the so-called BCP-SR-Full-MA. Here, "Full" refers to the MA, where $\Omega = 10\,000$ and population size is set to 800. When comparing the optimality gap_g obtained by running these two methods, we observe similar results to those presented in Table 3.13: both pricing strategies produce results of almost the same quality. None of these techniques is significantly better than the other one in terms of running time. As well as BCP-0.5, the BCP based on pricing with MA solutions did not succeed to prove optimality for any of the instances from `pa561, d1291, d2103`, for which the optimum was not already known.

3.6 Summary

Within this chapter we proposed two basic algorithms for solving the vertex biconnectivity augmentation problem: The memetic algorithm that finds locally optimal solutions, and the branch-and-cut-and-price algorithm that finds provably optimal solutions. Furthermore, an effective deterministic preprocessing which substantially reduces the search space in most cases is given.

The main features of the proposed memetic algorithm are: the local improvement procedure which guarantees local optimality with respect to the number of augmentation edges of any candidate solution, and the strong heritability and locality of the proposed recombination, respectively mutation. Furthermore, the biasing of initialization and recombination to make

the inclusion of the low-cost edges more likely, respectively the biasing of mutation to remove more expensive edges with higher probability, play significant roles.

Supporting data structures established during preprocessing allow efficient implementations of initialization, recombination, mutation, and local improvement. Empirical tests indicate that the algorithm calculates solutions of high quality, which are optimal in many cases and usually substantially better than those of the other three heuristics from the literature. Although a theoretical upper bound for the computational costs of preprocessing is $O(|V|^2|E|)$, it is in practice also very fast, even on large problem instances and the memetic algorithm usually dominates the total computation time. Within the memetic algorithm, local improvement dominates the computational costs. The theoretical worst-case time complexity for local improvement of a solution is $O(|V_T|^3)$, however, we have argued that the expected computational costs are substantially smaller. Empirical results support this and show that the approach scales well to instances of large size.

For solving V2AUG exactly, we developed the branch-and-cut algorithm that relies on the minimum-cut formulation with an exponential number of constraints. The root node of the branch-and-bound tree is initialized with simple degree constraints. Separation of violated edge- and node-connectivity constraints can be done exactly by applying the polynomial-time algorithm for finding the minimum-weight cut of a graph.

In our computational experiments, we show that small and randomly generated problem instances can be solved exactly by using only this straight-forward method. For these instances, the exact approach is even faster than the proposed MA.

We also investigated the incorporation of the column generation method into the branch-and-cut algorithm. Our computational results indicate that, using pricing, we can substantially improve the algorithm's performance. For the detection of the inactive variables that should be priced in, we used the reserve graph technique proposed in [89]. For the fast calculation of the variable's reduced costs, we used special data structures. Furthermore, the number of inactive edges that should be checked for pricing is reduced by applying the ideas given in [131]. We have also shown that the well-designed primal heuristics can further improve the quality of the algorithm.

The BCP algorithm relies on the MA, since it uses its high-quality solutions as starting solutions and initial bounds. The obtained results indicate that for small and medium-size instances, finding high-quality upper bounds by means of the MA can slow down the optimization. However, for large instances, it is advantageous to combine both approaches, in order to reduce optimality gaps. The largest instances we have tested have 561, 1291 and 2103 nodes and $\approx 150\,000$, $\approx 300\,000$ and $\approx 250\,000$ edges, respectively. For these instances we did not find optimal solutions, but the obtained optimality gap is less than 1.4% on average.

Chapter 4

The Prize-Collecting Steiner Tree Problem

The recent deregulation of public utilities such as electricity and gas in Austria has shaken up the classical business model of energy companies and opened up the way towards new opportunities. Of particular interest in this field is the planning and expansion of district heating networks. This area of energy distribution is characterized by extremely high investment costs but also by an unusually loyal customer base and limited competition. Moreover, the required reduction of greenhouse emissions forces many energy companies to seek ways of improving their ecological balance sheet. A very attractive possibility to meet this goal is the use of biomass for heat generation. The combination of these two factors has made the planning of heating networks one of the major challenges for companies in this field [74].

In a typical planning scenario the input is a set of potential customers with known or estimated heat demands (represented by discounted future profits), and a potential network for laying the pipelines (which is usually identical to the street network of the district or town). Costs of the network are dominated by labor and right-of-way charges for laying the pipes and the costs for building the heating plant.

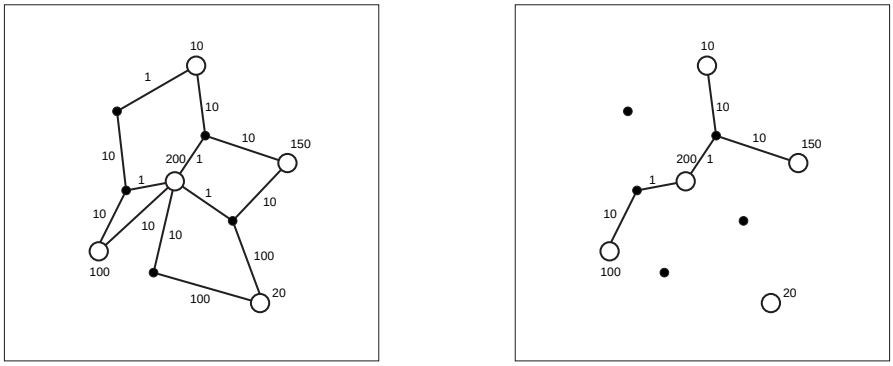
Recent advances in technology have made fiber optic connections for households economically feasible. Companies that design augmentation of fiber optic networks are faced with a similar problem: a set of potential customers with known estimated profits need to be added to an existing network. The fiber may be laid down through the streets – in this case the costs of lying the fiber directly correspond to streets' length, but may vary depending on the importance or function of each particular street. The fiber can be also laid through public properties, in which case special costs need to be considered.

Essentially, in both network design problems, the decision process faced by a profit oriented company consists of two parts: On one hand, a subset of particular profitable customers has to be selected from a total set of all potential customers. On the other hand, a network has to be designed to connect all selected customers in a feasible way. The natural trade-off between maximizing the sum of profits over all selected customers and minimizing the cost of the network leads to a prize-collecting objective function.

We can formulate this problem as an optimization problem on an undirected graph $G = (V, E, c, p)$, where the vertices V are associated with profits, $p : V \to \mathbb{R}^{\geq 0}$, and the edges E with costs, $c : E \to \mathbb{R}^{+}$. The graph in our application corresponds to the local street map, with the edges representing street segments and vertices representing street intersections and the location of potential customers. The profit p associated with a vertex is an estimate of the potential gain of revenue caused by that customer if connected to the network and receives its service. Vertices corresponding to street intersections have profit zero. The cost c associated with an edge is the cost of establishing the connection, i.e., of laying the pipe on the corresponding street segment.

The formal definition of the problem can be given as follows

Definition 9 (The Prize-Collecting Steiner Tree Problem, PCST). *Let $G = (V, E, c, p)$ be an undirected weighted graph as defined above. The* Linear Prize-Collecting Steiner Tree problem *(PCST) consists of finding a connected subgraph $T = (V_T, E_T)$ of G, $V_T \subseteq V$, $E_T \subseteq E$*



(a) (b)

Figure 4.1: Example of a PCST instance: (a) Each connection has fixed costs, hollow circles and filled circles represent customer and non-customer vertices, respectively. Goal: Build a network that maximizes the profit, defined as the difference between the gains and the expenses. (b) A feasible, but not optimal solution is given.

that maximizes

$$profit(T) = \sum_{v \in V_T} p(v) - \sum_{e \in E_T} c(e) \ . \tag{4.1}$$

It is easy to see that every optimal solution T will be a tree. Otherwise T would contain a cycle and removing any edge from the cycle would increase $profit(T)$ without violating connectivity of T. Throughout this thesis we will distinguish between *customer vertices*, defined as

$$R = \{v \in V \mid p(v) > 0\} \ ,$$

and *non-customer vertices*, in our application corresponding to street intersections, and we assume that $R \neq \emptyset$. Figure 4.1 illustrates an example of a PCST instance and a feasible solution for that instance.

The profit function given above is known in the literature as a function describing the *Net Worth Maximization* Problem (NW) [84]. In the so-called *Goemans and Williamson Minimization* Problem (GW) [66] the goal is to find a subtree $T = (V_T, E_T)$ that minimizes the following function:

$$GW(T) = \sum_{v \notin V_T} p(v) + \sum_{e \in E_T} c(e) \ . \tag{4.2}$$

Here, $p(v)$ is interpreted as penalty for *not* connecting a vertex v. As far as optimization is concerned, the NW and GW formulations are equivalent, since for all subtrees T of G

$$GW(T) + NW(T) = \sum_{v \in V} p(v) = const.$$

The objective function given by (4.1) can trivially be replaced by the minimization of the following difference:

$$c(T) = \sum_{e \in E_T} c(e) - \sum_{v \in V_T} p(v) \ . \tag{4.3}$$

In this formulation the problem can be seen as finding a network connecting a subset of customers so that the total costs are minimized. Note that the solutions obtained minimizing this objective function are usually negative; otherwise, an empty network would represent the optimal solution.

In this thesis we are going to concentrate on minimizing (4.2) as an objective function, as it has been considered in the literature before (see [66, 111, 23]).

PCST Variants. In practice, we often face additional side constraints. The planning problem of the heating network clearly requires that the heating plant is connected to the network. Fiber optic networks usually need to be augmented by connecting some already existing parts of the network to new customers. We can model both problems as a PCST by introducing a special vertex for the plant having a very high profit. In general, the *rooted prize-collecting Steiner tree problem* (RPCST), is defined as a variant of PCST with an additional source vertex $v_s \in V$ (representing a depot or repository, or an existing network[1] which must be a part of every feasible solution T.

An interesting variant of PCST arises if the energy company in our application chooses not to maximize the absolute gain of a project but rather the return on investment (RoI). Thus, we have to maximize the ratio of profits over costs. Formally, the resulting problem is called *fractional prize-collecting Steiner tree problem* (FPCST), with the following objective:

$$\max \frac{\sum_{v \in V_T} p(v)}{c_0 + \sum_{e \in E_T} c(e)} \tag{4.4}$$

over all subtrees T of G, where $c_0 > 0$ represents the fixed cost of the project, e.g., the setup costs of the heating plant in our application. Note that without the inclusion of c_0 in the definition, the empty set, which is a trivial feasible solution, would produce an undefined objective function value. Again, the rooted version of FPCST is a relevant special case to consider. Note that it cannot be tackled in an analogous way as above, since a vertex with artificially high profit would distort the ratio in (4.4).

Obviously, we cannot directly use integer linear programming to solve this problem with fractional objective. On the other side, since both numerator and denominator are linear functions, the problem belongs to the broader group of so-called *linear fractional combinatorial optimization problems* (LFCOs). In the recent paper [98] we discuss the solution of FPCST by using Newton's iterative method (cf. Radzik [138]), provided the corresponding linear rooted PCST instances can be solved to optimality. In the special case where the given graph G is a

[1] Note that when considering the existing network as a root, we have to shrink all the vertices of that network, and from multiple edges leaving the root to keep only the cheapest one (see, for example, Section 3.2.1, where a similar idea has been applied).

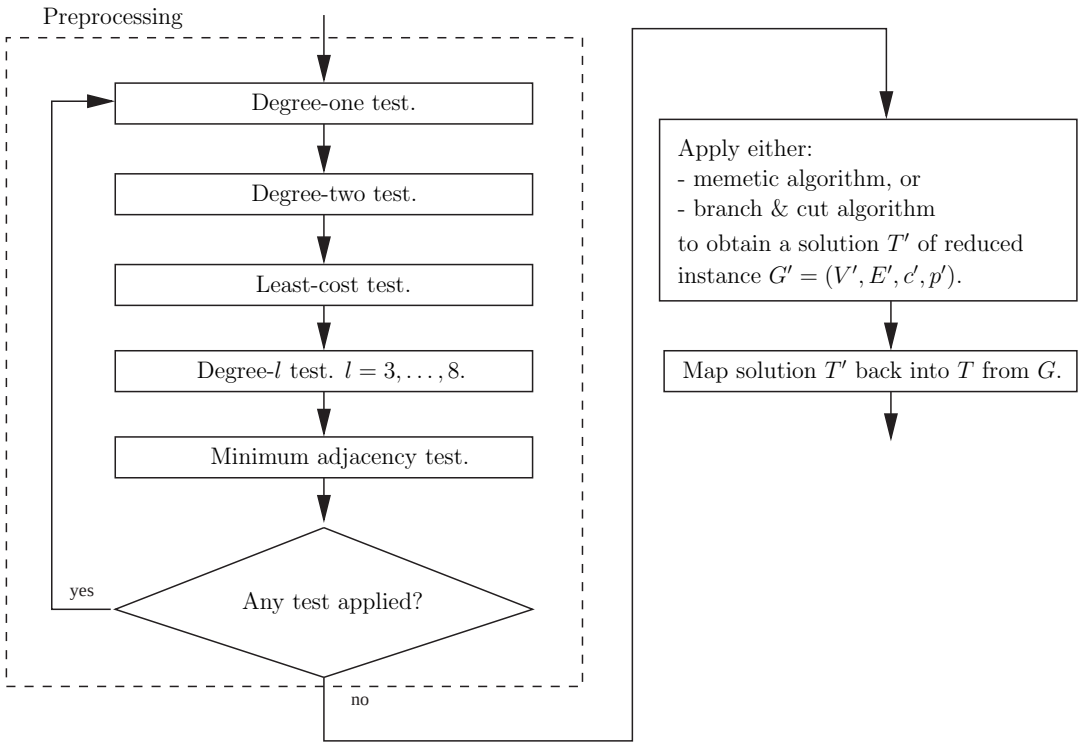


Figure 4.2: General approach for solving the PCST: the preprocessing procedure transforms the input graph $G = (V, E, c, p)$ into a reduced, undirected graph $G' = (V', E', c', p')$. The feasible solution T' obtained either by means of a memetic algorithm or by means of a branch-and-cut approach is finally mapped back into the subtree T of G.

tree, PCST can indeed be solved in $O(|V|)$ time which can be further exploited to construct an $O(|V| \log |V|)$ algorithm for FPCST in this special case.

Outline of this Chapter In the next section we give a short overview of previous work on PCST and some of its relatives. Preprocessing, which helps to significantly reduce the size of many instances, is treated in Section 4.2. In Section 4.3, we propose a MA used for finding approximate solutions for the prize-collecting Steiner tree problem. Extensive computational results are also provided. Different ILP models for PCST are presented and discussed in Section 4.4. In Section 4.4.3 we introduce our cut-based ILP model and describe how it can be solved in an efficient branch-and-cut framework in Section 4.5. Extensive computational experiments reported in Section 4.5.4 are followed by a concluding section.

The outline of our general approach for solving the prize-collecting Steiner tree problem is presented in Figure 4.2.

4.1 Previous Work

In 1987, Segev [149] considered for the first time the so-called *Node Weighted Steiner Tree Problem* (NWST) – the Steiner tree problem with node weights in addition to regular edge weights in which the sum of edge-costs and node-weights is minimized. NWST differs from PCST in the sense that the former requires a set of terminal vertices to be included in the solution (as in the classical Steiner tree problem). Segev noted that NWST can be turned into a directed Steiner tree problem, if the node weights are non-negative and the root of a solution tree is given.

His contribution concerns a special case of NWST, called the *single point weighted Steiner tree* problem (SPWST), where we are given a special vertex to be included in the solution. The weights on the remaining vertices are non-positive profit values, while non-negative weights on the edges reflect the costs incurred in obtaining or collecting these profits. Negating the node weights to make them positive and thus subtracting them from the edge costs in the objective function, immediately yields the objective function (4.3). Thus, as long as optimization is concerned, SPWST is equivalent to our definition of RPCST.

Balas [13] introduced the term "prize-collecting" in the context of the traveling salesman problem. The node weights in his model are non-negative and can be seen as penalties for not including nodes in a solution. In this way, his objective function resembles (4.2).

4.1.1 Approximation Algorithms

The first approximation algorithm for both the PCST and the prize-collecting traveling salesman problem has been proposed by Bienstock et al. [18], with approximation guarantees of 3 and 5/2, respectively.

Goemans and Williamson presented in [66] a purely combinatorial general approximation technique for a large class of constrained forest problems. Their algorithm is based on a primal-dual schema, runs in $O(n^2 \log n)$ time ($n := |V|$), and yields solutions within a factor of $2 - \frac{1}{n-1}$ of optimality for most of the considered problems: the generalized Steiner tree problem, the T-join problem, the minimum-weight perfect matching problem, etc.

The authors also provided an extension of the basic algorithm and proposed, in particular, algorithms for the prize-collecting Steiner tree and prize-collecting TSP problems. To solve the unrooted PCST, the Goemans-Williamson algorithm is performed for each vertex as a possible root. Thus, the total running time of the algorithm is $O(n^3 \log n)$.

Recently, Johnson et al. [84] improved the Goemans-Williamson algorithm by enhancing the second phase, the so-called *pruning phase*. The new algorithm is slightly faster and provides solutions that are provably at least as good and in practice significantly better. The authors also provided a modification to the growth phase to make the algorithm independent of the choice of the root vertex, thus giving a $(2 - \frac{1}{n-1})$–approximation algorithm for the general unrooted PCST which runs in $O(n^2 \log n)$ time. In exhaustive tests by Minkoff [121] the performance of the original Goemans-Williamson algorithm and the proposed improvement are compared on benchmark instances obtained from county street maps and randomly generated instances.

Feofiloff et al. [47] present a revised proof for the $(2 - \frac{1}{n-1})$–approximation algorithm by Johnson et al. and give an example showing that this ratio is tight. The authors also proposed a modification of the Goemans-Williamson algorithm based on a slightly different linear programming formulation. The new algorithm achieves a ratio of $2 - \frac{2}{n}$ and runs in $O(n^2 \log n)$ time.

4.1.2 Lower Bounds and Polyhedral Studies

Both Section 4.4 and 4.4.3 are devoted to ILP-formulations for PCST. Therefore, we will in this section only point out references without going into details.

In [149], Segev presented single- and multi-commodity flow formulations for SPWST (cf. Section 4.4.2). Furthermore, the author developed two bounding procedures based on Lagrangian relaxations of the corresponding flow formulations which were embedded in a branch-and-bound procedure. In addition, heuristics to compute feasible solutions were also included. The proposed algorithm was tested on a set of benchmark instances with up to 40 vertices.

Fischetti [51] studied the facial structure of a generalization of the problem, the so-called *Steiner arborescence* (or *directed Steiner tree*) problem and pointed out that the NWST can be transformed into it. The author considered several classes of valid inequalities and introduced a new inequality class with arbitrarily large coefficients, showing that all of them define distinct facets of the underlying polyhedron.

Goemans provided in [65] a theoretical study on the polyhedral structure of the node-weighted Steiner tree problem NWST and showed that this characterization is complete in the case that the input graph is series-parallel. There, SPWST, i.e., RPCST appears as the r-tree problem.

Engevall et al. [44] proposed another ILP formulation for the NWST, based on the *shortest spanning tree* problem formulation, introduced originally by Beasley [16] for the Steiner tree problem. In their formulation, besides the given root vertex r, an artificial root vertex 0 is introduced, and an edge between vertex 0 and r is set. They searched for a tree with additional constraints: each vertex v connected to vertex 0 must have degree one. The solution is interpreted in the following way: the vertices adjacent to vertex 0 are not considered as a part of the final solution. For the description of the tree, the authors use a modification of the generalized subtour elimination constraints (cf. Section 4.4.1). For finding good lower bounds, the authors use a Lagrangian heuristic and subgradient procedure based on the shortest spanning tree formulation. Experimental results presented for instances with up to 100 vertices indicated that the new approach outperformed Segev's algorithm.

Lucena and Resende [111] presented a cutting plane algorithm for the PCST based on generalized subtour elimination constraints (see again Section 4.4.1). Their algorithm contains basic reduction steps similar to those already given by Duin and Volgenant [41], and was tested on two groups of benchmark instances: the first group contains instances adopted from Johnson et al. [84], ranging from 100 vertices and 284 edges to 400 vertices and 1507 edges. The second group is derived from the Steiner problem instances (series C and D) of the OR-Library, with sizes ranging from 500 vertices and 625 edges to 1000 vertices and 25000 edges.

The proposed algorithm solved many of the considered instances to optimality, but not all of them (cf. Section 4.5.4).

4.1.3 Metaheuristics

Canuto et al. [23] developed a multi-start local-search-based algorithm with perturbations. Perturbations are done by changing the parameters of the input graph, either by setting profits of potential customers to zero, or by modifying the profits of non-zero vertices. Feasible solutions are obtained by the Goemans-Williamson algorithm, followed by a local search procedure. Within local search, the complete 1-flip neighborhood is examined, and the best solution found so far is selected and inserted in a pool of high-quality elite solutions. Between a randomly selected solution from the pool, and the solution found in the current iteration, *path relinking* is applied, exploring trajectories that connect these two solutions. A variable neighborhood search method is finally applied as a post-optimization step. The algorithm found optimal solutions on nearly all test instances for which the optimum is known.

4.2 Preprocessing

In this section, we briefly describe reduction techniques adopted from the work of Duin and Volgenant [41] for the NWST, which have been partially used also in [111]. From the implementation point of view, we transform the graph $G = (V, E, c, p)$ into a reduced graph $G' = (V', E', c', p')$ by applying the steps described below and maintain a *back-mapping* function to transform each feasible solution T' of G' into a feasible solution T of G.

Least-Cost Test Let d_{ij} represent the shortest path length between any two vertices i and j from V (considering only edge-costs). If $\exists e = (i, j)$ such that $d_{ij} < c_{ij}$ then edge e can simply be discarded from G.

Degree-l Test Consider a vertex $v \notin R$ of degree $l \geq 3$, connected to vertices from $Adj(v) = \{v_1, v_2, \ldots, v_l\}$. For any subset $K \subset V$, denote with $\mathrm{MST}_d(K)$, the minimum spanning tree of K with distances d_{ij}. If

$$\mathrm{MST}_d(K) \leq \sum_{w \in K} c_{vw}, \quad \forall K \subseteq Adj(v), \quad |K| \geq 3, \tag{4.5}$$

then v's degree in an optimal solution must be zero or two. Hence, we can remove v from G by replacing each pair (v_i, v), (v, v_j) with (v_i, v_j) either by adding a new edge $e = (v_i, v_j)$ of cost $c_e = c_{v_i v} + c_{v v_j} - p_v$ or in case e already exists, by defining $c_e = \min\{c_e, c_{v_i v} + c_{v v_j} - p_v\}$.

It is straightforward to apply a simplified version of this test to all vertices $v \in V$ with $l = 1$ and $l = 2$.

Minimum Adjacency Test This test is also known as $V \setminus K$ *reduction test* from [41]. If there are adjacent vertices $i, j \in R$ such that:

$$\min\{p_i, p_j\} - c_{ij} > 0 \text{ and } c_{ij} = \min_{it \in E} c_{it} \ ,$$

then i and j can be fused into one vertex of weight $p_i + p_j - c_{ij}$.

Summary of the Preprocessing Procedure We apply the steps described above iteratively, as long as any of them changes the input graph (see Fig. 4.2). The total number of iterations is bounded by the number of edges in G. Each iteration is dominated by the time complexity of the least-cost test, i.e., by the computation of all-pair shortest paths, which is $O(|E||V|+|V|^2 \log |V|)$. Thus, the preprocessing procedure requires $O(|E|^2|V|+|E||V|^2 \log |V|)$ time in the worst case, in which the input graph would be reduced to a single vertex. However, in practice, the running time is much lower, as documented in Section 4.2.1. The space complexity of preprocessing does not exceed $O(|E|^2)$.

4.2.1 Impacts of preprocessing

To test our preprocessing algorithm as well as one upper and one lower bounding procedure for the PCST, we consider the following groups of instances:

- Johnson et al. [84] tested their approximation algorithm on two sets of randomly generated instances. In the so-called `P` class, instances are unstructured and designed to have constant expected degree and profit to weight ratio. The `K` group comprises random geometric instances designed to have a structure somewhat similar to street maps. A detailed description of the generators for these instances can be found in [121]. In our tests, we considered a part of these instances with up to 400 vertices and 1 576 edges that have also been used by Lucena and Resende [111] and Canuto et al. [23].

- Canuto et al. [23] generated a set of 80 test problems derived from the Steiner problem instances of the well-known OR-Library[2]. For each of the 40 problems from series `C` and `D`, two sets of instances were generated by assigning zero profits to non-terminal vertices and randomly generated profits in the interval $[1, \mathit{maxprize}]$ to terminal vertices. Here, $\mathit{maxprize} = 10$ for problems in set `A`, and $\mathit{maxprize} = 100$ for problems in set `B`. Instances of group `C` contain 500 vertices, and between 625 and 12 500 edges, while instances of group `D` contain 1 000 vertices and between 1 250 and 25 000 edges.

  Following this schema, we generated an additional set of 40 larger benchmark instances derived from series `E` of the Steiner problem instances in the OR-Library. These new instances contain 2 500 vertices and between 3 125 and 62 500 edges.

Instance sets `K`,`P`,`C` and `D` of instances are available at `http://www.research.att.com/~mgcr/data/index.html`. All other problem instances used in this chapter are available in

[2]OR-library: J. E. Beasley, `http://mscmga.ms.ic.ac.uk/info.html`

Table 4.1: Preprocessing results on instances derived by Johnson et al. [84]. Reduced graphs $(V_{\mathrm{LR}}, E_{\mathrm{LR}})$ obtained after preprocessing of Lucena & Resende [111] (LR), our reduced graphs (V', E') and the running time of the preprocessing.

Instance	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_{\mathrm{LR}}\rvert$	$\lvert E_{\mathrm{LR}}\rvert$	$\lvert V'\rvert$	$\lvert E'\rvert$	$\frac{\lvert V'\rvert}{\lvert V\rvert}$ [%]	$\frac{\lvert E'\rvert}{\lvert E\rvert}$ [%]	t_{prep} [s]
P100	100	284	83	221	66	163	66.0	57.4	0.1
P100.1	100	284	91	211	84	196	84.0	69.0	0.1
P100.2	100	297	83	201	75	187	75.0	63.0	0.1
P100.3	100	316	94	243	91	237	91.0	75.0	0.0
P100.4	100	284	83	221	69	186	69.0	65.5	0.1
P200	200	587	172	447	166	438	83.0	74.6	0.2
P400	400	1144	356	972	345	1002	86.2	87.6	2.0
P400.1	400	1212	352	1025	323	983	80.8	81.1	2.5
P400.2	400	1196	364	1040	341	997	85.2	83.4	2.1
P400.3	400	1175	358	1008	334	969	83.5	82.5	2.3
P400.4	400	1144	356	972	344	949	86.0	83.0	1.7
K100	100	319	37	111	45	191	45.0	59.9	0.1
K100.1	100	319	37	111	42	185	42.0	58.0	0.1
K100.2	100	339	33	118	24	83	24.0	24.5	0.1
K100.3	100	407	20	87	26	123	26.0	30.2	0.1
K100.4	100	364	36	132	29	113	29.0	31.0	0.1
K100.5	100	358	38	140	31	120	31.0	33.5	0.1
K100.6	100	307	29	81	22	64	22.0	20.8	0.1
K100.7	100	315	25	71	25	93	25.0	29.5	0.1
K100.8	100	343	49	173	43	144	43.0	42.0	0.1
K100.9	100	333	21	67	22	70	22.0	21.0	0.1
K100.10	100	319	37	111	27	78	27.0	24.5	0.1
K200	200	691	99	361	81	271	40.5	39.2	0.5
K400	400	1507	211	855	231	914	57.8	60.7	3.0
K400.1	400	1507	211	855	217	854	54.2	56.7	2.7
K400.2	400	1527	217	935	228	948	57.0	62.1	3.0
K400.3	400	1492	195	694	210	806	52.5	54.0	3.8
K400.4	400	1426	190	747	197	784	49.2	55.0	2.9
K400.5	400	1456	223	799	220	799	55.0	54.9	2.8
K400.6	400	1576	239	986	241	1035	60.2	65.7	4.0
K400.7	400	1442	225	883	225	867	56.2	60.1	2.2
K400.8	400	1516	245	1036	235	987	58.8	65.1	3.1
K400.9	400	1500	205	803	211	862	52.8	57.5	3.0
K400.10	400	1507	211	855	221	923	55.2	61.2	3.7

Table 4.2: Preprocessing results on class C of instances derived by Canuto et al. [23]. Reduced graphs $(V_{\mathrm{LR}}, E_{\mathrm{LR}})$ obtained after preprocessing of Lucena & Resende [111] (LR), our reduced graphs (V', E') and the running time of the preprocessing.

Instance	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_{\mathrm{LR}}\rvert$	$\lvert E_{\mathrm{LR}}\rvert$	$\lvert V'\rvert$	$\lvert E'\rvert$	$\frac{\lvert V'\rvert}{\lvert V\rvert}$ [%]	$\frac{\lvert E'\rvert}{\lvert E\rvert}$ [%]	t_{prep} [s]
C1-A	500	625	116	214	116	214	23.2	34.2	1.2
C1-B	500	625	125	226	125	226	25.0	36.2	1.2
C2-A	500	625	110	209	109	207	21.8	33.1	1.1
C2-B	500	625	112	211	111	209	22.2	33.4	1.1
C3-A	500	625	174	293	160	277	32.0	44.3	1.1
C3-B	500	625	204	325	185	304	37.0	48.6	1.3
C4-A	500	625	207	331	178	300	35.6	48.0	1.2
C4-B	500	625	247	371	218	341	43.6	54.6	1.3
C5-A	500	625	254	375	163	274	32.6	43.8	1.2
C5-B	500	625	326	447	199	314	39.8	50.2	1.7
C6-A	500	1000	356	823	355	822	71.0	82.2	2.1
C6-B	500	1000	356	823	356	823	71.2	82.3	2.1
C7-A	500	1000	366	843	365	842	73.0	84.2	2.6
C7-B	500	1000	366	843	365	842	73.0	84.2	2.5
C8-A	500	1000	382	866	367	849	73.4	84.9	2.7
C8-B	500	1000	385	869	369	850	73.8	85.0	3.0
C9-A	500	1000	412	903	387	877	77.4	87.7	2.4
C9-B	500	1000	416	907	389	879	77.8	87.9	2.8
C10-A	500	1000	431	920	359	841	71.8	84.1	3.3
C10-B	500	1000	440	929	323	798	64.6	79.8	3.4
C11-A	500	2500	489	2143	489	2143	97.8	85.7	9.4
C11-B	500	2500	489	2143	489	2143	97.8	85.7	9.5
C12-A	500	2500	485	2189	484	2186	96.8	87.4	6.8
C12-B	500	2500	485	2189	484	2186	96.8	87.4	6.8
C13-A	500	2500	488	2167	472	2113	94.4	84.5	9.8
C13-B	500	2500	488	2167	471	2112	94.2	84.5	9.8
C14-A	500	2500	493	2168	466	2081	93.2	83.2	7.5
C14-B	500	2500	493	2168	459	2048	91.8	81.9	7.5
C15-A	500	2500	496	2153	406	1871	81.2	74.8	6.5
C15-B	500	2500	496	2153	370	1753	74.0	70.1	6.0
C16-A	500	12500	500	12500	500	4740	100.0	37.9	2.4
C16-B	500	12500	500	12500	500	4740	100.0	37.9	2.4
C17-A	500	12500	500	12500	498	4694	99.6	37.6	2.4
C17-B	500	12500	500	12500	498	4694	99.6	37.6	2.3
C18-A	500	12500	500	12500	469	4569	93.8	36.6	2.6
C18-B	500	12500	500	12500	465	4538	93.0	36.3	2.9
C19-A	500	12500	500	12500	430	3982	86.0	31.9	2.9
C19-B	500	12500	500	12500	416	3867	83.2	30.9	2.8
C20-A	500	12500	500	12500	241	1222	48.2	9.8	6.1
C20-B	500	12500	500	12500	133	563	26.6	4.5	5.0

Table 4.3: Preprocessing results on class D of instances derived by Canuto et al. [23]. Reduced graphs $(V_{\text{LR}}, E_{\text{LR}})$ obtained after preprocessing of Lucena & Resende [111] (LR), our reduced graphs (V', E') and the running time of the preprocessing.

Instance	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_{\text{LR}}\rvert$	$\lvert E_{\text{LR}}\rvert$	$\lvert V'\rvert$	$\lvert E'\rvert$	$\frac{\lvert V'\rvert}{\lvert V\rvert}$ [%]	$\frac{\lvert E'\rvert}{\lvert E\rvert}$ [%]	t_{prep} [s]
D1-A	1000	1250	233	443	231	440	23.1	35.2	4.9
D1-B	1000	1250	233	443	233	443	23.3	35.4	4.9
D2-A	1000	1250	261	485	257	481	25.7	38.5	4.9
D2-B	1000	1250	264	488	264	488	26.4	39.0	4.9
D3-A	1000	1250	340	571	301	529	30.1	42.3	5.5
D3-B	1000	1250	400	634	372	606	37.2	48.5	6.3
D4-A	1000	1250	381	616	311	541	31.1	43.3	5.6
D4-B	1000	1250	458	694	387	621	38.7	49.7	7.2
D5-A	1000	1250	521	768	348	588	34.8	47.0	7.6
D5-B	1000	1250	660	907	411	649	41.1	51.9	11.5
D6-A	1000	2000	741	1709	740	1707	74.0	85.3	14.4
D6-B	1000	2000	741	1709	741	1708	74.1	85.4	14.7
D7-A	1000	2000	735	1706	734	1705	73.4	85.2	11.3
D7-B	1000	2000	736	1707	736	1707	73.6	85.3	11.4
D8-A	1000	2000	794	1772	764	1738	76.4	86.9	11.7
D8-B	1000	2000	800	1780	778	1757	77.8	87.8	12.3
D9-A	1000	2000	791	1758	752	1716	75.2	85.8	17.9
D9-B	1000	2000	800	1767	761	1724	76.1	86.2	20.9
D10-A	1000	2000	844	1825	694	1661	69.4	83.0	14.6
D10-B	1000	2000	860	1842	629	1586	62.9	79.3	18.5
D11-A	1000	5000	986	4658	986	4658	98.6	93.2	27.7
D11-B	1000	5000	986	4658	986	4658	98.6	93.2	23.6
D12-A	1000	5000	992	4641	991	4639	99.1	92.8	23.1
D12-B	1000	5000	992	4641	991	4639	99.1	92.8	22.3
D13-A	1000	5000	990	4614	966	4572	96.6	91.4	27.7
D13-B	1000	5000	990	4614	961	4566	96.1	91.3	28.0
D14-A	1000	5000	991	4621	946	4500	94.6	90.0	35.5
D14-B	1000	5000	991	4621	931	4469	93.1	89.4	37.2
D15-A	1000	5000	993	4622	832	4175	83.2	83.5	47.1
D15-B	1000	5000	993	4622	747	3896	74.7	77.9	49.2
D16-A	1000	25000	1000	25000	1000	10595	100.0	42.4	10.8
D16-B	1000	25000	1000	25000	1000	10595	100.0	42.4	10.8
D17-A	1000	25000	1000	25000	999	10534	99.9	42.1	10.8
D17-B	1000	25000	1000	25000	999	10534	99.9	42.1	10.7
D18-A	1000	25000	1000	25000	944	9949	94.4	39.8	11.7
D18-B	1000	25000	1000	25000	929	9816	92.9	39.3	12.0
D19-A	1000	25000	1000	25000	897	9532	89.7	38.1	12.4
D19-B	1000	25000	1000	25000	862	9131	86.2	36.5	13.1
D20-A	1000	25000	1000	25000	488	2511	48.8	10.0	37.3
D20-B	1000	25000	1000	25000	307	1383	30.7	5.5	32.9

Table 4.4: Preprocessing results on class E of instances derived from OR-Library. The size of our reduced graphs (V', E') and the preprocessing's running time are shown.

Instance	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	$\frac{\|V'\|}{\|V\|}$ [%]	$\frac{\|E'\|}{\|E\|}$ [%]	t_{prep} [s]
E01-A	2500	3125	651	1246	26.0	39.9	21.5
E01-B	2500	3125	655	1250	26.2	40.0	21.8
E02-A	2500	3125	694	1304	27.8	41.7	20.7
E02-B	2500	3125	697	1307	27.9	41.8	20.7
E03-A	2500	3125	813	1414	32.5	45.2	29.4
E03-B	2500	3125	962	1572	38.5	50.3	30.9
E04-A	2500	3125	829	1425	33.2	45.6	24.9
E04-B	2500	3125	980	1588	39.2	50.8	26.2
E05-A	2500	3125	893	1502	35.7	48.1	36.9
E05-B	2500	3125	1029	1644	41.2	52.6	45.0
E06-A	2500	5000	1821	4283	72.8	85.7	37.9
E06-B	2500	5000	1821	4283	72.8	85.7	37.6
E07-A	2500	5000	1863	4339	74.5	86.8	39.3
E07-B	2500	5000	1865	4341	74.6	86.8	39.4
E08-A	2500	5000	1902	4379	76.1	87.6	40.1
E08-B	2500	5000	1911	4387	76.4	87.7	50.4
E09-A	2500	5000	1909	4388	76.4	87.8	50.5
E09-B	2500	5000	1918	4397	76.7	87.9	54.7
E10-A	2500	5000	1716	4181	68.6	83.6	60.6
E10-B	2500	5000	1594	4045	63.8	80.9	84.6
E11-A	2500	12500	2491	12063	99.6	96.5	145.1
E11-B	2500	12500	2491	12063	99.6	96.5	146.2
E12-A	2500	12500	2490	12090	99.6	96.7	82.5
E12-B	2500	12500	2490	12090	99.6	96.7	85.5
E13-A	2500	12500	2430	11949	97.2	95.6	148.2
E13-B	2500	12500	2407	11915	96.3	95.3	146.7
E14-A	2500	12500	2366	11872	94.6	95.0	144.2
E14-B	2500	12500	2311	11737	92.4	93.9	145.7
E15-A	2500	12500	2044	10845	81.8	86.8	207.8
E15-B	2500	12500	1864	10264	74.6	82.1	234.3
E16-A	2500	62500	2500	29332	100.0	46.9	82.2
E16-B	2500	62500	2500	29332	100.0	46.9	81.9
E17-A	2500	62500	2500	29090	100.0	46.5	81.6
E17-B	2500	62500	2500	29090	100.0	46.5	81.8
E18-A	2500	62500	2378	28454	95.1	45.5	85.7
E18-B	2500	62500	2347	28269	93.9	45.2	86.9
E19-A	2500	62500	2156	25011	86.2	40.0	92.2
E19-B	2500	62500	2085	23641	83.4	37.8	94.0
E20-A	2500	62500	1525	12770	61.0	20.4	107.3
E20-B	2500	62500	861	3881	34.4	6.2	231.5

our online database for PCST instances and solutions at the following URL: `http://www.ads.tuwien.ac.at/pcst`. Our computational experiments were performed on a Pentium IV/2.8GHz PC with 2 GB RAM memory.

We compare our preprocessing results against those obtained by Lucena and Resende in [111] (to which we refer as LR). The differences between our and LR's preprocessing are the following:

1. In [111], degree-one and degree-two tests are applied to the non-customer vertices only. Note that such changes are safe, in the sense that the objective function value on the reduced and on the original instance are always the same. In our case, however, the objective function value of the original instance can only be obtained by means of the back-mapping function that transforms the reduced instance into the original one.

2. Our degree-n test has been applied for all values $1 \leq n \leq 8$, while the highest value of n for LR preprocessing is not published.

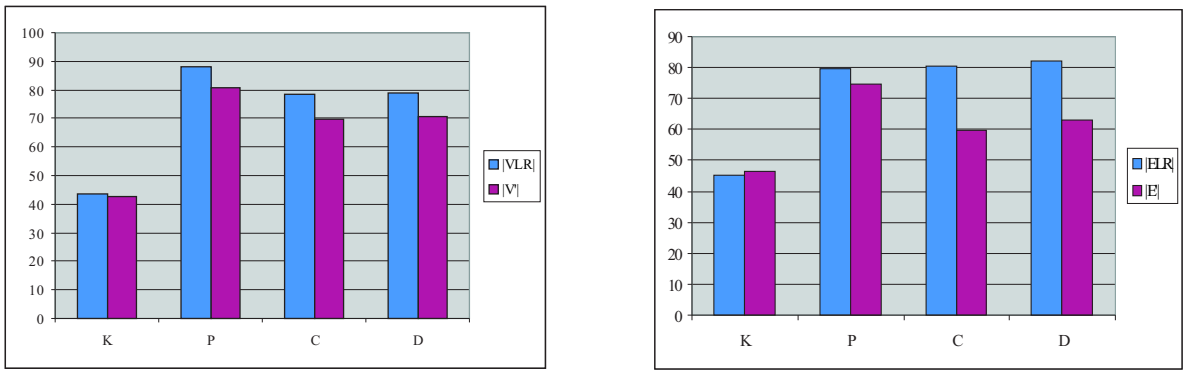
3. In [111], minimum adjacency test was not applied at all.

Tables 4.1, 4.2, 4.3 and 4.4 show the results of the proposed preprocessing. The number of vertices $|V|$ and the number of edges $|E|$ of the original graph are given. In ($|V_{\mathrm{LR}}|$,$|E_{\mathrm{LR}}|$) and ($|V'|, |E'|$), we show the number of vertices and edges of the instances obtained using LR and our new preprocessing, respectively. We also present savings in percent obtained using the new reductions ($\frac{|V'|}{|V|}$ [%] and $\frac{|E'|}{|E|}$ [%]) as well as the preprocessing's running time in seconds (t_{prep} [s]). Figure 4.3(a) illustrates summarized average comparison results for each of the groups `K, P, C, D`.

The results indicate that the proposed preprocessing may significantly reduce the size of input graphs within a short running time. Indeed, the average reductions on the number of vertices are between 28% (group `E`) and 57% (group `K`), while the number of edges can be reduced on average for between 34% (group `E`) and 53% (group `K`). The running time of preprocessing of Lucena and Resende is not published, thus we only compare the average reductions. Reductions of the number of vertices after LR preprocessing are between 1% (group `K`) and 8.7% (group `C`) worse than reductions after our preprocessing. Regarding the reductions on the number of edges, for group `K` the LR preprocessing is for about 1% better than our preprocessing. On the other side, for groups `C` and `D`, our reductions on the number of edges are for about 20% better than those of LR.

When considering the largest instances from groups `C` and `D`, it is easy to see that the minimum adjacency test plays the crucial role. Indeed, the LR preprocessing was not able to reduce these instances at all. On the other side, the minimum adjacency test reduced the size of these instances as follows: the number of vertices could be reduced by 17% and 15.7% for groups `C` and `D`, respectively. The number of edges is reduced by 70% and 66% for groups `C` and `D`, respectively. Drastic reductions are obtained for instances `C20-B` and `D20-B`, where the number of edges is reduced by a factor of 20(!).

The preprocessing's average running time indicates that the theoretical upper bound of $O(|E|^2|V| + |E||V|^2 \log |V|)$ usually does not occur in practical situations and that all the



(a) (b)

Figure 4.3: Summarized preprocessing results: (a) The number of vertices (in %) averaged per group. (b) The number of edges (in %) averaged per group.

instances we considered could be significantly reduced within a very short running time. Indeed, in the worst case, our preprocessing algorithm needed 235 seconds (instance `E15-B`), or 82 seconds on average for the group `E`.

4.3 A Memetic Algorithm for the PCST

The memetic algorithm we propose in this section has the basic structure shown in Algorithm 3 on page 25. All individuals in the population represent local optima with respect to their subtrees, which is ensured by applying a local improvement algorithm after a candidate solution's generation and after the application of the evolutionary variation operators *recombination* and *mutation*. After a pair of solutions is selected from the current population, recombination is always applied, while mutation is applied only with a probability p_{mut}, where $0 \leq p_{\text{mut}} \leq 1$. A new created candidate solution replaces always the worst solution in the population with one exception: To guarantee a minimum diversity, a new candidate solution identical to a solution already contained in the population is discarded [142].

Within this section, we propose problem-dependent variation operators, a clustering procedure used within mutation and a local improvement procedure. Extensive experiments on the benchmark instances mentioned in Section 4.3.7 show that the MA compares favorably to previously published results. While the solution values are almost always the same (or just slightly worse), a significant reduction of running time for medium and large instances is obtained.

4.3.1 Clustering

In the design of district heating or fiber optic networks, it is often the case that in small settlements the customers are grouped together, and that it either pays off to take all of them

at once, or not to take them at all. By employing *clustering* as a grouping procedure within variation operators, we can group the subsets of vertices and insert or delete them at once.

The clustering we implemented, the so-called *customer-based clustering*, sets up one cluster for each customer vertex and assigns all non-customer vertices over these clusters.

During the clustering we decompose the set V' of all vertices into disjoint cluster sets. As proposed by Mehlhorn in [115], we can define for each customer vertex $z \in R' = \{v \in V' \mid p'(v) > 0\}$, a cluster set $N(z)$:

$$N(z) := \{v \in V' \setminus R' \mid \forall c \in R' : \ d_{G'}(v, z) \leq d_{G'}(v, c)\} \cup \{z\}$$

where $d_{G'}(u, v)$ denotes the length of the shortest path between two vertices u and v in G', i.e. each non-customer vertex is assigned to the cluster set of its nearest customer vertex. For each vertex $v \in N(z)$ we call z the *base* of v, written $base(v) = z$. Note that the sets $N(z)$ correspond to the Voronoi regions in Euclidean plane.

As next step, we construct the so-called *cluster graph* $G_N = (V_N, E_N)$. Each vertex of $v \in V_N$ corresponds to the cluster of some customer vertex $z = base(v)$ and is assigned the set of vertices $V_N(v) = \{u \in V \mid base(u) = base(v) = z\}$ which belong to this cluster and the list of edges $E_N(v) = \{(u, w) \in E \mid base(u) = base(w) = z\}$ which lie fully inside this cluster. Two cluster vertices $u_N, v_N \in V_N$ are adjacent iff there exist two vertices $u, v \in V'$ assigned to u_N and v_N, respectively, that are adjacent in G'.

Runtime complexity for the preprocessing is dominated by the shortest paths calculations. Using Mehlhorn's algorithm [115], the clustering can be implemented in $O(|V'| \log |V'| + |E'|)$ time.

4.3.2 Edge-Set Encoding

Each possible PCST solution is a subtree of graph G'. In order to obtain an efficient EA, the desired tree-encoding should satisfy the following requirements (according to Palmer and Kershenbaum [132]):

- The encoding must be *coverable*, i.e. capable of representing all possible phenotypes, in our case all feasible subtrees of G'.
- The frequency with which each phenotype is represented should be the same (*unbiased* encoding).
- The mapping from the phenotype to the genotype space must be *injective*, i.e. one genotype may correspond to at most one phenotype.
- It should be *computationally easy* to decode a genotype.
- An efficient tree-encoding must provide *locality*: small changes in the genotype should correspond to small changes in the phenotype.

Raidl and Julstrom [143] pointed out other important principles of an efficient tree-encoding:

- *Heritability:* Offsprings of crossover should represent solutions that combine substructures of their parental solutions. In our implementation, offsprings should represent trees consisting mostly of parental edges.
- *Constraints:* The decoding of chromosomes and the variation operators should be able to enforce problem-specific constraints.
- *Hybrids:* The operators should be able to incorporate problem-dependent heuristics. Here, such a heuristic might favor edges of lower cost, or adjacent edges of vertices with higher profit.

In the same paper, Raidl and Julstrom [143] proposed a direct representation of spanning trees as sets of their edges. Traditional initialization and variation operators applied to this representation rarely generate feasible solutions. Thus, the authors proposed new problem-dependent operators based on random spanning tree algorithms. In combination with these operators, the edge-set encoding exhibited significant advantages over the previous encoding of spanning trees.

Following these results, PCST solutions in our MA are encoded using sets of their edges. The solution edges are represented in form of a hash-table, requiring only $O(|V'|)$ space. Insertion and deletion of edges, but also checking for existence of an edge, can be done in expected constant time. For this encoding we designed special problem-dependent variation operators which are described below.

4.3.3 Initialization

For creating initial candidate solutions we used a modified *distance network heuristic* for the Steiner tree problem (see, for example, [135, pp. 88–90]). Note that there is a certain correspondence between customer vertices of the graph G' and so-called *terminal vertices* of the Steiner tree problem (STP) in graphs (i.e. the vertices that must be spanned by the final solution). For each input graph $G' = (V', E', c', p')$ of the PCST, we build a *distance network* $G_D(R', E_D, c_D)$ as follows. The set of vertices R' corresponds to the customer vertices from V', i.e. $R' = \{v \in V' \mid p'(v) > 0\}$. The distance network is a complete graph whose edge costs are given by:

$$c_D(u, v) = d_{G'}(u, v), \text{for all } u, v \in R' \ ,$$

where $d_{G'}(u, v)$ denotes a shortest path between the vertices u and v in G'. The simple distance network heuristic for the Steiner tree problem in graphs represents a 2-approximation algorithm for the STP, and according to Mehlhorn [115], it can be implemented to run in $O(|V'| \log |V'| + |E'|)$ time.

In our initialization procedure, we randomly select a subset $V'_{init} \subset R'$ of customer vertices and apply the distance network heuristic on it. The procedure works as follows:

1. Randomly select a subset $V'_{init} \subset R'$ of size $\lceil p_{init} \cdot |R'| \rceil, p_{init} \in (0, 1)$;
2. Construct the minimum spanning tree (MST) T'_{init} on the subgraph of G_D induced by V'_{init};

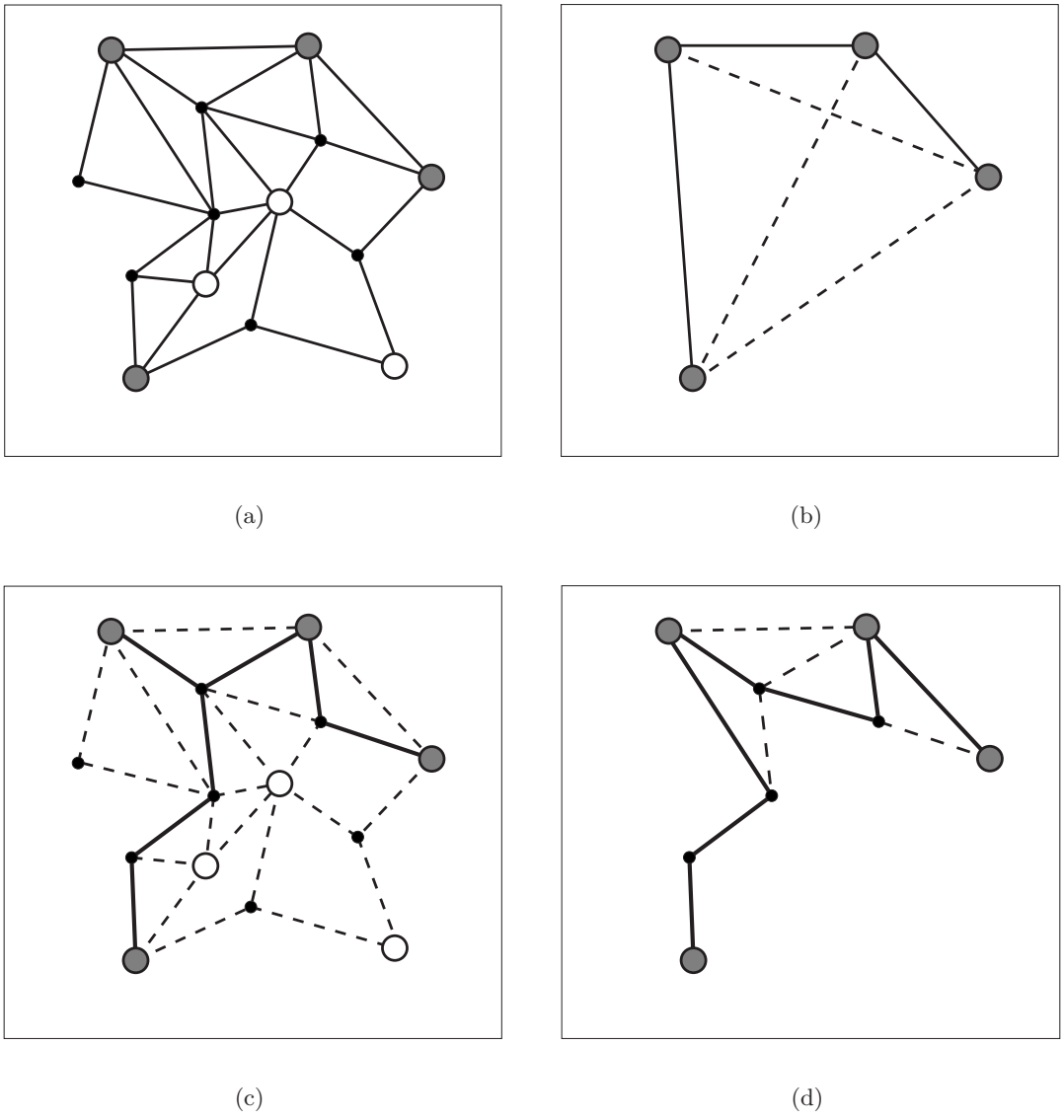


Figure 4.4: Initialization operator: (a) an input graph in which a subset of customer vertices V'_{init} is selected and treated as terminals (shadowed vertices); (b) distance network and its minimum spanning tree T'_{init} shown by bold lines; (c) subgraph G'_r represented with bold lines; (d) subgraph induced by the vertices of G'_r and its minimum spanning tree represented with bold lines.

3. Obtain a reduced subgraph G'_r of G' by replacing each edge of T'_{init} by its corresponding shortest path in G';

4. Find the MST T'_r on the subgraph induced by the vertices of G'_r;

5. Apply local improvement on T'_r;

Assuming that the computation of the distance network is done once in the preprocessing phase of the algorithm, and the edge-costs of the distance network are pre-sorted, the running time of initialization is dominated by the computation of the MST, which can be done in $O(|E'| \cdot \alpha(|E'|, |V'|))$ time.

4.3.4 Recombination

The recombination operator is designed with inheritance in mind; so we try to adopt the structural properties of two parental solutions. If the two solutions to be combined share at least one vertex, we just construct the spanning tree over the union of their edge sets. Due to the deterministic nature of the local improvement procedure and to avoid premature convergence, we build a random spanning tree on the union of parental edges.

When the parent solutions are disjoint, we randomly choose a customer vertex out of each solution, lookup the shortest path between these two vertices and add for each vertex n_p along the path all edges that belong to the cluster of n_p (including n_p itself). Finally, we build a random spanning tree over all these edges.

Runtime complexity is dominated by the computation of the random spanning tree, thus $O(|E'| \cdot \alpha(|E'|, |V'|))$.

4.3.5 Mutation

The aim of the mutation operator is to make small changes in the current solution which we achieve by connecting one or more new customers to the solution. We do this by adding clusters which are adjacent to the candidate solution and contain new customers.

To find an appropriate cluster to add, the algorithm chooses a *border vertex* n_b randomly. A border vertex is a vertex which is adjacent to at least one vertex outside our current solution. We incorporate the vertices of the cluster of n_b into our solution and employ the cluster graph to find a neighboring cluster whose customer (base) vertex is preferably not yet element of the current solution; the vertices contained in this cluster will be added to our solution. Finally we construct a minimum spanning tree to obtain a valid edge set.

To make the mutation operator more aggressive, this procedure may be iteratively applied $p_{itermut}$ times. Assuming that the edges are pre-sorted in increasing order with respect to their costs, the running time is dominated by the calculation of the spanning tree, and is $O(|E'| \cdot \alpha(|E'|, |V'|))$.

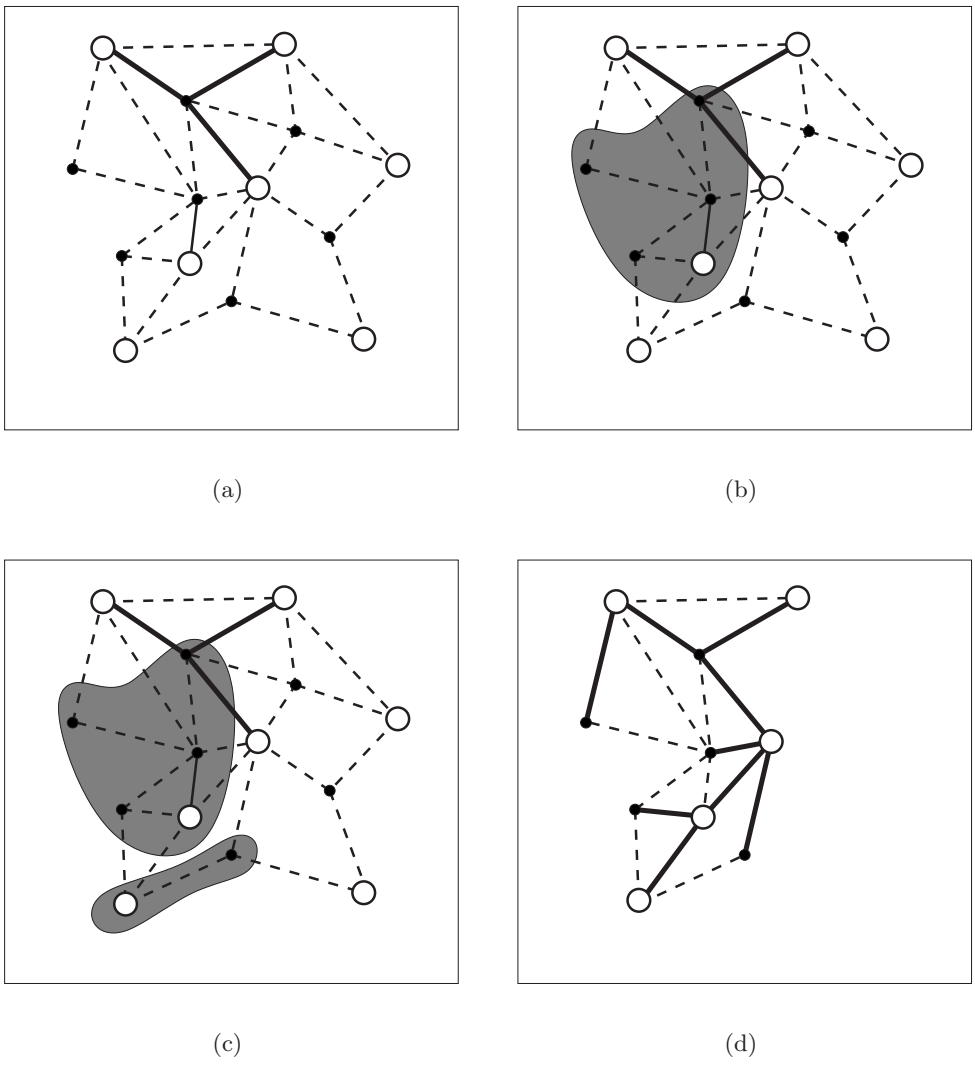


Figure 4.5: Mutation operator: (a) an input feasible solution T (bold lines represent the solution, dashed lines represent the rest of the input graph); (b) first, a border vertex and a cluster assigned to it are selected; (c) a neighboring cluster is selected; (d) from a subgraph induced by the vertices of the initial solution together with the vertices of two selected clusters, the minimum spanning tree (bold lines) is returned.

4.3.6 Local Improvement

Given a solution of the PCST, a tree T, the neighborhood of T is defined as the set of trees that can be reached by applying small modifications to T. Local improvement is performed to yield trees (intermediate and final) that are optimal or near optimal in the local context, i.e. there are no better trees in the neighborhood.

The local improvement procedure is applied after the recombination and mutation operators. Solutions obtained by means of the initialization procedure are also locally improved and then inserted into the population. The method we describe below is based on consecutive deletion of edges from the tree, in a bottom-up order. The algorithm is based on the "first improvement" principle.

Edge-Deletion Local Improvement

The algorithm described in this section solves the net worth *maximization* problem to optimality, if the input graph is a tree. A closely related dynamic programming algorithm can be found in [160] (where trees with node-weights only are considered). Note also that this algorithm has been mentioned by Johnson et al. [84] where it is called *strong pruning*, and is used within the second phase of the Goemans-Williamson algorithm.

We first consider the rooted PCST problem, where the maximization of the profit is requested, i.e. the objective function is given by (4.1). If $T = (V_T, E_T)$ is a tree with root r, then the function parent(v) assigns every vertex $v \in V_T \setminus \{r\}$ a unique vertex u which is the vertex following v on the path from v to r. The *subtree* rooted at v consists of all vertices and edges reachable from v without passing vertex parent(v). The set $C(v)$ of *children* of v is the set that contains all vertices u with parent$(u) = v$. A subtree of T is *optimal*, if there is no other subtree of T with a higher profit. To every vertex $v \in V_T$ we assign a label $l(v)$ and a subtree $T(v)$. The labels $l(v)$ are recursively defined as follows:

$$l(v) = p(v) + \sum_{u \in C(v)} \max\{0, l(u) - c(u,v)\} \ . \tag{4.6}$$

Subtree $T(v) = (V(v), E(v))$ has profit $l(v)$ and is given by:

$$\begin{aligned} V(v) &= \{v\} \cup \bigcup_{u \in C(v)} \{V(u) \mid l(u) - c(u,v) \geq 0\} \ , \\ E(v) &= \bigcup_{u \in C(v)} \{(u,v) \cup E(u) \mid l(u) - c(u,v) \geq 0\} \ . \end{aligned}$$

If $c(u,v) > l(u)$ for a vertex u with parent$(u) = v$ it does not pay off to include the subtree rooted at u via edge (u,v) (the only possible connection towards r), and we decide to cut off the edge (u,v) together with the corresponding subtree. This decision can be made locally, as soon as the value $l(u)$ is known. Thus, the algorithm labels the vertices in a bottom-up order, by starting with the leaves and ending at the root vertex r (see Algorithm 11 for an outline).

It is easy to see that the optimal subtree rooted at v is $T(v)$ with $l(v)$ as its profit (the correctness of this algorithm follows easily by induction). In the case when $l(u) - c(u,v) = 0$

we can decide to include or exclude the edge (u, v) from the solution, depending if our goal is to obtain the optimal subtree with the maximum or minimum number of vertices and edges.

Data : A tree $T = (V_T, E_T)$ with a fixed root $r \in V_T$, a profit function p on the vertices and a costs function c on the edges.
Result: For each $v \in V_T$ an optimal subtree $T(v) = (V(v), E(v))$ and a label $l(v) = \mathit{profit}(T(v))$.

$G^* = (V^*, E^*) = T$;
for $v \in V_T$ **do**
  $l(v) = p(v)$; $V(v) = \{v\}$; $E(v) = \emptyset$;
end
repeat
  $L = \{v \in V_T \setminus \{r\} \mid \deg_{G^*}(v) = 1\}$;
  for $v \in L$ **do**
    $u = \text{parent}(v)$;
    $l(u) = l(u) + \max\{0, l(v) - c(u, v)\}$;
    if $l(v) \geq c(u, v)$ **then**
      $V(u) = V(u) \cup V(v)$;
      $E(u) = E(u) \cup \{(u, v)\} \cup E(v)$;
    end
  end
  Remove the vertices of L from G^*;
until $V^* = \{r\}$;

Algorithm 11: Edge-Deletion Local Improvement Algorithm

The edge-deletion local improvement algorithm solves the rooted PCST on trees in $O(|V_T|)$ time. For solving the unrooted PCST, we only need to find the vertex v^* such that:

$$v^* = \arg\max_{v \in V_T} l(v) \ .$$

The corresponding subtree $T(v^*)$ represents the optimal solution, and can also be found in linear time.

4.3.7 Computational Results

In this section we provide computational results for the memetic algorithm proposed above on the set of instances already introduced in Section 4.2.1. We compare results of the memetic algorithm to those of Canuto et al. (denoted by CRR) obtained using multi-start local search with perturbations and variable neighborhood search [23].

The following setup was used for the memetic algorithm as it proved to be robust in preliminary tests: Population size $|P| = 800$; group size for tournament selection $k = 5$; parameter for initializing solutions $p_{init} = 0.9$; mutation probability $p_{\text{mut}} = 0.3$. Each run was terminated when no new best solution could be identified during the last $\Omega = 10\,000$ iterations.

Table 4.5: Comparing the performance of Canuto et al. [23] (CRR) against our memetic algorithm (MA) on the set of Johnson et al.'s [84] instances. Running times from Canuto et al. to be divided by 10 for comparison (cf. SPEC comparison).

Instance	*OPT*	CRR		MA				
		%-*gap*	*t* [*s*]	%-*gap*	σ	*t* [s]	*evals*	*sr* [%]
P100	803300	0.0	15	0.0	0.0	2.7	826	100.0
P100.1	926238	0.0	14	0.0	0.0	3.3	529	100.0
P100.2	401641	0.0	5	0.0	0.0	2.4	2708	100.0
P100.3	659644	0.0	10	0.0	0.0	2.8	629	100.0
P100.4	827419	0.0	10	0.0	0.0	2.3	560	100.0
P200	1317874	0.0	72	0.0	0.0	7.2	4596	100.0
P400	2459904	0.0	397	0.2	0.2	29.3	20453	23.3
P400.1	2808440	0.0	382	0.0	0.0	19.3	5238	33.3
P400.2	2518577	0.0	396	0.0	0.0	17.8	6208	100.0
P400.3	2951725	0.0	500	0.0	0.0	23.1	12358	0.0
P400.4	2852956	0.0	565	0.5	0.1	21.4	10912	0.0
K100	135511	0.0	2	0.0	0.0	1.7	500	100.0
K100.1	124108	0.0	2	0.0	0.0	1.6	500	100.0
K100.2	200262	0.0	3	0.0	0.0	1.6	500	100.0
K100.3	115953	0.0	3	0.0	0.0	1.5	500	100.0
K100.4	87498	0.0	6	0.0	0.0	1.9	500	100.0
K100.5	119078	0.0	2	0.0	0.0	1.4	500	100.0
K100.6	132886	0.0	2	0.0	0.0	1.2	500	100.0
K100.7	172457	0.0	2	0.0	0.0	1.6	500	100.0
K100.8	210869	0.0	2	2.3	0.0	1.5	500	0.0
K100.9	122917	0.0	2	0.0	0.0	1.4	500	100.0
K100.10	133567	0.0	1	0.0	0.0	1.4	500	100.0
K200	329211	0.0	9	0.0	0.0	2.4	500	100.0
K400	350093	0.0	68	0.0	0.0	6.9	500	100.0
K400.1	490771	0.0	194	0.0	0.0	6.7	500	100.0
K400.2	477073	0.2	234	0.2	0.0	7.1	1050	0.0
K400.3	415328	0.0	140	0.0	0.0	7.4	500	100.0
K400.4	389451	0.0	204	0.1	0.0	7.7	5793	26.7
K400.5	519526	0.0	122	0.3	0.0	7.1	574	0.0
K400.6	374849	0.0	60	0.0	0.0	8.3	523	100.0
K400.7	474466	0.1	306	0.1	0.2	7.7	1017	90.0
K400.8	418614	0.0	42	0.0	0.0	6.7	500	100.0
K400.9	383105	0.0	76	0.1	0.2	7.5	5153	0.0
K400.10	394191	0.4	231	0.7	0.2	8.7	3080	0.0

Tables 4.5-4.7 contain the results of our comparison: For each instance, we show the optimal value OPT, obtained by lower bounding procedure described by Lucena and Resende [111], or alternatively by our new branch-and-cut algorithm (see Section 4.4). Solution values of CRR, $v(T)$, or alternatively, percentage gaps for solution values of CRR, %-*gap*, are given in Tables 4.6-4.7 and in Table 4.5, respectively. The total running time in seconds (t [s]) for CRR is also provided. Given solution value $v(T)$, the percentage gap (%-*gap*) is calculated according to the following formula:

$$\%\text{-}gap = \frac{v(T) - OPT}{OPT} \times 100\% \ .$$

Recall that the values $v(T)$ represent the Goemans-Williamson minimization objective function.

Because of its stochastic nature, the MA was performed 30 times on each instance and the average results are presented in Tables 4.5-4.7. In Table 4.5 the average percentage gap, %-*gap*, and its standard deviation $\sigma(c)$ are given. Tables 4.6-4.7 provide the average solution values $v(T)_{avg}$ and their standard deviation σ_v. Furthermore, we show the average CPU-time and the average number of evaluated solutions until the best solution was found (t [s], respectively *evals*), and the success rates (sr), i.e. the percentage of instances for which optimal solutions could be found. Note that when presenting t [s] values, the preprocessing times are also taken into account.

When comparing our running time data (achieved on a Pentium IV with 2.8 GHz, 2 GB RAM, SPECint2000=1204) with the results of Canuto et al. [23] (Pentium II with 400 MHz, 64 MB RAM), the widely used SPEC© performance evaluation (`www.spec.org`) does not provide a direct scaling factor. However, taking a comparison to the respective benchmark machines both for SPEC 95 and SPEC 2000 into account, we can argue by a conservative estimate that dividing the Canuto et al. running times by a factor of 10 gives a very reasonable basis of comparison to our data.

Figure 4.6 summarizes our comparison results over all benchmark instances used in [23]. The computational results indicate that our memetic algorithm combined with preprocessing is always faster than CRR: for groups `K` and `P` the MA is on average almost 2 times faster, for group `C` this factor is 5, while for the group `D` the MA is almost 14 times faster. On the other side, the average solution quality of the MA is slightly worse than that obtained by CRR. The CRR algorithm found optimal solution for almost all instances of `K,P` and `C` groups, while the average percentage gap was 0.4% for group `D`. Our memetic algorithm performed slightly worse, but the average percentage gap per group was about 1% in the worst case (groups `C` and `D`).

We conclude that the MA is competitive against the heuristic approach proposed by Canuto, Resende and Ribeiro. Although the obtained MA values are not always optimal, the average gap and its standard deviation indicate a stable performance and the reliability of the memetic algorithm.

4.3.8 Performance Analysis of Variation Operators

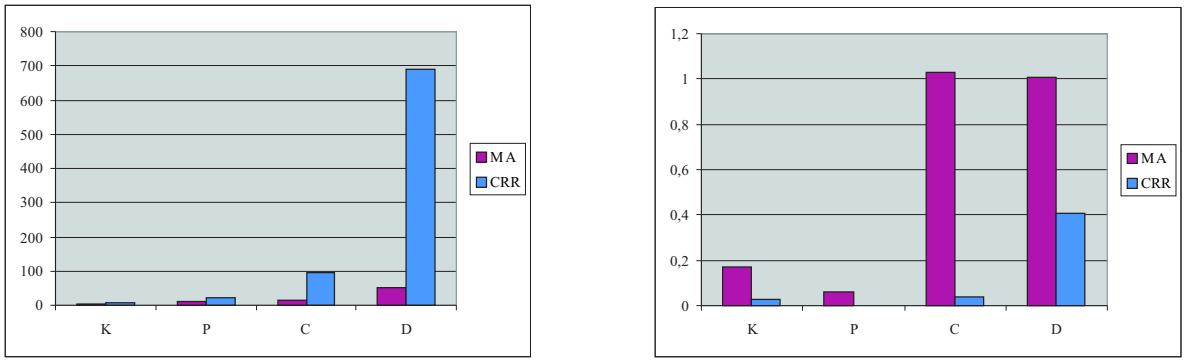
In this section we investigate the role of variation operators and the local improvement procedure introduced in Sections 4.3.4, 4.3.5 and 4.3.6, respectively. Our analysis is based on the

Table 4.6: Comparing the performance of Canuto et al. [23] (CRR) against our memetic algorithm (MA) on the set of instances derived from OR-Library. Running times from Canuto et al. to be divided by 10 for comparison (cf. SPEC comparison).

Instance	OPT	CRR		MA				
		$v(T)$	t [s]	$v(T)_{avg}$	σ_v	t [s]	*evals*	*sr* [%]
C1-A	18	18.0	3	18.0	0.0	1.9	500	100.0
C1-B	85	85.0	58	85.0	0.0	2.3	590	100.0
C2-A	50	50.0	7	50.0	0.0	1.8	500	100.0
C2-B	141	141.0	54	141.0	0.0	2.2	500	100.0
C3-A	414	414.0	87	414.0	0.0	2.9	500	100.0
C3-B	737	737.0	294	737.0	0.0	10.0	1569	100.0
C4-A	618	618.0	148	618.0	0.0	4.5	590	100.0
C4-B	1063	1063.0	387	1066.9	0.8	39.0	22176	0.0
C5-A	1080	1080.0	447	1080.0	0.0	9.4	730	100.0
C5-B	1528	1528.0	397	1528.1	0.3	20.1	4245	86.7
C6-A	18	18.0	9	18.0	0.0	4.1	500	100.0
C6-B	55	55.0	179	55.9	0.3	5.6	1359	6.7
C7-A	50	50.0	34	50.0	0.0	3.5	500	100.0
C7-B	102	103.0	167	106.0	0.0	5.7	1321	0.0
C8-A	361	361.0	313	362.7	0.7	8.5	1622	13.3
C8-B	500	500.0	404	502.0	1.1	29.0	16493	0.0
C9-A	533	533.0	475	533.1	0.3	13.4	3419	93.3
C9-B	694	694.0	583	699.3	2.4	38.5	16426	0.0
C10-A	859	859.0	628	860.3	0.9	35.4	12753	20.0
C10-B	1069	1069.0	474	1070.0	0.8	39.4	12692	33.3
C11-A	18	18.0	128	18.0	0.0	6.1	500	100.0
C11-B	32	32.0	140	32.0	0.0	9.1	1103	100.0
C12-A	38	38.0	162	38.7	0.5	9.0	2456	33.3
C12-B	46	46.0	156	46.0	0.0	8.7	590	100.0
C13-A	236	237.0	1050	237.0	0.2	17.9	5326	0.0
C13-B	258	258.0	733	258.5	0.7	35.9	15455	60.0
C14-A	293	293.0	829	293.0	0.0	21.0	3163	100.0
C14-B	318	318.0	766	318.6	0.5	29.8	9211	43.3
C15-A	501	501.0	957	502.2	0.8	45.4	14727	20.0
C15-B	551	551.0	837	551.8	0.9	45.7	15607	46.7
C16-A	11	11.0	1920	12.0	0.0	10.6	500	0.0
C16-B	11	11.0	1758	12.0	0.0	11.5	503	0.0
C17-A	18	18.0	549	19.0	0.0	11.2	620	0.0
C17-B	18	18.0	434	18.2	0.4	12.7	1951	76.7
C18-A	111	111.0	3990	112.4	0.7	24.1	7446	6.7
C18-B	113	113.0	3262	115.0	0.7	26.2	8361	6.7
C19-A	146	146.0	3928	146.2	0.4	17.9	5402	80.0
C19-B	146	146.0	3390	149.0	0.6	15.8	4035	0.0
C20-A	266	266.0	4311	266.0	0.0	7.3	598	100.0
C20-B	267	267.0	3800	267.0	0.0	5.2	500	100.0

Table 4.7: Comparing the performance of Canuto et al. [23] (CRR) against our memetic algorithm (MA) on the set of instances derived from OR-Library. Running times from Canuto et al. to be divided by 10 for comparison (cf. SPEC comparison).

Instance	*OPT*	CRR		MA				
		$v(T)$	t [s]	$v(T)_{avg}$	σ_v	t [s]	*evals*	*sr* [%]
D1-A	18	18.0	6	18.0	0.0	3.1	500	100.0
D1-B	106	106.0	257	106.0	0.0	3.8	1950	100.0
D2-A	50	50.0	7	50.0	0.0	3.5	500	100.0
D2-B	218	228.0	486	218.3	1.0	7.3	4157	93.3
D3-A	807	807.0	734	807.0	0.0	7.4	500	100.0
D3-B	1509	1510.0	2184	1516.2	1.3	51.0	15976	0.0
D4-A	1203	1203.0	1263	1203.8	0.4	10.4	974	16.7
D4-B	1881	1881.0	2233	1885.2	2.0	49.6	9671	0.0
D5-A	2157	2157.0	3352	2157.0	0.0	29.1	1963	100.0
D5-B	3135	3135.0	2555	3137.7	0.9	65.1	7316	0.0
D6-A	18	18.0	20	18.0	0.0	7.7	500	100.0
D6-B	67	70.0	702	72.6	0.8	10.5	1192	0.0
D7-A	50	50.0	195	50.0	0.0	8.2	500	100.0
D7-B	103	105.0	711	105.0	0.0	9.5	520	0.0
D8-A	755	755.0	1727	755.5	0.5	19.1	2788	50.0
D8-B	1036	1038.0	3175	1045.7	3.9	123.8	36313	0.0
D9-A	1070	1072.0	4109	1074.7	1.0	52.1	13718	0.0
D9-B	1420	1420.0	2754	1436.4	3.0	151.2	31361	0.0
D10-A	1671	1671.0	4193	1674.4	1.4	122.2	21289	0.0
D10-B	2079	2079.0	2644	2089.8	2.1	107.3	14598	0.0
D11-A	18	18.0	540	18.0	0.0	15.4	500	100.0
D11-B	29	30.0	1280	29.0	0.0	17.4	814	100.0
D12-A	42	42.0	844	42.0	0.0	13.9	500	100.0
D12-B	42	42.0	687	42.0	0.0	15.1	620	100.0
D13-A	445	445.0	5047	446.7	0.5	58.7	14308	0.0
D13-B	486	486.0	4288	491.7	1.9	97.2	22843	0.0
D14-A	602	602.0	6388	605.6	1.2	102.3	21486	0.0
D14-B	665	665.0	6178	674.2	1.4	102.8	17746	0.0
D15-A	1042	1042.0	7840	1048.7	1.3	145.7	18343	0.0
D15-B	1108	1108.0	5220	1114.7	0.8	95.6	11026	0.0
D16-A	13	13.0	1397	14.0	0.0	23.1	500	0.0
D16-B	13	13.0	1043	13.3	0.4	26.4	1313	73.3
D17-A	23	23.0	3506	23.0	0.0	24.8	1983	100.0
D17-B	23	23.0	2089	23.0	0.0	23.7	948	100.0
D18-A	218	218.0	30044	220.8	0.7	81.4	19864	0.0
D18-B	223	224.0	36643	230.2	1.3	98.7	25585	0.0
D19-A	306	308.0	40955	317.7	2.7	87.6	18480	0.0
D19-B	310	311.0	38600	317.8	2.2	81.9	17912	0.0
D20-A	536	536.0	28139	537.0	0.0	18.4	1036	0.0
D20-B	537	537.0	22104	537.0	0.0	12.7	1587	100.0



(a) (b)

Figure 4.6: Comparing our memetic algorithm (MA) with the Canuto et al. [23] approach: (a) The running time in seconds averaged per group - running times of Canuto et al. are divided by 10 for comparison (see text) (b) The percentage gap %-*gap* averaged per group.

Table 4.8: Average performance over 30 runs of different MA-variants, for `K`, `P`, `C` and `D` groups of PCST instances.

Grp.	MA					C+LI					M+LI					C+M+LI				
	%-*gap*	σ	t [s]	*evals*	*sr*	%-*gap*	σ	t [s]	*evals*	*sr*	%-*gap*	σ	t [s]	*evals*	*sr*	%-*gap*	σ	t [s]	*evals*	*sr*
K	0.2	0.0	4	1095	74.6	0.2	< 0.1	4	592	69.1	0.2	< 0.1	4	907	70.1	0.3	< 0.1	4	727	70.3
P	0.1	0.0	12	5910	68.8	0.3	< 0.1	10	5076	46.1	0.3	0.1	12	7478	27.3	0.6	0.1	6	3040	19.1
C	1.0	0.2	16	4926	55.7	2.2	0.1	17	6222	41.7	3.9	0.2	18	4264	24.6	2.4	0.2	11	1313	28.8
D	1.0	0.3	50	9092	40.8	1.9	0.3	61	11582	27.4	3.7	0.9	65	9479	20.2	3.5	0.2	37	1697	18.2

results obtained on the set of benchmark instances mentioned in previous section.

Table 4.8 illustrates the importance of using both, recombination and mutation operators, and that it is necessary to apply local improvement immediately after each variation operator. Shown are average results of 30 runs for the following four scenarios of the memetic algorithm: In MA, our default implementation, local improvement is applied after both variation operators. In C+LI, new candidate solutions are created only by recombination followed by local improvement. M+LI applies always only mutation followed by local improvement. In C+M+LI, recombination and mutation are used, and local improvement is performed before a solution is inserted into the population. All strategy parameters were set as in the previous experiments with the only exception that in M+LI, the probability of applying mutation was $p_{\text{mut}} = 1$.

The best performance is obtained when candidate solutions are locally improved after both, recombination and variation: the average percentage gap over all instances is 0.6% in that case and in Tables 4.5-4.7 we have seen detailed results of this default scenario.

Comparing the other three scenarios, we can observe:

- C+M+LI converged fastest, but the obtained solutions were in nearly all cases substantially poorer (1.7% of average gap over all instances). This points out that mutating of candidate solutions that are not already locally optimal, leads to the premature convergence of the underlying memetic algorithm. In other words, once the solution is being mutated, due to the deterministic nature of the local improvement algorithm, the probability of getting stuck into poor suboptimal solutions is getting higher.

- C+LI, on the other side, generally needed much more evaluations and also more time to converge. Although its total running time hardly deviates from the running time of our default MA implementation, the average gap obtained over all instances was 1.2%. This shows that incorporation of the mutation operator is needed in order to preserve diversity of the memetic algorithm.

- Finally, the worst results were obtained by running M+LI, with 2% of average gap, which clearly indicates that the recombination of genetic material represents an essential part of our memetic algorithm.

4.4 ILP Formulations of the Problem

In this section we provide a review of different ILP formulations used in literature for solving the PCST to optimality. We also study their relationships and provide new sets of strengthening inequalities.

First, we show an ILP formulation based on generalized subtour elimination constraints and two flow formulations based on the representation of the solutions as rooted trees. Finally, we concentrate on a connectivity-based formulation which represents the basis of our branch-and-cut algorithm presented in Section 4.5.

Within this section, for every integer programming formulation (P) the optimal solution value of the resulting LP-relaxation will be denoted by $c(LP_P)$.

4.4.1 Formulation Based on Generalized Subtour Elimination Constraints

This ILP formulation has been used in Lucena and Resende [111], and follows from an extended formulation of the Steiner problem in graphs studied also by Goemans [65] and Margot et al. [112].

To every subtree $T' = (V'_T, E'_T)$ of the input graph $G' = (V', E', c', p')$, we associate two incidence vectors:

$$X_{ij} = \begin{cases} 1 & (i,j) \in E'_T \\ 0 & \text{otherwise} \end{cases} \forall (i,j) \in E', \quad y_i = \begin{cases} 1 & i \in V'_T \\ 0 & \text{otherwise} \end{cases} \forall i \in V'$$

The PCST can be formulated as the following ILP (the edge-variables are denoted by a capital X to distinguish them from the variables for the directed formulations we will introduce later):

$$(GSEC) \quad \min \quad \sum_{ij \in E'} c'_{ij} X_{ij} + \sum_{i \in V'} p'_i (1 - y_i) \tag{4.7}$$

$$\text{subject to} \quad \sum_{ij \in E'} X_{ij} = \sum_{i \in V'} y_i - 1 \tag{4.8}$$

$$\sum_{i,j \in S} X_{ij} \leq \sum_{i \in S \setminus \{k\}} y_i \qquad \forall S \subseteq V', |S| \geq 2, \forall k \in S \tag{4.9}$$

$$0 \leq X_{ij} \leq 1 \qquad \forall (i,j) \in E' \tag{4.10}$$

$$0 \leq y_i \leq 1 \qquad \forall i \in V' \tag{4.11}$$

$$y_i \in \{0,1\} \qquad \forall i \in V' \tag{4.12}$$

Constraints (4.8) and (4.9) describe the tree structure of the solution. Constraint (4.8) excludes the empty tree from the set of feasible solutions. Constraints (4.9) are called *generalized subtour elimination* constraints. They guarantee that the solution is cycle free. Note that if $y_i = 1, \forall i \in S$, then (4.9) reduces to a classical subtour elimination constraint (SEC) as that for the TSP. The validity of the provided formulation follows from Edmonds' characterization of the spanning tree polytope (see, for example, [42]).

4.4.2 Rooted Tree Flow-Formulations

Directed tree formulations rely on a transformation of the PCST to the problem of finding a minimum subgraph in a related, directed graph as proposed by Fischetti [51]. We transform the reduced graph $G' = (V', E', c', p')$ that results from the application of preprocessing into the directed edge-weighted graph $G_{\mathrm{SA}} = (V_{\mathrm{SA}}, A_{\mathrm{SA}}, c'')$, and solve the so-called *Steiner arborescence problem* described below. See [14] for a detailed introduction into digraphs.

The Steiner Arborescence Problem

Suppose we are given a graph $G_r = (V_r, E_r)$, with a distinguished root vertex r. Suppose also that a (possibly empty) set of *terminal* vertices (also called *target* vertices) $T \subset V_r \setminus \{r\}$ is given. In the jargon of the Steiner tree problem of graphs, the vertices $V_r \setminus (T \cup \{r\})$ are called *Steiner vertices*. We relate to the graph G_r a digraph $G'_r = (V_r, A_r)$ whose set of arcs is given as:

$$A_r = \{(i,j) \mid i \in V_r, j \in V_r \setminus \{i, r\}\} \ .$$

Definition 10. [Steiner Arborescence]
Given a digraph $G'_r = (V_r, A_r)$, a Steiner Arborescence (SA) is a partial digraph $G'_{\mathrm{SA}} = (V'_r, A'_r)$ of G'_r such that:

- *all the* target *vertices $i \in T$ have in-degree equal to one in G'_{SA},*
- *all Steiner vertices have in-degree ≤ 1. Note that the construction of the digraph G_{SA} assures that the root vertex r has in-degree equal to 0.*
- *for each vertex $i \in V_r \setminus \{r\}$ whose in-degree is equal to one, there is a directed path from the root r to i.*

With each partial digraph $G'_{\mathrm{SA}} = (V'_r, A'_r)$ of G'_r, a *characteristic vector* $x \in \{0,1\}^{|A_r|}$ is associated:

$$x_{ij} = \begin{cases} 1 & (i,j) \in A'_r \\ 0 & \text{otherwise} \end{cases} \quad \forall (i,j) \in A_r \ .$$

Definition 11. The Steiner Arborescence Problem
The Steiner arborescence polytope *is defined as a convex hull of the characteristic vectors of all Steiner arborescences of G'_r. The* Steiner arborescence problem *consists of minimizing a linear objective function $\sum_{ij \in A_r} c_{ij} x_{ij}$ over the SA polyhedron.*

Note that if the set of terminal vertices is empty, minimization over the SA polyhedron only makes sense if there are negative edge-costs; otherwise the optimal solution contains only the root vertex. Fischetti [51] studied the Steiner arborescence problem and related polyhedra. He also showed that the node-weighted Steiner tree problem can be transformed into it. In what follows, we will show how the PCST problem can be transformed into the Steiner arborescence problem.

Transformation of PCST into the Steiner Arborescence Problem

Suppose we are given an instance of the reduced PCST problem $G' = (V', E', c', p')$. We transform the graph G' into a Steiner arborescence graph $G_{\text{SA}} = (V_{\text{SA}}, A_{\text{SA}}, c'')$ in the following way:

- In addition to the vertices of the input graph G', the vertex set of the transformed graph contains an artificial root r, i.e. $V_{\text{SA}} = V' \cup \{r\}$.
- The arc set A_{SA} contains two directed edges (i, j) and (j, i) for each edge $(i, j) \in E'$ plus a set of arcs from the root r to the customer vertices $R' = \{i \in V' \mid p_i > 0\}$, i.e.

$$A = \{(i, j) \mid i \in V, j \in V \setminus \{i, r\}\} \cup \{(r, i) \mid i \in R'\} \ .$$

- The set of target vertices is empty.
- We define the cost vector c'' as follows:

$$c''_{ij} = \begin{cases} c'_{ij} - p'_j & \forall (i, j) \in A_{\text{SA}},\ i \neq r \\ -p'_j & \forall (r, j) \in A_{\text{SA}} \ . \end{cases}$$

A *Steiner arborescence* G'_{SA} of G_{SA} is called a *feasible arborescence* if G'_{SA} corresponds to a solution of the PCST in which r has degree 1. In particular, a feasible arborescence with minimal total edge costs corresponds to an optimal prize-collecting Steiner tree.

It is easy to see that the cost of the corresponding PCST solution in G' can be found as the sum of the arc-costs of G'_{SA} plus the sum of all vertex weights in G'.

In a Steiner arborescence graph G_{SA} we will also consider customer vertices whose set R_{SA} is defined as $R_{\text{SA}} = \{i \in V_{\text{SA}} \mid i \neq r, p_i > 0\}$. Note that the non-customer vertices $v \in V_{\text{SA}} \setminus R_{\text{SA}}, v \neq r$ cannot be leaves of an optimal solution.

An example of this transformation can be found in Figures 4.8 (a) and (b) on page 126.

We model the problem of finding a minimum Steiner arborescence $G'_{\text{SA}} = (V'_{\text{SA}}, A'_{\text{SA}})$ by means of an integer linear program. Therefore, we introduce a variable vector $x \in \{0, 1\}^{|A_{\text{SA}}|}$ with the following interpretation:

$$x_{ij} = \begin{cases} 1 & (i, j) \in A'_{\text{SA}} \\ 0 & \text{otherwise} \end{cases} \quad \forall (i, j) \in A_{\text{SA}} \ .$$

The small letters x indicate arc variables in the directed model whereas capital letters X were used in Section 4.4.1 for edges in the undirected case. Furthermore, to indicate which of the vertices from $V_{\text{SA}} \setminus \{r\}$ belong to the solution, we use a variable vector $y \in \{0, 1\}^{|V_{\text{SA}}|-1}$:

$$y_i = \begin{cases} 1 & i \in V'_{\text{SA}} \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in V_{\text{SA}} \setminus \{r\} \ .$$

In the sequel, we provide three different ILP formulations of the PCST problem on Steiner arborescence graph.

Single-Commodity Flow Formulation

This is one of the simplest ILP formulations of the problem. Segev [149] used a very similar formulation for the single vertex-weighted Steiner tree problem, under the name *tree-type formulation*. A related formulation for the Steiner tree problem has been studied by several authors, see, e.g., [40].

This formulation describes the structure of a rooted arborescence by a flow using the variables f_{ij} for the amount of flow on arc (i,j) for all $(i,j) \in A_{\mathrm{SA}}$. One unit of flow is sent from the root vertex r to every customer vertex in the solution tree and f_{ij} represents the sum of all such flows via arc (i,j). In the optimal solution the values of x_{ij} indicate directed paths from r to every selected vertex. The resulting ILP formulation can be written as follows:

$$(SF) \qquad \min \quad \sum_{ij \in A_{\mathrm{SA}}} c''_{ij} x_{ij} + \sum_{i \in V_{\mathrm{SA}}} p'_i \tag{4.13}$$

$$\text{subject to} \quad \sum_{ji \in A_{\mathrm{SA}}} x_{ji} = y_i \qquad \forall i \in V_{\mathrm{SA}} \setminus \{r\} \tag{4.14}$$

$$\sum_{ji \in A_{\mathrm{SA}}} f_{ji} - \sum_{ij \in A_{\mathrm{SA}}} f_{ij} = y_i \qquad \forall i \in R_{\mathrm{SA}} \tag{4.15}$$

$$\sum_{ji \in A_{\mathrm{SA}}} f_{ji} - \sum_{ij \in A_{\mathrm{SA}}} f_{ij} = 0 \qquad \forall i \in V_{\mathrm{SA}} \setminus R_{\mathrm{SA}}, i \neq r \tag{4.16}$$

$$0 \leq f_{ij} \leq (|V_{\mathrm{SA}}| - 1) \cdot x_{ij} \qquad \forall (i,j) \in A_{\mathrm{SA}} \tag{4.17}$$

$$\sum_{ri \in A_{\mathrm{SA}}} x_{ri} = 1 \tag{4.18}$$

$$y_i, x_{ij} \in \{0,1\} \qquad \forall i \in V_{\mathrm{SA}} \setminus \{r\}, \forall (i,j) \in A_{\mathrm{SA}} \tag{4.19}$$

The constant term in the objective function is added such that (SF) yields the desired overall solution value (4.2). The so-called *in-degree* equation (4.14) guarantees that every selected vertex has exactly one predecessor on its path from the root. The classical flow preservation constraints are given by (4.15) and (4.16), where the former requires that every selected vertex receives one unit of flow to be consumed in this vertex. Constraints (4.17) force the arcs which are used by any flow to be included in the final directed tree of paths to the vertices. Finally, the so-called *root-degree* constraint (4.18) makes sure that the artificial root r is connected only to a single vertex, i.e. that we obtained a feasible arborescence.

Because of the so-called "big M" constraints resembled in (4.17), this formulation is known to be computationally inefficient in the sense that the LP-solution value is likely to deviate considerably from the optimal ILP value. An example of a solution of the LP-relaxation of (SF) can be found in Figure 4.8 (c) on page 126.

Multi-Commodity Flow Formulation

A straightforward extension of the previous formulation is the so-called *multi-commodity flow formulation* of the problem. Here, we split the flow from (SF) into separate commodities for every selected customer vertex. The variable f^k_{ij} describes the flow value of a single commodity

k on arc (i, j) on the directed path from the root vertex r to a selected customer vertex $k \in R_{\mathrm{SA}}$ with $y_k = 1$.

$$
\begin{aligned}
(MCF) \quad \min \quad & \sum_{ij \in A_{\mathrm{SA}}} c''_{ij} x_{ij} + \sum_{i \in V_{\mathrm{SA}}} p'_i && && (4.20)\\
\text{subject to} \quad & \sum_{ji \in A_{\mathrm{SA}}} x_{ji} = y_i && \forall i \in V_{\mathrm{SA}} \setminus \{r\} && (4.21)\\
& \sum_{ji \in A_{\mathrm{SA}}} f^i_{ji} - \sum_{ij \in A_{\mathrm{SA}}} f^i_{ij} = y_i && \forall i \in R_{\mathrm{SA}} && (4.22)\\
& \sum_{ji \in A_{\mathrm{SA}}} f^k_{ji} - \sum_{ij \in A_{\mathrm{SA}}} f^k_{ij} = 0 && \forall i \in V_{\mathrm{SA}} \setminus \{k, r\}, \forall k \in R_{\mathrm{SA}} && (4.23)\\
& 0 \leq f^k_{ij} \leq x_{ij} && \forall (i, j) \in A_{\mathrm{SA}}, \forall k \in R_{\mathrm{SA}} && (4.24)\\
& \sum_{ri \in A_{\mathrm{SA}}} x_{ri} = 1 && && (4.25)\\
& y_i, x_{ij} \in \{0, 1\} && \forall i \in V_{\mathrm{SA}} \setminus \{r\}, \forall (i, j) \in A_{\mathrm{SA}} && (4.26)
\end{aligned}
$$

The meaning of the constraints is almost equivalent to the (SF) formulation. Of course the flow preservation constraints are extended to hold for every single commodity in (4.22) and (4.23). The former selects commodity i as the only possibility to deliver any flow into vertex i. Conditions (4.24) force an arc to be included in the solution tree as soon as the flow of any commodity traverses through. A multi-commodity flow formulation is well known for the standard Steiner tree problem (see e.g. [134]) and it was applied to the SPWST by Segev [149].

It is not surprising that (MCF) strictly dominates the (SF) formulation. This means that every solution of the LP-relaxation of (MCF) can be mapped into an equivalent LP-solution of (SF) by simply merging the commodities on every arc into a single flow. Figure 4.8 (c) and (d) on page 126 shows an example where $c(LP_{MCF}) > c(LP_{SF})$ holds, and so the domination relation between the two formulations is strict.

4.4.3 Cut Formulation

A different ILP formulation (introduced in [51] for the NWST) concentrates on the connectedness of the solution. Therefore, *cuts* are introduced with the fairly simple condition that for every selected vertex which is separated from r by a cut there must be an arc crossing this cut.

The corresponding ILP model then reads as follows:

$$(CUT) \qquad \min \quad \sum_{ij \in A_{\mathrm{SA}}} c''_{ij} x_{ij} + \sum_{i \in V_{\mathrm{SA}}} p'_i \tag{4.27}$$

$$\text{subject to} \quad \sum_{ji \in A_{\mathrm{SA}}} x_{ji} = y_i \qquad \forall i \in V_{\mathrm{SA}} \setminus \{r\} \tag{4.28}$$

$$x(\delta^-(S)) \geq y_k \qquad k \in S, r \notin S, \forall S \subset V_{\mathrm{SA}} \tag{4.29}$$

$$\sum_{ri \in A_{\mathrm{SA}}} x_{ri} = 1 \tag{4.30}$$

$$x_{ij}, y_i \in \{0,1\} \qquad \forall (i,j) \in A_{\mathrm{SA}}, \forall i \in V_{\mathrm{SA}} \setminus \{r\} \tag{4.31}$$

The cut constraints (4.29) are also called *connectivity inequalities*. They guarantee that for each vertex v in the solution, there must be a directed path from r to v. Note that disconnectivity would imply the existence of a cut S separating r and v which would clearly violate the corresponding cut constraint.

As already observed in [51], the connectivity inequalities (4.29) can be put in an LP equivalent form by adding together $-x(\delta^-(S)) \leq -y_k$ and the in-degree equations $\sum_{ji \in A_{\mathrm{SA}}} x_{ji} = y_i$ for all $i \in S$, to produce the generalized subtour elimination constraint $\sum_{i,j \in S} x_{ij} \leq \sum_{i \in S \setminus \{k\}} y_i$, the directed counterpart of (4.9). Chopra and Rao [28] have shown for the Steiner tree problem that directed GSECs dominate directed counterparts of several other facet defining inequalities of the undirected (GSEC) formulation. This is also the reason why the directed formulation is preferable in practice.

The following theorem shows the equivalence of the LP-relaxations of (CUT) and (MCF). Note that this fact has already been observed in a related setting for the classical Steiner tree problem (see for example [40]).

Theorem 6. *The polytopes of the LP-relaxations for (MCF) and (CUT) are identical.*

Proof. Let x_{ij} be feasible for (MCF) and assume that there exist S and k violating (4.29) in (CUT), i.e., $x(\delta^-(S)) < y_k$. Considering (4.22) for the same vertex k, it follows that there is a flow of value y_k from r to k in the directed network defined by the arc capacities x_{ij}. The classical max-flow min-cut theorem implies that every cut separating r and k must have a cut value at least y_k in contradiction to the assumption.

Let x_{ij} be feasible for (CUT). We want to construct a corresponding feasible multi-commodity flow for (MCF). If $y_k = 0$ we simply set $f^k_{ij} = 0$ for all $(i,j) \in A_{\mathrm{SA}}$. For $y_k > 0$ consider the network with source r and sink k', where k' is connected only to k by an arc (k,k') with capacity y_k and capacities x_{ij} for all other arcs $(i,j) \in A_{\mathrm{SA}}$. The maximum flow in this network delivers the flow of commodity k. Its flow value f^k is exactly y_k as required in (4.22). If this maximum flow f^k were smaller than y_k, then by the max-flow min-cut theorem there must exist a minimal cut between r and k with capacity less than y_k thus violating (4.29). Exceeding the flow value of y_k is prevented by the introduction of the artificial vertex k' which cannot receive a larger inflow. The feasibility of all other constraints in (MCF) is obvious. □

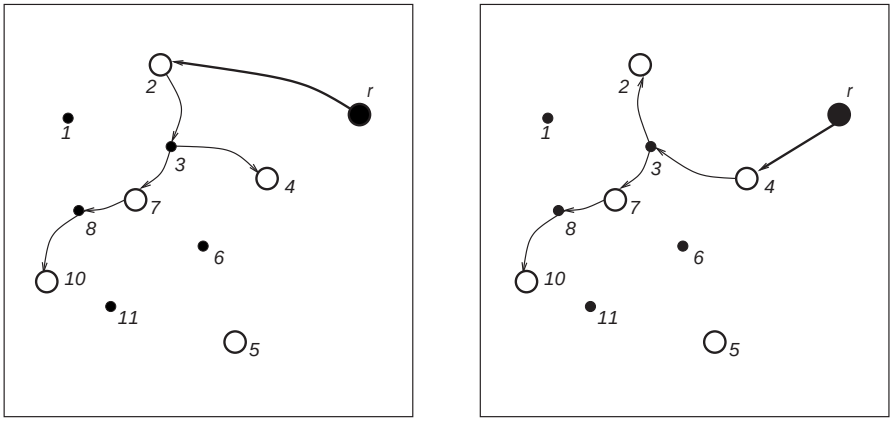


Figure 4.7: Two feasible Steiner arborescences representing the same PCST solution. Using the asymmetry inequalities only the solution on the left-hand side is considered as feasible.

4.4.4 Asymmetry Constraints

In order to create a bijection between arborescence and PCST solutions, we introduce the so-called *asymmetry constraints*:

$$x_{rj} \leq 1 - y_i, \; \forall i < j, \; i \in R \tag{4.32}$$

These inequalities assure that for each PCST solution the customer vertex adjacent to the root is the one with the smallest index. Figure 4.7 illustrates an example. Computational results have shown that they significantly reduce the computation time, because they exclude many symmetric solutions.

4.4.5 Strengthening the Formulation

Each feasible solution of the Steiner arborescence problem can be seen as a set of flows sending one unit from the root to all customer vertices j with $y_j = 1$.

Considering the tree structure of the solution it is obvious that in every non-customer vertex, which is not a branching vertex in the Steiner arborescence, in-degree and out-degree must be equal, whereas in a branching non-customer vertex in-degree is always less than outgoing degree. Thus, we have:

$$\sum_{ji \in A_{\mathrm{SA}}} x_{ji} \leq \sum_{ij \in A_{\mathrm{SA}}} x_{ij} \;\; , \quad \forall i \notin R, \quad i \neq r \;\; . \tag{4.33}$$

These so-called *flow-balance constraints* were introduced by Koch and Martin in [99] for the Steiner tree problem. They indeed represent a strengthening of the LP-relaxation of (4.28)-(4.32), as can be shown by an example in Figures 4.8 (d) and (e). For the classical Steiner tree problem, an analogous example can be found in [134].

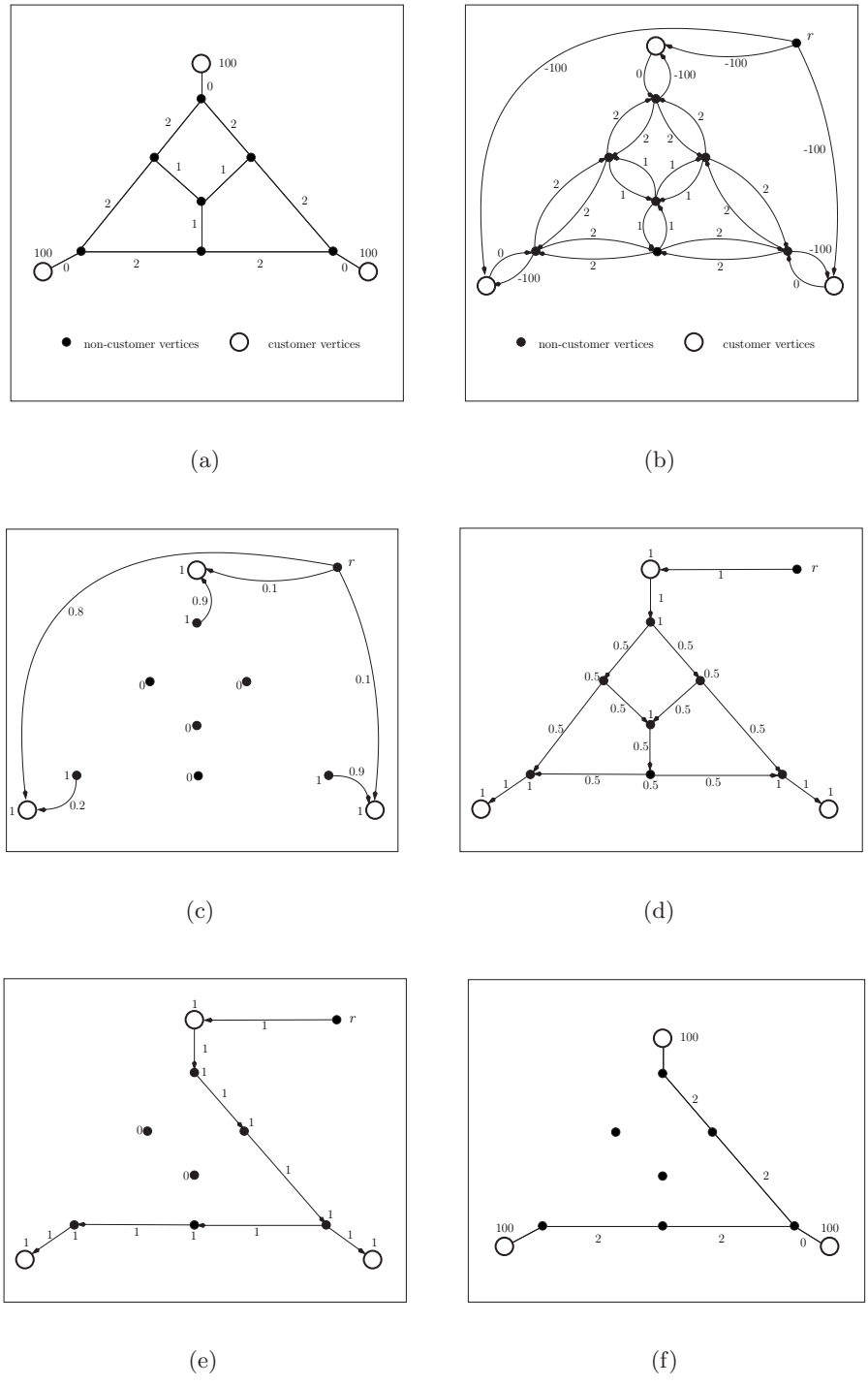


Figure 4.8: (a) an input graph G; (b) after transformation into the Steiner arborescence problem; (c) solution of (SF) LP-relaxation, $c(LP_{SF}) = 0$. LP-values of x and y variables are shown; (d) solution of (MCF) LP-relaxation, $c(LP_{MCF}) = c(LP_{CUT}) = 7.5$; (e) solution of (CUT) LP-relaxation augmented with flow-balance constraints has cost 8 and corresponds to the optimal solution (f).

4.5 Branch-and-Cut Algorithm

In the last section we gave an overview of the existing ILP formulations for the prize-collecting Steiner tree problem. Within this section, we describe an implementation of the branch-and-cut algorithm to solve the (CUT) formulation of the problem. The branch-and-cut algorithm is designed as follows: At each node of the branch-and-bound tree we solve the LP-relaxation (CUT), obtained by replacing the integrality requirements (4.31) by the simple bounds: $0 \leq y_i \leq 1, \forall i \in V_{\mathrm{SA}} \setminus \{r\}$ and $0 \leq x_{ij} \leq 1, \forall (i,j) \in A_{\mathrm{SA}}$. For solving the LP-relaxations and as a generic implementation of the branch-and-cut approach, we used the commercial packages ILOG CPLEX and ILOG Concert Technology (version 8.1).

4.5.1 Initialization

There are exponentially many constraints of type (4.29), so we do not insert them at the beginning but rather *separate* them during the optimization process using the separation procedure described below.

At the root node of the branch-and-bound tree, we start with in-degree, root-degree, flow-balance and asymmetry constraints. Furthermore, we add the following group of inequalities:

$$x_{ij} + x_{ji} \leq y_i, \quad \forall i \in V_{\mathrm{SA}} \setminus \{r\}, \quad (i,j) \in A_{\mathrm{SA}} \tag{4.34}$$

These constraints express the trivial fact that every arc adjacent to a vertex in the solution tree can be oriented only in one way. They are also a special case of the connectivity constraints written in their equivalent GSEC form, a directed counterpart of (4.9), with $S := \{i, j\}$. Although the LP may become large by adding all of these inequalities at once they offer a tremendous speedup since they do not have to be separated implicitly during the branch-and-cut algorithm. Further details are discussed in Section 4.5.4. An example that illustrates an advantage of such initialization is given in Figure 4.10.

4.5.2 Separation

During the separation phase which is applied at each node of the branch-and-bound tree, we add constraints of type (4.29) that are violated by the current solution of the LP-relaxation. Usually, this model is less dense than the equivalent directed (GSEC) model, so it may be computationally preferable within the branch-and-cut implementation.

These violated cut constraints can be found in polynomial time using a maximum flow algorithm on the *support graph* with arc-capacities given by the current solution. For finding the maximum flow in a directed graph, we used an adaptation of Cherkassky & Goldberg's maximum flow algorithm [26][3].

The outline of the separation procedure is given in Algorithm 12. Given a support graph $G_s = (V_{\mathrm{SA}}, A_{\mathrm{SA}}, x)$, we search for violated inequalities by calculating the maximum flow for all (r, i) pairs of vertices, $i \in R_{\mathrm{SA}}, y_i > 0$. The maximum flow algorithm $MaxFlow(G, x', r, i, S_r, S_i)$ returns the flow value f and two sets of vertices:

[3] Available at `http://www.avglab.com/andrew/CATS/maxflow_solvers.htm`

Data : A support graph $G_s = (V_{\text{SA}}, A_{\text{SA}}, x)$.
Result: A set of violated inequalities incorporated into the current LP.

for $i \in R_{SA}, y_i > 0$ **do**
  $x' = x + EPS$;
  repeat
    $f = MaxFlow(G, x', r, i, S_r, S_i)$;
    Detect the cut $\delta^+(S_r)$ such that $x'(\delta^+(S_r)) = f$, $r \in S_r$;
    if $f < y_i$ **then**
      Insert the violated cut $x(\delta^+(S_r)) \geq y_i$ into the LP;
    end
    $x'_{ij} = 1, \forall (i,j) \in \delta^+(S_r)$;
    if *BACKCUTS* **then**
      Detect the cut $\delta^-(S_i)$ such that $x'(\delta^-(S_i)) = f$, $i \in S_i$;
      if $S_i \neq \overline{S_r}$ **then**
        Insert the violated cut $x(\delta^-(S_i)) \geq y_i$ into the LP;
        $x'_{ij} = 1, \forall (i,j) \in \delta^-(S_i)$;
      end
    end
  until $f \geq y_i$ *or MAXCUTS constraints added*;
end

Algorithm 12: Separation procedure.

- Subset $S_r \subset V_{\mathrm{SA}}$ contains root vertex r and induces a minimum cut closest to r, in other words, $x(\delta^+(S_r)) = f$;

- Subset $S_i \subset V_{\mathrm{SA}}$ contains vertex i and induces a minimum cut closest to i, i.e., $x(\delta^-(S_i)) = f$.

If $f < y_i$, we insert the violated cut $x(\delta^+(S_r)) \geq y_i$ into the LP. We then follow the idea of the so-called *nested cuts* [99]: we iteratively add further violated constraints induced by the minimum (r, i)-cut in the support graph in which the capacities of all the arcs $(u, v) \in \delta^+(S_r)$ are set to one. This iterative process is done as long as the total number of the detected violated cuts is less than *MAXCUTS* (100, in the default implementation), or there are no more such cuts. By setting the capacities of the edges in a cut to one, we are able to increase the number of violated inequalities found within one cutting plane iteration. To deal with numerical instabilities of the LP solver, we inserted only those cuts that are violated by at least some small value (10^{-4}, in our default implementation).

Chopra et al. [27] proposed the so-called *back-cuts*, also used in [99], for the Steiner tree problem. To speed up the process of detecting more violated cuts within the same separation phase, we consider the reversal flow in order to find the cut "closest" to i, for some $i \in R, y_i > 0$. The advantage of Goldberg's implementation is that only one maximum flow calculation is needed in order to find both sets $S_r, r \in S_r$ and $S_i, i \in S_i$ defining the minimum cut of value f. Note that back-cuts (controlled by the *BACKCUTS* parameter) are combined with nested cuts in our implementation.

Finally, we considered the possibility of adding the smallest cardinality cut by increasing all x_{ij} values by some value *EPS*. The smallest cardinality cuts may have a great influence on the density of the underlying LP, however the running time of the maximum flow calculations may also increase. Indeed, our computational results (cf. Section 4.5.4) confirm that for most of our instances setting *EPS* to a positive value increases the CPU time.

4.5.3 Primal Heuristic

Within a branch-and-bound approach we call primal heuristics in order to find feasible solutions, or in order to improve existing upper bounds. The branch-and-cut framework of CPLEX calls the primal heuristic when the linear program in a node of the tree is solved and no more violated inequalities are found just before a branch is performed.

The basic idea of our primal heuristic is that we first fix a set S of vertices (i.e. terminals) that will be contained in the heuristic solution. Then we apply the standard minimum spanning tree heuristic for the Steiner tree problem to the graph $G = (V, E, c)$ with terminal set S. Let T be the resulting tree. We solve the PCST on T optimally by the linear time algorithm described in Section 4.3.6.

For choosing the set S of terminal vertices, we use the values of the y-variables in the LP-solution of the current node in the branch-and-cut tree. We tested the following strategies:

- Using of y-values as probabilities for inserting a vertex into S (denoted by RMinSpan);

- Computing for each vertex the average of its LP-value in the fractional solution and the best known feasible solution and using this value as the probability for choosing the vertex (the so-called, Incumbent Heuristic);
- Choosing all vertices v_i where the value of y_i in the current fractional solution is at least 1/2. We call this heuristic a CutOff Heuristic.

Having chosen the vertices in S, we compute the *distance network* G_S for S where $G_S = (S, S \times S, d_S)$. The length d_S of an edge in G_S is the length of the shortest path connecting the two corresponding vertices in G. The length of a path is determined by the x-variables of the edges on the path. We assign to each edge (i, j) in the problem graph the cost $1 - \max\{x_{ij}, x_{ji}\}$ where x_{ij} is the value of the corresponding edge-variable in the fractional solution of the current branch-and-bound node. Thus, a path is short if its edges have high LP-values.

We compute a minimum spanning tree $T = (S, E_T)$ in G_S and define the set S' of vertices in G as the union of S and the set of all vertices on the shortest paths that correspond to edges in E_T. Let $G_H = (S', E_H, c)$ be the subgraph of G induced by the vertex set S'. In this graph, the cost of each edge is again the original cost in the problem instance.

This graph is connected and therefore we can compute a minimum spanning tree $T' = (S', E_{T'})$ for it. Finally, we use the linear time algorithm to solve the PCST optimally on the tree T' (see Section 4.3.6). The resulting graph is our heuristic solution. The computational results in Section 4.5.4 show that the primal heuristic can significantly improve the gap between the lower bound and the best known feasible solution for some of our most challenging problem instances.

4.5.4 Computational Results

In this section we provide computational results of the branch-and-cut algorithm described above. In the sequel we show that we have succeeded to solve all instances known from the literature to optimality. Furthermore, we introduce new sets of larger and more difficult instances and solve some of them to optimality. A set of real-world instances arising from the design of optical fiber networks with corresponding optimal solutions are also provided in this section. Finally. we also implemented a column generation approach for the (MCF) formulation provided in Section 4.4.2. In this section we compare several column generation strategies (including one based on the memetic algorithm described in Section 4.3). In the end we compare the performance of implementations of (MCF) and (CUT) formulations and draw some conclusions.

Testing the Branch-and-Cut Algorithm

Our new branch-and-cut approach has been extensively tested on the instances introduced in Section 4.2.1. In what follows, we compare our results against those recently obtained by Lucena and Resende [111] (denoted by LR). For groups `C` and `D`, Tables 4.9 and 4.10 list instance name, and the LR results: the best obtained lower bounds (*L. Bound*) and the CPU times in seconds required to prove optimality (t [s]). If the CPU time is not given, it means

that the LR algorithm terminated because of excessive memory consumption. For our new ILP approach, we provide the following values: the provably optimal solution value (OPT), the total running time in seconds (t [s]) (including preprocessing time), the number of violated cuts found by our separation procedure ($\#Cuts$), the number of violated Gomory fractional cuts automatically added by CPLEX ($\#G.\ Cuts$) and the total running time of the same algorithm without preprocessing (t_{noprep} [s]).

On all but 8 instances of groups `C` and `D` lower bounds obtained by LR were equal to known upper bounds obtained by Canuto et al. [23]. However, on 16 instances from `C` and `D`, the LR algorithm did not prove the optimality. Improving upon their results, our new ILP approach solved all instances known from the literature to proven optimality. The optimal solution values, that were not guaranteed to be optimal before, are marked with an asterisk while new optimal values are given in bold face.

Comparing our running time data (achieved on a Pentium IV with 2.8 GHz, 2 GB RAM, SPECint2000=1 204) with the results of Lucena and Resende [111] (done on SGI Challenge Computer 28 196 MHz MIPS R10000 processors with 7.6 GB RAM, each run used a single processor), the widely used SPEC© performance evaluation (`www.spec.org`) does not provide a direct scaling factor. However, taking a comparison to the respective benchmark machines both for SPEC 95 and SPEC 2000 into account, we obtain a scaling factor of 17.2. On the other side, in [38] the SGI machine is assigned a factor of 114 and to our machine the factor 1 414, which gives 12.4 as a scaling factor. Thus, we can argue by a conservative estimate that dividing the LR running times by a factor of 20 gives a very reasonable basis of comparison to our data.

Tables 4.9 and 4.10 document that our new approach is able to solve all the instances to optimality within a very short time, even if preprocessing is turned off. The running time comparison for those instances where LR running times are known, shows that our new approach is significantly faster. If we compare average running times for our ILP with and without preprocessing, we also see that there is no particular advantage of applying preprocessing on instances of groups `C` and `D`. For group `D`, the average total running time is even slightly better when preprocessing is turned off. This indicates that for these instances there is a trade-off between computational effort needed to preprocess the input graph, and the overall branch-and-cut running time.

Both algorithms, LR and our new ILP approach, solved all `P` and `K` instances to optimality. Thus, we omit the corresponding tables here and refer to Table 4.11 and the next subsection where we analyze the running time of our algorithm. All the instances of `K,P,C` and `D` groups are solved in the root node of the branch-and-cut tree.

Summarizing Results on `K,P,C` and `D` Groups In Table 4.11 we summarize experimental results of the branch-and-cut algorithm without running the reduction procedures proposed in Section 4.2. We represent the following values for each group `K,P,C` and `D`: for each instance where the LR algorithm proved optimality, and whose $t_{ILP} > 0.2$, we calculate the *speed-up factor* t_{LR}/t_{ILP}. We then present the average (AVG), minimum (MIN) and maximum

Table 4.9: Results obtained by Lucena and Resende (LR) and our new results, on the instances from Steiner series C. Running times in (LR) are divided by 20 for comparison (cf. text above). Asterisk marks new certificates of optimality for known values. New optimal solution values are given in bold face.

	LR		ILP				
Instance	L. Bound	t [s]/20	OPT	t [s]	# Cuts	# G. Cuts	t_{noprep} [s]
C1-A	18	0.0	18	1.3	0	0	0.1
C1-B	85	0.1	85	1.2	6	6	0.2
C2-A	50	0.0	50	1.1	0	0	0.1
C2-B	141	0.0	141	1.1	0	0	0.1
C3-A	414	0.1	414	1.2	0	0	0.3
C3-B	737	1.3	737	1.4	4	1	0.4
C4-A	618	0.1	618	1.3	2	6	0.9
C4-B	1063	14.4	1063	1.5	4	0	0.6
C5-A	1080	4.0	1080	1.4	0	0	8.6
C5-B	1528	174.4	1528	2.0	0	0	2.0
C6-A	18	0.0	18	2.2	0	0	0.1
C6-B	55	2.9	55	2.5	8	12	0.6
C7-A	50	0.1	50	2.6	0	0	0.1
C7-B	102	0.2	102	2.7	4	9	0.2
C8-A	361	1.7	361	2.9	0	0	0.6
C8-B	500	10.8	500	3.3	4	1	0.6
C9-A	533	4.2	533	3.6	12	16	0.9
C9-B	694	95.6	694	3.8	14	23	1.1
C10-A	859	8.0	859	4.0	8	0	2.5
C10-B	1069	175.1	1069	4.1	10	0	4.3
C11-A	18	0.2	18	9.7	2	0	0.2
C11-B	32	3.4	32	13.7	48	25	3.2
C12-A	38	1.9	38	7.1	4	0	0.3
C12-B	46	6.3	46	7.5	12	8	0.6
C13-A	236	16.6	236	10.5	4	1	2.1
C13-B	258	154.6	258	12.7	56	22	5.4
C14-A	293	87.5	293	8.0	2	0	0.6
C14-B	318	57.1	318	8.1	4	0	0.6
C15-A	501	2711.2	501	7.6	12	1	6.4
C15-B	551	—	*551	6.6	4	0	4.4
C16-A	11	10.2	11	3.8	6	9	3.7
C16-B	11	10.3	11	3.7	6	9	3.6
C17-A	18	12.5	18	4.8	12	30	2.8
C17-B	18	19.4	18	4.0	8	15	2.8
C18-A	111	1001.6	111	4.1	2	2	5.3
C18-B	113	—	*113	6.2	14	3	25.7
C19-A	146	7610.9	146	3.6	2	0	3.9
C19-B	146	950.0	146	3.4	2	0	4.1
C20-A	265	—	**266**	6.3	2	0	22.6
C20-B	267	—	*267	5.0	2	0	51.8
C-AVG	334.3	—	334.3	4.5	7	5.0	4.4

Table 4.10: Results obtained by Lucena and Resende (LR) and our new results, on the instances from Steiner series D. Running times in (LR) are divided by 20 for comparison (cf. text above). Asterisk marks new certificates of optimality for known values. New optimal solution values are given in bold face.

	LR		ILP				
Instance	L. Bound	t [s]/20	OPT	t [s]	# Cuts	# G. Cuts	t_{noprep} [s]
D1-A	18	0.0	18	4.9	0	0	0.2
D1-B	106	0.3	106	5.0	8	8	0.5
D2-A	50	0.0	50	4.9	0	0	0.2
D2-B	218	0.1	218	4.9	0	0	0.3
D3-A	807	0.6	807	5.7	0	0	1.1
D3-B	1509	16.6	1509	6.8	2	0	0.9
D4-A	1203	2.6	1203	6.0	0	0	3.0
D4-B	1881	77.6	1881	8.0	8	0	2.5
D5-A	2157	29.9	2157	8.5	6	0	93.8
D5-B	3135	—	*3135	12.9	6	0	7.3
D6-A	18	0.1	18	14.5	0	0	0.2
D6-B	67	11.3	67	16.0	20	4	3.0
D7-A	50	0.2	50	11.4	0	0	0.2
D7-B	103	7.7	103	11.7	2	1	0.4
D8-A	755	8.5	755	15.3	24	0	9.5
D8-B	1036	163.4	1036	13.2	2	1	3.1
D9-A	1070	67.3	1070	32.3	24	15	30.7
D9-B	1420	1252.6	1420	25.1	104	0	3.5
D10-A	1671	3129.5	1671	18.7	8	0	30.4
D10-B	2079	—	*2079	20.9	8	0	25.5
D11-A	18	1.7	18	29.1	2	4	0.5
D11-B	29	43.5	29	28.0	14	41	4.8
D12-A	42	14.1	42	24.6	10	2	1.6
D12-B	42	14.9	42	23.3	4	3	1.3
D13-A	445	1234.5	445	34.4	26	4	18.9
D13-B	486	223.2	486	31.4	12	0	3.3
D14-A	602	—	*602	39.2	12	0	60.2
D14-B	665	—	*665	41.9	14	1	22.4
D15-A	1040	—	**1042**	64.0	24	14	146.8
D15-B	1107	84595.9	**1108**	54.1	10	1	24.2
D16-A	13	497.9	13	14.4	8	4	19.3
D16-B	13	306.5	13	15.3	2	3	10.1
D17-A	23	847.0	23	15.3	6	27	38.0
D17-B	23	687.1	23	15.3	6	35	27.0
D18-A	218	—	*218	61.1	60	2	16.1
D18-B	223	—	**223**	17.7	10	0	15.8
D19-A	306	—	**306**	24.0	20	42	49.3
D19-B	310	—	**310**	24.4	28	2	105.5
D20-A	529	—	**536**	38.0	0	0	48.3
D20-B	530	—	**537**	33.2	2	0	122.1
D-AVG	650.4	—	650.9	22.0	12.3	5.4	23.8

(MAX) value of this factor per group. We also count the number of instances where we proved optimality for values that were not guaranteed to be optimal by LR, and also the number of instances where optimal solution were not known before. The last column shows the number of instances for which our ILP approach needed not more than 0.2 seconds to solve them.

If we assume the conservative hardware speed-up factor of 20, the results of Table 4.11 show that:

- our algorithm is on average about 30 times slower for the instances of group `K`. However, all of them could be solved to optimality by both LR and our algorithm within a short running time (within 1000 seconds, in the worst case).
- on the remaining instances that could be solved to optimality by the LR algorithm, our new approach is significantly faster, and the average running time speed-up factor lies between 11 and 156.
- our new approach is able to solve all the instances to optimality within a very short time, even if preprocessing (used also by Lucena and Resende [111]) is turned off.

Table 4.11: Comparison of running-time speed-up factors over all instances of a group: average, minimal and maximal factors for each group are given.

Group	$t_{LR}/(20 \cdot t_{noprep})$			new status of optimality		
	AVG	MIN	MAX	proven	new value	$t_{ILP} < 0.2$
`K`	0.1	0.03	0.2	-	-	7
`P`	11.9	3.3	28.7	-	-	6
`C`	116.9	0.1	1961.6	3	1	11
`D`	155.7	0.2	3498.6	5	7	12

The Advantage of Preprocessing for the Branch-and-Cut Algorithm The preprocessing techniques still play an important role in solving larger instances to optimality. This can be seen in Table 4.12 where the results of our new approach on the set `E` of benchmark instances with and without preprocessing are compared. For these more challenging instances both variant of the algorithm are terminated by setting the `cplexTimeLimit` parameter to 2 000 seconds. The results indicate the advantage of preprocessing when the size of the LPs increases.

For 5 out of 40 instances the algorithm did not find the optimal solution within the given time limit when preprocessing was turned off. On the rest of 35 instances there is only a slight improvement in running time due to preprocessing – the branch-and-cut algorithm without preprocessing is only for about 5% slower.

As before, the instances of this group are also solved without branching.

Table 4.12: Results obtained on the instances derived from Steiner series E.

		With preprocessing			Without preprocessing		
Instance	*OPT*	t [s]	# Cuts	# G. Cuts	t [s]	# Cuts	# G. Cuts
E01-A	13	21.6	0	0	0.5	0	0
E01-B	109	22.7	10	21	1.7	6	9
E02-A	30	20.8	0	0	0.5	0	0
E02-B	170	21.0	4	4	1.6	6	5
E03-A	2231	34.8	6	18	25.8	26	5
E03-B	3806	35.0	6	0	16.5	14	6
E04-A	3151	39.4	8	2	291.1	18	12
E04-B	4888	33.3	8	0	20.4	6	0
E05-A	5657	78.9	14	2	—	—	—
E05-B	7998	62.0	2	0	135.0	16	0
E06-A	19	38.3	0	0	0.7	0	0
E06-B	70	39.1	2	5	3.6	4	11
E07-A	40	39.8	0	0	0.7	0	0
E07-B	136	42.0	6	15	3.8	4	12
E08-A	1878	97.5	10	19	80.2	10	8
E08-B	2555	60.9	14	2	22.8	22	2
E09-A	2787	111.4	10	5	593.2	20	3
E09-B	3541	69.7	6	2	23.7	10	3
E10-A	4586	285.1	96	2	545.5	12	0
E10-B	5502	112.8	10	0	350.0	145	2
E11-A	21	147.7	0	0	5.7	0	0
E11-B	34	154.4	2	8	8.7	2	16
E12-A	49	86.8	2	4	2.0	0	0
E12-B	67	100.5	22	21	22.9	32	24
E13-A	1169	173.1	12	3	98.0	20	2
E13-B	1269	177.3	14	7	27.6	10	1
E14-A	1579	566.0	200	0	71.2	32	1
E14-B	1716	458.2	166	6	140.0	160	1
E15-A	2610	490.6	150	0	314.1	54	1
E15-B	2767	577.7	136	4	696.2	112	5
E16-A	15	110.6	8	17	80.4	16	27
E16-B	15	112.5	8	23	103.2	18	22
E17-A	25	162.9	20	19	146.9	26	27
E17-B	25	113.8	10	18	245.2	44	23
E18-A	555	813.3	170	0	—	—	—
E18-B	564	688.8	143	0	—	—	—
E19-A	747	331.0	96	5	—	—	—
E19-B	758	228.6	70	4	1143.2	84	7
E20-A	1331	129.2	16	1	1288.3	48	3
E20-B	1342	232.7	2	0	—	—	—
E-AVG	1645.6	178.1	36.5	5.9	—	—	—

Tuning Separation Strategy In our default implementation we used nested cuts and back-cuts. The parameter *EPS* was set to 0.0, which means that we refrained from the calculation of smallest cardinality cuts. Our computational experiments have shown that the usage of back-cuts is crucial for our implementation. By omitting the back-cuts, some of the larger instances (even of groups `C` and `D`) could not be solved to proven optimality - the algorithm terminated because of excessive memory consumption.

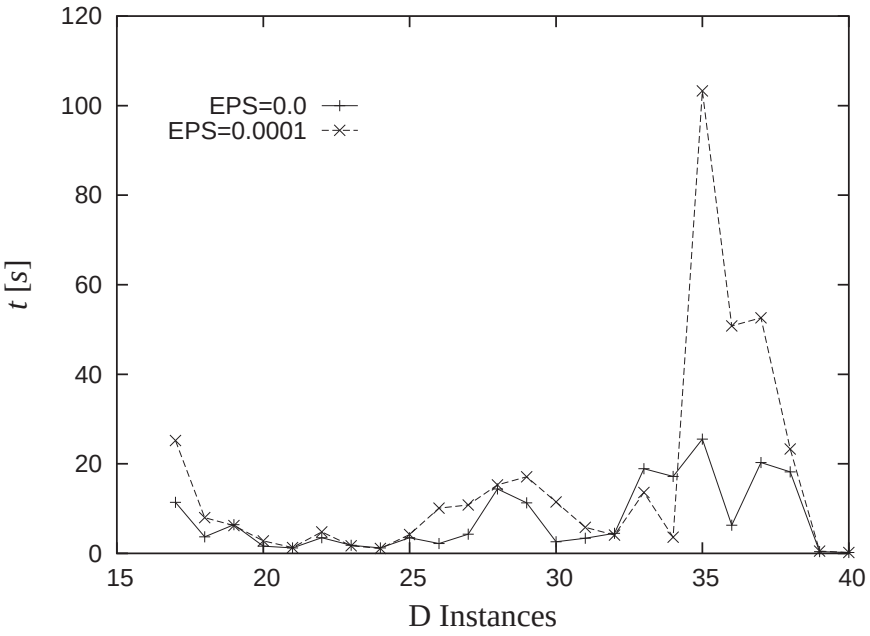
The role of parameter *EPS* within the separation is studied in Table 4.13 where we show CPU times in seconds averaged over each group of instances. All the runs were limited to 1 000 seconds (except for group `E`, where we set the limit to 2 000). As the first two columns in Table 4.13 document, the usage of minimum cardinality cuts within the separation plays the most important role for solving the instances in the `K` group. In contrary, for solving the instances in the `C`, `D` and `E` groups, computing the smallest cardinality cuts seems to be too expensive, i.e., there is a trade-off between the time needed to solve the maximum-flow algorithm and the time for solving a single LP-relaxation. Figures 4.9(a) and (b) depict the running time performance of the branch-and-cut algorithm with and without smallest cardinality cuts for the largest 24 instances of groups `D` and `E`, respectively. Although there are some instances where the usage of smallest cardinality cuts may reduce the total running time, the overall performance is getting worse. For two largest groups, `D` and `E`, the usage of smallest cardinality cuts slows down the running time for more than two times.

Table 4.13 also documents the crucial role of using inequalities (4.34) within the initialization procedure. These inequalities say that each edge can only be used in one direction in any feasible solution (see Section 4.5.1) and only if one of its incident vertices is in the solution. The last two columns represent results obtained by omitting inequalities (4.34) from the initialization.

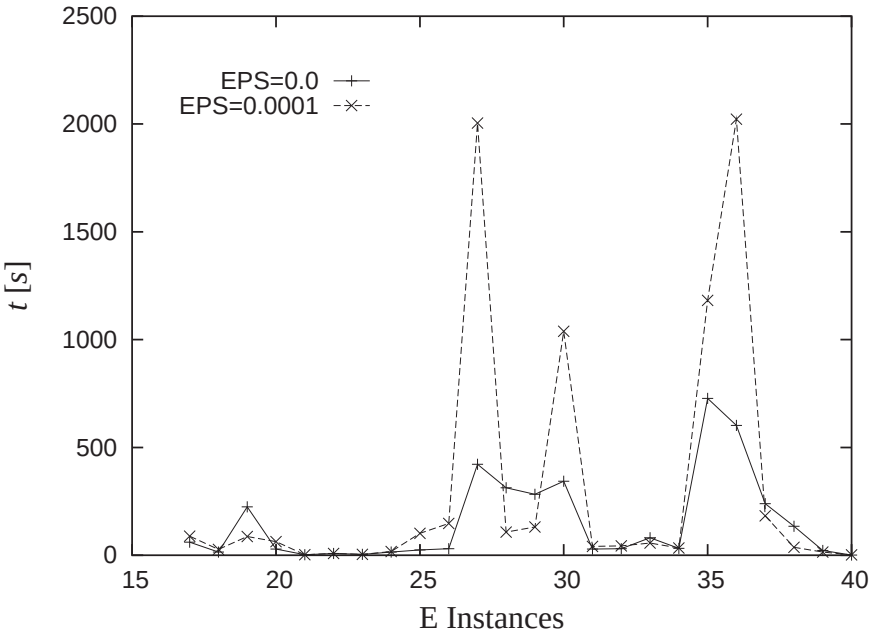
These inequalities that forbid subtours of size two play the most important role in our implementation. By separating these inequalities instead of inserting them in the initialization phase already, the algorithm is not able to solve many of the instances within a given time limit. This negative effect can not be compensated by the – sometimes considerable – improvement of running time brought about by the separation with smallest cardinality cuts (in particular for `K` and `P` groups).

The difficulties with subtours of size two usually arise when a customer vertex i is connected to a non-customer vertex j by an edge with cost $c_{ij} < p_i$. In this case, the initial LP-solution contains a directed arborescence rooted at j (note that $y_j = 0$, i.e., the in-degree of j is zero), with an arc (j, i) of negative cost. By adding $x_{ij} \leq y_i, \forall i \in V_{\mathrm{SA}} \setminus \{r\}$ inequalities, instead of (4.34), solutions of the initial LP contain subtours of size two on such pairs (i, j) of vertices. Figure 4.10 illustrates examples of fractional LP-solutions immediately after initialization, i.e. before any separation is called. Similarly, the problems appear also if two customer vertices i and j are connected by an edge whose $c_{ij} < \min\{p_i, p_j\}$.

Although the number of these inequalities is linear in number of edges $O(|A_{\mathrm{SA}}|)$, they usually do not slow down the performance of solving a single LP-relaxation. On the other side, they may save a huge number of separation calls.

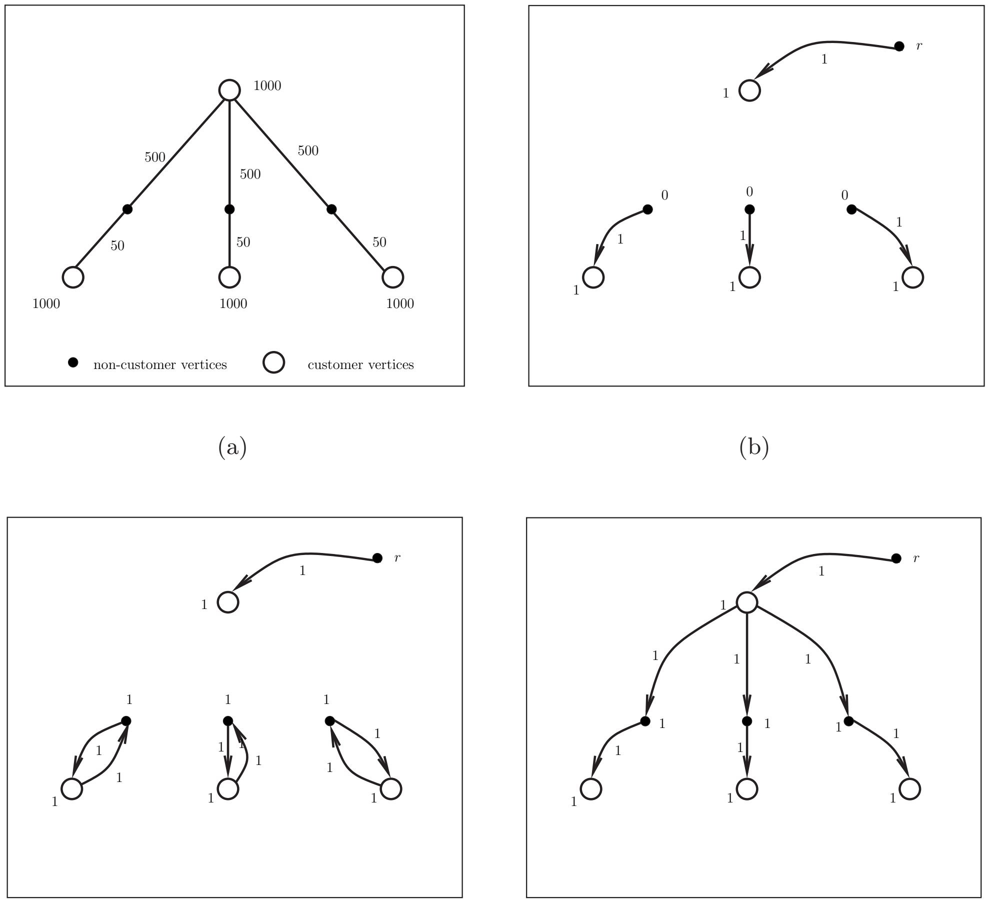


(a)



(b)

Figure 4.9: Running time comparison of separation with and without smallest cardinality cuts, $EPS = 0.0001$ and $EPS = 0.0$, respectively. The largest 24 instances of the group `D` (a) and `E` (b) are shown: from `D09-A` to `D20-B`, and from `E09-A` to `E20-B`, respectively. The total running time of the branch-and-cut algorithm (without preprocessing time) is shown.



(c) (d)

Figure 4.10: The advantage of explicit initialization with (4.34) constraints can be seen when comparing LP-solutions obtained immediately after initialization: (a) Given instance with edge costs and vertex prizes; (b) LP-solution when we refrain from (4.34) inequalities. LP-values of edges and vertices are shown, $c(LP) = 150$; (c) LP-solution if $x_{ij} \leq y_i, \forall i \in V_{\text{SA}} \setminus \{r\}$ are used in initialization. $c(LP) = 300$; (d) LP-solution when (4.34) are added explicitly in the initialization. $c(LP) = 1650$, the solution is already optimal, no separation needed.

Table 4.13: Comparison of average CPU times over all instances of a group for separation with or without smallest cardinality cuts ($EPS = 10^{-4}$ or $EPS = 0$, respectively) and for initialization with or without (4.34) constraints. Preprocessing times are discarded.

Group	Init. with (4.34) ineq.		Init. without (4.34) ineq.	
	$EPS = 0$	$EPS = 10^{-4}$	$EPS = 0$	$EPS = 10^{-4}$
K	48.8	7.6	88.8	2.9
P	0.2	0.3	21.4	2.2
C	0.8	0.9	—	0.8
D	4.5	9.6	—	—
E	95.1	199.1	—	—

4.5.5 Testing Real-World Instances

In this section we consider a set of 35 instances based on the real-world examples that have been used in the design of fiber optic networks for some German cities [10, 9][4]. The instances are generated according to GIS data bases: connections and positions of vertices are based on real infrastructures, but, for reasons of data protection, the choice of customers and their prizes are slightly changed. Our instances are divided in two groups: `Cologne1` and `Cologne2`. Basic properties of these instances, like the number of vertices $|V|$, the number of edges $|E|$ and the number of customers $|R|$ are shown in Table 4.5.5. For each of the two groups there are 5 subgroups, each of which contains a number of equal-sized instances. The number of instances of each subgroup is also shown in the table.

These instances contain an existing subnetwork that needs to be augmented to serve new customers. This subnetwork can be shrunk into a single vertex, in the same way as we did it in Section 3.2.1 for the V2AUG problem. Multiple edges are replaced by cheapest connections and self-loops are discarded. After this transformation, we only need to consider the rooted PCST problem, with the shrunk vertex as a root. Two examples of these instances are shown in Figure 4.11.

Since these instances are typically very dense (i.e. they are almost complete graphs), it does not pay off to apply degree-n test. Our experiments have shown that because the number of customers is typically very small, the minimum adjacency test does not help reducing the instances at all. Hence, we applied only the least-cost test, and, as Table 4.5.5 in column $|E'|$ documents, the savings obtained in that way are greater than 90%.

Table 4.15 shows the performance of our ILP approach on the real-world instances. We compare two approaches based on the initialization of the LP with and without (4.34), i.e. generalized subtour elimination constraints of size two. The results document that all the instances of `Cologne1` group could be solved to optimality in less than 2 400 seconds. For instances of `Cologne2` group, we present the percentage of the gap between global lower bound

[4]Instances are available at `http://www.ads.tuwien.ac.at/pcst`.

Figure 4.11: Examples of two instances from `Cologne1` group. Optimal solution of (a) `i02M2`; (b) `i04M3`. Shadowed vertices represent customers, while bold lines belong to the existing network. Grid lines in the background determine customers' prizes.

Table 4.14: Properties of two groups of real-world instances, `Cologne1` and `Cologne2`.

`Cologne1`						`Cologne2`					
Group	$\|V\|$	$\|E\|$	$\|R\|$	$\|E'\|$	# of inst.	Group	$\|V\|$	$\|E\|$	$\|R\|$	$\|E'\|$	# of inst.
`i01`	768	69077	10	6332	3	`i01`	1819	213973	9	16743	4
`i02`	769	69140	11	6343	3	`i02`	1820	213915	7	16740	4
`i03`	771	69100	13	6343	3	`i03`	1825	214095	12	16762	4
`i04`	761	68907	3	6293	3	`i04`	1817	213859	4	16719	4
`i05`	761	68934	3	6296	3	`i05`	1826	214013	13	16794	4

Table 4.15: Results on two groups of real-world instances. We compare two ILP approaches, where the initialization is done with and without constraints (4.34). All instances of `Cologne1` group are solved to optimality, while for the `Cologne2` group, we show the gap in percent obtained after the time limit of 2 hours was exceeded.

`Cologne1`				`Cologne2`					
Instance	Without (4.34) t [s]	With (4.34) t [s]	OPT	Instance	Without (4.34) %-gap	t [s]	With (4.34) %-gap	t [s]	OPT
i01M1	**0.5**	2.9	109271.5	i01M2	0.0	2.1	0.0	5.1	355467.7
i01M2	**252.3**	487.8	315925.3	i01M3	1.9	31025.7	**1.7**	27331.9	628833.6
i01M3	1371.4	**1195.8**	355625.4	i01M4	**2.1**	45002.1	3.9	40927.5	773398.3
i02M1	**0.5**	2.9	104065.8	i02M2	0.0	107.0	0.0	110.7	288946.8
i02M2	**431.8**	598.2	352538.8	i02M3	**0.5**	9034.4	2.4	14173.6	419184.2
i02M3	2353.6	**1810.9**	454365.9	i02M4	**2.1**	13322.0	4.7	19124.3	430034.3
i03M1	**0.5**	3.1	139749.4	i03M2	0.0	907.1	0.0	855.9	459918.9
i03M2	362.6	**326.8**	407834.2	i03M3	**3.3**	37416.1	5.7	42150.0	643062.0
i03M3	1140.2	**755.9**	456125.5	i03M4	**3.7**	42752.0	5.6	42237.7	677733.1
i04M1	**0.5**	2.8	25282.6	i04M2	0.0	2.5	0.0	5.4	161700.5
i04M2	**20.0**	22.6	89920.8	i04M3	**0.0**	5095.1	2.1	13259.2	245287.2
i04M3	**42.1**	77.7	97148.8	i04M4	**0.0**	4298.1	0.1	8700.1	245287.2
i05M1	**0.5**	2.8	26717.2	i05M2	0.0	2107.0	0.0	2568.7	571031.4
i05M2	**94.7**	122.9	100269.6	i05M3	**0.1**	11852.9	1.0	19655.4	672403.1
i05M3	443.8	**399.4**	110351.2	i05M4	**0.5**	16203.8	0.8	16343.5	713973.6

and optimum after two hours computation time: %-$gap = (OPT - LB_g)/OPT$. We also show total running times in seconds (t [s]) needed to prove optimality (the time limit was set to 45 000 seconds). Finally, we also provide optimal values in OPT columns.

While constraints (4.34) have shown to be very advantageous for the previous instances known from the literature, Table 4.15 documents that for the real-world instances there is a trade-off between the size of the underlying LP and the number of separation calls that can be saved. This can be explained by a very small percentage of customer vertices, which is less than 2% and 1%, for `Cologne1` and `Cologne2` groups, respectively. The number of subtours of size two usually depends on the number of negative edges which directly corresponds to the number of customer vertices. On the other side, using (4.34), we insert $2 \cdot |A_{\mathrm{SA}}|$ inequalities in the initialization phase, which obviously represents a disadvantage for such very large instances with only few customer vertices. All `*M1` instances of `Cologne2` group could be solved to optimality in less than 30 seconds, and they always represent single-vertex solutions, thus we omit them from Table 4.15.

We can conclude that our ILP approach shows that it can be used within real-world applications to solve instances that appear in practice. The approach is able to solve very large graphs (with up to 1 825 vertices and 214 095 edges) to proven optimality within less than 12 hours, in the worst case. Since we are dealing with off-line network design problems, such a

running time is still considered as reasonable.

Testing Primal Heuristics

All the instances mentioned above are solved in the root node of the branch-and-cut tree, thus no branching was needed. In order to test the performance and quality of primal heuristics described in Section 4.5.3, we introduced an additional set of difficult problem instances that cannot be reduced by the preprocessing algorithm described in Section 4.2.

Rosseti et al. [147] proposed three new sets of artificially generated and very difficult instances for the Steiner tree problem. We used the most difficult of them, the so-called *hypercubes*, to derive new test problems for the PCST.

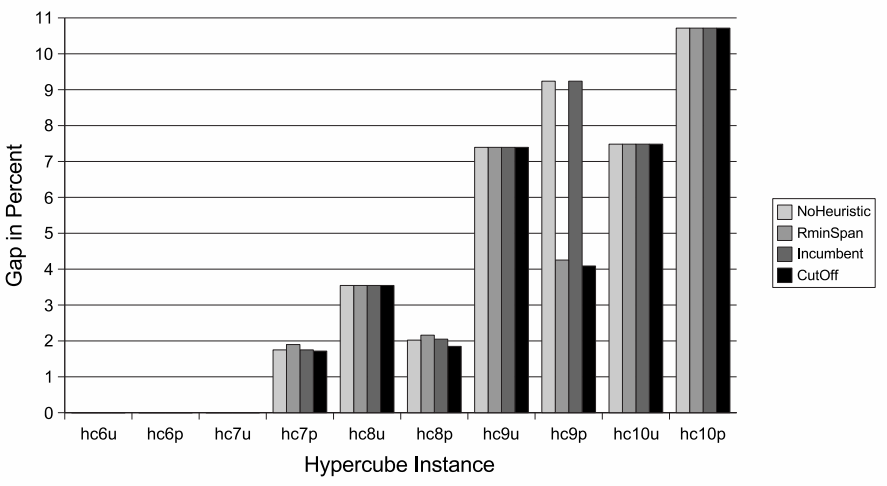


Figure 4.12: Comparing the performance of primal heuristics.

Graphs in this series for the Steiner tree problem are d-dimensional hypercubes with $d \in \{6, \ldots, 12\}$. For each value of d, the corresponding graph has 2^d vertices and $d \cdot 2^{d-1}$ edges. These graphs are bipartite and the terminals are defined as one set of the bipartition. The so-called *unperturbed* instances receive unit edge costs whereas edges of the so-called *perturbed* instances receive random integral costs distributed uniformly in the interval $[100, 110]$.

We derived the PCST hypercubes by assigning zero profits to non-terminal vertices and an integer profit randomly chosen from a uniform distribution over the interval $[1, 2]$ and $[100, 220]$ for unperturbed and perturbed instances, respectively. Our naming convention is as proposed in [147], thus `hc`d[`u`|`p`] denotes an unperturbed (`u`) or perturbed (`p`) d-dimensional hypercube instance.

The running time of the program was limited to 3 600 seconds. For each instance, we provided an initial feasible solution computed by the memetic algorithm described in Section 4.3. As primal heuristic for the branch-and-bound algorithm, we tested three simple minimum spanning tree heuristics described in Section 4.5.3 that use the fractional solution at the current branch-and-bound node to compute a feasible solution. The primal heuristic is called in

each node of the branch-and-bound tree. Note that our preprocessing method described in Section 4.2 did not manage to reduce these instances at all.

Table 4.16 shows the results for our experiments with the hypercube instances. Beside the size of the instance, we show the optimality gap for the algorithm obtained without primal heuristic (i.e using only CPLEX default mechanism) and using three heuristics described above: RMinSpan, Incumbent and CutOff. The optimality gap is obtained as

$$gap_g = \frac{UB_g - LB_g}{LB_g} \times 100\% \ ,$$

where UB_g represents the costs of the best subtree (obtained either within the MA, or during the branching), and LB_g is a global lower bound. The *optimality* gap_g expresses that the solution with costs UB_g is at most $gap_g\%$ more expensive than the optimal solution.

Table 4.16: Computational results for the hypercube instances with a time limit of 3 600 seconds. The least optimality gaps (or the least running times for the instances solved to optimality) are shown in bold.

Instance	$\|V\|$	$\|E\|$	No Heur. $gap_g(t\ [s])$	RMinSpan $gap_g(t\ [s])$	Incumbent $gap_g(t\ [s])$	CutOff $gap_g(t\ [s])$	LB_g	UB_g
hc6u	64	192	0 (0.1)	0 (0.1)	0 (0.1)	0 (0.1)	35.0	35.0
hc6p	64	192	0 (**60.6**)	0 (78.5)	0 (77.8)	0 (78.1)	3907.0	3907.0
hc7u	128	448	0 (1145.5)	0 (**813.1**)	0 (814.0)	0 (813.3)	71.0	71.0
hc7p	128	448	1.5	**1.3**	1.4	1.5	7723.0	7630.5
hc8u	256	1024	6.4	6.4	6.4	6.4	149.0	140.0
hc8p	256	1024	**2.3**	2.4	2.8	2.7	15337.0	14988.0
hc9u	512	2304	7.8	7.8	7.8	7.8	300.0	278.3
hc9p	512	2304	9.3	**3.9**	4.5	4.1	30795.0	29643.1
hc10u	1024	5120	7.5	7.5	7.5	7.5	593.0	551.6
hc10p	1024	5120	10.6	10.6	10.6	10.6	65185.0	58913.4

The three instances `hc6u`, `hc6p` and `hc7u` could be solved to optimality within the time bound. For these instances we show in brackets the running time in seconds each of the algorithms needed to prove the optimality. Although in some cases usage of the primal heuristic may slow down the performance (like in `hc6p` case), there are instances where (1) the primal heuristic can significantly speed up the time needed to prove optimality (like `hc7u`, running time reduced for about 30%) or (2) improve the optimality gap within a given timebound (for `hc9p`, for example, using the RMinSpan heuristic we reduced the optimality gap for about 5.5%). For the hypercubes with dimensions 11 and 12, no lower bound could be found because the initial linear program could not be solved before the end of the time limit.

Finally, although all the heuristics have widely similar performance, we can conclude that, on average the RMinSpan Heuristic provides the best results.

Our heuristic can only improve the result if the branch-and-bound tree has more than one node, otherwise it is never called. It follows that it is not useful for easy instances (where the

problem can be solved in the root node) or very large instances (where CPLEX cannot finish solving the linear program of the root node of the tree before the time limit). So we conclude that the heuristic is of limited use and should only be tried for instances where CPLEX without the heuristic has to branch.

4.5.6 Column Generation Approach for (MCF)

In this section we provide computational results for the multi-commodity flow formulation of the PCST proposed in Section 4.4.2 enhanced with the flow-balance constraints provided in Section 4.4.5. Since in this formulation we are dealing with a very large number of variables ($O(|A_{\mathrm{SA}}| \cdot |R_{\mathrm{SA}}|)$ which is $O(|V'|^3)$ in general case) we propose to use a column generation approach to solve the LP-relaxation of the problem. If the optimal solution of the LP-relaxation is not already integer, we switch to the CPLEX mixed integer programming optimizer that continues to search for the optimal solution by applying a branch-and-bound approach.

We tested the following three column generation settings:

- Jünger et al. [89] proposed to initialize sparse and reserve graphs for solving the TSP with the 5- and 10-nearest neighbor graph, respectively. Here, we initialized sparse and reserve graphs using 2- and 3-nearest neighbor graphs. The reason for this is that the solutions of the TSP span all the vertices of the underlying graph, which is in the PCST not always the case. We call this approach NN.

- We consider a combination with the memetic algorithm proposed in Section 4.3. The restricted master problem is initialized with the last MA population (the sparse graph) while all the edges contained in the solutions of the first MA's population, which do not belong to the sparse graph already, are building the reserve graph. We denote this approach by CGMA.

- We finally check what happens if the sparse and reserve graphs are empty, thus, a complete pricing is done in each iteration, and the sparse graph contains as few edges as possible. This strategy is denoted with EMP.

Recall that we are dealing with the unrooted PCST here. Thus, to assure the feasibility of the sparse graph, in addition to the sparse graph edges obtained as described above, we initialize the restricted master problem with all outgoing edges of the artificial root vertex.

Because of the size of the underlying LPs, we tested only small instances, like those belonging to `K` group as well as smaller instances of group `C` (from `C1` to `C8`). For all of these instances all three strategies ended up with optimal solutions. The values of obtained lower bounds were always optimal. However, total running times were different from instance to instance, ranging from 219 to 4827 seconds, for group `K`, for example. Figure 4.13 shows running times of the algorithms with the proposed three column generation techniques for the largest instances of group `K`. Instances `K100*` were all solved to optimality in less than a second, while for `K200` instance all approaches needed less than 10 seconds.

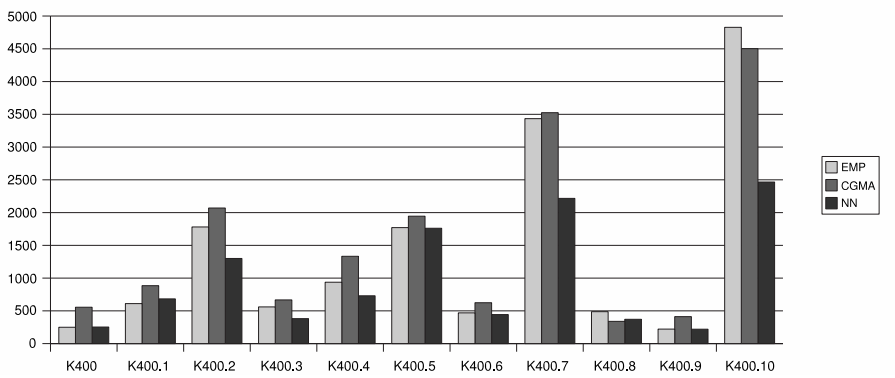


Figure 4.13: The running time of the column generation approach for the (MCF) formulation for the 11 largest K instances. Compared are three pricing strategies: EMP, CGMA and NN defined above. Preprocessing times are not included.

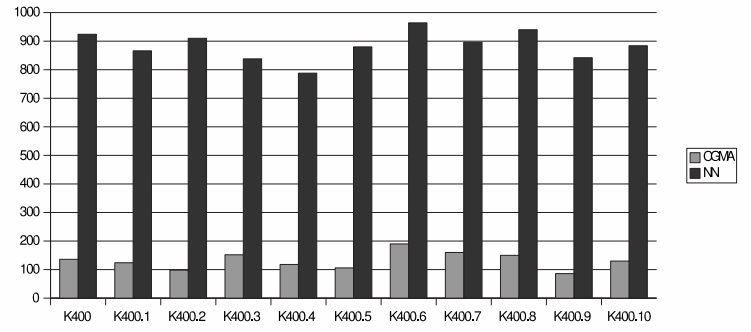


Figure 4.14: Testing K400 instances: Size of the sparse graph when it is initialized using CGMA and NN strategies.

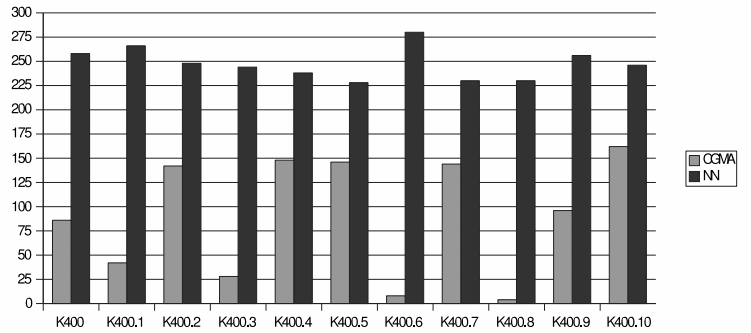


Figure 4.15: Testing K400 instances: Size of the reserve graph when it is initialized using CGMA and NN strategies.

Figures 4.14 and 4.15 show the size of the sparse and reserve graph, respectively, for NN and CGMA approaches. In this comparison, the outgoing edges from the artificial root vertex are taken out of consideration. The size of the sparse graph in CGMA approach is significantly smaller than when nearest neighbor strategy is used. This can be seen as an advantage for larger LP formulations, when solving one pricing iteration can be a very time-consuming process. On the other side, when sparse and reserve graphs are small, more edges are inserted during the pricing phase, i.e. the number of iterations may increase. This is shown in Figure 4.16, where the number of iterations the CGMA approach needed is for about 60% greater that the number of NN iterations.

Comparing Figures 4.13 and 4.16 one observes that for group `K`, there is a direct correspondence between the number of iterations and the running time of the algorithm. In that case, small sparse and reserve graphs represent a disadvantage because the underlying LP formulation can be solved quickly. It is interesting to see that sometimes, even when the sparse graph represents the optimal solution, the running time of the column generation approach does not directly depend on it. Indeed, for instances `K400.1, K400.2, K400.3, K400.6, K400.8` and `K400` the solution found by MA was already optimal. However, the number of iterations the CGMA approach needed to solve them is sometimes 100% greater than the number of iterations of NN approach (see `K100` instance, for example).

Figure 4.17 illustrates running times in seconds (not including preprocessing times) for EMP, CGMA and NN pricing strategies tested on the smallest 16 instances of group `C`. The running times indicate that when the size of the underlying LP increases, there is a trade-off between the size of the LP formulation and the number of iterations. In this case, the increase in the size of the sparse graph may significantly slow down the running time of a single pricing iteration. This can be seen on instances `C8-A` and `C8-B`. Figure 4.18 shows that the number of iterations CGMA approach needed to finish the optimization is on average for about 70% greater than NN number of iterations. However, the CGMA approach is significantly faster than both NN and EMP approaches, especially for the instances whose underlying LP is of moderate size. Indeed, in Table 4.17 the instances of `C` group are shown for which the CGMA outperformed the NN approach. We show the number of rows, the number of columns and the number of non-zero entries for CGMA and NN approaches, immediately after the initialization of the restricted master problem. One observes that when the size of the master problem exceeds a certain threshold value, the NN approach is not competitive anymore.

Finally, we can conclude that the the multi-commodity flow formulation of the PCST is very ineffective in practice. Figure 4.19 shows that the running times for the (MCF) formulation (for the largest 11 `K` instances) vary widely and consistently exceed the running time for the (CUT) model by a huge margin.

However, the proposed CGMA approach may be an advantageous one for those problems where there is no ILP formulation with an efficient separation algorithm. For different kinds of cutting and packing problems, for example, when LPs contain quadratic (or cubic) number of rows or columns, the way of choosing the sparse and reserve graph may be of crucial role (see [137]).

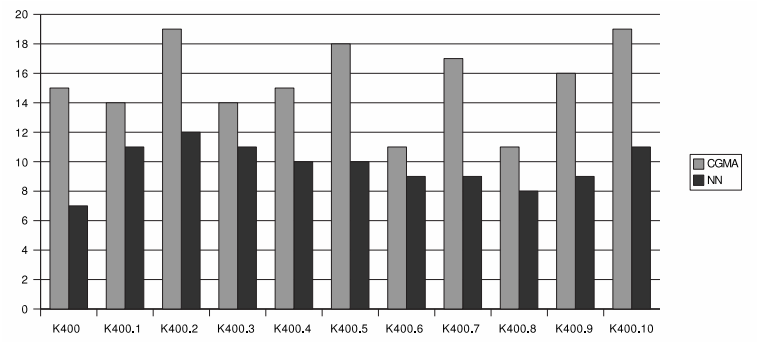


Figure 4.16: Testing `K400` instances: The number of iterations needed to solve the LP-relaxation using CGMA and NN strategies.

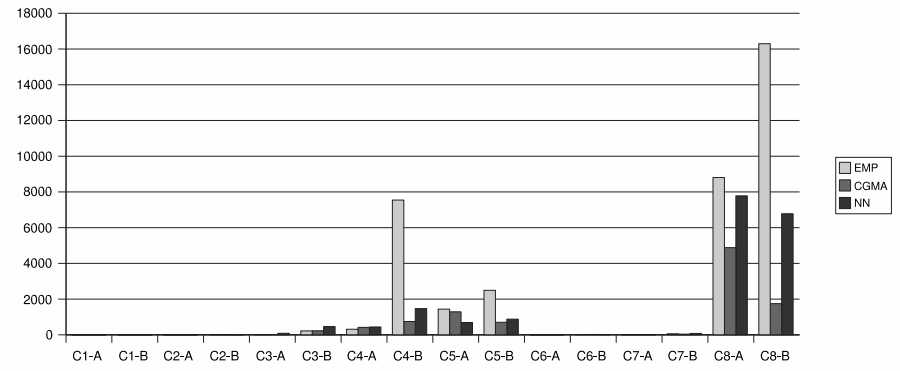


Figure 4.17: The running time in seconds EMP, CCGA and NN approach needed to solve the LP-relaxation for the smallest 16 instances of group `C`.

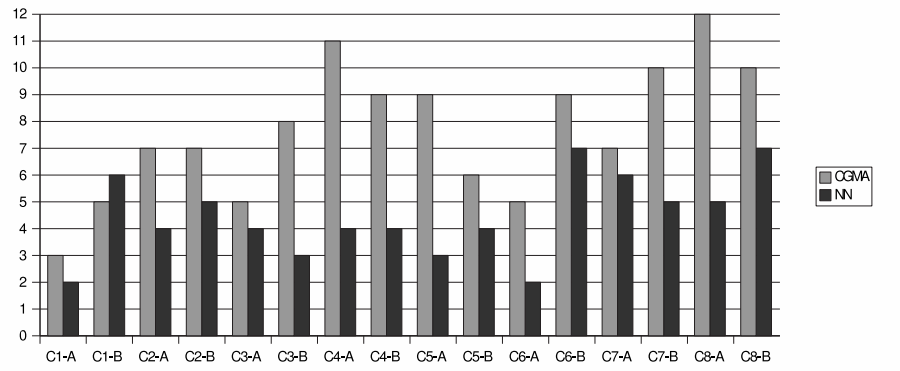


Figure 4.18: The number of iterations CCGA and NN approach needed to solve the LP-relaxation for the smallest 16 instances of group `C`.

Table 4.17: Size of the restricted master problem for CGMA and NN approaches.

Instance	CGMA			NN		
	# of rows	# of columns	# of nonzeros	# of rows	# of columns	# of nonzeros
C4-B	51208	30361	153755	79333	54164	244591
C5-B	66657	38486	205987	87425	57050	273708
C8-A	21852	13641	64862	94664	69110	284667
C8-B	32945	20592	97976	100173	73656	303776
K400.10	8359	5982	26508	35990	26334	108105

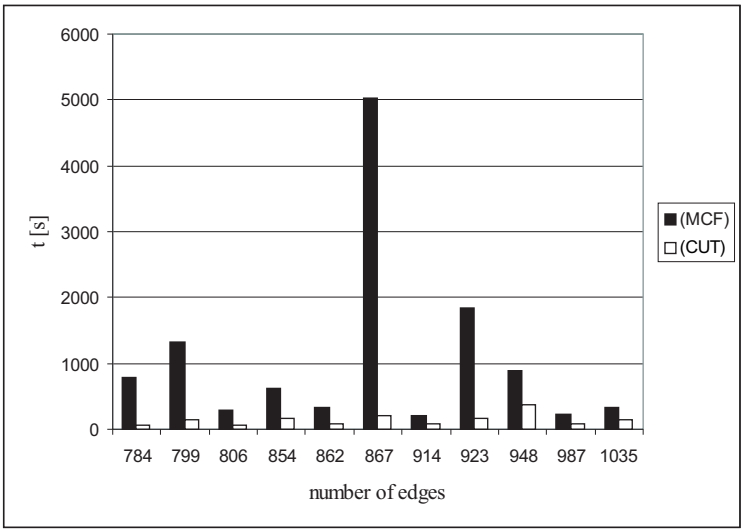


Figure 4.19: CPU times of (CUT) and (MCF) formulations for 11 K400 instances. The instances are sorted according to their number of edges after preprocessing.

4.6 Summary

The prize-collecting Steiner tree problem (PCST) formalizes in an intuitive way the planning problem encountered in the design of utility networks such as gas, or district heating, or fiber optic networks. Selecting the most profitable customers and connecting them by a least-cost network immediately leads to the problem of computing a Steiner tree, where the terminals are not fixed but can be chosen arbitrarily from a given set of vertices each one contributing a certain profit.

Two aims of this thesis were:

- To construct a part of the algorithmic framework to solve large and difficult instances of PCST to optimality within reasonable running time. The method of choice is a branch-and-cut approach based on an ILP formulation depending on connectivity inequalities which can be written as cuts between an artificial root and every selected customer vertex. While the choice of the ILP model is essential for the success of our method, it should also be pointed out that solving the basic ILP model by a default algorithm is by no means sufficient to reach reasonable results. Indeed, our experiments show that a satisfying performance can be achieved only by appropriate initialization and strengthening of the original ILP formulation and in particular by a careful analysis of the separation procedure.

- To develop an efficient metaheuristic approach that finds suboptimal solutions for very difficult problem instances, where the exact approach yields only lower bounds without finding any feasible solution. For this purpose we developed a memetic algorithm that incorporates a local improvement subroutine that solves the problem on trees to optimality. Furthermore, the algorithm is based on the efficient edge-set encoding and comprises problem-dependent variation operators.

In our computational results we have shown that the memetic algorithm is of an order of magnitude faster than the previous best known metaheuristic approach for the PCST. The quality of solutions found was on average not worse than 1% (of optimum) per group. Furthermore, using our ILP approach, we managed to solve to optimality (even without the usual preprocessing) all instances from the literature in a few seconds thereby deriving new optimal solution values and new certificates of optimality for a number of problems previously attacked. For these instances, the ILP approach was also significantly faster than the memetic algorithm itself.

For a number of new large instances constructed from Steiner tree instances, we also derived optimal solutions within reasonable running time. For these instances with more than 60 000 edges, our advanced preprocessing procedure proved to be an indispensable tool for still finding the optimum without branching.

We also tested real-world instances arising in the design of fiber optic networks. Even these instances with up to 1 825 vertices and 214 095 edges we succeeded to solve to optimality, but only after reducing them by applying the preprocessing. In the worst case, our ILP approach

needed about 12 hours, which is, for off-line network design problems, still considered to be a reasonable running time.

The so-called hypercube instances were the final performance test for our algorithms. The built-in difficulty of these artificial instances for the standard Steiner tree problem carries over in a natural way to PCST. For these cases, we used upper bounds found by the memetic approach to initialize the branch-and-cut approach. We added a primal heuristic to our framework to improve the upper bound in each node of the branch-and-cut tree. Our results show that, in some cases, this heuristic can dramatically improve the best feasible solutions found.

For the multi-commodity flow based formulation of the problem, we proposed a column generation algorithm as a lower bounding procedure to solve it. To speed up the pricing process we used the sparse and reserve graph strategy. To initialize sparse and reserve graphs, we suggested to use the results obtained after running the MA. Computational results for three different pricing strategies are presented. Due to a possibility to formulate the PCST by using efficiently separable connectivity constraints, the (MCF) formulation does not show to be a preferable one for the PCST. However, for some other difficult COPs, where column generation represents an essential part of the algorithmic framework, the usage of memetic algorithms within pricing represents a new and promising approach (see [137] for the *two-dimensional bin-packing* problem, for example).

Chapter 5

Discussion and Extensions

We developed metaheuristic and exact approaches for two combinatorial optimization problems that belong to the class of network design problems. The first one, vertex biconnectivity augmentation, appears in the design of survivable communication or electricity networks. The second problem, the prize-collecting Steiner tree problem, describes a natural trade-off between maximizing the sum of profits over all selected customers and minimizing the realization costs when designing a fiber optic or a district heating network.

For the selected problems, the aim of this thesis was to develop tools that find feasible high-quality solutions of practical relevance within reasonable running time. For this purpose, we developed new memetic algorithms (MAs) based on novel solution representation techniques, search operators, constraint handling techniques, local-improvement strategies, and heuristic biasing methods.

Another goal was to provide methods that enable us to estimate the quality of the heuristic solutions we obtained. Therefore, we proposed new branch-and-cut algorithmic frameworks that provided optimal values or, in case of exhausted computational resources, lower bounds that were used to determine optimality gaps for MA solutions.

Finally, we also investigated some possibilities of combining promising variants of exact algorithms and MAs, like incorporating exact algorithms that solve some special cases within MAs, biasing primal heuristics or guiding column generation using MA results. Our study on combinations between exact and memetic algorithms represents a pioneering work in this field that should lead to a better understanding of both, evolutionary and exact approaches. The purpose of this thesis was also to instantiate better interactions between these, so far independently pursued streams.

Vertex Biconnectivity Augmentation (V2AUG)

Given a graph $G_0 = (V, E_0)$ and a set of possible augmentation edges $E \setminus \{E_0\}$, our goal is to augment G_0 with a cheapest subset of augmentation edges, $A \subset E \setminus \{E_0\}$, such that $G_A = (V, E_0 \cup A)$ is biconnected.

Our optimization algorithms rely on the block-cut tree and block-cut graph data structures that are obtained after running a new deterministic preprocessing procedure. We also derived

additional data structures that enhanced the performance of the basic MA operators and of the primal heuristic. Our computational experiments on instances from the literature and on a set of newly generated instances have shown that the preprocessing procedure reduced substantially the size of input graphs.

Our memetic algorithm for V2AUG guarantees local optimality with respect to the number of augmentation edges of any candidate solution. This was achieved by applying a local improvement procedure after initialization, recombination, and mutation. The memetic algorithm usually dominates the total computation time, while the preprocessing does not influence it. Within the memetic algorithm, local improvement dominates the computational costs. The theoretical worst-case time complexity for local improvement of a single solution is $O(|V|^3)$. However, we argued that the expected computational costs are substantially smaller. The computational study supported this and showed that our new memetic algorithm derived the best feasible solutions, when compared to three existing approaches for V2AUG. Compared to a previously developed genetic algorithm, our MA provided significant improvements in terms of running time and quality of solutions. Although there is a very fast approximation algorithm for V2AUG proposed by Khuller and Thurimella in [95], the quality of obtained solutions was typically lower than the quality of MA solutions.

Our branch-and-cut algorithm relies on an ILP formulation which comprises connectivity constraints that assure vertex-biconnectivity of the augmented graph. The exact algorithm relies on a separation procedure that runs in polynomial time. Our computational experiments have shown that the branch-and-cut algorithm can be faster than the MA itself, when applied to small and randomly generated instances. We compared two separation approaches: one, in which only single violated vertex-biconnectivity cuts are added before the LP is resolved, and a second one that adds all violated vertex-biconnectivity constraints before resolving the LP. Computational experiments have shown that the latter represents an advantageous strategy that allows significant savings in the total number of solved LPs. We also studied the role of edge-connectivity constraints within the separation procedure. We learned that the edge-connectivity separation can significantly reduce the total number of inserted cuts, thus having a great impact on the total running time.

For solving larger benchmark instances we extended the proposed branch-and-cut algorithm with a column generation procedure. Our results indicate that the incorporation of pricing represents the only practical way to solve very large instances to proven optimality. Furthermore, we proposed a primal heuristic in which we restricted the set of augmentation edges only to those edges whose fractional value is greater than a certain threshold value. Our results have shown that this threshold value may have a great influence on the performance of the whole branch-and-cut-and-price (BCP) algorithm. We also compared the performance of two different MA settings that are used for the initialization of upper bounds. The collected results have shown that for the instances of moderate size the time needed to instantiate the upper bounds dominates the total BCP running time. We have also seen that for these instances the quality of upper bounds does not influence the rest of BCP running time. However, when the problem size becomes larger, it is recommended to run the MA to obtain solutions that are as good as possible in order to reduce the optimality gaps.

Finally, we investigated the performance of the branch-and-cut-and-price algorithm if, instead of using nearest-neighbor graphs, the MA solutions are used within the pricing procedure. The obtained results have shown that both pricing approaches are competitive, and that none of them is significantly better than the other in terms of running time.

The Prize-Collecting Steiner Tree Problem The Prize-Collecting Steiner Tree Problem (PCST) on a graph $G = (V, E)$ with edge costs, $c : E \mapsto \mathbb{R}^+$, and vertex profits, $p : V \mapsto \mathbb{R}^+$, asks for a subtree minimizing the sum of the total cost of all edges in the subtree plus the total profit of all vertices **not** contained in the subtree. PCST appears frequently in the design of fiber optic or utility networks where customers and the network connecting them have to be chosen in the most profitable way.

Before starting the optimization, we proposed running a preprocessing procedure that removes redundant edges or vertices from the input graph. The procedure generalizes tests proposed by Duin and Volgenant in [41] for the node-weighted Steiner tree problem.

We first proposed a memetic algorithm in which all individuals of the population represent local optima with respect to their subtrees. This is ensured by applying a linear-time local improvement algorithm that solves the PCST on trees to optimality. To enhance our problem-dependent variation operators, we proposed a clustering procedure that groups the subsets of vertices and allows insertion or deletion of all of them at once. We tested the MA against a multi-start local-search-based algorithm with perturbations developed by Canuto et al. in [23]. Extensive experiments on the benchmark instances used also in [23] have shown that the MA compares favorably to previously published results. While the solution values were almost always the same, we achieved substantial reductions of running time.

Our next contribution is the formulation of an integer linear program on a directed graph model based on connectivity inequalities corresponding to edge-cuts in the graph. The main advantage of this model is the efficient separation of sets of violated inequalities by a maximum flow algorithm. Moreover, we introduced new asymmetry constraints that reject multiple consideration of the same solution. Our new approach managed to solve all benchmark instances from the literature to optimality, including eight for which the optimum was previously not known. Compared to a recent algorithm by Lucena and Resende [111], our new method is faster by more than two orders of magnitude. For these instances, the ILP approach was also significantly faster than the memetic algorithm itself. Furthermore, we introduced a new class of larger randomly generated instances and reached optimal results for all of them.

A particularly interesting high-tech application arises in the planning of the access net domain (last mile) of fiber optic networks, which can be modeled as a PCST. We tested modified real-world instances obtained from the German company NetCologne (used for the extension of existing fiber optic networks). Even these instances with up to 1 825 vertices and 214 095 edges were successfully solved to provable optimality, but only after reducing them by applying the preprocessing procedure. In the worst case, our ILP approach needed about 12 hours, which is, for off-line network design problems, still considered to be a reasonable running time.

The so-called hypercube instances [147] were the final performance test for our MA and

ILP algorithms. The built-in difficulty of these artificial instances for the standard Steiner tree problem carries over in a natural way to PCST. For these cases, we used upper bounds found by the memetic approach to initialize the branch-and-cut approach. We added a primal heuristic to our framework to improve the upper bound in each vertex of the branch-and-cut tree. Our results have shown that in some cases this heuristic can dramatically improve the best feasible solutions found. For the largest hypercube instances, the only feasible solutions are obtained by running the MA, while our ILP approach provided only lower bounds.

For the solution of the multi-commodity flow (MCF) based formulation of the problem, we proposed a column generation algorithm as a lower bounding procedure. To speed up the pricing process we used the sparse and reserve graph strategy. As for V2AUG, we suggested to use the results obtained from running the MA to initialize sparse and reserve graphs. We presented computational results for three different pricing strategies. Because it is possible to formulate the problem by using efficiently separable connectivity constraints, the (MCF) formulation turns out not to be preferable for the PCST. However, for some other difficult COPs, where column generation represents an essential part of the algorithmic framework, the use of memetic algorithms within pricing represents a new approach with promising results (see [137] for the *two-dimensional bin-packing* problem, for example).

Finally, we learned that the choice of the ILP model was essential for the success of our method, but also that solving the basic ILP model by a default algorithm was by no means sufficient to reach reasonable results. Indeed, our experiments have shown that a satisfying performance could be achieved only by appropriate initialization and strengthening of the original ILP formulation and in particular by a careful analysis of the separation procedure.

Possible Extensions

Fractional and Piecewise Linear PCST

In the design of district heating networks, the energy companies are often interested not in maximizing the absolute gain of a project but rather in maximizing the return on investment (RoI). The corresponding problem, the so-called *fractional prize-collecting Steiner tree problem* (FPCST) [98] (see also Section 4), consists of maximizing the ratio of profits over costs:

$$\max \frac{\sum_{v \in V_T} p(v)}{c_0 + \sum_{e \in E_T} c(e)} \tag{5.1}$$

over all subtrees T of G, where $c_0 > 0$ represents the fixed cost of the project, e.g., the setup costs of the heating plant in our application.

Using Newton's iterative method (cf. Radzik [138]), each FPCST instance can be also solved to proven optimality, provided the corresponding linear rooted PCST instances can be solved to optimality. Thus, our branch-and-cut approach for PCST directly influences the performance of the exact algorithm for FPCST. Furthermore, for those instances where the branch-and-cut represents a bottleneck, we believe that the combination with our MA for solving linear PCST may yield a new efficient heuristic method for FPCST.

Another interesting problem arising in the design of district heating networks can be modeled as the so-called *piecewise linear prize-collecting Steiner tree problem* (PWPCST). In practical applications it is often the case that fixed implementation costs of building a pipe are not linearly dependent on its length. Furthermore, additional non-linear transportation costs dependent on the amount of flow through each particular pipe, need to be taken into consideration. If these cost functions are concave, we can approximate them by corresponding sets of linear functions and solve the PWPCST using mixed integer programming methods. Flow-based formulations of the linear PCST (see Section 4.4.2) can easily be extended to model the piecewise PCST. Finally, we strongly believe that the pricing strategy based on heuristic results, which we used for V2AUG and PCST, can also be applied for improving the performance of the exact algorithm for PWPCST.

$k_{\max}$-Survivability Network Design Problem

The access net domain (last mile) still represents the bottleneck of present fiber optic architectures [10]. The minimization of implementation costs of fiber optic networks within urban areas under some new problem-specific constraints is of substantial interest for most of the network carriers.

One of these minimization problems, the so-called $k_{\max}$ *survivable network design problem* ($k_{\max}$-SNDP), represents a relaxation of the well-known $\{0,1,2\}$ *survivable network design problem* ($\{0,1,2\}$-SNDP) [152]. In this problem we search for a compromise between the full survivability of a network against single link failures (edge-biconnectivity) and the minimization of implementation costs. In $k_{\max}$-SNDP a failure of a single connection is allowed if it does not destroy "the core" of the network. Hence, survivability of a connection between two vertices may be sacrificed if the diameter of the subnetwork that becomes disintegrated, is not longer than a given value $k_{\max}$.

The problem can be modeled as follows: A simple graph $G = (V, E)$ models the network with its set of vertices V, while its set of edges E corresponds to possible link connections. Each edge $e \in E$ has a fixed cost $c_e > 0$ representing the cost of establishing the direct link connection. The cost of establishing a network $N = (V_N, E_N)$ consisting of a subset $E_N \subset E$ of edges is given by $c(E_N) = \sum_{e \in E_N} c_e$, i.e. it is the sum of the costs of all individual links contained in E_N. Furthermore, each edge $e \in E$ has a fixed length $l_e > 0$. For a path $P \subset E_N$, its *length* is defined as $\sum_{e \in P} l_e$.

As in the standard survivable network design problem, we are dealing with the following three types of vertices:

- *special* vertices, for which a "high" degree of survivability has to be ensured in the network to be constructed;
- *ordinary* vertices, which only have to be connected to the network;
- *optional* vertices, which may or may not be a part of network at all. They directly correspond to the Steiner vertices of the classical Steiner tree problem.

The type of each vertex is modeled by an associated *survivability requirement* value $r_v \in \{0, 1, 2\}$. Special vertices are represented by vertices of type 2, while ordinary and optional vertices have type 1 and 0, respectively. In the standard $\{0, 1, 2\}$-SNDP, a network $N = (V_N, E_N)$ is said to satisfy *edge-connectivity requirements* if, for each pair $u, v \in V_N$ of distinct vertices, N contains at least

$$r_{uv} = \min\{r_u, r_v\}$$

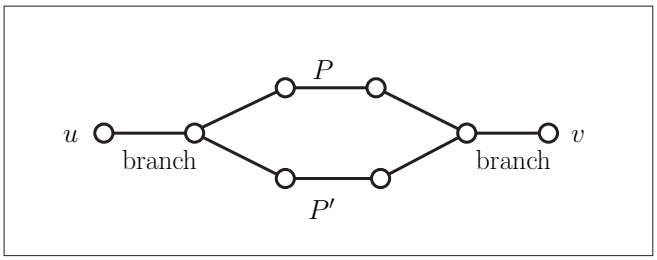
edge-disjoint paths.

For economical reasons, in $k_{\max}$-SNDP, we relax the edge-connectivity requirements and introduce a parameter $k_{\max} \geq 0$, the so-called *maximal branch length*. Therefore, our goal is to build a network of minimum cost such that for each two vertices u and v:

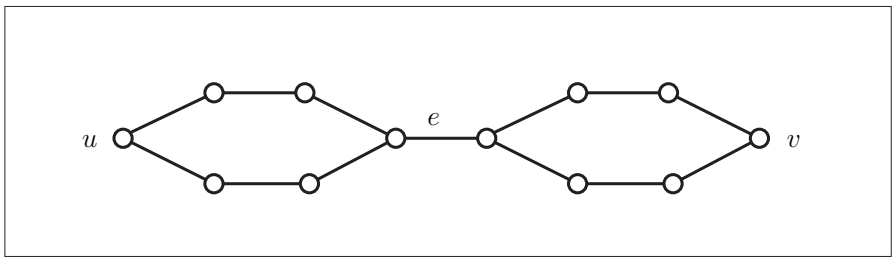
- if $r_{uv} = 1$, there is a path connecting u and v,
- if $r_{uv} = 2$, there are at least two edge-disjoint paths $P \subset E$ and $P' \subset E$ connecting u and v, with the only exception of sub-paths not longer than $k_{\max}$ emanating from u or v, where P and P' may coincide.

It is easy to see that 0-SNDP directly corresponds to the $\{0, 1, 2\}$-SNDP. Typically, a network in which all customers have $r_v = 2$ consists of an edge-biconnected component and paths attached to it. These paths are called *branches*. Some examples are given in Figure 5.1.

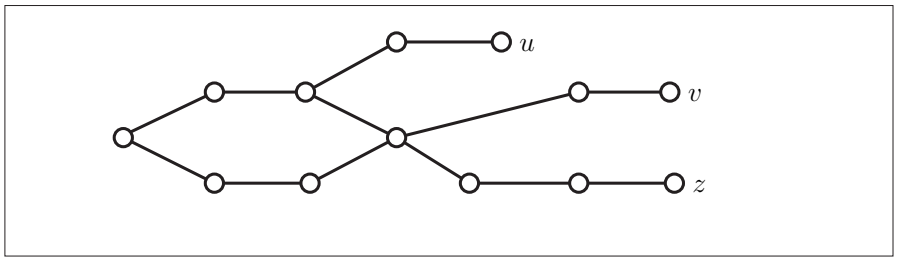
The $k_{\max}$-SNDP is a fresh problem for which there are a lot of open questions. To our knowledge, the only known heuristic approach is published in [10]. We believe that the methods proposed in this thesis may be successfully modified and applied to $k_{\max}$-SNDP. It is certainly promising to adapt our memetic approach in order to obtain high quality solution of the problem. Furthermore, a study about solving some special cases to optimality in polynomial time represents another interesting field of research. It is also important to investigate the problem-specific polytope in order to obtain a tighter characterization of the problem. Finally, we are convinced that our new branch-and-cut frameworks for V2AUG and PCST are flexible and powerful enough to build the basis of further successful and competitive combinatorial algorithms for $k_{\max}$-SNDP and similar survivability network design problems.



(a)



(b)



(c)

Figure 5.1: Examples of the 3-SNDP: Assume all edges have length $l_e = 1$, and all vertices $r_v = 2$. (a) A feasible solution. P and P' are two paths connecting u and v. There are two branches induced by $P \cap P'$, one that contains u and the other one that contains v. The length of both branches is one, thus the solution is feasible. (b) An infeasible solution. The bridge between two edge biconnected components contains neither u nor v. (c) A feasible solution. There are two edge-disjoint paths connecting v and z, but they have a common vertex. There are three branches and none of them is longer than 3, thus the solution is feasible.

Appendix A

Curriculum Vitae

- **Personal Information**
  - Title: Mag.rer.nat.
  - Date and place of birth: October 1, 1973, Prokuplje, Serbia & Montenegro
  - Citizenship: Serbia & Montenegro
  - Marital status: married since 1997 with Dr. Dragoslav Ljubić
  - Children: Natalija (1997) and Čedomir (1999)
  - Languages: Serbo-croatian (mother tongue), English (fluent), German (fluent), Russian, Spanish
  - URL: `www.ads.tuwien.ac.at/~ivana/`
- **Education**
  - **2000-2004:** PhD studies of computer science at the Faculty of Informatics, Vienna University of Technology, Austria. Supervisors: Prof. Petra Mutzel and Prof. Ulrich Pferschy.
  - **February 2000:** Graduation with distinction to Master of Science degree (MSc). Master thesis: *An Application of Genetic Algorithms on Connectivity Problems in Graphs.* Examiners: Prof. Djordje Dugošija, Prof. Dušan Tošić, Prof. Vera Kovačević - Vujčić.
  - **1996-2000:** Postgraduate studies at the Department for Optimization and Numerical Analysis, Faculty of Mathematics, University of Belgrade, Serbia & Montenegro.
  - **September 1996:** Graduation with distinction to Bachelor of Science degree (BSc).
  - **1992-1996:** Studies of Computer Science at the Faculty of Mathematics, University of Belgrade and at the Faculty of Science and Mathematics, University of Niš (1992-1994), Serbia & Montenegro.
  - **Jun 1992:** High school graduation (Matura).

  * **1988-1992:** High school *Bora Stanković*, Niš and *Gimnazija Prokuplje*, Prokuplje (1988-1990), Serbia & Montenegro.
  * **1980-1988:** Primary school, Prokuplje, Serbia & Montenegro.

- **Scholarships and Awards**

  * **Summer 2004:** Student Summer Research Grant of IEEE Neural Networks Society: cooperation with the University of La Laguna, Spain.
  * **2003-2004:** Doctoral Scholarship Programme of the Austrian Academy of Sciences (Doktorandenprogramm der Österreichischen Akademie der Wissenschaften, DOC).
  * **Summer 2001:** Student Summer Research Grant of IEEE Neural Networks Society.
  * **1994-1996:** Scholarship of the CIP - Institute of Transportation, Belgrade, Serbia & Montenegro.

- **Work and Teaching Experience**

  * **2000-present:** Research and teaching assistant at the Institute of Computer Graphics and Algorithms, Vienna University of Technology (in the framework of FWF Project 13602-INF and DOC Scholarship). In charge of teaching numerous tutorials, practical courses, and two diploma thesis.
  * **1996-2000:** Research and teaching assistant at the Department of Mathematics, Faculty of Civil Engineering, University of Belgrade. In charge of the exercises for the lectures *Mathematics I* and *Mathematics II*.
  * **Projects:**
    - *Hybrid Evolutionary Algorithms for Selected Graph Problems*, Project No. 13602-INF funded by Austrian Science Foundation (FWF) (02/2000-09/2000 and 08/2001-01/2003). Project leader Dr. Günther Raidl.
    - *Mathematical optimization models and methods with applications*, Project No. 1583 of Institute of Mathematics, Serbian Academy of Science, Serbia & Montenegro (01/2002-present, adjunct member). Project leader Dr. Nenad Mladenović.
    - *Combinatorial and Memetic Algorithms for Selected Network Design Problems* in the framework of DOC programme.

- **Publications**

  Refereed Conference Papers

  1. *Combining a Memetic Alorithm with Integer Programming to Solve the Prize-Collecting Steiner Tree Problem*, with G. W. Klau et al, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2004, volume 3102 of LNCS, pages 1304–1315, Springer, 2004.

2. *The Fractional Prize-Collecting Steiner Tree Problem on Trees*, with G. Klau, P. Mutzel, U. Pferschy, R. Weiskircher, ESA 2003, volume 2832 of LNCS, pages 691-702, Springer, 2003.
3. *A genetic algorithm for the index selection problem*, with J. Kratica and D.Tošić, Proceedings of EvoWorkshops 2003, volume 2611 of LNCS, pages 281-291, Springer, 2003.
4. *A memetic algorithm for vertex-biconnectivity augmentation*, with S. Kersting and G. R. Raidl, in S. Cagnoni et al., editors, Applications of Evolutionary Computing: EvoWorkshops 2002, volume 2279 of LNCS, pages 102-111. Springer, 2002.
5. *An evolutionary algorithm with hill-climbing for the edge-biconnectivity augmentation problem*, with G. R. Raidl, In E. J.-W. Boers et al., editors, Applications of Evolutionary Computing: EvoWorkshops 2001, volume 2037 of LNCS, pages 20-29, Springer, 2001
6. *A genetic algorithm for the uncapacitated network design problem*, with J. Kratica, D. Tošić and V. Filipović, in R. Roy et al., editors, Soft Computing in Industry - Recent Applications, Engineering series, pages 329-338. Springer, 2001.
7. *A hybrid GA for the edge-biconnectivity augmentation problem*, with G. R. Raidl, and J. Kratica, In K. Deb, G. Rudolph, X. Yao, and H.-P. Schwefel, editors, Proceedings of the 2000 Parallel Problem Solving from Nature VI Conference, volume 1917 of LNCS, pages 641-650, Springer, 2000.
8. *Fine grained tournament selection for the simple plant location problem*, with V. Filipovic, J. Kratica and D. Tošić, In 5th Online World Conference on Soft Computing Methods in Industrial Applications, WSC5, pages 152-158, Internet, 2000. ISBN: 951-22-5205-8.
9. *Genetic algorithm for designing a spread-spectrum radar polyphase code*, with J. Kratica, V. Filipovic and D. Tošić, In 5th Online World Conference on Soft Computing Methods in Industrial Applications, WSC5, pages 191-197, Internet, 2000. ISBN: 951-22-5205-8.
10. *A genetic algorithm for the biconnectivity augmentation problem*, with J. Kratica, in C. Fonseca et al., editors, Proceedings of the 2000 IEEE Congress on Evolutionary Computation, pages 89-96, IEEE Press, 2000.
11. *Some Methods for Solving the Traveling Salesman Problem by Genetic Algorithms*, with J. Kratica, V. Šešum, and V. Filipović, In M. M. Klarin, J. M. Cvijanović, and D. D. Milanović, editors, Proceedings of the 2nd International Symposium of Industrial Engineering, pages 281-284, 1998, in serbian.
12. *Application of Genetic Algorithms on the Minimum Steiner Tree Problem*, with J. Kratica, and V. Filipović, In M. M. Klarin, J. M. Cvijanović, and D. D. Milanović, editors, Proceedings of the 2nd International Symposium of Industrial Engineering, pages 277-280, 1998, in serbian.

Journal Articles

1. *A memetic algorithm for vertex-biconnectivity augmentation*, with G. R. Raidl, Journal of Heuristics 9(5):401-428, 2003.
2. *Evolutionary local search for the edge-biconnectivity augmentation problem*, with G. R. Raidl, Information Processing Letters, 82(1):39-45, 2002.
3. *Solving the simple plant location problem by genetic algorithms*, with J. Kratica, D. Tošić and V. Filipović, RAIRO - Operations Research, 35(1):127-142, 2001.

Technical Reports

1. *A branch-and-cut algorithm for the prize-collecting Steiner tree problem*, with R. Weiskircher et al., Report No. TR 186-1-04-01, Vienna University of Technology, Austria, 2004.

Bibliography

[1] K. Aardal and S. van Hoesel. Polyhedral techniques in combinatorial optimization I: Theory. *Statistica Nederlandica*, 50:3–26, 1996.

[2] K. Aardal and S. van Hoesel. Polyhedral techniques in combinatorial optimization II: Applications and computations. *Statistica Nederlandica*, 53:129–178, 1999.

[3] E. Aarts, J. Korst, and P. van Laarhoven. Simulated annealing. In Aarts and Lenstra [4], pages 91–120.

[4] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, New York, USA, 1997.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[6] E. Alba and B. Dorronsoro. Solving the vehicle routing problem by using cellular genetic algorithms. In Gottlieb and Raidl [68], pages 11–20.

[7] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding tours in the TSP. Technical Report 99885, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1999.

[8] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer Verlag, 1999.

[9] P. Bachhiesl. Carinthia Tech Institute, Klagenfurt, Austria. Personal communication.

[10] P. Bachhiesl, M. Prossegger, G. Paulus, J. Werner, and H. Stögner. Simulation and optimization of the implementation costs for the last mile of fiber optic networks. *Networks and Spatial Economics*, 3(4):467–482, 2003.

[11] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

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

[13] E. Balas. The prize-collecting traveling salesman problem. *Networks*, 19:621–636, 1989.

[14] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, algorithms and applications*. Springer, 2001.

[15] C. Barnhart, E. Johnson, G. Nemhauser, M. Savelsbergh, and P. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[16] J. E. Beasley. An SST-based algorithm for the Steiner problem in graphs. *Networks*, 19:1–16, 1989.

[17] D. Bertsimas and J. T. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, USA, 1997.

[18] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[19] C. Buchheim and M. Jünger. Detecting symmetries by Branch & Cut. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 178–188. Springer, 2002.

[20] L. S. Buriol, P. M. França, and P. Moscato. A new memetic algorithm for the asymmetric traveling salesman problem. *Journal of Heuristics*, 10(5), 2004. to appear.

[21] L. Caccetta and S. Hill. A branch and cut method for the degree-constrained minimum spanning tree problem. *Networks*, 37(2):74–83, 2001.

[22] S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. R. Raidl, editors. *Applications of Evolutionary Computing: EvoWorkshops 2002*, volume 2279 of *LNCS*. Springer, 2002.

[23] S. A. Canuto, M. G. C. Resende, and C. C. Ribeiro. Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks*, 38:50–58, 2001.

[24] J. Cheriyan and R. Thurimella. Approximating minimum-size k-connected spanning subgraphs via matching. *SIAM Jornal on Computing*, 30:528–560, 2000.

[25] J. Cheriyan, S. Vempala, and A. Vetta. An approximation algorithm for the minimum-cost k-vertex connected subgraph. In *Proceedings of the 34th ACM Symposium on the Theory of Computing (STOC)*, pages 306–312. ACM, Montréal, Canada, 2002.

[26] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[27] S. Chopra, E. Gorres, and M. R. Rao. Solving a Steiner tree problem on a graph using a branch and cut. *ORSA Journal on Computing*, 4:320–335, 1992.

[28] S. Chopra and M. R. Rao. The Steiner tree problem I: Formulations, compositions and extension of facets. *Mathematical Programming*, 64:209–229, 1994.

[29] T. Christof, M. Jünger, J. Kececioglu, P. Mutzel, and G. Reinelt. A branch-and-cut approach to physical mapping of chromosomes by unique end-probes. *Journal of Computational Biology*, 4(4):433–447, 1997.

[30] N. Christofides and C. A. Whitlock. Network synthesis with connectivity constraints - a survey. In *Proceedings of the 9th IFORS International Conference*, pages 705–723. Hamburg, Germany, 1981.

[31] L. W. Clarke and G. Anandalingam. A bootstrap heuristic for designing minimum cost survivable networks. *Computers & Operations Research*, 22:921–934, 1995.

[32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[33] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization*. McGraw Hill, Berkshire, England, 1999.

[34] C. Cotta. Scatter search and memetic approaches to the error correcting code problem. In Gottlieb and Raidl [68], pages 51–61.

[35] C. Cotta, A. S. Mendes, V. García, P. França, and P. Moscato. Applying memetic algorithms to the analysis of microarray data. In G. R. Raidl, S. Cagnoni, J. J. R. Cardalda, D. W. Corne, J. Gottlieb, A. Guillot, E. Hart, C. G. Johnson, E. Marchiori, J.-A. Meyer, and M. Middendorf, editors, *Applications of Evolutionary Computing: EvoWorkshops 2003*, volume 2611 of *LNCS*, pages 22–32. Springer, 2003.

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

[37] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, 2:393–410, 1954.

[38] J. J. Dongarra. Performance of various computers using standard linear equations software (linpack benchmark report). Technical Report CS-89-85, University of Tennessee, 2004.

[39] M. Dorigo and G. D. Caro. The ant colony optimization meta-heuristic. In Corne et al. [33], pages 11–32.

[40] C. Duin. *Steiner's Problem in Graphs*. PhD thesis, University of Amsterdam, 1993.

[41] C. W. Duin and A. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17(2):353–364, 1987.

[42] J. Edmonds. Submodular functions, matroids and certain polyhedra. In R. Guy, H. Hanani, N. Sauer, and J. Schönheim, editors, *Combinatorial Structures and Their Application*, pages 69–87. Gordon and Breach, 1970.

[43] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In Jünger and Naddef [88], pages 157–222.

[44] S. Engevall, M. Göthe-Lundgren, and P. Värbrand. A strong lower bound for the node weighted Steiner tree problem. *Networks*, 31(1):11–17, 1998.

[45] K. P. Eswaran. Representation of graphs and minimally augmented Eulerian graphs with applications in data base management. Technical Report RJ 1305, IBM, Yorktown Heights, N.Y., 1973.

[46] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[47] P. Feofiloff, C. Fernandes, C. Ferreira, and J. Pina. Primal-dual approximation algorithms for the prize-collecting Steiner tree problem. 2003. submitted.

[48] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1998.

[49] G. R. Filho and L. A. N. Lorena. Constructive genetic algorithm and column generation: an application to graph coloring. In *Proceedings of APORS 2000 - The Fifth Conference of the Association of Asian-Pacific Operations Research Societies within IFORS*, 2000.

[50] M. Finger, T. Stützle, and H. Lourenço. Exploiting fitness distance correlation of set covering problems. In Cagnoni et al. [22], pages 61–71.

[51] M. Fischetti. Facets of two Steiner arborescence polyhedra. *Mathematical Programming*, 51:401–419, 1991.

[52] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, New York, 1995.

[53] C. Fonseca, J.-H. Kim, and A. Smith, editors. *Proceedings of the 2000 IEEE Congress on Evolutionary Computation*. IEEE Press, 2000.

[54] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[55] B. Fortz. *Design of Survivable Networks with Bounded Rings*. Network Theory and Applications. Kluwer Academic Publishers, Universitè Libre de Bruxelles, Bruxelles, Belgium, 2000.

[56] G. N. Frederickson and J. Jájá. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

[57] G. N. Frederickson and J. Jájá. On the relationship between the biconnectivity augmentation and traveling salesman problems. *Theoretical Computer Science*, 19(2):203–218, 1982.

[58] A. P. French, A. C. Robinson, and J. M.Wilson. Using a hybrid genetic-algorithm/branch and bound approach to solve feasibility an optimization integer programming problems. *Journal of Heuristics*, 7:551–564, 2001.

[59] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. System Sci.*, 50:259–273, 1995.

[60] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[61] A. Galluccio and G. Proietti. Polynomial time algorithms for edge-connectivity augmentation of Hamiltonian paths. In P. Eades and T. Takaoka, editors, *ISAAC*, volume 2223 of *Lecture Notes in Computer Science*, pages 345–354. Springer, 2001.

[62] A. Galluccio and G. Proietti. Polynomial time algorithms for 2-edge-connectivity augmentation problems. *Algorithmica*, 36(4):361–374, 2003.

[63] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[64] F. W. Glover and G. A. Kochenberger, editors. *Handbook of Metaheuristics*. Kluwer Academic Publishers, Boston, 2003.

[65] M. X. Goemans. The Steiner tree polytope and related polyhedra. *Mathematical Programming*, 63:157–182, 1994.

[66] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 144–191. P. W. S. Publishing Co., 1996.

[67] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[68] J. Gottlieb and G. R. Raidl, editors. *Evolutionary Computation in Combinatorial Optimization, 4th European Conference, EvoCOP 2004, Coimbra, Portugal, April 5-7, 2004, Proceedings*, volume 3004 of *Lecture Notes in Computer Science*. Springer, 2004.

[69] J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. V. Gucht. Genetic algorithms for the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 160–168. Lawrence Erlbaum, 1985.

[70] M. Grötschel and O. Holland. Solving matching problems with linear programming. *Mathematical Programming*, 33:243–259, 1985.

[71] M. Grötschel, C. Monma, and M. Stoer. Polyhedral and computational investigations for designing communication networks with high survivability requirements. *Operations Research*, 43(6):1012–1024, 1995.

[72] D. Gusfield. A graph theoretic approach to statistical data security. *SIAM Journal on Computing*, 17(3):552–571, 1988.

[73] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 246–255. ACM-SIAM, 2001.

[74] J. Hackner. *Energiewirtschaftlich optimale Ausbauplanung kommunaler Fernwärmesysteme*. PhD thesis, Vienna University of Technology, Austria, 2004.

[75] P. Hansen and N. Mladenović. Variable neighbourhood search. In Glover and Kochenberger [64], pages 145 –184.

[76] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut of a graph. In *Proceedings of the 3th ACM-SIAM Symposium on Descrete Algorithms*, pages 165–174. ACM, 1992.

[77] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In Aarts and Lenstra [4], pages 121–136.

[78] R. Hinterding. Mapping, order-independent genes and the knapsack problem. In D. Schaffer, H.-P. Schwefel, and D. B. Fogel, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 13–17. IEEE Press, 1994.

[79] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[80] D. Holstein and P. Moscato. Memetic algorithms using guided local search: A case study. In Corne et al. [33], pages 235–243.

[81] T.-S. Hsu. Simpler and faster biconnectivity augmentation. *Journal of Algorithms*, 45(1):55–71, 2002.

[82] T. Ishii. *Studies on Multigraph Connectivity Augmentation Problems*. PhD thesis, Dept. of Applied Mathematics and Physics, Kyoto University, Kyoto, Japan, 2000.

[83] D. Johnson. Local optimization and the traveling salesman problem. In M. Paterson, editor, *Proceedings on 17th Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 446–461, Berlin, 1990. Springer.

[84] D. S. Johnson, M. Minkoff, and S. Phillips. The prize-collecting Steiner tree problem: Theory and practice. In *Proceedings of 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 760–769, San Francisco, CA, 2000.

[85] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann, 1995.

[86] M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

[87] M. Jünger and P. Mutzel. 2-Layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications (JGAA)*, 1(1):1–25, 1997.

[88] M. Jünger and D. Naddef, editors. *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions [based on a Spring School, Schloß Dagstuhl, Germany, 15-19 May 2000]*, volume 2241 of *Lecture Notes in Computer Science*. Springer, 2001.

[89] M. Jünger, G. Reinelt, and S. Thienel. Provably good solutions for the traveling salesman problem. *Zeitschrift für Operations Research*, 40:183–217, 1994.

[90] M. Jünger, G. Reinelt, and S. Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

[91] M. Jünger and S. Thienel. Introduction to ABACUS-A Branch-And-CUt System. *Operations Research Letters*, 22:83–95, 1998.

[92] M. Jünger and S. Thienel. The ABACUS system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30(11):1325–1352, 2000.

[93] S. Kersting, G. R. Raidl, and I. Ljubić. A memetic algorithm for vertex-biconnectivity augmentation. In Cagnoni et al. [22], pages 102–111.

[94] S. Khuller and B. Raghavachari. Improved approximation algorithms for uniform connectivity problems. *J. Algorithms*, 21:434–450, 1996.

[95] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.

[96] G. Klau. *A Combinatorial Approach to Orthogonal Placement Problems*. PhD thesis, University of Saarland, Germany, 2002.

[97] G. Klau, I. Ljubić, A. Moser, P. Mutzel, P. Neuner, U. Pferschy, and R. Weiskircher. Combining a memetic algorithm with integer programming to solve the prize-collecting

Steiner tree problem. In K. Deb, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, LNCS, pages 1304–1315. Springer-Verlag, 2004.

[98] G. Klau, I. Ljubić, P. Mutzel, U. Pferschy, and R. Weiskircher. The fractional prize-collecting Steiner tree problem on trees. In G. D. Battista and U. Zwick, editors, *ESA 2003*, volume 2832 of *LNCS*, pages 691–702. Springer-Verlag, 2003.

[99] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[100] G. Kortsarz, R. Krauthgamer, and J. R. Lee. Hardness of approximation for vertex-connectivity network-design problems. In K. Jansen, S. Leonardi, and V. V. Vazirani, editors, *APPROX*, volume 2462 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[101] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.

[102] N. Krasnogor and J. Smith. A memetic algorithm with self-adaptive local search: TSP as a case study. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*, pages 987–994. Morgan Kaufman, 2000.

[103] L. Ladányi, T. Ralphs, and L. Trotter. Branch, cut, and price: Sequential and parallel. In Jünger and Naddef [88], pages 223–260.

[104] M. Laguna and R. Marti. *Scatter Search: Methodology and Implementation in C*. Kluwer, 2003.

[105] A. Z.-Z. Lin, J. Bean, and I. C. C. White. A hybrid genetic/optimization algorithm for finite horizon partially observed markov decision processes. *Journal on Computing*, 16(1):27–38, 2004.

[106] I. Ljubić and J. Kratica. A genetic algorithm for the biconnectivity augmentation problem. In Fonseca et al. [53], pages 89–96.

[107] I. Ljubić and G. R. Raidl. An evolutionary algorithm with hill-climbing for the edge-biconnectivity augmentation problem. In E. J.-W. Boers, S. Cagnoni, J. Gottlieb, E. Hart, P. L. Lanzi, G. R. Raidl, R. E. Smith, and H. Tijink, editors, *Applications of Evolutionary Computing: EvoWorkshops 2001*, volume 2037 of *LNCS*, pages 20–29. Springer, 2001.

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

[109] I. Ljubić, R. Weiskircher, U. Pferschy, G. Klau, P. Mutzel, and M. Fischetti. Solving the prize-collecting Steiner tree problem to optimality. Technical Report TR 186–1–04–01, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004.

[110] H. R. Lourenco, O. Martin, and T. Stützle. Iterated local search. In Glover and Kochenberger [64], pages 321–353.

[111] A. Lucena and M. G. C. Resende. Strong lower bounds for the prize-collecting Steiner problem in graphs. *Discrete Applied Mathematics*, 2003. to appear.

[112] F. Margot, A. Prodon, and T. M. Liebling. Tree polyhedron on 2-tree. *Mathematical Programming*, 63:183–192, 1994.

[113] A. Marino, A. Prügel-Bennett, and C. A. Glass. Improving graph colouring with linear programming and genetic algorithms. In *Proceedings of Eurogen99*, pages 113–118, 1999.

[114] A. Martin. General mixed integer programming: Computational issues for branch-and-cut algorithms. In Jünger and Naddef [88], pages 1–25.

[115] K. Mehlhorn. A faster approximation for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[116] P. Merz. *Memetic Algorithms for Combinatorial Optimization Problems: Fitness Landscapes and Effective Search Strategies*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, 2000.

[117] P. Merz. On the performance of memetic algorithms in combinatorial optimization. In W. Hart, N. Krasnogor, and J. Smith, editors, *Second Workshop on Memetic Algorithms (2nd WOMA)*, pages 168–173, San Francisco, California, USA, 2001.

[118] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In Corne et al. [33], pages 245–260.

[119] P. Merz and B. Freisleben. Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE Transactions on Evolutionary Computation*, 4(4):337–352, 2000.

[120] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, 1996.

[121] M. Minkoff. The prize-collecting Steiner tree problem. Master's thesis, MIT, May, 2000.

[122] C. L. Monma and D. F. Shallcross. Methods for designing communications networks with certain two-connected survivability constraints. *Operations Research*, 37(4):531–541, 1989.

[123] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, 1989. C3P Report 826, Caltech Concurrent Computation Program.

[124] P. Moscato. Memetic algorithms: A short introduction. In Corne et al. [33], pages 219–234.

[125] P. Mutzel. *The Maximum Planar Subgraph Problem*. PhD thesis, University of Cologne, 1994.

[126] P. Mutzel. A polyhedral approach to planar augmentation and related problems. In P. G. Spirakis, editor, *ESA*, volume 979 of *Lecture Notes in Computer Science*, pages 494–507. Springer, 1995.

[127] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM Journal on Optimization*, 11(4):1065–1080, 2001.

[128] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Descrete Methods*, 5:54–66, 1992.

[129] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1998.

[130] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47:19–36, 1990.

[131] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[132] C. C. Palmer and A. Kershenbaum. Representing trees in genetic algorithms. In Bäck et al. [12], page G1.3:1..G1.3:8.

[133] C. Peterson and B. Söderberg. Artificial neural networks. In Aarts and Lenstra [4], pages 173–214.

[134] T. Polzin and S. V. Daneshmand. A comparison of Steiner tree relaxations. *Discrete Applied Mathematics*, 112:241–261, 2001.

[135] H. J. Prömel and A. Steger. *The Steiner Tree Problem*. Vieweg, 2002.

[136] J. S. Provan and R. C. Burk. Two-connected augmentation problems in planar graphs. *Journal of Algorithms*, 32(2):87–107, 1999.

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

[138] T. Radzik. Newton's method for fractional combinatorial optimization. In *Proceedings of 33rd Annual Symposium on Foundations of Computer Science*, pages 659–669, 1992.

[139] G. R. Raidl. Weight-codings in a genetic algorithm for the multiconstraint knapsack problem. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, A. Zalzala, and W. Porto, editors, *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*, pages 596–603. IEEE Press, 1999.

[140] G. R. Raidl. An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In Fonseca et al. [53], pages 104–111.

[141] G. R. Raidl and H. Feltl. An improved hybrid genetic algorithm for the generalized assignment problem. In H. M. Haddadd et al., editors, *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 990–995. ACM Press, 2004.

[142] G. R. Raidl and J. Gottlieb. On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms. In S. Brave and A. S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 204–211, Orlando, FL, 1999.

[143] G. R. Raidl and B. A. Julstrom. Edge-sets: An effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, 7(3):225–239, 2003.

[144] G. R. Raidl and I. Ljubić. Evolutionary local search for the edge-biconnectivity augmentation problem. *Information Processing Letters*, 82(1):39–45, 2002.

[145] G. Reinelt. Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing*, 4:206–217, 1992.

[146] K. Reinert, H.-P. Lenhof, P. Mutzel, K. Mehlhorn, and J. Kececioglou. A branch-and-cut algorithm for multiple sequence alignment. In *Proceedings of the First Annual International Conference on Computational Molecular Biology, RECOMB '97*, pages 241–249, 1997.

[147] I. Rosseti, M. P. de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. New benchmark instances for the Steiner problem in graphs. In *Extended Abstracts of the 4th Metaheuristics International Conference*, pages 557–561, Porto, Portugal, 2001.

[148] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Chichester, England, 1986.

[149] A. Segev. The node-weighted Steiner tree problem. *Networks*, 17:1–17, 1987.

[150] M. C. Souza, C. Duhamel, and C. C. Ribeiro. A GRASP heuristic for the capacitated minimum spanning tree problem using a memory-based local search strategy. Technical report, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2002.

[151] A. T. Staggemeier, A. R.Clark, U. Aickelin, and J. Smith. A hybrid genetic algorithm to solve a lot-sizing and scheduling problem. In *16th triannual Conference of the International Federation of Operational Research Societies*, Edinburgh, 2002.

[152] M. Stoer. *Design of Survivable Networks*, volume 1531 of *Lecture Notes in Mathematics.* Springer, 1992.

[153] T. Stützle and M. Dorigo. Local search and metaheuristics for the quadratic assignment problem. Technical Report AIDA-01-01, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2001.

[154] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, San Mateo, California, 1989.

[155] G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In *Foundations of Genetic Algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, California, 1991.

[156] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[157] S. Thienel. *A Branch-And-CUt System.* PhD thesis, University of Cologne, Germany, 1995.

[158] C. Voudouris and E. Tsang. Guided local search. Technical Report CSM-247, Department of Computer Science, University of Essex, 1995.

[159] T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *Journal of Computer and System Sciences*, 35(1):96–144, 1987.

[160] L. A. Wolsey. *Integer Programming.* John Wiley, New York, 1998.

[161] A. Zhu. A uniform framework for approximating weighted connectivity problems. B.Sc. thesis at the University of Maryland, MD, May 1999.

[162] A. Zhu, S. Khuller, and B. Raghavachari. A uniform framework for approximating weighted connectivity problems. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 937–938, 1999.

Index