Computational Intelligence Methods and Applications

Christian Blum

Construct, Merge, Solve & Adapt

A Hybrid Metaheuristic for Combinatorial Optimization

Springer

Computational Intelligence Methods and Applications

Founding Editors
Sanghamitra Bandyopadhyay
Ujjwal Maulik
Patrick Siarry

Series Editor

Patrick Siarry, LiSSi, E.A. 3956, Université Paris-Est Créteil, Vitry-sur-Seine, France

The monographs and textbooks in this series explain methods developed in computational intelligence (including evolutionary computing, neural networks, and fuzzy systems), soft computing, statistics, and artificial intelligence, and their applications in domains such as heuristics and optimization; bioinformatics, computational biology, and biomedical engineering; image and signal processing, VLSI, and embedded system design; network design; process engineering; social networking; and data mining.

Christian Blum

Construct, Merge, Solve & Adapt

A Hybrid Metaheuristic for Combinatorial Optimization

Springer

Christian Blum
IIIA-CSIC
Bellaterra, Spain

ISSN 2510-1765 ISSN 2510-1773 (electronic)
Computational Intelligence Methods and Applications
ISBN 978-3-031-60102-6 ISBN 978-3-031-60103-3 (eBook)
https://doi.org/10.1007/978-3-031-60103-3

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

This book is lovingly dedicated to my mother, Maria Blum (1949–2020), whose unwavering emotional support and belief in me have empowered me to confront life's challenges with optimism and resilience.

Preface

The work on CMSA started in 2015 during my years as an Ikerbasque Research Fellow at the University of the Basque Country in San Sebastian. It originated from the observation that large neighborhood search (LNS) algorithms based on the partial destruction of an incumbent solution at each iteration sometimes underperform in the context of optimization problems in which solutions contain rather few solution components. (As an example, think about a multi-dimensional knapsack problem instance with very tight resource constraints.) Our intention, in the context of the Ph.D. thesis of Pedro Pinacho Davidson, was then to develop an alternative hybrid algorithm that would work well in those cases in which LNS showed problems. In the meanwhile, two other Ph.D. students from my group at the IIIA-CSIC in Bellaterra (Barcelona), Mehmet Anıl Akbay and Jaume Reixach, have been working on different aspects of CMSA. Moreover, the initial paper on CMSA (published under the title "Construct, merge, solve & adapt: A new general algorithm for combinatorial optimization", which was published in 2016 in the journal *Computers and Operations Research*, has received 106 citations (Google Scholar, February 2024). Moreover, to date, CMSA has been applied to 20 different combinatorial optimization problems.

I am also happy to say that our work on CMSA has received two awards in recent years. The first one was the *best paper award* at the ECOM track of the GECCO 2016 conference for a paper on the application of CMSA to the multi-dimensional knapsack problem. The second award was the one for *The Best Methodological Contribution in Operations Research* jointly given by the Spanish Society of Statistics and Operations Research (SEIO) and the BBVA Foundation in 2021.

This book aims to give an account of the current state of the research efforts on CMSA. After shortly introducing the general line of research and the tools to be used in the book, the first chapter provides a didactical introduction to standard CMSA in the context of the minimum dominating set problem. In addition, the C++ program code used for part of the experiments presented in this chapter is offered in Appendix A. The following four chapters are dedicated to important CMSA variants (ADAPT_CMSA and LEARN_CMSA), respectively, to important topics for the practical application of CMSA: the use of set-covering-based ILP models for sub-instance solving, and the application of CMSA to optimization problems that are naturally modeled by non-binary ILPs. Finally, the last chapter outlines research lines that have not yet received much attention. Moreover, several avenues for current and future work are described. I believe that this book will be useful and inspiring for everyone who plans to apply CMSA to a specific optimization problem.

I am very grateful to the following people. The main idea of LEARN_CMSA presented in Chap. 3 of this book was contributed by Pedro Pinacho Davidson, who was my Ph.D. student at the University of the Basque Country, and who is nowadays an associate professor at the Universidad de Concepción, Chile. Moreover, Pedro prepared the initial implementation of LEARN_CMSA for the FFMS problem. The idea of ADAPT_CMSA presented in Chap. 2 was developed together with Mehmet Anıl Akbay, who was my Ph.D. student at the time of preparing this book. The same holds for the use of set-covering-based ILP models for sub-instance solving in the context of the EVRP-TW-SPD problem in Chap. 4. Mehmet provided the CMSA implementation for the EVRP-TW-SPD problem. Moreover, some of the text in this chapter was written based on his original texts. I am also grateful to Camilo Chacón Sartori, one of my latest Ph.D. students, who implemented the web application STNWeb for the generation of the nice and informative STN graphics presented in this book. Last but not least, thanks to Guillem Rodríguez, Jaume Reixach, and Camilo Chacón for proofreading (parts of) the book. Many thanks to all of you!

To end, promising research remains to be done in the context of the CMSA algorithm. Together with the optimization group at the IIIA-CSIC in Bellaterra (Barcelona), I will take on this endeavor during the coming years. We certainly hope that other research groups on metaheuristics and their hybrids will join this effort in the quest for increasingly efficient CMSA variants.

Sant Esteve Sesrovires, Spain Christian Blum
February 2024

Acknowledgments

During the time of writing this book, the author was supported as principal investigator of the following three grants funded by MCIN/AEI/10.13039/501100011033: PID2019-104156GB-I00, PID2022-136787NB-I00, and TED2021-129319B-I00. Moreover, the author would like to acknowledge the support of the Spanish National Research Council (CSIC) of which the author is a Senior Research Scientist.

Contents

Acronyms

ACO	Ant Colony Optimization
AI	Artificial Intelligence
BA	Bacterial Algorithm
BIP	Binary Integer Programming
BKPWC	Bounded Knapsack Problem With Conflicts
CD	Critical Difference
CMSA	Construct, Merge, Solve & Adapt
CP	Constraint Programming
DNA	Deoxyribonucleic Acid
EA	Evolutionary Algorithm
EV	Electric Vehicle
EVRP	Electric Vehicle Routing
EVRP-TW-SPD	Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickups and Deliveries
FFMS	Far From Most String
IG	Iterated Greedy
ILP	Integer Linear Programming
KP	Knapsack Problem
LNS	Large Neighborhood Search
LP	Linear Programming
MaxSAT	Maximum Satisfiability Problem
ML	Machine Learning
MCSP	Minimum Common String Partition
MDKP	Multi-Dimensional Knapsack Problem
MDS	Minimum Dominating Set
MPIDS	Minimum Positive Influence Dominating Set
OR	Operations Research
PBIG	Population-Based Iterated Greedy
PSO	Particle Swarm Optimization
RFLCS	Repetition-Free Longest Common Subsequence
SA	Simulated Annealing

SPD	Simultaneous Pickup and Delivery
STN	Search Trajectory Network
TS	Tabu Search
TSP	Travelling Salesman Problem
TW	Time Window
VNS	Variable Neighborhood Search
VSBP	Variable-Sized Bin Packing
WID	Weighted Independent Domination

Chapter 1
Introduction to CMSA

Abstract Construct, Merge, Solve & Adapt (CMSA) is an award-winning, hybrid algorithm for solving hard combinatorial optimization problems. The main idea consists in the iterated application of an exact approach—such as, for example, an integer linear programming (ILP) solver—to sub-instances of the original problem instances to be solved. These sub-instances are extended at each iteration by adding solution components from a set of valid solutions that are obtained either by probabilistic solution construction or by any other means. In this first chapter, we will give an introduction to CMSA including related work and the application of basic CMSA variants to a well-known combinatorial optimization problem known as the Minimum Dominating Set (MDS) problem in undirected graphs. In addition, we will describe all the tools that are used for the experimental evaluation of the algorithms presented in this book. This includes the parameter tuning software called `irace`, an R-based tool for the statistical comparison of multiple algorithms called `scmamp`, and a web-based tool for the graphical comparison of multiple algorithms called `STNWeb`.

1.1 Introduction to Optimization

Optimization refers to the process of finding a best solution or outcome from a set of possible choices, generally to maximize or minimize a particular objective or criterion. It is a fundamental concept in various fields, including mathematics, engineering, economics, computer science, and more. In fact, in our increasingly technological world, the need for solving hard optimization problems has been growing constantly over the last decades. Optimization problems are prevalent in numerous practical applications across different fields. Examples are to be found, among others, in the following major fields. For each one, we provide an exemplifying reference.

1. **Supply Chain Optimization** [29]: Companies aim to optimize their supply chains by determining the most efficient way to source, produce, and deliver

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3_1

goods while minimizing costs and maintaining inventory levels to meet customer demand.

2. **Portfolio Optimization** [53]: In the realm of finance, investors aim to optimize their investment portfolios by selecting the most suitable mix of assets to maximize returns while effectively mitigating risks. This entails the pursuit of an ideal asset allocation.
3. **Production Scheduling** [84]: Manufacturers want to optimize production schedules to minimize production costs, reduce lead times, and meet customer demand reliably. This requires determining the optimal allocation of resources and scheduling production runs.
4. **Transportation Routing** [101]: In logistics and transportation, companies aim to derive optimal routes and delivery schedules for delivery vehicles, ships, or airplanes to minimize fuel costs, reduce travel times, and increase delivery efficiency. Increasingly complex optimization problems must be solved in so-called electric vehicle routing problems.
5. **Project Management** [61]: Project managers seek to optimize project schedules and resource allocation to complete projects on time and within budget. Critical path analysis and resource leveling are techniques used, for example, for the optimization of project schedules.
6. **Network Design** [82]: Companies and service providers need to optimize the design and layout of their networks—such as, for example, telecommunications networks, computer networks, or transportation networks—to maximize efficiency and minimize costs.
7. **Inventory Management** [96]: Retailers and manufacturers optimize inventory levels to balance the costs of holding excess inventory against potentially lost sales due to the lack of stock. This is often achieved through so-called Economic Order Quantity (EOQ) and Just-In-Time (JIT) inventory systems.
8. **Energy Management** [75]: Organizations and companies aim to optimize energy consumption in buildings and manufacturing processes to reduce energy costs and minimize environmental impact. This involves scheduling equipment and systems to operate at their most energy-efficient levels.
9. **Agricultural Planning** [24]: Farmers and agricultural organizations optimize crop planting, irrigation, and harvesting schedules to maximize yield, minimize resource usage, and adapt to changing weather conditions.
10. **Sensor Placement** [65]: In environmental monitoring, for example, optimizing the placement of sensors or surveillance cameras to maximize coverage and minimize costs is crucial.
11. **Drug Design** [58]: In the field of drug development, several optimization problems arise, often with the goal of identifying and developing effective and safe pharmaceutical compounds. Some of these optimization problems concern (1) compound screening, (2) optimizing the molecular structure of a compound to improve its potency, selectivity, and safety, and (3) clinical trial design.
12. **Staff Rostering and Resource Allocation** [42]: The optimal assignment of personnel to shifts and the need for an allocation of resources arises in a wide range of organizations and industries. Hospitals and healthcare facilities, for

example, must optimize staff scheduling, operating room allocation, and patient appointment scheduling to improve patient care and reduce costs.

13. **Traffic Management** [10]: Cities use optimization to manage traffic flow, for example, through optimizing traffic signal timings, leading to reduced congestion and improved traffic efficiency.

Obviously, these are just examples, and optimization is used in many more fields and scenarios to improve decision-making, resource allocation, and overall efficiency. Moreover, optimization is an essential factor in many fields of research. In fact, without efficient optimization techniques, many fields of research would not be able to advance at the same speed as they are doing today.

Modelling an Optimization Problem

In order to solve an optimization problem, it must first be modeled in a mathematical, respectively technical, way. Key elements of an optimization problem model include the following ones.

1. *Objective Function:* A model consists of at least one objective function that quantifies the goal or criterion to be optimized. Such a function may represent a quantity to be maximized (e.g., profit, efficiency, performance) or minimized (e.g., cost, error, time). In the presence of exactly one objective function, we talk about single-objective optimization, while several—usually conflicting—objective functions characterize a multi-objective problem.
2. *Decision Variables:* Optimization problems involve decision variables together with their domains. Each candidate solution to a problem is characterized by a different setting (value assignment) of the decision variables.
3. *Constraints:* They define which candidate solutions (value-assignments of the decision variables) correspond to feasible solutions, in contrast to infeasible solutions. Constraints can be equality constraints (e.g., fixed budget) or inequality constraints (e.g., resource availability).
4. *Optimization Objective:* As already mentioned in the context of describing the concept of an objective function (see above), objective functions might be maximized or minimized. This is called the optimization objective.

Often the *goal of optimization* is to find an optimal solution, which is a feasible solution with an objective function value better or equal to the objective function value of all other feasible solutions. Instead, the goal of optimization might simply be to find a good enough solution in a reasonable computation time.

Optimization problems come in various forms and can be categorized into different types based on their characteristics, constraints, and objectives. In general, we distinguish between *continuous (or numerical) optimization problems* [77] and *discrete (or combinatorial) optimization problems* [80]. Hereby, continuous optimization problems refer to models in which decision variables have continuous (real-valued) domains that may be bounded or unbounded. In discrete, respectively combinatorial, optimization problems, the decision variables are restricted to domains of discrete values. In the following, we provide two examples for each of these problem categories.

1.1.1 Examples of Continuous Optimization Problems

Examples of continuous (or numerical) optimization problems are analytical problems such as the minimization of the **Rastrigin function** which is plotted in Fig. 1.1 in two dimensions. The formula of this function (in two dimensions) is as follows:

$$f(x_1, x_2) = 20 + \sum_{i=1}^{2}(x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i)) \ , \tag{1.1}$$

with $x_1, x_2 \in [-5.12, 5.12]$. A more practical example of a continuous optimization problem with real-world relevance is the **parameter estimation in nonlinear models** problem. This problem frequently arises in various fields, including science, engineering, economics, and biology, where researchers or analysts need to estimate the parameters of a complex—generally nonlinear—model to fit observed data. Figure 1.2 shows a graphical illustration.

Given the observed data, a model must be chosen. Subsequently, the optimization problem consists of determining the optimal values of the model's parameters in order to best fit the data. The objective is to minimize the difference between the model's predictions and the observed data, typically expressed as the sum of squared residuals (least squares). The decision variables correspond to the model's parameters that need to be estimated. Constraints are based on parameter value restrictions (e.g., bounds or relationships between parameters).

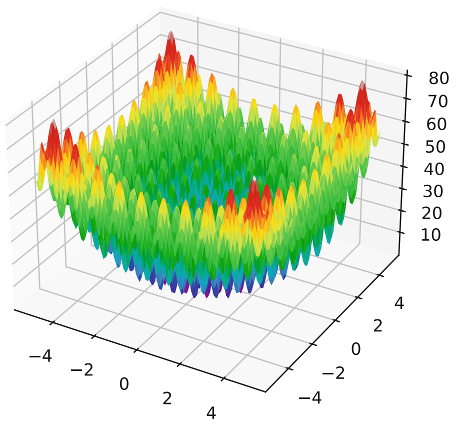


Fig. 1.1 Rastrigin function in two dimensions

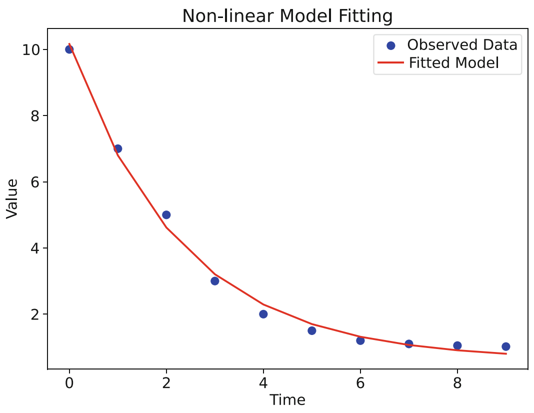


Fig. 1.2 Example of parameter estimation in nonlinear models

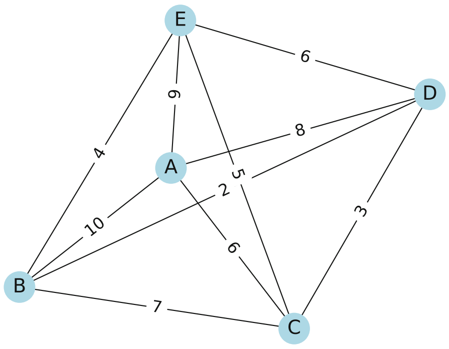


Fig. 1.3 Example of a TSP problem instance with five cities

1.1.2 Examples of Combinatorial Optimization Problems

One of the most emblematic combinatorial optimization problems is the so-called **traveling salesman problem (TSP)**. This problem owes its name to the objective of the problem. A traveling salesman must pass through a number of cities exactly once, before returning to the city in which the journey started. The optimization objective is to minimize the traveled distance. This can be modeled by means of a completely connected graph in which the nodes represent the cities that must be visited, and weights on the edges correspond to the distances between the cities. Each feasible solution corresponds to a Hamiltonian cycle of this graph. Hereby, a Hamiltonian cycle is a cyclic route that contains each vertex exactly once. A graphical illustration is given in Fig. 1.3.

Another well-known combinatorial optimization problem is the so-called **knapsack problem (KP)**. Given is a set of items, whereby each item has a profit and, for example, a weight. Given is also a knapsack with an upper limit for the total weight of the objects it can carry. The objective of the problem is to select a set of items such that they fit into the knapsack—that is, their weights may sum to at most the

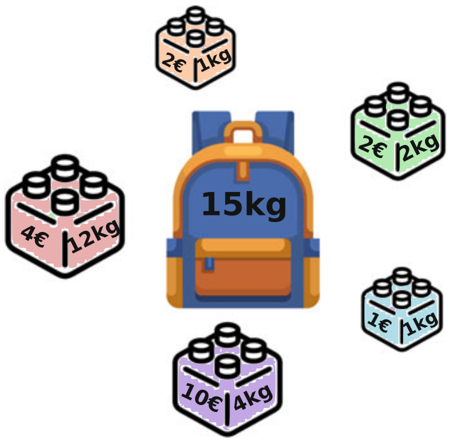


Fig. 1.4 Example of a small knapsack problem instance

weight limit of the knapsack—and the sum of the profits of the selected items is maximized. A graphical illustration is provided in Fig. 1.4.

1.1.3 Modelling an Optimization Problem

As mentioned before, to solve an optimization problem employing an optimization technique—that is, an algorithm—it must first be modeled in a way depending on its characteristics; see [33, 55, 81]. The two continuous optimization examples from Sect. 1.1.1 are modeled as global optimization problems with non-linear objective functions. In the case of a linear objective function, linear constraints, and a convex search space, a continuous optimization problem can be modeled as a linear programming (LP) problem and then be solved by LP techniques from Operations Research (OR). In contrast, the two combinatorial optimization problems outlined in Sect. 1.1.2 can be modeled as *integer linear programming (ILP)* problems, that is, in terms of models that are characterized by linear objective functions and constraints, and decision variables with discrete domains. Note that most optimization problems treated in this book are of this type. However, the general idea of CMSA is also applicable to solving optimization problems modeled in other ways.

For demonstration purposes, we provide two different ILP models of the TSP. Given is a set N of n cities, that is, $N = \{1, \ldots, n\}$. Moreover, let $A = \{(i, j) \mid i, j \in N, i \neq j\}$ be the complete set of arcs connecting any ordered pair of cities. Finally, let $c_{ij} > 0$ be the distance for traveling from city i to city j. For modeling this problem, first, the following set of binary decision variables is introduced: $\{x_{ij} \in \{0, 1\} \mid i, j \in N, i \neq j\}$, that is, for each arc (i, j) we introduce a binary decision variable x_{ij}. Hereby, in case $x_{ij} = 1$, arc (i, j) forms part of the solution.

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} && && (1.2)\\
\text{subject to} \quad & \sum_{j\in N, j\neq i} x_{ij} = 1 && \forall\, i \in N && (1.3)\\
& \sum_{j\in N, j\neq i} x_{ji} = 1 && \forall\, i \in N && (1.4)\\
& \sum_{i,j\in S, i\neq j} x_{ij} \leq |S| - 1 && \forall\, S \subset N \text{ with } 2 \leq |S| \leq n-1 && (1.5)\\
& x_{ij} \in \{0,1\} && \forall\, (i,j) \in A
\end{aligned}
$$

The objective function (1.2) to be minimized sums the distances of all used arcs. Furthermore, constraints (1.3) and constraints (1.4) ensure that each city is visited exactly once, respectively, is left exactly once. Finally, the so-called subtour elimination constraints (1.5) make sure that the finally selected arcs form exactly one cyclic tour (in contrast to several shorter ones).

In order to show that an optimization problem can potentially be modeled in different ways, we additionally provide a second, alternative ILP model for the TSP.

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} && && (1.6)\\
\text{subject to} \quad & y_{1,1} = 1 && && (1.7)\\
& \sum_{k=1}^{n} y_{ik} = 1 && i = 1,\ldots,n && (1.8)\\
& \sum_{i=1}^{n} y_{ik} = 1 && k = 1,\ldots,n && (1.9)\\
& \sum_{(i,j)\in A} x_{ij} = n && && (1.10)\\
& y_{i,k-1} + y_{jk} - x_{ij} \leq 1 && \forall\, (i,j) \in A, k \geq 2 && (1.11)\\
& y_{in} + y_{1,1} - x_{i1} \leq 1 && \forall\, (i,1) \in A && (1.12)\\
& y_{ik} \in \{0,1\} && i,k = 1,\ldots,n\\
& x_{ij} \in \{0,1\} && \forall\, (i,j) \in A
\end{aligned}
$$

This alternative model works on the basis of two sets of binary decision variables. In addition to variables x_{ij} for all $(i,j) \in A$, this model also features a binary variable y_{ik} for all $i,k = 1,\ldots,n$. If $y_{ik} = 1$, this means that city i is the k-

th visited city on the tour of the traveling salesman. Thus, the model prevents the generation of subtours by building a permutation of all cities in the following way. First, constraints (1.8) ensure that each city is assigned to exactly one position of the permutation, while constraints (1.9) require that there is exactly one city assigned to each position of the permutation. Without loss of generality, fixing $y_{1,1}$ to 1, causes the permutation to start with city 1 on position 1.[1] Furthermore, constraint (1.10) requires to choose exactly n arcs for the tour, while constraints (1.11) ensure that the arc variable x_{ij} is set to 1, if both variables $y_{i,k-1}$ and y_{jk} are set to 1. In other words, if cities i and j are placed on consecutive positions of the permutation, then arc (i, j) must form part of the tour. Finally, note that constraints (1.12) only cover the special case of position 1 of the permutation being the successor of position n.

> Types of Optimization Problems Considered in This Book

As mentioned above, in this book we consider problems that can be modeled in terms of ILPs. Put differently, we deal with the solution of problems that can be modeled on the basis of a set of discrete decision variables. Moreover, the considered optimization problems can be modeled by means of a linear objective function and linear constraints. However, note that this does not exclude that the algorithmic ideas presented in this book can also be applied to other types of optimization problems.

1.1.4 Basic Optimization Techniques

After modeling the optimization problem under consideration in a convenient way, an optimization technique is required to solve the problem. Exact and approximate methods for optimization are two broad categories of techniques used to find optimal—or simply good-enough—solutions to optimization problems. These two categories comprise techniques of different characteristics that are suited to different types of problems and objectives. Their key features can be summarized as follows.

Exact methods guarantee to find an optimal solution in finite time. This is the case if an optimal solution exists, and if given enough time and resources. These methods are usually deterministic. They systematically explore the solution/search space to find an optimal solution. Exact methods are practical for small to moderately-sized optimization problems, where the search space is not too large. Examples of exact methods include linear programming, integer programming, dynamic programming, branch and bound, and branch and cut, just to name a few [11, 103].

[1] Note, in this context, that instead of y_{11} we use here the notation $y_{1,1}$ in order to avoid a misunderstanding.

Knowledge Required from Readers on Exact Techniques

In this book, we assume that readers generally know about the existence and the high-level functioning of exact techniques. However, we do not expect a reader to be an expert on exact techniques. Neither do we expect that a reader is able to design and program exact techniques. This level of difficulty will be relegated completely to the use of black-box ILP solvers. For the experiments presented in this book, we make use of CPLEX[2], a commercial software product offered by IBM, which is free for academic purposes.

Despite the advantages of exact methods as outlined above, they are not always the best choice for solving the optimization problem at hand. First, the problem instances to be solved might be too large to be handled by the exact method. Second, the required computation time and/or the required amount of resources—for example, in terms of computer memory—might be excessive. Therefore, research has also focused on so-called approximate methods, which are typically of a heuristic and/or stochastic nature. They generally do not guarantee finding an optimal solution but aim to find a good enough solution within a reasonable time frame. In other words, approximate methods are often much faster than exact methods, making them suitable for large and complex optimization problems. At the same time, they require fewer computational resources and are, therefore, often more practical for real-world, large-scale problems. Examples of approximate methods range from simple, deterministic greedy heuristics [72] to more sophisticated metaheuristics [20, 48] including, for example, evolutionary algorithms [23], simulated annealing [62], particle swarm optimization [30, 59], and variable neighborhood search [74].

Greedy heuristics—also referred to as greedy algorithms—build a solution from scratch by making a sequence of deterministic choices. They are called “greedy” because, at each step of the solution construction process, they make a locally optimal choice with the hope that this will lead to a good solution overall. They are generally easy to implement and computationally efficient, but they generally do not provide any guarantees on the quality of the generated solutions.

Metaheuristics are more sophisticated approximate algorithms. They are designed to efficiently search through large search spaces and find high-quality solutions, although they may not guarantee anything about the quality of the solutions found. Some common metaheuristics[3] for combinatorial optimization include the following ones:

- **Evolutionary Algorithms (EAs)** [23]: These algorithms are inspired by the process of natural evolution. At each iteration, they maintain a population of

[2] https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio.

[3] Note that the order in which metaheuristics are described here has no meaning.

potential solutions and apply genetic operators such as mutation, crossover, and selection to evolve and improve the solutions over multiple iterations.
- **Simulated Annealing (SA)** [62]: SA is inspired by the annealing process in metallurgy. It starts with an initial solution and iteratively explores neighboring solutions. Worse solutions are accepted with a probability that is decreasing over time, allowing the algorithm to escape from local optima, that is, sub-optimal solutions surrounded by worse solutions.
- **Tabu Search (TS)** [51]: TS maintains a list of "tabu" or forbidden moves (the so-called tabu list) to avoid revisiting previously explored solutions. It explores the neighborhood of the current solution while avoiding moves forbidden by the tabu list.
- **Ant Colony Optimization (ACO)** [38]: ACO is inspired by the foraging behavior of ants. It uses a population of artificial ants to explore the solution space. Ants deposit pheromones on paths they traverse, and the pheromone levels guide the search process.
- **Particle Swarm Optimization (PSO)** [30, 59]: PSO is inspired by the social behavior of birds and fish. It maintains a population of particles that move through the search space. Particles adjust their positions based on their own experience and the experience of their neighbors.
- **Variable Neighborhood Search (VNS)** [74]: VNS explores the solution space by changing the neighborhood structure in which it searches. The standard version of VNS tends to start with local neighborhoods. Moreover, the algorithm tends to move to more disruptive neighborhoods in case nothing better can be found in more local neighborhoods.

In summary, we could say that optimization techniques can vary widely, ranging from simple trial-and-error approaches to more sophisticated mathematical methods. The choice of method depends on the characteristics of the considered optimization problem, its complexity, the available computational resources, and the available time budget.

1.1.5 Hybrid Optimization Techniques

Extensive research efforts have been devoted to addressing combinatorial optimization problems over the past decades, both in Operations Research (OR) and Artificial Intelligence (AI). Consequently, both researchers and practitioners now possess a diverse toolbox of techniques, encompassing both exact and approximate methods, to tackle optimization problems of this nature. However, when confronted with the task of solving large-scale instances of complex combinatorial optimization problems, even metaheuristic techniques can become disoriented within the vast search spaces that characterize such scenarios. It has become increasingly evident that a practical approach lies in harnessing the complementary strengths of both exact and heuristic methods, particularly when addressing large-scale problem instances.

The field of *hybrid metaheuristics for combinatorial optimization* [19, 97] has become increasingly popular in recent years due to the ability of such approaches to combine the strengths of different ways of solving optimization problems within a single algorithm.[4] Algorithms known as *large neighborhood search (LNS)* [86] and *very large-scale neighborhood search* [1] are probably among the most well-known techniques from this field. Another related branch of work is the one on ejection chain approaches [50]. In principle, there are many ways of generating so-called large neighborhoods for a given problem. However, many LNS approaches are based on the principle of *ruin-and-recreate* [95], also sometimes found as *destroy-and-recreate* or *destroy-and-rebuild*. At each iteration, first, the incumbent solution is partially destroyed. Then, either an exact technique or any other appropriate technique is applied to find—among all solutions that include the produced partial solution—a solution that improves the incumbent solution. Generally, a time limit is imposed on this step. Many examples of this type of LNS can be found in the literature, including [34, 43, 94], just to name a few. The large neighborhoods generated in this context are known as destruction-based large neighborhoods.

Apart from methods based on partial solution destruction, there are alternative ways of defining large neighborhoods that are used in algorithms such as local branching [45], the corridor method [26], and POPMUSIC [63]. In the latter approach (POPMUSIC), at each iteration, a large neighborhood is generated as follows. The incumbent solution is first split into parts. A so-called seed-part is then chosen and extended by adding other parts that are close to the seed-part in order to form a sub-problem. This step depends on some distance measure between solution parts. Finally, the generated sub-problem is solved by an approximate or an exact solution approach. This process is repeated until the incumbent solution does not contain a sub-problem that can be improved.

1.2 Tools Used in This Book

First of all, note that all approaches described in this book were implemented in `C++` [56] and compiled with one of the latest `Gnu` compilers.[5] Moreover, as ILP solver we used `CPLEX` version 22.1,[6] if not otherwise stated. Apart from these fundamental tools for the research described in this book, we made use of the following three tools, which are described in the following.

[4] These algorithms are also often labelled as *matheuristics* [22].

[5] https://gcc.gnu.org/.

[6] https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio.

1.2.1 `irace`: A Tool for Parameter Tuning

One of the challenges in conducting experimental evaluations of stochastic optimization algorithms is the issue of parameter tuning. This task involves configuring the algorithm's parameters to optimal values, ensuring its peak performance on the designated set of benchmark instances. The literature offers several scientific tools for this purpose, including `SMAC3` [68], `ParamILS` [57] and `HyperBand` [67].

In this book, we make use of `irace` [70],[7] one of the parameter tuning tools most used for tuning the parameters of optimization algorithms. `irace` is particularly useful when working with algorithms that have multiple parameters and where finding the right combination of parameter settings can significantly impact their performance. The `irace` tool automates this process. Key features and functionalities of `irace` include the following ones:

1. **Iterative Racing:** `irace` uses an iterative racing mechanism to evaluate different combinations of algorithm parameters. It systematically explores numerous parameter settings, ranking them based on the algorithm's performance on a predefined set of benchmark instances.
2. **Parallelization:** `irace` is designed to work in parallel, which can significantly speed up the parameter tuning process. It distributes the evaluation of parameter configurations across the available processors or machines, making it suitable for high-performance computing environments. For the application in this book, we used `irace` on a high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.
3. **Generality:** Even though `irace` is programmed in `R`,[8] it can be used for different algorithms, that is, it is not limited to a specific algorithm, domain, or programming language. It only requires an executable of the algorithm whose parameters are to be tuned. Users can apply it to optimize a wide range of algorithms and techniques, including machine learning models, optimization algorithms, and more.
4. **Scalability:** `irace` is scalable, making it suitable for both small-scale and large-scale optimization tasks. It adapts to the available computational resources and allows users to balance the trade-off between the quality of results and the computational effort required.
5. **Robustness:** `irace` is robust in handling noisy or stochastic algorithms. It can efficiently deal with algorithms that produce variable results due to randomness or external factors.

In summary, `irace` is a powerful tool for researchers and practitioners who need to find well-working parameter settings for their algorithms efficiently and

[7] https://mlopez-ibanez.github.io/irace/.

[8] https://www.r-project.org/.

effectively. It can save a significant amount of time and computational resources by automating the parameter-tuning process and helping users discover the best configuration for their specific tasks.

> Use of `irace` in This Book

In this book, we will employ `irace` to generate the optimal parameter values for all algorithms considered in the presented computational evaluations. This approach is essential to guarantee that the comparative analyses presented are both fair and informative.

1.2.2 `STNWeb`: A Tool for the Graphical Comparison of Algorithms

A picture is worth a thousand words.
Arthur Brisbane, 1911

Visual representations of complex concepts may significantly enhance our ability to grasp digital information more effectively. This principle extends across various domains within Computer Science and AI. Remarkably, the research community in combinatorial optimization has yet to achieve significant progress in the development of visual aids, despite the growing demand for innovative tools that facilitate the comparison of optimization algorithms. Over the past few decades, the conventional approach to comparing optimization algorithms has centered around gathering numerical data from runs of different algorithms. This data is subsequently analyzed using conventional tools, such as tables and classical data charts (e.g., line plots, bar plots, and scatter plots). It has also become customary to complement this form of algorithm comparison with statistical analyses of the collected data. In recent years, an increasing number of researchers have recognized the importance of incorporating supplementary graphical tools to gain a more profound understanding of the behavior of optimization algorithms, particularly metaheuristics [20, 48].

As previously mentioned, the research community dedicated to optimization algorithms has not shown a high level of productivity in the development of visual tools. Nonetheless, there have been several attempts to visualize the behavior of optimization algorithms, as evidenced in various studies, including [31, 71, 73, 87]. These approaches typically employ dimensionality reduction techniques to project complex search spaces into two or three dimensions, enabling a basic tracking of the search progress. The currently best tool for this purpose was introduced only

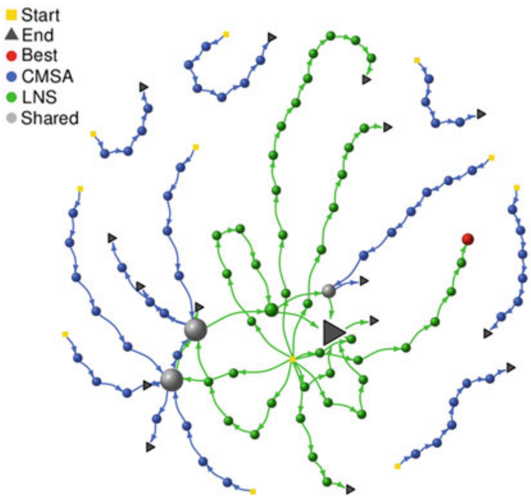


Fig. 1.5 Example of an `STNWeb` graphic

recently in [28]. This tool—labeled `STNWeb`[9]—is a web application based on the concept of so-called *Search Trajectory Networks (STNs)* [78]. STNs are graph structures with nodes and directed edges for visualizing the search process of iterative optimization algorithms and aiding in the analysis of their progress. Notably, STNs offer a means to represent multiple trajectories of various optimization algorithms applied to the same problem instance in a graphical format.

Figure 1.5 illustrates a straightforward example of an `STNWeb` graphic. This visual representation compares the performance of two distinct algorithms when applied to an instance of the well-known and challenging *multi-dimensional knapsack problem*. The graphic showcases the trajectories resulting from ten separate runs of each algorithm on this problem instance. Each vertex within this visualization represents a solution to the problem instance. Nevertheless, it is worth noting that this may not always hold true, as explained further below. The graphic employs various visual elements to convey information effectively. The colors, shapes, and sizes of the vertices have specific meanings, which are elucidated as follows:

- Distinct algorithms' trajectories are presented using different colors, as denoted in the legend of each `STNWeb` graphic. For instance, in Fig. 1.5, the 10 trajectories generated by the CMSA algorithm are depicted in blue, while those generated by the LNS algorithm are represented in green.[10]

[9] STNWeb is freely accessible to anyone interested and can be accessed at this URL: https://www.stn-analytics.com/.

[10] In this context it is not important to know the nature of these algorithms.

- The initial points of the trajectories are marked with yellow squares. It is interesting to observe, for example, that in Fig. 1.5, the 10 runs of the CMSA algorithm commence from distinct initial solutions, whereas the 10 runs of the LNS algorithm all originate from the same initial solution.
- Trajectory endpoints are represented in two ways: dark grey triangles and red dots. Dark grey triangles signify endpoints that do not correspond to the best-found solution across all algorithm runs, while red dots indicate endpoints that correspond to best-found solutions.
- Pale grey dots represent solutions that are shared across trajectories of at least two distinct algorithms.
- Lastly, the size of a vertex or dot conveys the count of algorithm trajectories passing through it: the larger the vertex, the greater the number of traversing algorithm trajectories.

As mentioned above, the `STNWeb` graphic of Fig. 1.5 offers a comparative analysis of two different algorithms applied ten times to the same problem instance. Notably, it reveals the presence of an attraction area within the search space, particularly for the CMSA algorithm. Out of the ten CMSA trajectories, five are notably drawn toward the region marked by the presence of large, pale grey dots. Interestingly, although many trajectories traverse this area, none of them stops in one of the two solutions represented by those dots, underscoring that these solutions are not the best ones within that region. In contrast, the LNS algorithm exhibits less inclination toward this specific area in the search space, with only one of its trajectories passing through it. This aspect of trajectory behavior is a facet often overlooked in contemporary optimization research. Nevertheless, it can prove invaluable for gaining insights into why an algorithm performs exceptionally well or, conversely, why it faces challenges in delivering good results.

Note that—either due to many or long algorithm trajectories, or to specific problem instance characteristics—STN graphics in which dots are solutions may sometimes appear cluttered and hard to interpret. For this reason, STNWeb comes with so-called *search space partitioning* techniques that divide the search space into chunks containing closely related solutions. In STN graphics after search space partitioning, dots are those chunks of the search space containing at least one of the solutions from the considered set of algorithm trajectories.

Use of `STNWeb` in This Book

In this book, we will make use of the STNWeb tool for generating graphics that will help us understand the behavioral differences of different algorithms.

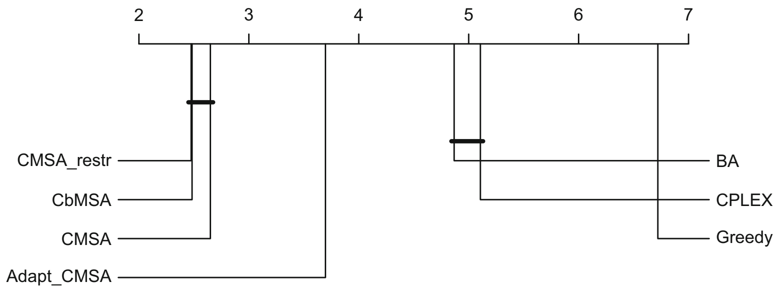


Fig. 1.6 Example of a critical difference (CD) plot generated with the `scmamp` package

1.2.3 `scmamp`: *A Tool for the Statistical Comparison of Algorithms*

In addition to the graphical `STNWeb` tool described in the previous section, algorithms are also compared on the basis of numerical results. In order to do this on a scientific basis we make use of statistical comparison, which involves employing statistical methods to assess and differentiate the performance of various optimization techniques. Such a statistical comparison aims to determine whether observed differences in the outcomes are statistically significant or merely due to chance. The goal is to provide a robust and objective basis for selecting the most effective optimization approach in a given context.

For this purpose, we make use of the R package `scmamp` [25].[11] Several works (see, for example, [35, 46, 47]) have outlined a fundamental classification of general machine learning scenarios and the corresponding statistical tests suitable for each scenario. The `scmamp` package implements the recommendations from these works and aims specifically for a comparative analysis of multiple algorithms across multiple problem instances. The package also provides functions for the generation of so-called *critical difference (CD) plots* based on the results of the statistical comparisons performed. An example of such a CD plot in which seven different algorithms are compared over a range of problem instances is shown in Fig. 1.6. Vertical whiskers indicate the average ranking of the algorithms over the considered set of problem instances. Algorithm CMSA_restr, for example, is the best-ranked algorithm, while algorithm Greedy is the one with the lowest average rank. Moreover, if two algorithm whiskers are joined by a bold horizontal bar, it suggests that the difference in their performance is not statistically significant. In the example shown in Fig. 1.6, no statistical difference is detected, for example, concerning the performance of algorithms BA and CPLEX. Conversely, if two algorithm whiskers are not joined by a bold horizontal bar, their performance is considered significantly different.

[11] https://github.com/b0rxa/scmamp.

In essence, CD plots help researchers and practitioners make informed decisions about the relative performance of multiple algorithms across a set of problem instances. They provide a clear visual representation of which algorithms are statistically different in terms of their performance and which are not, aiding in the selection of the most suitable algorithm for a specific set of problem instances.

> Use of `scmamp` in This Book

`scmamp` is used in all experimental evaluations of this book for the generation of CD plots in order to convey statistical information about the relative performance of the compared algorithms.

1.3 CMSA: Construct, Merge, Solve & Adapt

As mentioned at the beginning of this book, CMSA (Construct, Merge, Solve & Adapt) is an algorithmic idea for solving hard combinatorial optimization problems. The algorithm was first presented in a seminal paper in [18] in which it was applied to the minimum covering arborescence problem and to the minimum common string partition problem. The currently existing applications of CMSA are summarized in Table 1.1.

> Awards for Work on CMSA

Work on CMSA has received two awards in recent years. The first one was the *best paper award* at the ECOM track of the GECCO 2016 conference for a paper on the application of CMSA to the multi-dimensional knapsack problem [17].[12] The second award was the one for *The Best Methodological Contribution in Operations Research* jointly given by the Spanish Society of Statistics and Operations Research (SEIO) and the BBVA Foundation in 2021.[13]

Initially, the idea for the development of CMSA originated from an observation of a possible weakness of an older hybrid technique known as large neighborhood search (LNS); see also Sect. 1.1.5. More specifically, in the case of (1) problems—respectively, problem instances—that are characterized by large sets of solution components and (2) solutions that consist of comparatively few solution components, we noticed that partial-destruction-based LNS sometimes easily gets stuck in local minima. Therefore, the motivation was to generate a hybrid approach in which this would not happen.

[12] GECCO 2016 Awards Webpage. Accessed on 17/11/2023.

[13] SEIO-FBBVA 2021 Award Webpage. Accessed on 17/11/2023.

Table 1.1 Currently existing applications of (variants of) CMSA

Optimization problem	Publication	Year
Minimum common string partition	[12, 16, 18]	2016, 2020, 2021
Minimum covering arborescence	[18]	2016
Repetition-free longest common subsequence	[13, 14]	2016, 2018
Multi-dimensional knapsack	[16, 17, 69]	2016, 2017, 2021
Maximising the net present value of project schedules	[98]	2019
Binary optimization	[15]	2019
Minimum capacitated dominating set	[83]	2019
Routing cooperative air-ground robots with fuel constraints	[8]	2019
Maximum happy vertices problem	[49, 66, 100]	2019, 2022
Score-constrained packing	[54]	2020
Refueling and maintenance planning of nuclear power plants	[39]	2021
Test data generation in software product lines	[44]	2021
Bus driver scheduling	[90]	2022
Minimum positive influence dominating set	[4]	2022
Unit disk cover	[5]	2022
Electric vehicle routing	[2, 3]	2022, 2023
Multi-way multi-dimensional number partitioning	[37]	2023
Closest String	[79]	2023
Maximum disjoint dominating set	[88, 89]	2023, 2024
Rooted max tree coverage	[104]	2024

1.3.1 Standard CMSA

We start our introduction to CMSA by describing a standard algorithm variant, which is henceforth simply called CMSA. However, before doing so, the general idea of CMSA is briefly described. At each iteration, CMSA first generates several (generally valid) solutions to the tackled problem instance in a probabilistic way. Next, the components of these solutions are added to an initially empty sub-instance C'. This sub-instance is then passed to an exact solver (if convenient, this can be an ILP solver) which delivers the best solution found in a limited time. Based on this solution, the incumbent sub-instance is adapted. In particular, based on an aging mechanism, seemingly useless solution components are removed from C' in order not to slow down the solver in the next algorithm iteration.

The first action that needs to be taken for applying CMSA to a combinatorial optimization problem is defining the set C of solution components, that is, those components of which solutions to the considered problem are composed. Later we will provide specific examples for such a definition. For the moment, however, let $C = \{c_1, \ldots, c_n\}$ be a generic set of solution components. Moreover, any valid solution $S \in \mathcal{S}$ to the considered optimization problem—where $\mathcal{S}$ is the set of all valid solutions—can be expressed as a subset of C, that is, $S \subseteq C$ for all $S \in \mathcal{S}$.

Algorithm 1.1: Pseudo-code of standard CMSA

```
1: input 1: Complete set of solution components $C$
2: input 2: values for CMSA parameters $n_a$, $age_{\max}$, and $t_{\mathrm{ILP}}$
3: $S^{\mathrm{bsf}} := \emptyset$
4: $C' := \emptyset$
5: $age[c] := 0$ for all $c \in C$
6: while CPU time limit not reached do
7:    for $i := 1, \ldots, n_a$ do
8:       $S :=$ ProbabilisticSolutionConstruction($C$)
9:       if $f(S) < f(S^{\mathrm{bsf}})$ then $S^{\mathrm{bsf}} := S$ endif
10:      for all $c \in S$ and $c \notin C'$ do
11:         $age[c] := 0$
12:         $C' := C' \cup \{c\}$
13:      end for
14:   end for
15:   $S^{\mathrm{ILP}} :=$ SolveSubinstance($C'$, $t_{\mathrm{ILP}}$)
16:   if $f(S^{\mathrm{ILP}}) < f(S^{\mathrm{bsf}})$ then $S^{\mathrm{bsf}} := S^{\mathrm{ILP}}$ end if
17:   Adapt($C'$, $S^{\mathrm{ILP}}$, $age_{\max}$)
18: end while
19: output: $S^{\mathrm{bsf}}$
```

Finally, let $f : \mathcal{S} \mapsto \mathbb{N}^+$ for the following discussion be the objective function to be minimized, and let $f(\emptyset) := \infty$.

Algorithm 1.1 provides the pseudo-code of the standard version of CMSA. The algorithm starts by initializing both the best-so-far solution S^{bsf} and the sub-instance C', which is always a subset of C, to the empty set. Furthermore, the so-called age values of all solution components are initialized to zero, that is, $age[c] := 0$ for all $c \in C$. The main loop of CMSA consists of four actions.

1. First, in the **construct step** of CMSA, a number of n_a valid solutions to the considered problem are probabilistically generated (see line 8 of Algorithm 1.1).
2. Second, in the **merge step** of CMSA, the current sub-instance C' is updated with the solution components found in these n_a solutions (see lines 10–13). That is, those solution components that (1) are found in at least one of the n_a constructed solutions and (2) do currently not form part of C', are added to C' and their age value is set to zero.
3. Third, in the **solve step** of CMSA, an ILP solver is applied in order to find the best possible solution that only contains components from sub-instance C', within a time limit of t_{ILP} CPU seconds (see line 15).
4. Fourth, in the **adapt step** of CMSA, sub-instance C' is adapted based on the solution S^{ILP} returned by the ILP solver; see function Adapt(C', S^{ILP}, $age_{\max}$) in line 17. In particular, in this function, sub-instance C' is adapted in the following way. First, the age values of all components in $C' \setminus S^{\mathrm{ILP}}$ are incremented by one. Second, the age values of all components in S^{ILP} are set to zero. The final action in the *adapt step* consists of removing all those components from C' whose age value has reached the maximum allowed age ($age_{\max}$). This is done in order to

prevent components that never appear in S^{ILP} from slowing down the ILP solver in subsequent CMSA iterations. Note that the age value $age[c]$ of a solution component $c \in C$, at any time, indicates the number of consecutive CMSA iterations for which c has formed part of sub-instance C' without having been included in the ILP-solution to the sub-instance C'.

These four steps are iterated until a given CPU time limit is reached. The output of the algorithm is S^{bsf}, the best solution found during the whole process.

Note that the *construct step* and the *solve step* of such a CMSA algorithm are problem-dependent. In particular, for the *construct step* of the algorithm, generally, a greedy heuristic for the tackled problem is used in a randomized way, and the *solve step* depends on the availability of an ILP model for the tackled problem. Moreover, the way in which an ILP model is exactly generated based on sub-instance C' leaves room for variation. In contrast, the *merge step* and the *adapt step* are problem-independent.

1.4 Application to Minimum Dominating Set

In the following, we show how standard CMSA can be applied to the so-called Minimum Dominating Set problem. In particular, we will show two different ways of defining the set of solutions components, which results in slightly different ways of generating the ILP models corresponding to the sub-instances at each algorithm iteration.

Requirements for the Application of CMSA

In order to apply the standard CMSA algorithm from the previous section (see Sect. 1.3.1) to any combinatorial optimization problem, we need to define two algorithm components:

1. A probabilistic way of generating valid solutions
2. An ILP model of the tackled problem

The Minimum Dominating Set (MDS) problem is a well-known combinatorial optimization problem in Computer Science. Given an undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of n vertices and E is the set of edges, the goal of the problem is to find a smallest dominating set $D^* \subseteq V$. Hereby, a set $D \subseteq V$ is called a *dominating set* of G if and only if for every vertex $v \in V$ it holds that

1. $v \in D$, or
2. $\exists\, v' \in D$ such that $(v, v') \in E$.

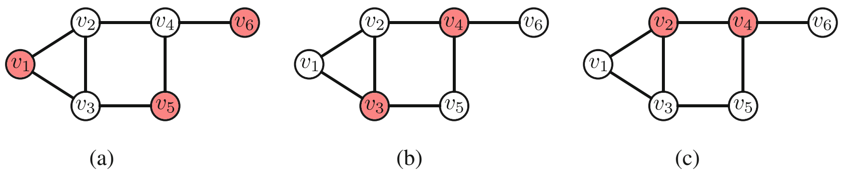


Fig. 1.7 An undirected graph with three different MDS solutions. (**b**) and (**c**) show alternative optimal solutions. (**a**) Sub-optimal solution. (**b**) An optimal solution. (**c**) Another optimal solution

In other words, a subset of the vertices of graph G is called a dominating set of G if every vertex of G is either in the set or adjacent to at least one vertex from the set; see Fig. 1.7 for an example. In this context, note that the set of neighbors of v in G is denoted by $N(v)$, that is, $N(v) := \{v' \in V \mid (v, v') \in E\}$. Moreover, $N[v] := N(v) \cup \{v\}$ is called the *closed neighborhood* of v.

The standard ILP model for the MDS problem makes use of a binary variable x_i for each vertex $v_i \in V$. The model can be stated as follows.

$$\min \sum_{v_i \in V} x_i \tag{1.13}$$

$$\text{subject to} \sum_{v_j \in N(v_i)} x_j + x_i \geq 1 \qquad \forall\, v_i \in V \tag{1.14}$$

$$x_i \in \{0, 1\} \qquad \forall\, v_i \in V$$

The constraints (1.14) make sure that any valid solution either contains a node $v_i \in V$ or at least one of its neighbors.

Finally, note that the MDS problem is known to be NP-hard, meaning that there is no known polynomial-time algorithm to solve it for arbitrary graphs unless $P = NP$. As a result, research often focuses on finding efficient algorithms for special cases or developing heuristic algorithms.

1.4.1 An Intuitive Way of Defining the Solution Components

Programming Code Availability of This Application

Note that the C++ program code for this way of applying standard CMSA to the MDS problem is included in Appendix A of this book.

The first important step for the development of any CMSA algorithm is the definition of the set of solution components (C). In the case of the MDS problem,

Algorithm 1.2: MDS solution construction

```
1: input: a graph $G = (V, E)$
2: $s := \emptyset$
3: $\overline{V} := \{v_i \in V \mid N[v_i \mid s] \neq \emptyset\}$
4: while $\overline{V} \neq \emptyset$ do
5:     $v_j :=$ ChooseFrom($\overline{V}$)
6:     $s := s \cup \{v_j\}$
7:     $\overline{V} := \{v_i \in V \mid N[v_i \mid s] \neq \emptyset\}$
8: end while
9: output: a valid solution $s$
```

there is an intuitive way of doing so. Hereby, set C simply contains a component c_i for each vertex $v_i \in V$. Therefore, for the following discussion, remember that v_i and c_i both refer to vertex v_i of the input graph G. The standard CMSA for the MDS problem based on the intuitive set of solution components is henceforth labeled CMSA_INT.

1.4.1.1 Constructing Solutions to the MDS Problem

In the following, we say that, if a vertex v_i is added to a solution s under construction, then v_i *covers itself and all its neighbors*, that is, it covers v_i (itself) and all $v_j \in N(v_i)$. Moreover, assuming that s is a partial solution under construction, we denote by $N[v_i \mid s] \subseteq N[v_i]$ the *set of uncovered vertices* (with respect to s) from the closed neighborhood $N[v_i]$ of $v_i \in V$.

The solution construction mechanism is shown in Algorithm 1.2. It starts with an empty solution $s = \emptyset$. For each construction step, $\overline{V} \subseteq V$ is defined as the set of vertices that can cover at least one of the vertices not covered yet by s. At each construction step, exactly one vertex is chosen from $\overline{V}$ in function ChooseFrom($\overline{V}$). This works as follows. First, a random number r is sampled uniformly at random from [0, 1]. In case $r \leq d_{\text{rate}}$, $v_j \in \overline{V}$ is chosen such that

$$v_j := \max_{v_i \in \overline{V}} \left\{ \left| N[v_i \mid s] \right| \right\} \tag{1.15}$$

Otherwise—that is, in case $r > d_{\text{rate}}$—a number of $\min\{l_{\text{size}}, |\overline{V}|\}$ vertices from $\overline{V}$ is pre-selected and placed into a candidate set $L \subseteq \overline{V}$ such that

$$\left| N[v_i \mid s] \right| \geq \left| N[v_k \mid s] \right| \quad \text{for all } v_i \in L, v_k \in \overline{V} \setminus L \tag{1.16}$$

A vertex $v_j \in L$ is then chosen uniformly at random. Finally, note that each solution s, after finalizing its construction, is converted into a corresponding solution S that contains for each $v_i \in s$ the corresponding solution component c_i.

1.4.1.2 Solving Sub-instances of the MDS Problem

Next, we will describe how to use the ILP model of the MDS problem to solve a sub-instance C' within CMSA_INT. In this context, remember that—in any CMSA algorithm—a sub-instance C' contains those solution components that may appear in valid solutions to C'.

In order to solve a sub-instance C' by means of the application of an ILP solver in function SolveSubinstance(C', t_{ILP}) of Algorithm 1.1, the MDS ILP model is extended in CMSA_INT by adding the following set of constraints:

$$x_i = 0 \quad \text{for all } c_i \in C \setminus C' \tag{1.17}$$

In this way, only solution components (resp. vertices) that form part of the sub-instance C' may be selected to appear in solutions.

Note that the function SolveSubinstance(C', t_{ILP})—see line 15 of Algorithm 1.1—returns the ILP solver solution in terms of a set S^{ILP} of solution components. Moreover, the application of the ILP solver is subject to a time limit of t_{ILP} CPU seconds, which means that the solution S^{ILP} returned by the function SolveSubinstance(C', t_{ILP}) is not necessarily an optimal solution to the sub-instance C'.

ILP Solver Dependent Settings Considered in This Book

Please be aware that while a commercial ILP solver like CPLEX (utilized throughout this book) can be used with its default parameter settings, this may not always yield the best results. Therefore, we explore the following options to possibly enhance performance beyond the default settings:

1. **Heuristic emphasis:** CPLEX offers a parameter to balance between the speed of proving optimality and the speed to improve the best-found solution during execution. As proving optimality is not a priority in CMSA algorithms, we consider a parameter $\text{cplex}_{\text{emphasis}} \in \{\texttt{true}, \texttt{false}\}$, where $\text{cplex}_{\text{emphasis}} = \texttt{false}$ refers to the default setting of CPLEX and $\text{cplex}_{\text{emphasis}} = \texttt{true}$ indicates a setting of the emphasis parameter to a value of five (highest heuristic emphasis value).
2. **Warm start:** When started in default mode, an ILP solver initially does not know any valid solution to the tackled problem. In some cases, providing the ILP solver with an initial valid solution can speed up the solving process. In the context of CPLEX, this is called a warm start. We consider a parameter $\text{cplex}_{\text{warmstart}} \in \{\texttt{true}, \texttt{false}\}$, where $\text{cplex}_{\text{warmstart}} = \texttt{false}$ does not provide an initial solution to CPLEX, whereas $\text{cplex}_{\text{warmstart}} = \texttt{true}$ provides the best-so-far solution S^{bsf} to CPLEX as initial solution.
3. **Aborting a CPLEX call:** Even with an increased value for the heuristic emphasis, CPLEX spends computation time on bound computations that eventually lead to proving optimality. In some applications, a lot of computation time might be spent on these efforts. In fact, in our experience, it is sometimes beneficial to

abort CPLEX when the first solution is found that improves over the best-so-far solution S^{bsf}. This CPLEX behavior is invoked in the applications of this book with a parameter $\mathrm{cplex}_{\mathrm{abort}} \in \{\texttt{true}, \texttt{false}\}$.

This completes the description of the application of standard CMSA based on the intuitive set of solution components to the MDS problem.

1.4.2 *A Generic Way of Defining the Solution Components*

Depending on the optimization problem to be solved, sometimes, there might be no intuitive way of defining the set of solution components. However, a generic way of doing so works as follows. For this purpose, we assume to have a binary ILP model for the tackled problem at hand.[14] The set C of generic solution components will contain for each binary variable x_i two solution components:

1. Component c_i^0: corresponding to $x_i = 0$.
2. Component c_i^1: corresponding to $x_i = 1$.

In case a solution s to the problem is characterized by $x_i = 0$, the corresponding CMSA-solution S contains component c_i^0. Similarly, if $x_i = 1$ in a solution s, then S contains component c_i^1. In the case of the MDS problem, for example, the generic set of solution components C is defined as follows:

$$C := \{c_1^0, \ldots, c_n^0, c_1^1, \ldots, c_n^1\} \quad \text{where } n := |V| \tag{1.18}$$

For example, the solution in Fig. 1.7a at page 21a in CMSA format would contain the following set of solution components: $\{c_1^1, c_2^0, c_3^0, c_4^0, c_5^1, c_6^1\}$. Henceforth the standard CMSA algorithm for the MDS based on the generic set of solution components is labeled CMSA_GEN.

1.4.2.1 Solution Construction

For constructing solutions in CMSA_GEN we use exactly the same way as described in the context of CMSA_INT in Sect. 1.4.1.1. In other words, function ProbabilisticSolutionConstruction(C) first generates a solution s which consists of a set of vertices. The only difference to the procedure described in Sect. 1.4.1.1 consists in the conversion of s into a solution S of CMSA_GEN. In particular, for each $v_i \in V$

[14] Note that any integer ILP can be transformed into a binary ILP (BIP). This is well known in OR and will be demonstrated employing an example in Chap. 5.

that forms part of s, solution component c_i^1 is added to S, and for each $v_i \in V$ which does not form part of s, solution component c_i^0 is added to S.

1.4.2.2 Sub-instance Solving

A sub-instance C' in CMSA_GEN is solved by adding the following additional constraints to the MDS ILP model (for $i = 1, \ldots, |V|$) and solving it with CPLEX:

$$x_i = 0 \qquad \text{if } c_i^0 \in C' \text{ and } c_i^1 \notin C' \tag{1.19}$$

$$x_i = 1 \qquad \text{if } c_i^0 \notin C' \text{ and } c_i^1 \in C' \tag{1.20}$$

In other words, if C' only contains the solution component c_i^0 corresponding to $x_i = 0$, then the value of x_i is fixed to zero. Similarly, if C' only contains the solution component c_i^1 corresponding to $x_i = 1$, then the value of x_i is fixed to one. Otherwise, if C' contains both solution components c_i^0 and c_i^1 then variable x_i is left free, which means that v_i might, or not, be included in a solution to the sub-instance.

In this context, note that the way of generating the ILP model corresponding to C' in CMSA_GEN is potentially more restrictive than in CMSA_INT, at least in the case of our application to the MDS problem. This can easily be seen as follows. Let us assume that both algorithm variants are in the first iteration (that is, C' is empty), $n_a = 2$ (that is, two solutions are generated per iteration), and the generated solutions are the ones from Fig. 1.7b and c on page 21. Therefore, in CMSA_INT, after solution construction it holds that $C' = \{c_2, c_3, c_4\}$. In the case of CMSA_GEN, it holds that $C' = \{c_1^0, c_2^0, c_2^1, c_3^0, c_3^1, c_4^1, c_5^0, c_6^0\}$. This means that the ILP model corresponding to C' in CMSA_INT is obtained by adding constraints $x_i = 0$ $(i \in \{1, 5, 6\})$ to the MDS ILP model. In contrast, the ILP model corresponding to C' in CMSA_GEN is obtained by also adding constraints $x_i = 0$ $(i \in \{1, 5, 6\})$, but additionally constraint $x_4 = 1$ is added. This means that v_4 must be part of a valid solution to the sub-instance in CMSA_GEN.

1.4.3 Experimental Evaluation

The following algorithms are included in the experimental evaluation presented in this section:

1. GREEDY: A greedy heuristic obtained by executing the heuristic from Algorithm 1.2 on page 22 in a deterministic way.
2. CPLEX: Application of CPLEX 22.1 to each considered problem instance with the default parameter values of CPLEX.
3. CMSA_INT: The standard CMSA algorithm, based on the intuitive way of defining the set of solution components.

4. CMSA_GEN: The standard CMSA algorithm, based on the generic way of defining the set of solution components.

Note that CPLEX 22.1 is used—both in standalone mode (CPLEX) and within the CMSA variants—in sequential mode. For conducting the experiments we used the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

1.4.3.1 MDS Benchmark Sets

In order to have a controlled experimentation environment, we decided to produce benchmark sets consisting of graphs of varying sizes and densities, making use of the following three network models:

1. **Erdös-Rényi graphs**: The Erdös-Rényi model [40] is a random graph model named after mathematicians Paul Erdös and Alfréd Rényi. This model is one of the earliest and simplest models for generating random graphs. The earliest version was introduced in 1959 and is characterized by two parameters: the number of nodes (n) and the probability of an edge existing between any pair of nodes (p).
2. **Watts-Strogatz graphs:** The Watts-Strogatz model is a mathematical model for generating small-world networks, which are networks characterized by a small average shortest path length between vertices while still exhibiting a high level of local clustering. This model was proposed by Duncan J. Watts and Steven Strogatz in 1998 [102].
3. **Barabási-Albert graphs:** The Barabási-Albert model is a preferential attachment model for generating scale-free networks. It was introduced by Albert-László Barabási and Réka Albert in 1999 [9]. The key feature of the Barabási-Albert model is its ability to generate networks with a scale-free degree distribution, meaning that the distribution of node degrees follows a power-law, where a small number of nodes have a very high degree while the majority have lower degrees.

In particular, we generated 30 graphs with each of the three models, and for each combination of $|V| \in \{500, 1000, 1500, 2000\}$ and four different graph densities. Note that the graph density is controlled in Erdös-Rényi graphs by the edge probability (p), in Watts-Strogatz graphs by a parameter k, and in Barabási-Albert graphs by a parameter m. In total, this benchmark set consists of 480 graphs from each of the three network models. For the generation of all these graphs we used the implementations of the three graph models from the `igraph` library.[15]

[15] https://igraph.org/.

1.4.3.2 Parameter Tuning

As mentioned already in Sect. 1.2.1, the `irace` tool was used for tuning the parameters of CMSA_INT and CMSA_GEN. The following is a list of parameters that were considered for parameter tuning:

- n_a: The number of solution constructions per CMSA iteration.
- age_{max}: The maximum age solution components may reach before being removed from the sub-instance C'.
- d_{rate}: The determinism rate for solution construction (see Sect. 1.4.1.1).
- l_{size}: The candidate set size for solution construction (see Sect. 1.4.1.1).
- t_{ILP}: The CPU time limit (in seconds) for the application of CPLEX for solving a sub-instance C'.
- $\text{cplex}_{\text{emphasis}}$: Use of heuristic emphasis in CPLEX.
- $\text{cplex}_{\text{warmstart}}$: Use of warm start in CPLEX.
- $\text{cplex}_{\text{abort}}$: Aborting CPLEX whenever then best-so-far solution S^{ILP} is improved.

Both CMSA variants were tuned exactly once for the entire benchmark set. As tuning instances, additional problem instances were generated. More specifically, for each combination of $|V|$ and graph density, exactly one tuning instance was generated by each of the three network models. This makes a total of 24 tuning instances. As computation time limit, 150 CPU seconds was used for all graphs with $|V| = 500$, 300 CPU seconds for all graphs with $|V| = 1000$, 450 CPU seconds for all graphs with $|V| = 1500$, and 600 CPU seconds for all graphs with $|V| = 2000$. Finally, `irace` was given a budget of 3000 algorithm runs.

The tuning results for both CMSA_INT and CMSA_GEN are shown in Table 1.2. In general, the preferred parameter settings of the two CMSA variants are rather similar. The number of solution constructions is rather low and the maximum age limit is rather low. This means that, in the context of the MDS problem, sub-instances should not be too large. This is because probably CPLEX is running into problems with larger sub-instances, which is also indicated by a rather high CPU time limit for CPLEX for solving sub-instances. The differences in the parameter setting for the two CMSA variants is to be found in the setting of d_{rate}, where CMSA_GEN seems to require less determinism than CMSA_INT. Another difference is shown in the use of CPLEX abort. While CMSA_INT does not make use of the abort mechanism, CMSA_GEN does make use of it.

Table 1.2 Parameters, domains and tuning results for the MDS problem

Parameter	Domain	CMSA_INT	CMSA_GEN
n_a	$\{1, \ldots, 50\}$	4	1
age_{max}	$\{1, \ldots, 10\}$	3	7
d_{rate}	[0.0, 0.99]	0.29	0.04
l_{size}	$\{3, \ldots, 50\}$	35	37
t_{ILP}	$\{1, \ldots, 20\}$	13	15
$\text{cplex}_{\text{emphasis}}$	{`true`, `false`}	`true`	`true`
$\text{cplex}_{\text{warmstart}}$	{`true`, `false`}	`false`	`false`
$\text{cplex}_{\text{abort}}$	{`true`, `false`}	`false`	`true`

1.4.3.3 Results

All four algorithmic techniques (Greedy, Cplex, Cmsa_Int and Cmsa_Gen) were applied exactly once to each of the problem instances from the benchmark set. The computation time limit for Cplex, Cmsa_Int and Cmsa_Gen was the same as the one used for tuning (see previous section). The results are shown in the form of box plots in Figs. 1.8, 1.9, and 1.10. Note that there is exactly one graphic

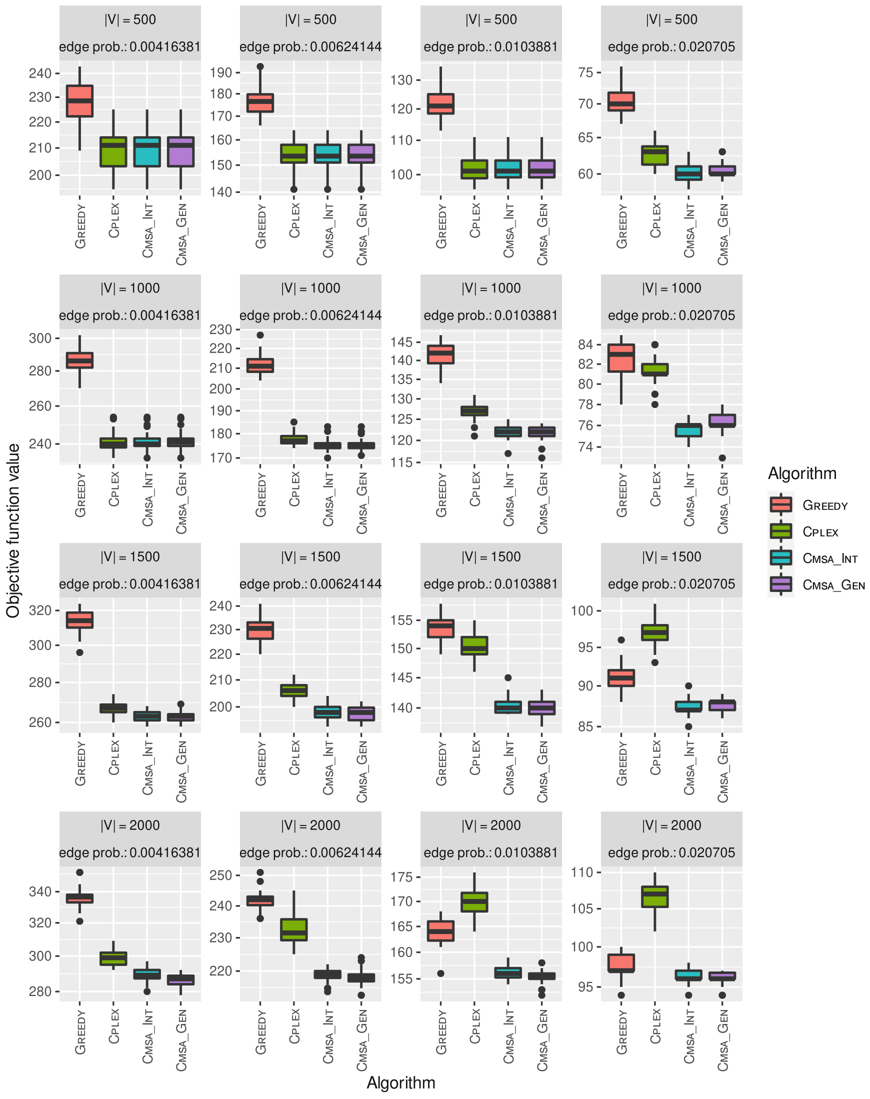


Fig. 1.8 Results for Erdös-Rényi graphs

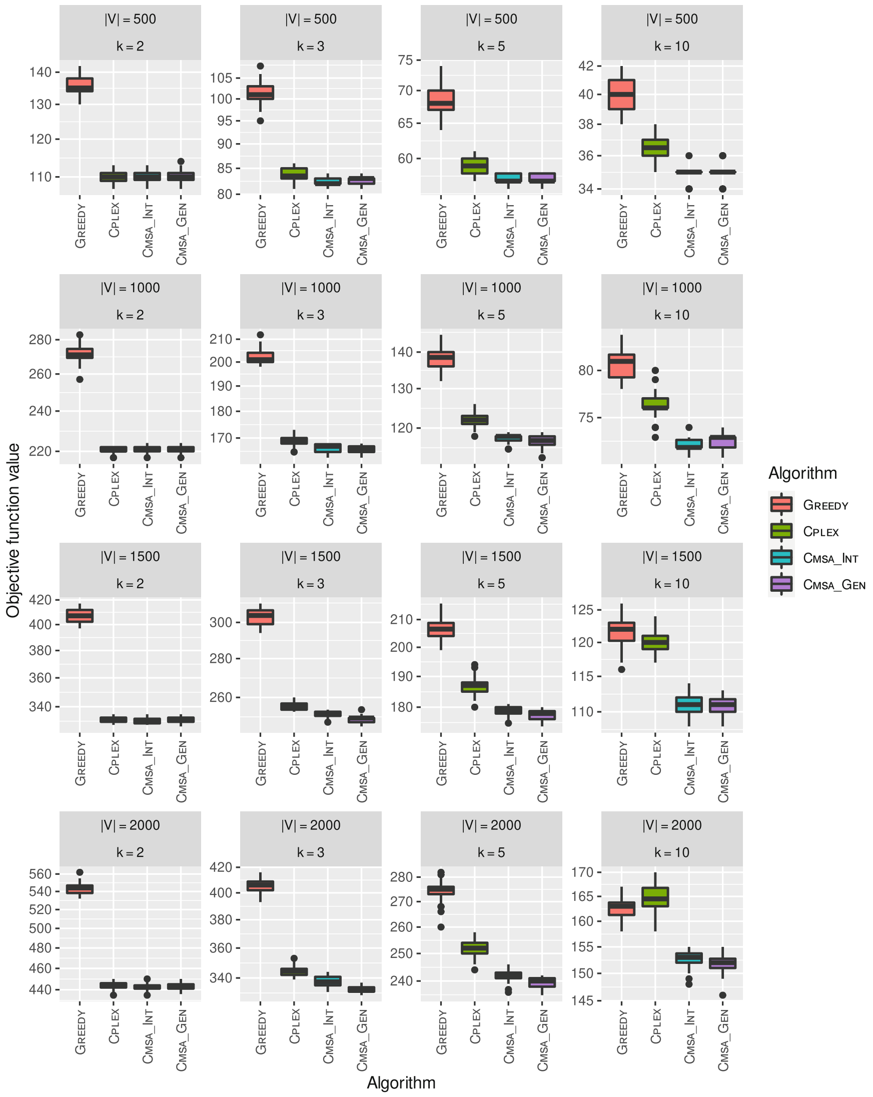


Fig. 1.9 Results for Watts-Strogatz graphs

for each network model (Erdös-Rényi, Watts-Strogatz, and Barabási-Albert). Each of these graphics contains a 4×4 grid of box plots. Hereby, the rows present the results (from top to bottom) for graphs of increasing size, and the columns (from left to right) present the results for graphs of increasing density.

To be able to support the analysis of the results with claims about the statistical significance, so-called CD plots are provided separately for the graphs of each of the three network models in Figs. 1.11, 1.12, and 1.13; see Sect. 1.2.3 on page 16 for a

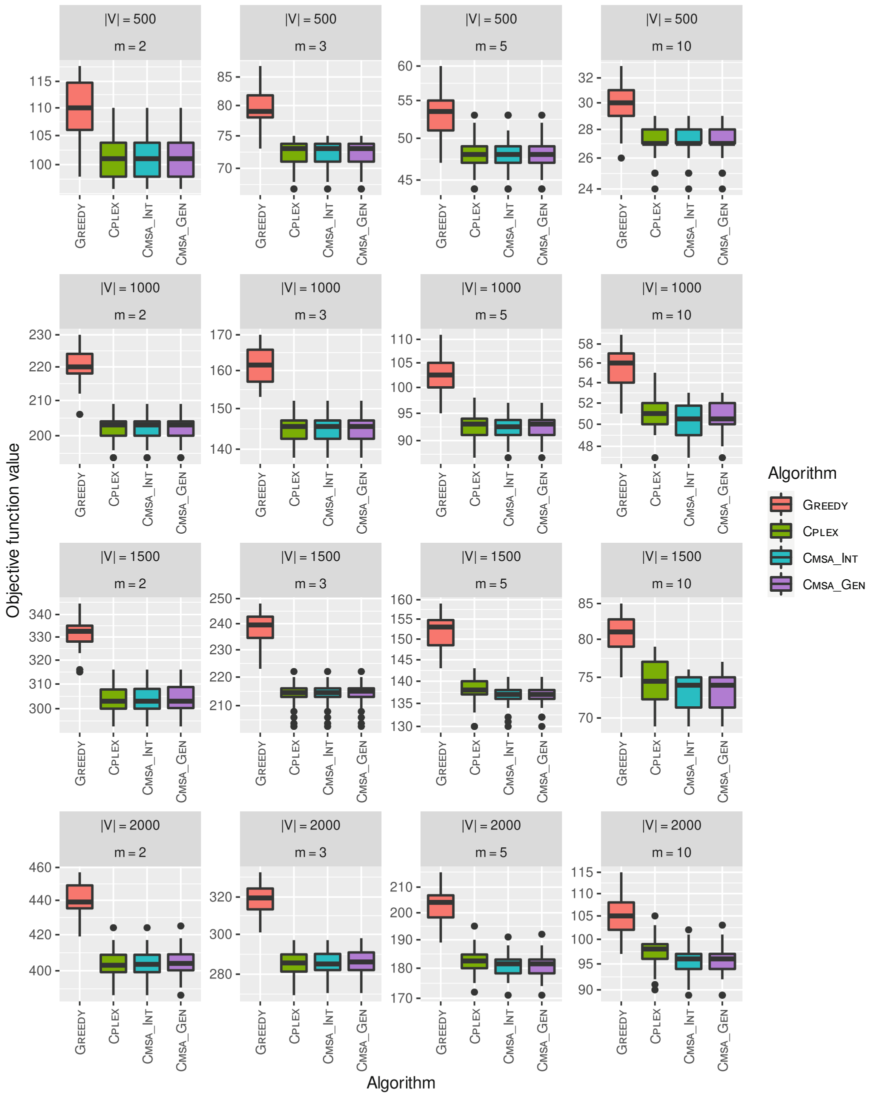


Fig. 1.10 Results for Barabási-Albert graphs

general description of CD plots. Each of the CD plot figures contains five graphics. The first one (topmost) of these graphics provides statistical information over the complete set of graphs (concerning the respective network model). The remaining four CD plot graphics convey statistical information concerning all graphs of a certain density.

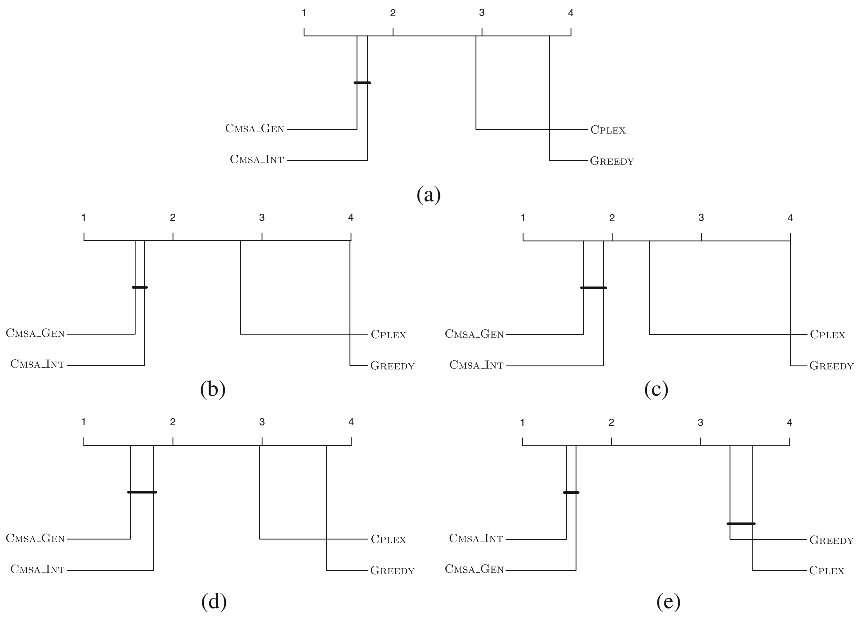


Fig. 1.11 Critical Difference (CD) plots concerning Erdös-Rényi graphs. (**a**) All graphs. (**b**) Density $p = 0.00624144$. (**c**) Density $p = 0.00416381$. (**d**) Density $p = 0.0103881$. (**e**) Density $p = 0.020705$

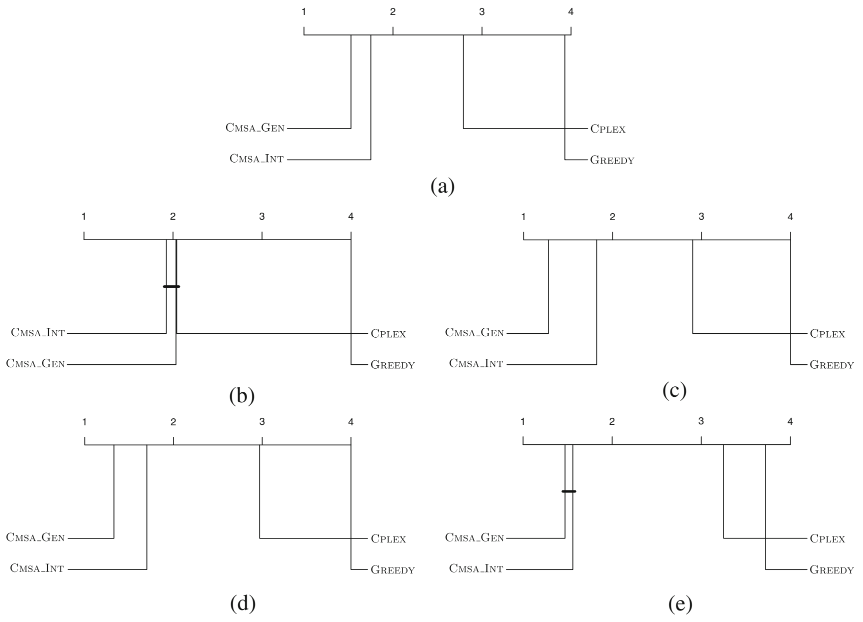


Fig. 1.12 Critical Difference (CD) plots concerning Watts-Strogatz graphs. (**a**) All graphs. (**b**) Density $k = 2$. (**c**) Density $k = 3$. (**d**) Density $k = 5$. (**e**) Density $k = 10$

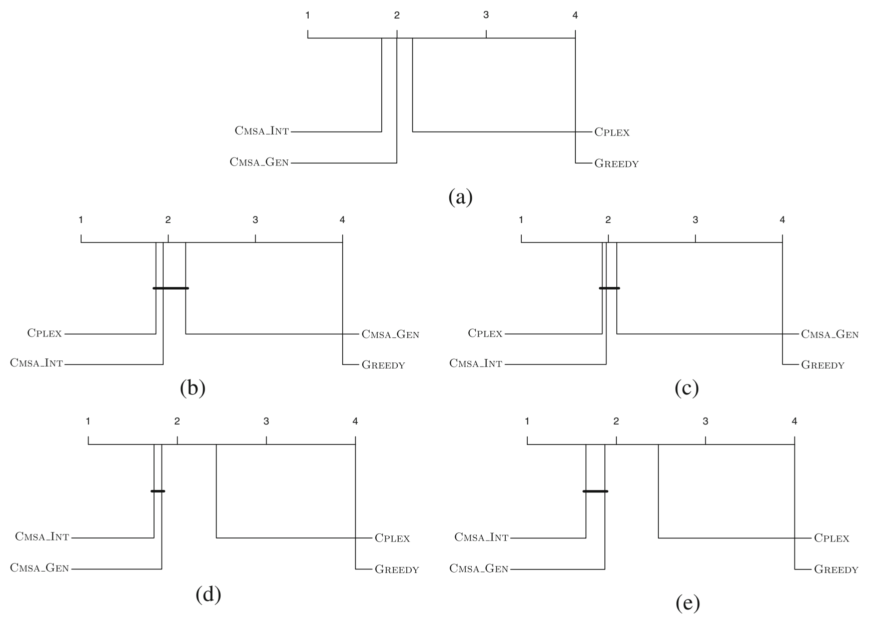


Fig. 1.13 Critical Difference (CD) plots concerning Barabási-Albert graphs. (**a**) All graphs. (**b**) Density $m = 2$. (**c**) Density $m = 3$. (**d**) Density $m = 5$. (**e**) Density $m = 10$

⊳ Main Observations Concerning the MDS Results

1. First, and most importantly, both CMSA variants generally outperform both GREEDY and CPLEX.
2. The comparison with CPLEX shows exactly the pattern that one would expect from the comparison of a hybrid technique with an exact technique: in the context of small and/or sparse graphs, for which CPLEX performs strongly, the performance of the two CMSA variants is comparable to the one of CPLEX. With growing graph size and/or density, however, both CMSA variants clearly start to outperform CPLEX. Note that CPLEX even performs worse than GREEDY in the case of the largest and densest Erdös-Rényi (Fig. 1.8) and Watts-Strogatz (Fig. 1.9) graphs.
3. Concerning the comparison between the two CMSA variants (CMSA_INT vs. CMSA_GEN) we can observe that, even though the performance of both is rather similar, CMSA_GEN outperforms CMSA_INT in the context of Erdös-Rényi and Watts-Strogatz graphs, whereas the opposite is the case for the Baragási-Albert graphs. Moreover, this claim has statistical significance in the case of the Watts-Strogatz graphs (in favor of CMSA_GEN) and in the case of the Barabási-Albert graphs (in favor of CMSA_INT). Nevertheless, when looking at the differences between the algorithm performances shown by the box plots, we can see that

these differences are rather small, with the exception of the largest and densest Watts-Strogatz graphs.

4. For what concerns the differences in algorithm performance depending on the graph models used to generate the problem instances, it can clearly be said that (at least for the algorithms studied here) the MDS is much easier to solve in Barabási-Albert graphs (resembling scale-free networks) than in Erdös-Rényi (random graphs) and Watts-Strogatz graphs (small-world networks).

Finally, we also present some STNWeb graphics that show the different behavior of CMSA_INT and CMSA_GEN in Fig. 1.14; see Sect. 1.2.2 on page 13 for a description of this type of graphic. Figure 1.14a shows the STN consisting of 10 runs of CMSA_INT and CMSA_GEN for the first Erdös-Rényi graph with 2000 vertices and the lowest density, a case for which the box plots in Fig. 1.8 show that CMSA_GEN outperforms CMSA_INT. This STNWeb graphic shows that two of the CSMA_GEN runs obtain the overall best solution found. Moreover, it shows that the trajectories of CMSA_GEN are generally longer than those of CMSA_INT. This can be explained by the fact that CMSA_GEN only generates one new solution at each iteration, which leads to a small step-size when moving through the search space. Moreover, it makes use of the abort functionality for stopping CPLEX whenever a better solution than the currently best-so-far solution S^{ILP} is obtained. This leads to the fact that the start of the algorithm trajectories of CMSA_GEN are of much worse quality than those of CMSA_INT, which in turn leads to longer search trajectories. Curiously, the 20 search trajectories do not show any overlaps. Therefore, we also produced the same STN after search space partitioning, which is obtained after clustering similar solutions and coarsening in this way the search space. The corresponding STNWeb graphic is shown in Fig. 1.14b. In particular, it can be seen that most CMSA_GEN search trajectories are attracted by the same area of the search space, marked by the two large red dots. Moreover, it can also be seen that those CMSA_INT runs that are attracted by this area of the search space, stop in inferior solutions shortly before reaching the best solutions of that area (see the large gray triangles in the south-west direction of the large red dots).

The graphics in Fig. 1.14c and d show the same for the first Watts-Strogatz graph with 1000 nodes and the highest density. In this case, the box plots of Fig. 1.9 show that CMSA_INT generally outperforms CMSA_GEN. And in fact, the STNWeb graphic after search space partitioning (Fig. 1.14d) shows that CMSA_INT runs rather quickly end up in one of two best-found solutions, whereas some of the CMSA_GEN trajectories end in the large gray triangle to the left of the smaller red dot, while other CMSA_GEN trajectories do not seem to be attracted by that area of the search space.

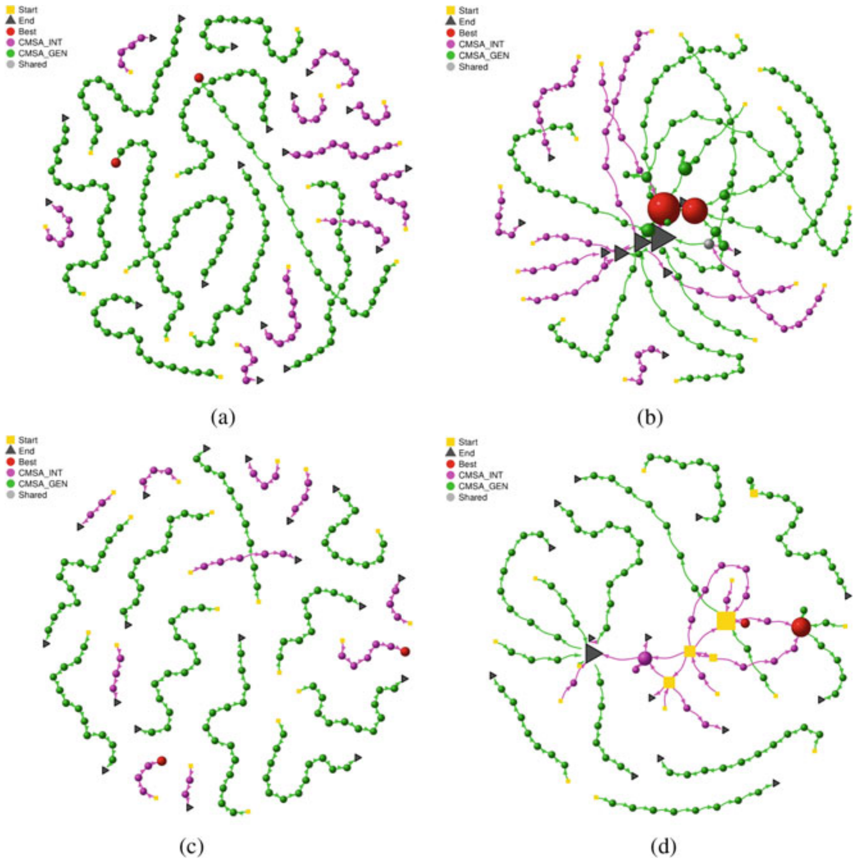


Fig. 1.14 STNWeb graphics. (**a**) and (**b**) show 10 runs of CMSA_INT and CMSA_GEN for the first Erdös-Rényi graph with 2000 vertices and the lowest density. While (**a**) shows the complete STN, (**b**) shows the same STN after partitioning. (**c**) and (**d**) show the same for the first Watts-Strogatz graph with 1000 nodes and the highest density

1.5 Algorithmic Proposals Related to CMSA

The core concept of CMSA closely aligns with that found in numerous LNS variants.[16] Both CMSA and various LNS variants operate on the principle of iteratively applying an exact technique to reduced problem instances, that is, sub-instances derived from the original problem instances. However, the manner in which CMSA generates these sub-instances differs from the conventional approach in LNS. In CMSA, there is no imposition of a predefined partial solution when

[16] See Sect. 1.1.5 for a discussion on LNS. Moreover, see Sect. 6.3.2 in Chap. 6 of this book for a comparison between CMSA and LNS.

utilizing the exact technique. Instead, CMSA narrows down the viable options for constructing a feasible solution and makes use of the exact technique for finding (if possible) the optimal solution that can be generated from this refined set of choices.

Concepts related to CMSA are explored in the works of Applegate et al. [7] and Cook and Seymour [32], who addressed the classical traveling salesman problem (TSP) as follows. Initially, in a preliminary phase, they employ a metaheuristic to generate a collection of high-quality TSP solutions. Subsequently, these solutions are merged to create a reduced problem instance, which is then solved by an exact solver. Similar approaches are known from the field of set covering and vehicle routing problems. In [27], for example, Cavaliere et al. tackle the Capacitated Vehicle Routing Problem (CVRP) by, first, applying a local search approach in order to create a pool of good routes; second, the pool of routes is refined by the application of column generation; and lastly, an exact approach is used to solve the CVRP only allowing routes from the generated pool of routes. This is done in an iterative way.

Another example of related work concerns the so-called *generate-and-solve* (GS) framework that was originally presented in [76]. This framework is a two-phase approach which works as follows:

1. **Generation phase:** In the first phase, a set of candidate solutions is generated. This process often involves using heuristics, metaheuristics, or other optimization methods to explore the solution space and find feasible and possibly high-quality solutions. In other words, techniques such as genetic algorithms, simulated annealing, or particle swarm optimization may be used to explore the solution space efficiently and discover diverse solutions.
2. **Solving phase:** The solutions generated in the first phase are typically used to create a reduced or modified version of the original problem instance. This reduction might involve simplifying the problem by fixing certain variables or constraints based on the generated solutions. The reduced problem instance is then solved optimally using an exact optimization method. Exact solvers guarantee to find the globally optimal solution within the reduced problem space.

Recent applications of this framework include the ones in [36, 85, 92].

Another prominent example of related work concerns *kernel search* [6], which is a heuristic framework based on the identification of a restricted set of promising solution components (called the kernel) and on the exact solution of sub-instances by ILP solvers. Applications include [52, 64, 93].

The concept of *solution merging* in evolutionary algorithms (EAs) typically involves combining information from multiple candidate solutions to generate new solutions. EAs, which are optimization techniques inspired by natural selection and genetics, often employ mechanisms to create diverse and potentially improved solutions over successive iterations. Solution merging is one such mechanism that aims to exploit the strengths of different candidate solutions. By merging information from different solutions, the evolutionary algorithm seeks to explore a broader space of potential solutions, facilitating the discovery of novel and high-quality solutions to the optimization problem. The specific mechanisms for solution merging can vary based on the algorithm and the nature of the optimization problem being addressed. In fact, both heuristic and exact techniques are used in the context of solution merging. Applications of solution merging include [21, 41, 91].

A last related line of work that we would like to mention here is *merge search* (MS) [60]. Just like CMSA, MS generates at each iteration a sub-instance to the original problem instance and tries to solve this sub-instance by means of an exact solver. The primary distinction between CMSA and MS lies in the way in which sub-instances are generated. CMSA primarily aims at pinpointing a substantial set of variables with fixed values in high-quality solutions. The optimization focus then shifts to the remaining set of variables. In contrast, MS seeks aggregations of variables, specifically groups with consistent (identical) values across good solutions. However, the exact values within these groups remain subject to optimization. One of the most recent applications of MS can be found in [99].

References

1. Ahuja, R.K., Orlin, J.B., Sharma, D.: Very large-scale neighborhood search. International Transactions in Operational Research **7**(4–5), 301–317 (2000)
2. Akbay, M.A., Kalayci, C.B., Blum, C.: Application of CMSA to the electric vehicle routing problem with time windows, simultaneous pickup and deliveries, and partial vehicle charging. In: Metaheuristics International Conference, pp. 1–16. Springer (2022)
3. Akbay, M.A., Kalayci, C.B., Blum, C.: Application of Adapt-CMSA to the two-echelon electric vehicle routing problem with simultaneous pickup and deliveries. In: European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar), pp. 16–33. Springer (2023)
4. Akbay, M.A., López Serrano, A., Blum, C.: A self-adaptive variant of CMSA: application to the minimum positive influence dominating set problem. International Journal of Computational Intelligence Systems **15**(1), 44 (2022)
5. Alves Viana, L.G.: Uma meta-heurística híbrida para o problema de cobertura de discos ponderados. Bachelor's thesis, Universidade Federal de Alagoas, Instituto de Computação, Maceió, Brazil (2022)
6. Angelelli, E., Mansini, R., Speranza, M.G.: Kernel search: A general heuristic for the multi-dimensional knapsack problem. Computers & Operations Research **37**(11), 2017–2026 (2010)
7. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Finding tours in the TSP. Tech. rep., Forschungsinstitut für Diskrete Mathematik, University of Bonn, Germany (1999)
8. Arora, D., Maini, P., Pinacho-Davidson, P., Blum, C.: Route planning for cooperative air-ground robots with fuel constraints: an approach based on CMSA. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 207–214 (2019)
9. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
10. Barceló, J.: Models, Traffic Models, Simulation, and Traffic Simulation, pp. 1–62. Springer New York, New York, NY (2010)
11. Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966)
12. Blum, C.: Minimum common string partition: on solving large-scale problem instances. International Transactions in Operational Research **27**(1), 91–111 (2020)

13. Blum, C., Blesa, M.J.: Construct, merge, solve and adapt: application to the repetition-free longest common subsequence problem. In: Evolutionary Computation in Combinatorial Optimization: 16th European Conference, EvoCOP 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings 16, pp. 46–57. Springer (2016)
14. Blum, C., Blesa, M.J.: A comprehensive comparison of metaheuristics for the repetition-free longest common subsequence problem. Journal of Heuristics **24**(3), 551–579 (2018)
15. Blum, C., Gambini Santos, H.: Generic CP-supported CMSA for binary integer linear programs. In: Hybrid Metaheuristics: 11th International Workshop, HM 2019, Concepción, Chile, January 16–18, 2019, Proceedings 11, pp. 1–15. Springer (2019)
16. Blum, C., Ochoa, G.: A comparative analysis of two matheuristics by means of merged local optima networks. European Journal of Operational Research **290**(1), 36–56 (2021)
17. Blum, C., Pereira, J.: Extension of the CMSA algorithm: an LP-based way for reducing sub-instances. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 285–292 (2016)
18. Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, merge, solve & adapt: a new general algorithm for combinatorial optimization. Computers & Operations Research **68**, 75–88 (2016)
19. Blum, C., Raidl, G.R.: Hybrid Metaheuristics – Powerful Tools for Optimization. Springer (2016)
20. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys **35**(3), 268–308 (2003)
21. Borisovsky, P., Dolgui, A., Eremeev, A.: Genetic algorithms for a supply management problem: MIP-recombination vs greedy decoder. European Journal of Operational Research **195**(3), 770–779 (2009)
22. Boschetti, M.A., Maniezzo, V., Roffilli, M., Bolufé Röhler, A.: Matheuristics: Optimization, simulation and control. In: M.J. Blesa, C. Blum, L. Di Gaspero, A. Roli, M. Sampels, A. Schaerf (eds.) Proceedings of HM 2009 – 6th International Workshop on Hybrid Metaheuristics, *Lecture Notes in Computer Science*, vol. 5818, pp. 171–177. Springer Berlin Heidelberg (2009)
23. Bäck, T.: Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press (1996)
24. Caicedo Solano, N.E., García Llinás, G.A., Montoya-Torres, J.R.: Towards the integration of lean principles and optimization for agricultural production systems: a conceptual review proposition. Journal of the Science of Food and Agriculture **100**(2), 453–464 (2020)
25. Calvo, B., Santafé, G.: scmamp: Statistical comparison of multiple algorithms in multiple problems. The R Journal **8**(1) (2016)
26. Caserta, M., Voß, S.: A corridor method based hybrid algorithm for redundancy allocation. Journal of Heuristics **22**(4), 405–429 (2016)
27. Cavaliere, F., Bendotti, E., Fischetti, M.: An integrated local-search/set-partitioning refinement heuristic for the capacitated vehicle routing problem. Mathematical Programming Computation **14**(4), 749–779 (2022)
28. Chacón Sartori, C., Blum, C., Ochoa, G.: STNWeb: a new visualization tool for analyzing optimization algorithms. Software Impacts **17**, 100558 (2023)
29. Chopra, S., Meindl, P.: Supply Chain Management. Strategy, Planning & Operation, pp. 265–275. Gabler, Wiesbaden (2007)
30. Clerc, M.: Particle Swarm Optimization. ISTE Ltd (2006)
31. Collins, T.D.: Applying software visualization technology to support the use of evolutionary algorithms. Journal of Visual Languages & Computing **14**(2), 123–150 (2003)
32. Cook, W., Seymour, P.: Tour merging via branch-decomposition. INFORMS Journal on Computing **15**(3), 233–248 (2003)
33. Dantzig, G.B.: Maximization of a linear function of variables subject to linear inequalities. Activity Analysis of Production and Allocation **13**, 339–347 (1951)
34. Demir, E., Bektaş, T., Laporte, G.: An adaptive large neighborhood search heuristic for the pollution-routing problem. European Journal of Operational Research **223**(2), 346–359 (2012)

35. Demšar, J.: Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research **7**, 1–30 (2006)
36. Dias Saraiva, R., Nepomuceno, N., Rogério Pinheiro, P.: A two-phase approach for single container loading with weakly heterogeneous boxes. Algorithms **12**, 67 (2019)
37. Djukanović, M., Kartelj, A., Blum, C.: Self-adaptive CMSA for solving the multidimensional multi-way number partitioning problem. Expert Systems with Applications p. 120762 (2023)
38. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
39. Dupin, N., Talbi, E.G.: Matheuristics to optimize refueling and maintenance planning of nuclear power plants. Journal of Heuristics **27**(1–2), 63–105 (2021)
40. Erdös, P., Rényi, A.: On random graphs I. Publ. math. debrecen **6**(290–297), 18 (1959)
41. Eremeev, A.V., Kovalenko, Y.V.: A memetic algorithm with optimal recombination for the asymmetric travelling salesman problem. Memetic Computing **12**(1), 23–36 (2020)
42. Ernst, A.T., Jiang, H., Krishnamoorthy, M., Sier, D.: Staff scheduling and rostering: A review of applications, methods and models. European Journal of Operational Research **153**(1), 3–27 (2004)
43. Eskandarpour, M., Dejax, P., Péton, O.: A large neighborhood search heuristic for supply chain network design. Computers & Operations Research **80**, 23–37 (2017)
44. Ferrer, J., Chicano, F., Ortega-Toro, J.A.: CMSA algorithm for solving the prioritized pairwise test data generation problem in software product lines. Journal of Heuristics **27**, 229–249 (2021)
45. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1), 23–47 (2003)
46. García, S., Fernández, A., Luengo, J., Herrera, F.: Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. Information Sciences **180**(10), 2044–2064 (2010)
47. Garcia, S., Herrera, F.: An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. Journal of Machine Learning Research **9**(12) (2008)
48. Gendreau, M., Potvin, J.Y. (eds.): Handbook of Metaheuristics, 3rd edn. Springer Publishing Company, Incorporated (2019)
49. Ghirardi, M., Salassa, F.: A simple and effective algorithm for the maximum happy vertices problem. Top **30**(1), 181–193 (2022)
50. Glover, F.: Ejection chains, reference structures and alternating path methods for traveling salesman problems. Discrete Applied Mathematics **65**(1–3), 223–253 (1996)
51. Glover, F., Laguna, M.: Tabu Search. Springer Science+Business Media. LLC (1997)
52. Guastaroba, G., Speranza, M.G.: Kernel search: An application to the index tracking problem. European Journal of Operational Research **217**(1), 54–68 (2012)
53. Harvey, C.R., Liechty, J.C., Liechty, M.W., Müller, P.: Portfolio selection with higher moments. Quantitative Finance **10**(5), 469–485 (2010)
54. Hawa, A.: Exact and evolutionary algorithms for the score-constrained packing problem. Ph.D. thesis, Cardiff University (2020)
55. Hillier, F.S., Lieberman, G.J.: Introduction to Operations Research, 11th edn. McGraw-Hill Education (2018)
56. Horton, I., Vn Weert, P.: Beginning C++17: From Novice to Professional. Apress (2018)
57. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009)
58. Katsila, T., Spyroulias, G.A., Patrinos, G.P., Matsoukas, M.T.: Computational approaches in target identification and drug discovery. Computational and Structural Biotechnology Journal **14**, 177–184 (2016)
59. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of ICNN'95 – International Conference on Neural Networks, vol. 4, pp. 1942–1948. IEEE (1995)
60. Kenny, A., Li, X., Ernst, A.T.: A merge search algorithm and its application to the constrained pit problem in mining. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 316–323 (2018)
61. Kerzner, H.: Project management: a systems approach to planning, scheduling, and controlling. John Wiley & Sons (2017)

62. Kirkpatrick, S., Gelatt Jr, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
63. Lalla-Ruiz, E., Voß, S.: POPMUSIC as a matheuristic for the berth allocation problem. Annals of Mathematics and Artificial Intelligence **76**(1–2), 173–189 (2016)
64. Lamanna, L., Mansini, R., Zanotti, R.: A two-phase kernel search variant for the multidimensional multiple-choice knapsack problem. European Journal of Operational Research **297**(1), 53–65 (2022)
65. Lee, S.Y., Lee, I.B., Yeo, U.H., Kim, R.W., Kim, J.G.: Optimal sensor placement for monitoring and controlling greenhouse internal environments. Biosystems Engineering **188**, 190–206 (2019)
66. Lewis, R., Thiruvady, D., Morgan, K.: Finding happiness: An analysis of the maximum happy vertices problem. Computers & Operations Research **103**, 265–276 (2019)
67. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. The Journal of Machine Learning Research **18**(1), 6765–6816 (2017)
68. Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., Hutter, F.: SMAC3: a versatile bayesian optimization package for hyperparameter optimization. The Journal of Machine Learning Research **23**(1), 2475–2483 (2022)
69. Lizárraga, E., Blesa, M.J., Blum, C.: Construct, merge, solve and adapt versus large neighborhood search for solving the multi-dimensional knapsack problem: Which one works better when? In: Evolutionary Computation in Combinatorial Optimization: 17th European Conference, EvoCOP 2017, Amsterdam, The Netherlands, April 19–21, 2017, Proceedings 17, pp. 60–74. Springer (2017)
70. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
71. Lorenzo, A.D., Medvet, E., Tušar, T., Bartoli, A.: An analysis of dimensionality reduction techniques for visualizing evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. ACM (2019)
72. Martí, R., Pardalos, P.M., Resende, M.G.: Handbook of Heuristics. Springer Publishing Company, Incorporated (2018)
73. Michalak, K.: Low-dimensional Euclidean embedding for visualization of search spaces in combinatorial optimization. IEEE Transactions on Evolutionary Computation **23**(2), 232–246 (2019)
74. Mladenović, N., Hansen, P.: Variable neighborhood search. Computers & Operations Research **24**(11), 1097–1100 (1997)
75. Najjar, M., Figueiredo, K., Hammad, A.W., Haddad, A.: Integrated optimization with building information modeling and life cycle assessment for generating energy efficient buildings. Applied Energy **250**, 1366–1382 (2019)
76. Nepomuceno, N., Pinheiro, P., Coelho, A.L.: Tackling the container loading problem: a hybrid approach based on integer linear programming and genetic algorithms. In: Evolutionary Computation in Combinatorial Optimization: 7th European Conference, EvoCOP 2007, Valencia, Spain, April 11-13, 2007. Proceedings, pp. 154–165. Springer (2007)
77. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer Series in Operations Research and Financial Engineering. Springer (2006)
78. Ochoa, G., Malan, K.M., Blum, C.: Search trajectory networks: A tool for analysing and visualising the behaviour of metaheuristics. Applied Soft Computing **109**, 107492 (2021)
79. de Oliveira, E.B., da Silva Batista, M., Pinheiro, R.G.S.: Uma abordagem híbrida CMSA para o problema da cadeia de caracteres mais próxima. In: Proceedings of the Simpósio Brasileiro de Pesquisa Operacional, vol. 55 (2023)
80. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications (1998)
81. Pardalos, P.M., Resende, M.G.: Handbook of Applied Optimization. Oxford University Press (2002)

82. Piliouras, T.C.: Network design: management and technical perspectives. CRC press (2004)
83. Pinacho-Davidson, P., Bouamama, S., Blum, C.: Application of CMSA to the minimum capacitated dominating set problem. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 321–328 (2019)
84. Pinedo, M.L.: Scheduling, vol. 29. Springer (2012)
85. Pinheiro, P.R., Coelho, A.L.V., Aguiar, A.B., Sobreira Neto, A.d.M.: Towards aid by generate and solve methodology: application in the problem of coverage and connectivity in wireless sensor networks. International Journal of Distributed Sensor Networks **8**(10), 790459 (2012)
86. Pisinger, D., Røpke, S.: Large Neighborhood Search. In: M. Gendreau, J.Y. Potvin (eds.) Handbook of Metaheuristics, *International Series in Operations Research & Management Science*, vol. 146, pp. 399–419. Springer US (2010)
87. Pohlheim, H.: Multidimensional scaling for evolutionary algorithms – visualization of the path through search space and solution space using Sammon mapping. Artificial Life **12**, 203–209 (2006)
88. Rosati, R.M., Bouamama, S., Blum, C.: Construct, merge, solve and adapt applied to the maximum disjoint dominating sets problem. In: L. Di Gaspero, P. Festa, A. Nakib, M. Pavone (eds.) Metaheuristics, pp. 306–321. Springer International Publishing, Cham (2023)
89. Rosati, R.M., Bouamama, S., Blum, C.: Multi-constructor CMSA for the maximum disjoint dominating sets problem. Computers & Operations Research **161**, 106450 (2024)
90. Rosati, R.M., Kletzander, L., Blum, C., Musliu, N., Schaerf, A.: Construct, merge, solve and adapt applied to a bus driver scheduling problem with complex break constraints. In: International Conference of the Italian Association for Artificial Intelligence, pp. 254–267. Springer (2022)
91. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
92. Sá Santos, J.V., Nepomuceno, N.: Computational performance evaluation of column generation and generate-and-solve techniques for the one-dimensional cutting stock problem. Algorithms **15**(11), 394 (2022)
93. Santos-Peñate, D.R., Campos-Rodríguez, C.M., Moreno-Pérez, J.A.: A kernel search matheuristic to solve the discrete leader-follower location problem. Networks and Spatial Economics **51**, 1–26 (2019)
94. Schmid, V.: Hybrid large neighborhood search for the bus rapid transit route design problem. European Journal of Operational Research **238**(2), 427–437 (2014)
95. Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., Dueck, G.: Record breaking optimization results using the ruin and recreate principle. Journal of Computational Physics **159**(2), 139–171 (2000)
96. Silver, E.A., Pyke, D.F., Peterson, R.: Inventory management and production planning and scheduling, vol. 3. Wiley New York (1998)
97. Talbi, E. (ed.): Hybrid Metaheuristics, *Studies in Computational Intelligence*, vol. 434. Springer (2013)
98. Thiruvady, D., Blum, C., Ernst, A.T.: Maximising the net present value of project schedules using CMSA and parallel ACO. In: Hybrid Metaheuristics: 11th International Workshop, HM 2019, Concepción, Chile, January 16–18, 2019, Proceedings 11, pp. 16–30. Springer (2019)
99. Thiruvady, D., Blum, C., Ernst, A.T.: Solution merging in matheuristics for resource constrained job scheduling. Algorithms **13**(10), 256 (2020)
100. Thiruvady, D., Lewis, R.: Recombinative approaches for the maximum happy vertices problem. Swarm and Evolutionary Computation **75**, 101188 (2022)
101. Toth, P., Vigo, D.: The Vehicle Routing Problem. SIAM (2001)
102. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)
103. Winston, W.L.: Operations Research: Applications and Algorithms. Thomson Learning, Inc. (2004)
104. Zhou, J., Zhang, P.: Simple heuristics for the rooted max tree coverage problem. In: W. Wu, J. Guo (eds.) Combinatorial Optimization and Applications, pp. 252–264. Springer Nature Switzerland, Cham (2024)

Chapter 2
Self-adaptive CMSA

Abstract While the standard CMSA algorithm has proven its utility across a range of different combinatorial optimization problems, certain applications have highlighted its susceptibility to variations in parameter settings. In order to deal with this issue, an innovative self-adaptive variant of the CMSA algorithm, termed ADAPT_CMSA, was developed and will be presented in this chapter. The primary objective is to mitigate the parameter sensitivity that might occur in the standard CMSA variant. The merits of this novel CMSA variant are substantiated through its application to the Minimum Positive Influence Dominating Set (MPIDS) problem and to the Far From Most String (FFMS) problem. Notably, ADAPT_CMSA distinguishes itself from standard CMSA by not presenting the need for a computationally intensive parameter tuning process across subsets within the considered set of problem instances.

2.1 Introduction

The research community on metaheuristics [16] acknowledges a prevalent issue related to an algorithm's high sensitivity to variations in parameter values. In this context, a metaheuristic algorithm is deemed *parameter sensitive* when two conditions are met: (1) the algorithm's performance for particular instances or instance groups significantly relies on specific parameter values, and (2) the necessary parameter values differ notably across distinct instances or instance groups. The research community views excessive sensitivity to parameter settings as an unfavorable aspect.

A noticeable susceptibility to parameter values has also been observed in certain applications of CMSA as documented in the literature. An illustrative case is the initial application of standard CMSA to the Minimum Positive Influence Dominating Set (MPIDS) problem [1], an NP-hard combinatorial optimization problem. In response to this issue, a self-adaptive variant of CMSA named ADAPT_CMSA was proposed in [4]. The primary objective during the development of ADAPT_CMSA was to obtain a CMSA variant that exhibits reduced sensitivity to parameter values. To validate its effectiveness, in this chapter, we present the initial application of

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3_2

Algorithm 2.1: Pseudo-code of self-adaptive CMSA: ADAPT_CMSA

```
 1: input 1: Complete set of solution components $C$
 2: input 2: Values for ADAPT_CMSA parameters $t_{\text{prop}}$, $t_{\text{ILP}}$
 3: input 3: Values for solution construction parameters $\alpha^{\text{LB}}$, $\alpha^{\text{UB}}$, $\alpha_{\text{red}}$
 4: $S^{\text{bsf}} :=$ GenerateGreedySolution($C$)
 5: $n_{\text{a}} := 1$; $\alpha_{\text{bsf}} := \alpha^{\text{UB}}$; $C' := S^{\text{bsf}}$
 6: while CPU time limit not reached do
 7:   for $i := 1, \ldots, n_{\text{a}}$ do
 8:     $S :=$ ProbabilisticSolutionConstruction($C$, $S^{\text{bsf}}$, $\alpha_{\text{bsf}}$)
 9:     $C' := C' \cup S$
10:   end for
11:   $(S^{\text{ILP}}, t_{\text{solve}}) :=$ SolveSubinstance($C'$, $t_{\text{ILP}}$) {This function returns two objects: (1) the obtained solution ($S^{\text{ILP}}$), (2) the required computation time ($t_{\text{solve}}$)}
12:   if $t_{\text{solve}} < t_{\text{prop}} \cdot t_{\text{ILP}}$ and $\alpha_{\text{bsf}} > \alpha^{\text{LB}}$ then $\alpha_{\text{bsf}} := \alpha_{\text{bsf}} - \alpha_{\text{red}}$ end if
13:   if $f(S^{\text{ILP}}) < f(S^{\text{bsf}})$ then
14:     $S^{\text{bsf}} := S^{\text{ILP}}$
15:     $n_{\text{a}} := 1$
16:   else
17:     if $f(S^{\text{ILP}}) > f(S^{\text{bsf}})$ then
18:       if $n_a = 1$ then $\alpha_{\text{bsf}} := \min\{\alpha_{\text{bsf}} + \frac{\alpha_{\text{red}}}{10}, \alpha^{\text{UB}}\}$ else $n_a = 1$ end if
19:     else
20:       $n_{\text{a}} := n_{\text{a}} + 1$
21:     end if
22:   end if
23:   $C' := S^{\text{bsf}}$
24: end while
25: output: $S^{\text{bsf}}$
```

CMSA to the MPIDS problem. In addition, we describe its application to a second NP-hard combinatorial optimization problem known as the Far From Most String (FFMS) problem.

The results obtained will demonstrate multiple advantages of ADAPT_CMSA over the standard CMSA. Firstly, ADAPT_CMSA eliminates the need for fine-tuning specific parameters for subsets within the designated benchmark set. Following a single round of parameter tuning, ADAPT_CMSA performs effectively across the entire benchmark set, encompassing instances of diverse sizes. Secondly, in the realm of large-scale problem instances, ADAPT_CMSA distinctly outperforms standard CMSA. Even with specialized tuning efforts, standard CMSA cannot match the competitive performance achieved by ADAPT_CMSA in handling such large-scale problem instances.

Note that ADAPT_CMSA has already been used in [2–4, 8].

2.2 Self-adaptive CMSA: General Description

The pseudo-code for ADAPT_CMSA is outlined in Algorithm 2.1. A prominent distinction from the standard CMSA—see Algorithm 1.1 on page 19—is the absence of age values. ADAPT_CMSA operates with a fixed maximum age of one, meaning that after each iteration, all solution components, except those forming part of the best-so-far solution S^{bsf}, are purged from the sub-instance C' (refer to line 23). This is done because ADAPT_CMSA disposes of an alternative way of regulating the size of the sub-instances. Another variation is evident in function ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}), responsible for the probabilistic generation of solutions in each algorithm iteration (see line 8). Notably, this function takes as input, in addition to the set of all possible solution components (C), the current best-so-far solution S^{bsf}, and a parameter α_{bsf} (where $0 \leq \alpha_{\mathrm{bsf}} < 1$). This parameter influences the construction of new solutions, biasing them towards the best-so-far solution S^{bsf}. Specifically, a higher value of α_{bsf} results in a greater similarity of the solutions generated in ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}) to S^{bsf}.

The dynamic adjustment of the α_{bsf} value is a self-adaptive feature in ADAPT_CMSA. Initially, ADAPT_CMSA requires as input lower and upper bounds, α^{LB} and α^{UB} respectively, for the α_{bsf} value. Additionally, the step size, α_{red}, for reducing α_{bsf} must be provided as input. The algorithm commences by setting α_{bsf} to its maximum value, α^{UB} (see line 5).[1] If the resulting ILP can be solved within a computation time t_{solve} that is below a fraction t_{prop} of the maximally allowed computation time t_{ILP}, the α_{bsf} value is decreased by α_{red} (see line 12). The rationale behind this adjustment lies in the following reasoning: when the ILP is easily solvable, it indicates a relatively small search space due to a low number of free variables. To increase the freedom in the ILP by introducing more free variables, the solutions generated in ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}) should exhibit greater dissimilarity to S^{bsf}. This goal is achieved by reducing the α_{bsf} value.

The second self-adaptive aspect managed by ADAPT_CMSA pertains to the number of solution constructions per iteration (n_a), outlined in lines 13–22. Initially, the algorithm sets n_a to 1 (see line 5). Additionally, if the solution of the reduced ILP (S^{ILP}) improves upon the best-so-far solution S^{bsf}, n_a is reset to one (see line 15). Conversely, if the S^{ILP} solution is strictly inferior to S^{bsf}, it indicates that the corresponding sub-instance was too large (resp. difficult) for the ILP solver to complete the solving-process within t_{ILP} seconds. In this scenario, if $n_a = 1$, the α_{bsf} value is marginally increased (by $\frac{\alpha_{\mathrm{red}}}{10}$); otherwise, n_a is reset to one. In the event that $f(S^{\mathrm{ILP}}) = f(S^{\mathrm{bsf}})$, n_a is incremented by one (see line 20). This adjustment is made because the sub-instance did not yield a superior solution to S^{bsf}, while still being

[1] This implies that solutions generated in this manner will exhibit greater similarity to S^{bsf} compared to lower values of α_{bsf}.

solved to optimality within the allotted computation time of t_{ILP} seconds, indicating that the sub-instance's size should be expanded.

2.3 Application to the MPIDS Problem

In this section, the application of ADAPT_CMSA to the MPIDS problem will be presented. This problem has some relevance in the realm of social networks. Picture a scenario where the vertices and edges within such a social network symbolize individuals (persons) and their respective relationships/interactions. Generally, information disseminated within social networks holds the potential to exert a substantial impact, be it positive or negative, on segments of society. The social norms theory indicates that individuals' behavior can be influenced by the perception of others' thoughts and actions [9]. Consequently, leveraging relationships among people in social networks can be advantageous for attaining economic and/or societal benefits. In this context, the objective of the MPIDS problem is to identify a small subset of influential individuals (or key individuals) to accelerate the dissemination of positive influence in a social network [11, 12]. The MPIDS problem also finds alternative applications in e-learning software [18], online business [14], as well as issues related to drinking, smoking, and other substance-related concerns [17].

Here is a technical description of the MPIDS problem: Consider an undirected graph $G = (V, E)$ with $|V| = n$ vertices containing neither loops nor parallel edges.[2] A subset $s \subseteq V$ is considered a valid solution if it satisfies the following condition: for each vertex $v \in V$, at least half of its neighbors must be part of s. It is important to note that if G is connected, any valid solution s also is a dominating set of G.[3] The goal of the MPIDS problem is to identify a valid solution $s^* \subseteq V$ with the minimum possible size. In other words, for any valid solution $s \subseteq V$, the objective function value is defined as $f(s) := |s|$. It is worth mentioning that the solution $s := V$ is a trivial one for the problem. An example of the MPIDS problem is provided in Fig. 2.1.

Regarding complexity, it is important to highlight that the MPIDS problem falls into the category of NP-hard problems. Several algorithmic methods have been proposed in the literature to address the MPIDS problem, with the current leading approach being a method based on local search as detailed in [15]. However, it is crucial to emphasize that the primary focus of this chapter is to showcase the merits of ADAPT_CMSA in comparison to standard CMSA, rather than engaging in a comparison with the current state-of-the-art methods.

[2] Loops (also known as self-loops or self-edges) in undirected graphs are edges (v, v) from a node v to itself.

[3] See Sect. 1.4 of Chap. 1 for the definition of a dominating set.

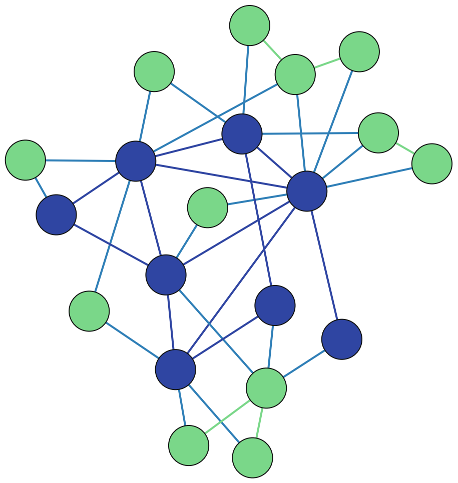


Fig. 2.1 An undirected graph on 20 vertices. The optimal MPIDS solution contains the vertices marked in blue color

A well-known ILP model for the MPIDS problem from the related literature is based on a binary variable x_i for each vertex $v_i \in V$. This ILP model can be expressed as follows.

$$\min \quad \sum_{i=1}^{n} x_i \tag{2.1}$$

$$\text{subject to} \quad \sum_{v_j \in N(v_i)} x_j \geq \left\lceil \frac{deg(v_i)}{2} \right\rceil \quad \forall\, v_i \in V \tag{2.2}$$

$$x_i \in \{0, 1\}$$

In this context, $N(v_i)$ represents the neighborhood of vertex v_i in the input graph G. Additionally, $deg(v_i)$ stands for the degree of vertex v_i, with $deg(v_i) := |N(v_i)|$. Equation (2.2) imposes a requirement on any valid solution, ensuring that it includes at least half of the neighbors of each vertex $v_i \in V$.

2.3.1 Generic Definition of the Solution Components

As in Sect. 1.4.1 on page 21 we will define the set C of solution components for the application of standard CMSA_GEN and ADAPT_CMSA in a generic way.

In other words, we introduce two solution components, namely c_i^0 and c_i^1, for each binary variable x_i where i ranges from 1 to the total number of vertices $|V|$. Specifically, c_i^0 corresponds to $x_i = 0$, and c_i^1 corresponds to $x_i = 1$. The set $C = \{c_1^0, \ldots, c_n^0, c_1^1, \ldots, c_n^1\}$ encompasses the entire set of $2n$ solution components.

Algorithm 2.2: Solution construction procedure for the MPIDS problem

```
 1: CMSA input: solution construction parameters $d_{\text{rate}}$, $l_{\text{size}}$
 2: ADAPT_CMSA input: solution construction parameter $\alpha_{\text{bsf}}$
 3: $s := s_{\text{par}}$ $\{s_{\text{par}} \subset V$ is obtained from a pre-processing procedure$\}$
 4: while $s$ is not a valid solution do
 5:   Let $U \subseteq V$ be the set of uncovered vertices
 6:   $v := \text{argmin}\{deg(v') \mid v' \in U\}$
 7:   while $|N(v) \cap s| < \left\lceil \frac{deg(v)}{2} \right\rceil$ do
 8:     $\hat{v} :=$ ChooseFrom$(N(v) \setminus s)$
 9:     $s := s \cup \{\hat{v}\}$
10:   end while
11: end while
12: output: valid solution $s$
```

A *candidate solution* S constitutes a subset of C with a cardinality of $|S| = n$. Additionally, S includes precisely one of the two components, c_i^0 or c_i^1, for each i within the range of 1 to $|V|$. Finally, a candidate solution S attains the status of a valid solution if it satisfies the constraints of the considered optimization problem.

2.3.2 Constructing Solutions to the MPIDS Problem

First of all, note that—as in the case of the MDS problem in Sect. 1.4.1.1 on page 22—a valid MPIDS solution is constructed in terms of a set of vertices. Afterwards, it is converted into the corresponding set of solution components. Valid MPIDS solutions must be constructed in the following three algorithmic functions:

1. Function ProbabilisticSolutionConstruction(C) of standard CMSA_GEN; see line 8 of Algorithm 1.1 on page 19.
2. Function GenerateGreedySolution(C) of ADAPT_CMSA; see line 4 of Algorithm 2.1 on page 42.
3. Function ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}) of ADAPT_CMSA; see line 8 of Algorithm 2.1 on page 42.

All three functions utilize the solution construction mechanism from the greedy procedure outlined in [7]. However, the first and third functions mentioned above employ this greedy procedure in a probabilistic manner. It is worth noting that the method of introducing probabilistic elements into this greedy procedure differs between these two functions. For the ensuing discussion, a vertex $v \in V$ is deemed *covered* concerning a (partial) solution s if and only if at least half of its neighbors are included in s. Conversely, if this condition is not met, v is said to be *uncovered*.

The construction mechanism employed is outlined in Algorithm 2.2. Initially, each solution s undergoing construction is initialized with a set $s_{\text{par}} \subset V$ comprising nodes that must provenly form part of any optimal solution, as indicated in line 3. It is important to note that s_{par} is generated by the application of a pre-processing procedure as detailed in [7].

During each step of the solution construction process, the following is done. Firstly, the set U of all vertices not yet covered by the vertices in the (partial) solution s is identified; see line 5. Subsequently, a node $v \in U$ is chosen such that $deg(v) \leq deg(v')$ holds for all $v' \in U$ (line 6). Next, nodes from $N(v) \setminus s$ are incrementally added to s while $|N(v) \cap s| < \left\lceil \frac{deg(v)}{2} \right\rceil$, as illustrated in lines 7-10. Here, the function ChooseFrom($N(v) \setminus s$) is responsible for selecting exactly one vertex from $N(v) \setminus s$ in each iteration of the while loop.

> Implementation of ChooseFrom($N(v) \setminus s$) in CMSA_GEN

First, a candidate list L is initialized, containing all vertices $v' \in N(v) \setminus s$. Each vertex v' within L is evaluated by its *cover degree* $cov_{\text{deg}}(v')$, denoting the number of uncovered adjacent vertices to v'. It is noteworthy that the vertices in L are arranged in descending order based on their cover degree values. Subsequently, a uniform random number r is generated within the interval [0, 1]. If $r \leq d_{\text{rate}}$ (where d_{rate} is the determinism rate), the vertex with the highest cover degree is chosen and incorporated into s. Conversely, if r exceeds d_{rate}, a vertex is randomly selected from the restricted candidate list, which comprises the initial l_{size} vertices of L. Here, l_{size} denotes the size of the restricted candidate list, and all vertices within it have an equal probability of $\frac{1}{l_{\text{size}}}$ for selection.

> Implementation of ChooseFrom($N(v) \setminus s$) in ADAPT_CMSA

First, for each vertex $v_i \in N(v) \setminus s$ where $c_i \in S^{\text{bsf}}$, a value

$$q(v_i) := (cov_{\text{deg}}(v_i) + 1) \cdot \alpha_{\text{bsf}} \tag{2.3}$$

is assigned, while for all other vertices $v_j \in N(v) \setminus s$, the assigned value is

$$q(v_j) := (cov_{\text{deg}}(v_j) + 1) \cdot (1 - \alpha_{\text{bsf}}) \ . \tag{2.4}$$

Subsequently, a vertex $\hat{v}$ is selected from $N(v) \setminus s$ based on the following probabilities:

$$\mathbf{p}(v') := \frac{q(v')}{\sum_{v'' \in N(v) \setminus s} q(v'')} \quad \forall\, v' \in N(v) \setminus s \tag{2.5}$$

Note that the bias towards the best-so-far solution S^{bsf} is determined by the parameter $\alpha_{\text{bsf}} \in [0, 1]$. A higher value of α_{bsf} strengthens this bias. It is important to note that such bias is absent in the standard CMSA.

Finally, note that—after the construction of a solution s—this solution is transformed into a corresponding solution S containing solution component c_i^1 for each $v_i \in s$, and solution component c_i^0 for each $v_i \in V \setminus s$.

2.3.3 Sub-instance Solving

A sub-instance C' is solved in function SolveSubinstance(C', t_{ILP}) of Algorithm 1.1 (see page 19) and of Algorithm 2.1 (see page 42) in exactly the same way. In particular, the following additional constraints are added to the MPIDS ILP model (for $i = 1, \ldots, |V|$) and the resulting model is solved by CPLEX:

$$x_i = 0 \qquad \text{if } c_i^0 \in C' \text{ and } c_i^1 \notin C' \tag{2.6}$$

$$x_i = 1 \qquad \text{if } c_i^0 \notin C' \text{ and } c_i^1 \in C' \tag{2.7}$$

In other words, if C' only contains the solution component c_i^0 corresponding to $x_i = 0$, then the value of x_i is fixed to zero. Similarly, if C' only contains the solution component c_i^1 corresponding to $x_i = 1$, then the value of x_i is fixed to one. Otherwise, if C' contains both solution components c_i^0 and c_i^1 then variable x_i is left free, which means that v_i might, or not, be included in a solution to the sub-instance. Note that this way of solving sub-instances is exactly the same as outlined in the context of CMSA_GEN for the MDS problem in Sect. 1.4.2.2 on page 25. The only difference between CMSA_GEN and ADAPT_CMSA is that function SolveSubinstance(C', t_{ILP})—in the case of ADAPT_CMSA—returns also the solving time t_{solve} in addition to the CPLEX solution S^{ILP}.

2.3.4 Experimental Evaluation

The experimental evaluation in this section encompasses the following algorithms:

1. GREEDY: The heuristic obtained by the execution of the algorithm presented in Algorithm 2.2 on page 46 in a deterministic manner.
2. CPLEX: Application of CPLEX 22.1 to each considered problem instance, utilizing the default parameter values of CPLEX.
3. CMSA_GEN: The standard CMSA algorithm, making use of the generic approach to defining the set of solution components.
4. ADAPT_CMSA: The self-adaptive CMSA approach from this chapter.

Note that CPLEX 22.1 is used—both in standalone mode (CPLEX) and within the CMSA variants—in sequential mode. For conducting the experiments we used the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

2.3.4.1 MPIDS Benchmark Sets

As the MPIDS problem was defined in the context of applications in social networks, we used the `igraph` library[4] for generating input graphs with the Barabási-Albert model. This is because the Barabási-Albert model generates scale-free networks which are often used to simulate social networks. In particular, we generated 30 graphs for each combination of $|V| \in \{1000, 5000, 10{,}000, 50{,}000, 100{,}000\}$ and three different graph densities as controlled by a parameter m in the Barabási-Albert model. In total, this benchmark set consists of 450 graphs, ranging from rather small graphs with 1000 nodes to large-scale graphs of 100,000 vertices and up to 2,000,000 edges.

2.3.4.2 Parameter Tuning

As in the case of the MDS problem in Sect. 1.4.3.2, the `irace` tool was used for tuning the parameters of the considered CMSA variants. The list of parameters of the CMSA_GEN variant is the same as the one already outlined in Sect. 1.4.3.2. In contrast, ADAPT_CMSA does not make use of parameters n_a and $age_{\max}$ because n_a is handled in a self-adaptive way in ADAPT_CMSA, while $age_{\max}$ is fixed to one. Moreover, ADAPT_CMSA does not utilize parameters d_{rate} and l_{size}, because the way of making the solution construction probabilistic is different from the one in CMSA_GEN. Finally, ADAPT_CMSA does not use CPLEX-parameter $\text{cplex}_{\text{abort}}$, because the information about the CPLEX computation time is important for the self-adaptive mechanism.[5] Therefore, CPLEX is never aborted before either the sub-instance is solved to optimality, or the maximum computation time of t_{ILP} CPU seconds is reached. Instead, ADAPT_CMSA makes use of the following list of additional parameters:

- α^{LB}: The lower bound for α_{bsf}, which determines the bias of the best-so-far solution S^{bsf} on the solution construction process.
- α^{UB}: The upper bound for α_{bsf}.
- α_{red}: The amount by which the value of α_{bsf} is reduced if necessary (see line 12 of Algorithm 2.1 on page 42).
- t_{prop}: The parameter used for deciding if α_{bsf} is to be reduced at each iteration (again, see line 12 of Algorithm 2.1 on page 42).

In the first attempt, both CMSA variants were tuned exactly once for the entire benchmark set. As tuning instances, additional problem instances were generated. More specifically, for each combination of $|V|$ and graph density, exactly one tuning

[4] https://igraph.org/.

[5] Remember that a description of the CPLEX parameters possibly used within CMSA is provided on page 23.

Table 2.1 Parameters, domains, and tuning results for the MPIDS problem

Parameter	Domain	CMSA_GEN	ADAPT_CMSA
n_a	$\{1, \ldots, 50\}$	44	n.a.
$age_{\max}$	$\{1, \ldots, 10\}$	3	n.a.
d_{rate}	$[0.0, 0.99]$	0.99	n.a.
l_{size}	$\{3, \ldots, 50\}$	8	n.a.
t_{ILP}	$\{1, \ldots, 50\}$	41	13
$\text{cplex}_{\text{emphasis}}$	{`true`, `false`}	`true`	`true`
$\text{cplex}_{\text{warmstart}}$	{`true`, `false`}	`false`	`false`
$\text{cplex}_{\text{abort}}$	{`true`, `false`}	`true`	n.a.
α^{LB}	$[0.6, 0.99]$	n.a.	0.91
α^{UB}	$[0.6, 0.99]$	n.a.	0.97
α_{red}	$[0.01, 0.1]$	n.a.	0.02
t_{prop}	$[0.1, 0.8]$	n.a.	0.41

instance was generated, resulting in 15 tuning instances. As computation time limit we used 600 CPU seconds for all problem instances.

The tuning results for both CMSA_GEN and ADAPT_CMSA are shown in Table 2.1. They give already several indications for possible problems of CMSA_GEN. The determinism rate (d_{rate}), for example, is very high. Moreover, the CPU time limit for CPLEX (t_{ILP}) is rather high too. Both settings might indicate that CMSA_GEN is having problems solving sub-instances for larger problem instances. Regarding the parameter settings of ADAPT_CMSA, a rather high bias for constructing solutions in the vicinity of the best-so-far solution S^{bsf} can be observed.

2.3.4.3 First Results

All four algorithmic techniques (GREEDY, CPLEX, CMSA_GEN and ADAPT_CMSA) were applied exactly once to each of the problem instances from the benchmark set. The computation time limit for CPLEX, CMSA_GEN, and ADAPT_CMSA was the same as the one used for tuning (600 CPU seconds per run). The results are shown in the form of box plots in Figs. 2.2 (smaller instances with $|V| \in \{1000, 5000, 10{,}000\}$ and 2.3 (larger instances with $|V| \in \{50{,}000, 100{,}000\}$). Each of these graphics contains a grid of box plots. Rows in the grid present the results (from top to bottom) for graphs of increasing size, and grid columns (from left to right) present the results for graphs of increasing density.

To facilitate the analysis of the results along with assertions regarding statistical significance, we include CD plots presented in Fig. 2.4. For a comprehensive explanation of CD plots, please refer to Sect. 1.2.3 on page 16. Figure 2.4 comprises six CD plots. The first one (Fig. 2.4a) presents statistical details for the entire set of

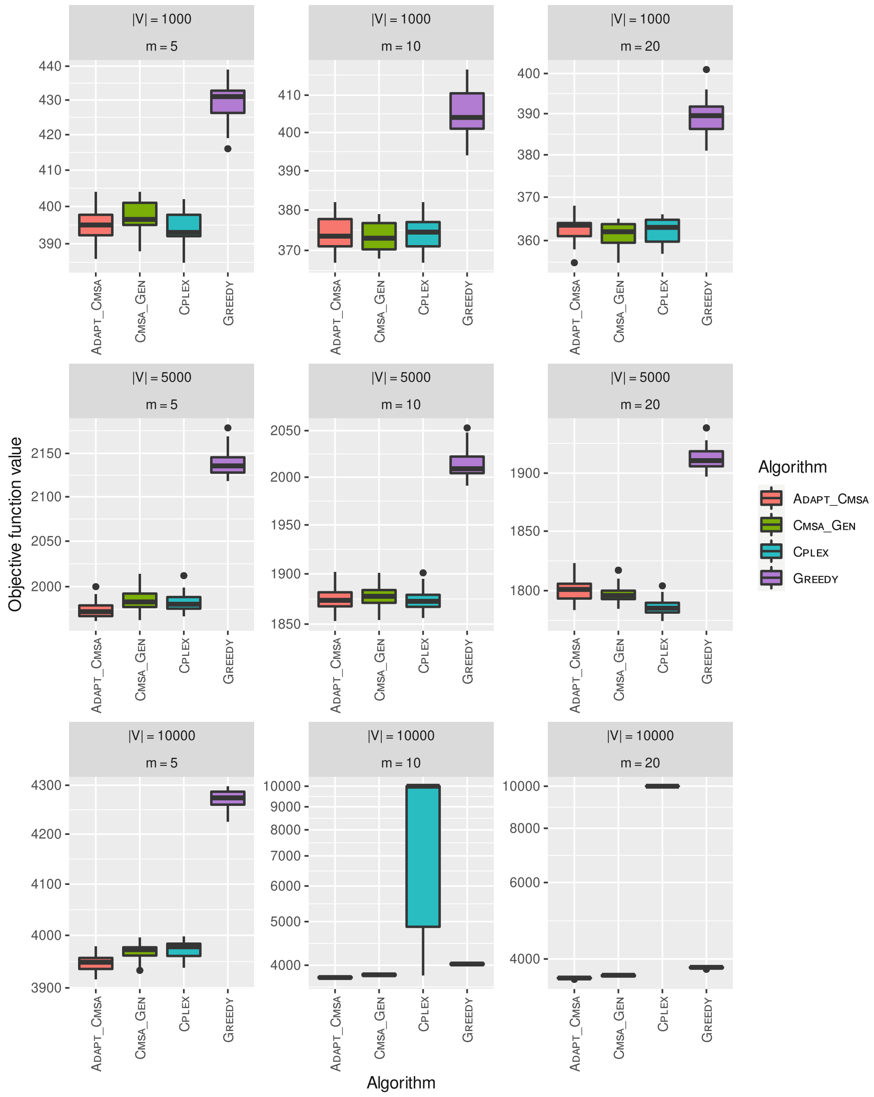


Fig. 2.2 MPIDS results for graphs with $|V| \in \{1000, 5000, 10{,}000\}$ based on a single parameter tuning application per algorithm

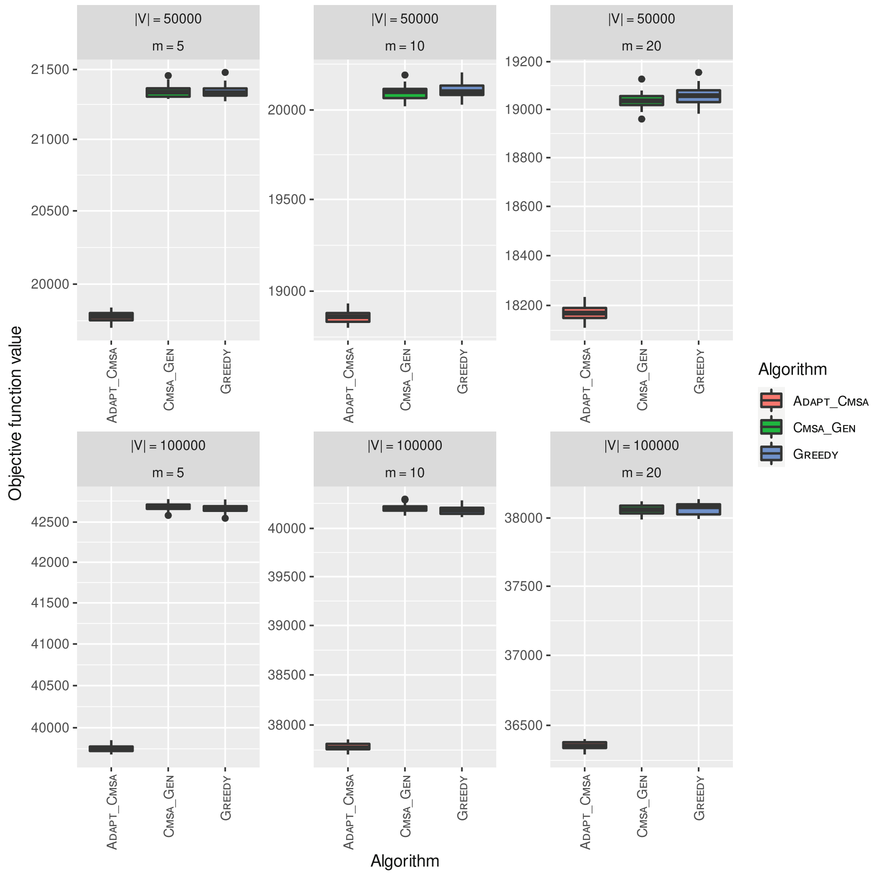


Fig. 2.3 MPIDS results for graphs with $|V| \in \{50{,}000, 100{,}000\}$ based on single parameter tuning application per algorithm

graphs, while the subsequent five CD plots provide statistical insights specific to graphs of particular sizes.

Main Observations Concerning the MPIDS Results

1. CPLEX is only able to compete with the CMSA variants in the context of the denser ones of the small problem instances ($|V| \in \{1000, 5000\}$). In fact, for problem instances with 10,000 vertices of medium and high density, CPLEX starts to fail. For the large problem instances, CPLEX is only able to generate the trivial solutions (containing all vertices of a graph) within the allotted computation time.

2. For graphs with $|V| \in \{1000, 5000, 10{,}000\}$, Cmsa_Gen works well, but—apart from the denser graphs with $|V| = 1000$—Cmsa_Gen is already slightly inferior to Adapt_Cmsa. However, for graphs with $|V| \in \{50{,}000, 100{,}000\}$ Cmsa_Gen fails. This can be seen by the fact that it is only slightly better than Greedy. This indicates that the sub-instances generated within Cmsa_Gen cannot be solved anymore by CPLEX for graphs of that size.
3. In contrast to Cmsa_Gen, Adapt_Cmsa works very well for problem instances over the whole benchmark set.
4. The CD plots from Fig. 2.4 confirm that Adapt_Cmsa outperforms all other approaches with statistical significance.

Additionally, we plotted STNWeb graphics of the obtained results; see Sect. 1.2.2 on page 13 for a description of the STNWeb tool and the type of graphics that are produced. Figure 2.5 shows the STN (complete vs. partitioned) for the first problem instance with 10,000 vertices and $m = 5$ (sparsest graphs). The complete STN (Fig. 2.5a) shows that the Adapt_Cmsa search trajectories are much longer. This is for two reasons. First, the algorithm does smaller steps in the search space (due to a lower CPU time limit for CPLEX in comparison to Cmsa_Gen). Second, starting from solutions of a similar quality as Cmsa_Gen, the algorithm produces clearly better final results. The partitioned STN (Fig. 2.5b) indicates a specific property of

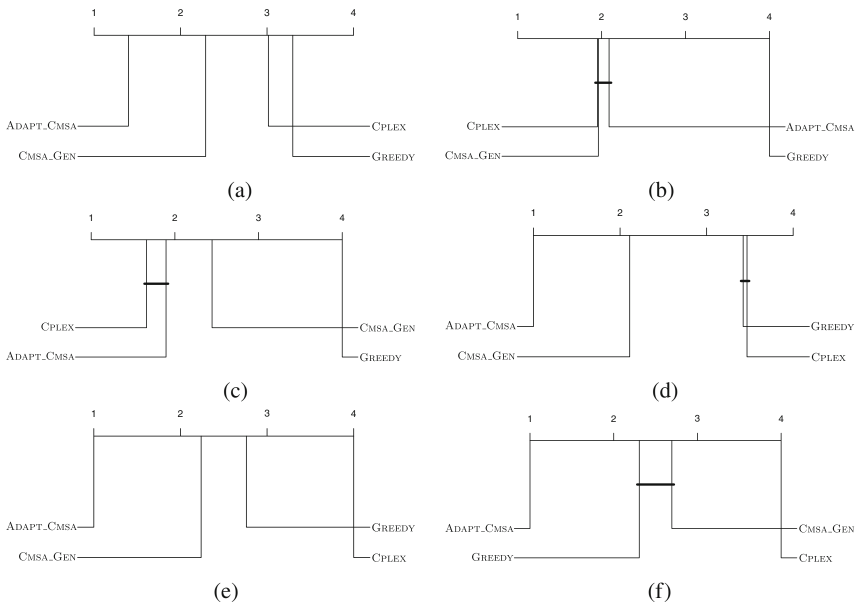


Fig. 2.4 Critical Difference (CD) plots concerning the MPIDS results. (**a**) All graphs. (**b**) $|V| = 1000$. (**c**) $|V| = 5000$. (**d**) $|V| = 10{,}000$. (**e**) $|V| = 50{,}000$. (**f**) $|V| = 100{,}000$

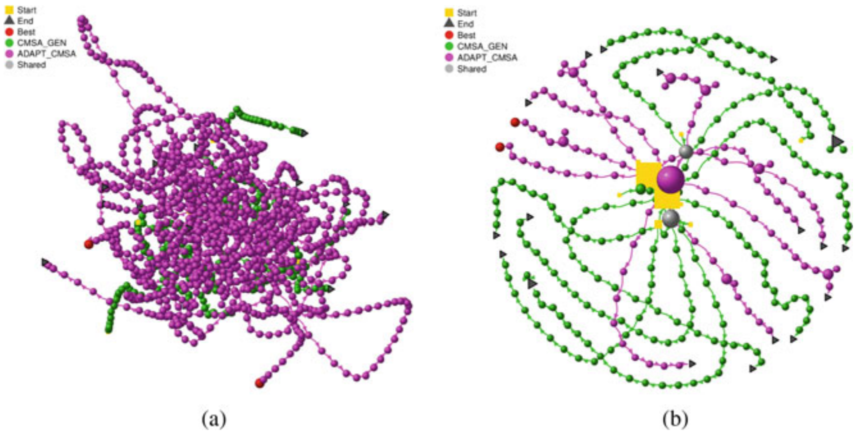


Fig. 2.5 STNWeb graphics. (**a**) and (**b**) show 10 runs of CMSA_GEN and ADAPT_CMSA for the first problem instance with 10,000 vertices and $m = 5$ (sparse). While (**a**) shows the complete STN, (**b**) shows the same STN after partitioning

the MPIDS problem. Observe that trajectory overlaps are only found at the start of algorithm trajectories, both concerning trajectories of different algorithms (see the two larger grey dots) and between trajectories of the same algorithm (see the large pink dot). This indicates that different good solutions to an MPIDS instance might have quite different structures. In order to confirm this, the following experiment was made. The scatter plots in Fig. 2.6 show for each pair of same-quality solutions from the search trajectories of CMSA_GEN and ADAPT_CMSA their difference (in terms of the number of vertices that are different in both solutions). Especially with growing instance size (starting from the second row of scatter plots) these graphics nicely confirm what was indicated already by the STNWeb graphics from Fig. 2.5. The better a pair of same-quality solutions is, the larger their difference.

2.3.4.4 Results with a Specialized Parameter Tuning

In a second experiment, the aim is to study the change of performance both of CMSA_GEN and ADAPT_CMSA (if any) when specifically tuned for each of the five considered graph sizes. The parameter values obtained with `irace` from these specialized tuning runs are provided in Table 2.2.

Again, the results are provided in terms of box plots (see Figs. 2.7 and 2.8) and the corresponding CD plots (see Fig. 2.9). Note that in these graphics, the results of both CMSA_GEN and ADAPT_CMSA obtained with the two available parameter settings (single tuning vs. specialized tuning) are compared. For this purpose the algorithms, when using the specialized parameter setting, are called CMSA_GEN_T and ADAPT_CMSA_T.

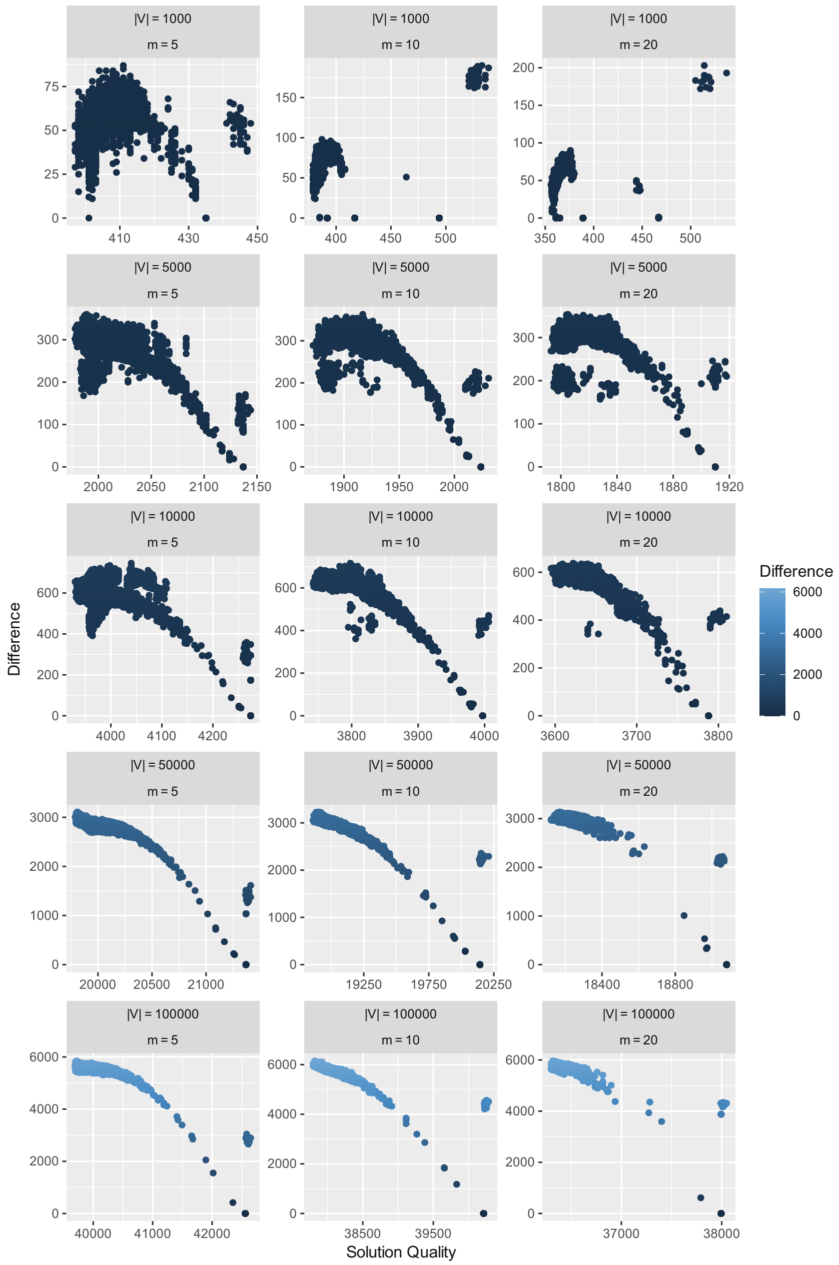


Fig. 2.6 Differences between MPIDS solutions of the same quality. The x-axes of all plots indicate the solution quality (that is, the objective function values), while the y-axes show the differences between solutions of the same quality from the considered search trajectories

Table 2.2 Specialized tuning results for the MPIDS problem

	CMSA_GEN					ADAPT_CMSA				
Parameter	$\lvert V\rvert = 1000$	$\lvert V\rvert = 5000$	$\lvert V\rvert = 10{,}000$	$\lvert V\rvert = 50{,}000$	$\lvert V\rvert = 100{,}000$	$\lvert V\rvert = 1000$	$\lvert V\rvert = 5000$	$\lvert V\rvert = 10{,}000$	$\lvert V\rvert = 50{,}000$	$\lvert V\rvert = 100{,}000$
n_a	9	37	2	2	18	n.a.	n.a.	n.a.	n.a.	n.a.
$age_{\max}$	3	1	7	3	8	n.a.	n.a.	n.a.	n.a.	n.a.
d_{rate}	0.42	0.8	0.91	0.98	0.99	n.a.	n.a.	n.a.	n.a.	n.a.
l_{size}	32	23	20	7	40	n.a.	n.a.	n.a.	n.a.	n.a.
t_{ILP}	32	47	49	10	2	9	21	12	30	4
cplex$_{\text{emphasis}}$	`true`	`true`	`true`	`true`	`false`	`false`	`true`	`true`	`false`	`true`
cplex$_{\text{warmstart}}$	`false`	`true`	`false`	`false`	`false`	`true`	`true`	`true`	`true`	`false`
cplex$_{\text{abort}}$	`true`	`false`	`true`	`true`	`true`	n.a.	n.a.	n.a.	n.a.	n.a.
α^{LB}	n.a.	n.a.	n.a.	n.a.	n.a.	0.62	0.69	0.93	0.87	0.8
α^{UB}	n.a.	n.a.	n.a.	n.a.	n.a.	0.81	0.86	0.99	0.94	0.92
α_{red}	n.a.	n.a.	n.a.	n.a.	n.a.	0.07	0.02	0.03	0.09	0.03
t_{prop}	n.a.	n.a.	n.a.	n.a.	n.a.	0.55	0.27	0.32	0.42	0.4

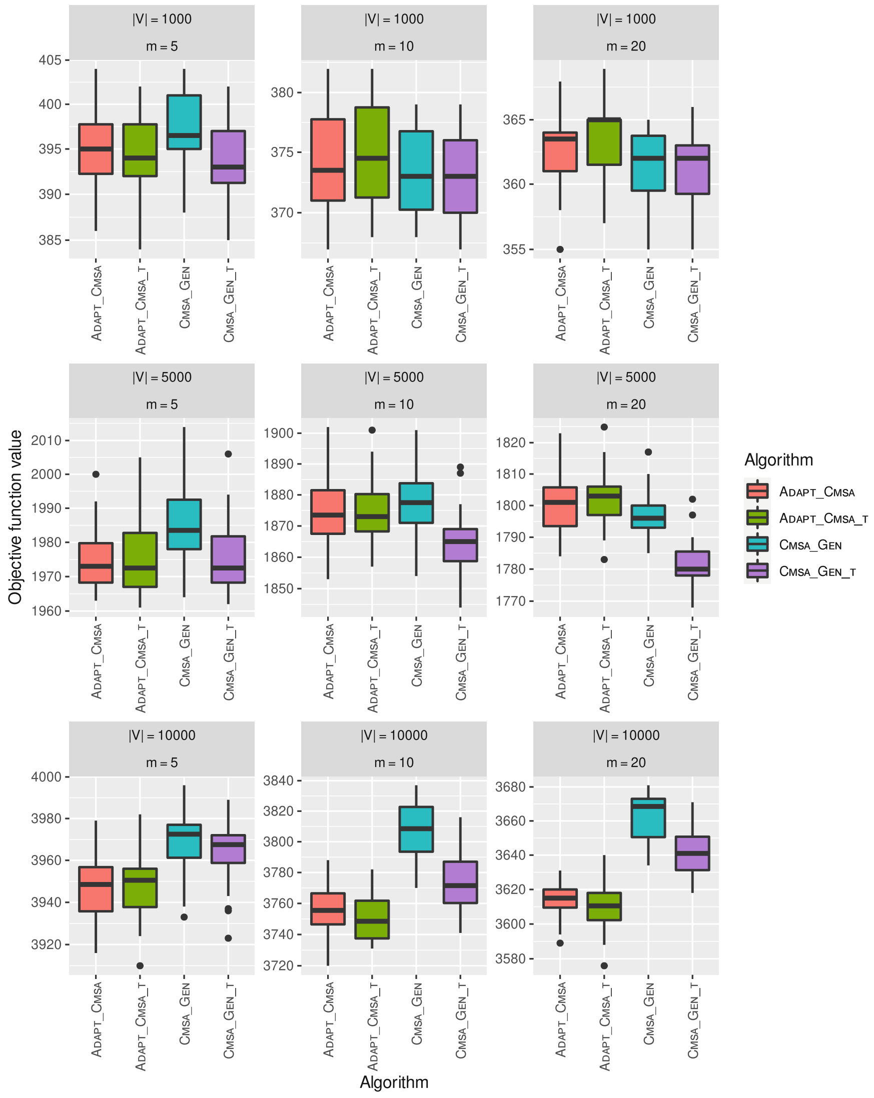


Fig. 2.7 MPIDS results for graphs with $|V| \in \{1000, 5000, 10{,}000\}$ based on instance-subset-specific parameter tuning

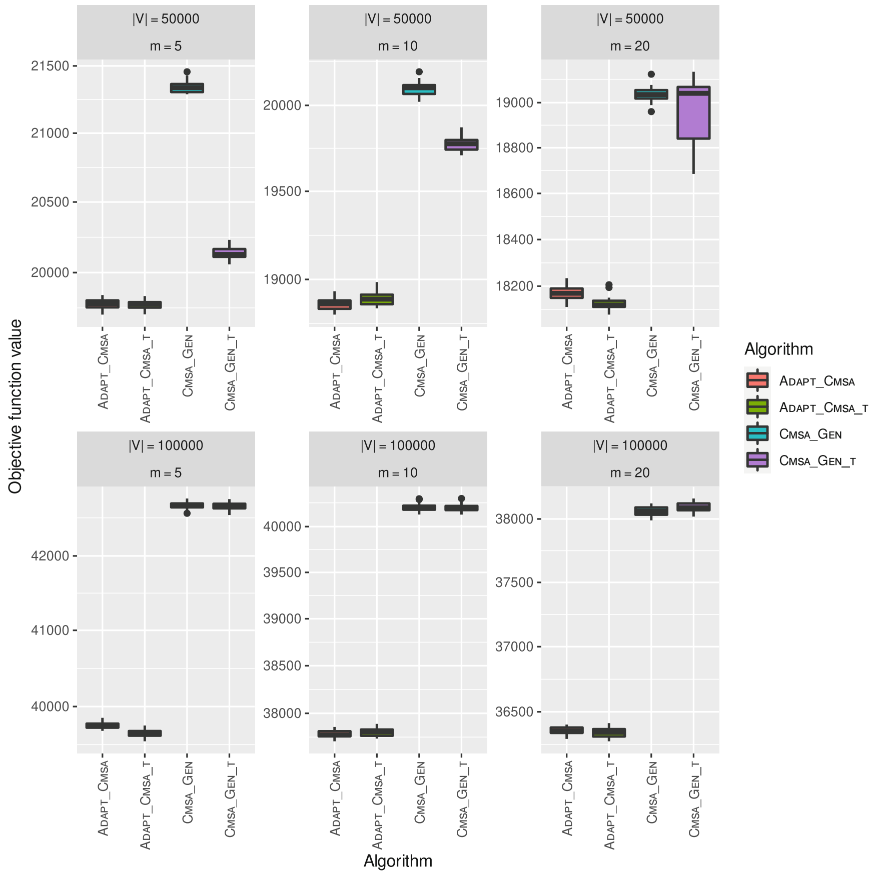


Fig. 2.8 MPIDS results for graphs with $|V| \in \{50{,}000, 100{,}000\}$ based on instance-subset-specific parameter tuning

Main Observations Concerning Single Tuning vs. Specialized Tuning

1. ADAPT_CMSA_T still outperforms CMSA_GEN_T with statistical significance.
2. The most important observation is that the results of CMSA_GEN can be significantly improved by specialized tuning up to a graph size of 50,000 vertices. In contrast, the results of ADAPT_CMSA are not significantly improved by specialized tuning. This indicates that the self-adaptive mechanism of ADAPT_CMSA works in an excellent way. This results not only in an excellent performance in the context of large-scale problem instances of problems such as the MPIDS, but it also results in potentially much less parameter tuning efforts in comparison to CMSA_GEN.
3. The results of CMSA_GEN_T for graphs of size $|V| = 100{,}000$ indicate the continuing disability of the algorithm to produce sub-instances of a size and nature that can be solved by CPLEX.

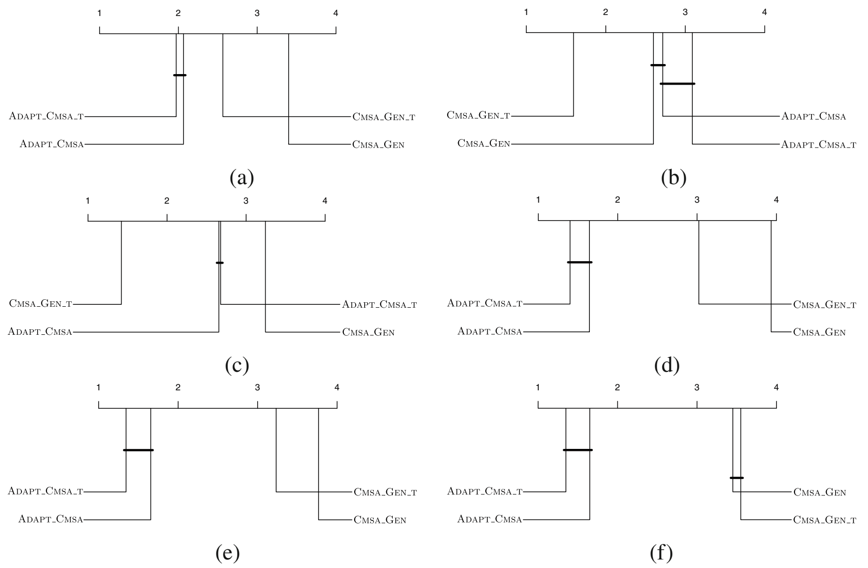


Fig. 2.9 Critical Difference (CD) plots concerning the MPIDS results after specialized parameter tuning. (**a**) All graphs. (**b**) $|V| = 1000$. (**c**) $|V| = 5000$. (**d**) $|V| = 10{,}000$. (**e**) $|V| = 50{,}000$. (**f**) $|V| = 100{,}000$

2.4 Application to the FFMS Problem

As a second application of Adapt_Cmsa, we introduce its use in addressing the *far from most string* (FFMS) problem [13], a combinatorial optimization problem that falls under the category of NP-hard problems. This problem is part of the sequence consensus problems family, where the goal is to find a consensus sequence for a finite set of sequences, representing them in the best way possible. Sequence consensus problems often involve various and sometimes conflicting objectives. In the context of the FFMS problem, the objective is to identify a new sequence that significantly differs from the majority of the given input sequences (refer to the technical description below). The FFMS problem finds applications in diverse fields, such as molecular biology, where its use extends to creating diagnostic probes for bacterial infections or identifying potential drug targets.

A specific instance of the FFMS problem is represented as (Ω, t), where $\Omega = \{s_1, \ldots, s_n\}$ forms a set of n input strings over a finite alphabet Σ. Each input string s_i in Ω has a length of m, denoted as $|s_i| = m$ for all $s_i \in \Omega$. Additionally, a fixed threshold value is provided, with $0 < t < m$. In the subsequent discussion, the j-th character of a string s_i is referred to as $s_i[j]$. The *Hamming distance* between two equal-length strings $s_i \neq s_j \in \Omega$, denoted as $d_H(s_i, s_j)$, represents the count of positions where corresponding characters in the two strings differ. In other words:

$$d_H(s_i, s_j) = \left|\left\{k \in \{1, \ldots, m\} \mid s_i[k] \neq s_j[k]\right\}\right| \tag{2.8}$$

Any string s with a length of m over alphabet Σ is considered a feasible solution to the FFMS problem. The objective function value $f_{\text{orig}}(s)$ for any such string s is defined as follows:

$$f_{\text{orig}}(s) := |\{s_i \in \Omega \mid d_H(s, s_i) \geq t\}| \tag{2.9}$$

This implies that the objective function value for a solution or string s is determined by the count of input strings whose Hamming distance with s is greater than or equal to the specified threshold value t. Over the past two decades, a range of different algorithmic approaches have been presented in the related literature for addressing the FFMS problem. Presently, the negative learning ACO approach proposed in [6] and the memetic algorithm introduced in [10] are considered state-of-the-art methods for tackling the FFMS problem.

Example of the FFMS Problem

Given is the following problem instance:

- *Number of input strings:* $n = 3$
- *Length of the input strings:* $m = 4$
- *Alphabet:* $\Sigma = \{0, 1\}$ (that is, the alphabet size is 2)
- *Threshold value:* $t = 3$
- *Instance data (input strings):* $s_1 = 0101$, $s_2 = 0111$, $s_3 = 0011$

In the following we will analyze two possible solutions:

1. **A first valid solution:** $s = 1100$
   - Hamming distances:
     - $\mathbf{d_H(s = 1100, s_1 = 0101) = 2}$
     - $\mathbf{d_H(s = 1100, s_2 = 0111) = 3}$
     - $\mathbf{d_H(s = 1100, s_3 = 0011) = 4}$
   - Objective function value: $f_{\text{orig}}(s) = 2$
2. **A second valid solution:** $s^* = 1000$ (optimal solution)
   - Hamming distances:
     - $\mathbf{d_H(s^* = 1000, s_1 = 0101) = 3}$
     - $\mathbf{d_H(s^* = 1000, s_2 = 0111) = 4}$
     - $\mathbf{d_H(s^* = 1000, s_3 = 0011) = 3}$
   - Objective function value: $f_{\text{orig}}(s) = 3$

The FFMS problem can be modeled in terms of an ILP model, as originally described and introduced in [5]. This model makes use of two sets of binary variables. The first set contains a variable $x_{j,a}$ for each combination of a position $j = 1, \ldots, m$ of a possible solution and a character $a \in \Sigma$. The second one contains a binary variable y_i for each input string $s_i \in \Omega$ ($i = 1, \ldots, n$). The ILP model can then be stated as follows.

$$\max \quad \sum_{i=1}^{n} y_i \tag{2.10}$$

$$\text{subject to} \quad \sum_{a \in \Sigma} x_{j,a} = 1 \quad j = 1, \ldots, m \tag{2.11}$$

$$\sum_{j=1}^{m} x_{j,s_i[j]} \leq m - t \cdot y_i \quad i = 1, \ldots, n \tag{2.12}$$

$$x_{j,a}, y_i \in \{0, 1\}$$

Note that constraints (2.11) ensure that exactly one character from Σ is chosen for each position j of a solution string. Moreover, constraints (2.12) ensure that a variable y_i can only have value one if and only if the Hamming distance between input string $s_i \in \Omega$ and a solution string (as defined by the setting of the variables $x_{j,a}$) is greater or equal than the threshold value t.

2.4.1 Augmented Objective Function

The FFMS problem poses a significant challenge not only for exact techniques but also for metaheuristics. One of the main difficulties arises from the limited range of distinct objective function values. Specifically, for an instance with n input strings, the set of possible objective function values is constrained to $\{0, \ldots, n\}$. This characteristic leads to wide plateaus in the search space of an FFMS problem instance. Consequently, similar solutions often share identical objective function values. For a metaheuristic, this implies that the search space provides minimal (or no) guidance on how to progress and explore during the search process. Consequently, metaheuristics frequently encounter difficulties navigating these plateaus. In consideration of these challenges, [6] conducted experiments incorporating four augmented objective functions in addition to the original objective function. In this chapter, we will adopt the augmented objective function from [5], which we will refer to as $f_{\text{aug}}()$. This function is a lexicographic objective function, with the primary criterion being the original objective function. The second criterion makes use of the following function:

$$h(s) := \sum_{\{s_i \in \Omega \mid d_H(s,s_i) \geq t\}} d_H(s, s_i) + \max_{\{s_i \in \Omega \mid d_H(s,s_i) < t\}} \{d_H(s, s_i)\} \tag{2.13}$$

In simpler terms, $h(s)$ computes the sum of Hamming distances between s and the input strings $s_i \in \Omega$ with a Hamming distance of at least t. It also takes into account the maximum Hamming distance between s and the input strings $s_i \in \Omega$ with a Hamming distance less than t. The original objective function and $h()$ are then combined using a lexicographic approach.

$$\begin{aligned} f_{\text{aug}}(s) > f_{\text{aug}}(s') \textbf{ iff } f_{\text{orig}}(s) > f_{\text{orig}}(s') \textbf{ or} \\ (f_{\text{orig}}(s) = f_{\text{orig}}(s') \textbf{ and } h(s) > h(s')) \end{aligned} \tag{2.14}$$

The rationale behind $h()$ can be explained as follows: a higher value of $h(s)$ indicates a lower probability that minor alterations to s will lead to a decrease in the original objective function.

2.4.2 Intuitive Definition of the Solution Components

In the case of the application of different CMSA variants to the FFMS problem, we decided for the following intuitive definition of the set C of solution components: each combination of a position j in the solution string (where $j = 1, \ldots, m$) and a letter $a \in \Sigma$ is a solution component $c_{j,a}$. That is, $C := \{c_{j,a} \mid j = 1, \ldots, m \text{ and } a \in \Sigma\}$. Any feasible solution S is a subset of C such that for each position $j = 1, \ldots, m$, S contains exactly one of the solution components from $C_j := \{c_{j,a} \mid a \in \Sigma\}$. Similarly, the sub-instance C' is always a subset of C. Note that, in the following, $f_{\text{aug}}(S) := f_{\text{aug}}(s)$, where s is the solution string which is derived in a well-defined way from the solution components contained in S. Moreover, let $f_{\text{aug}}(\emptyset) := 0$.

2.4.3 Constructing Solutions to the FFMS Problem

A solution s with a length of m (represented as a string) is generated by selecting one character from the alphabet Σ for each position $j = 1, \ldots, m$. This selection is based on greedy information, for which we employ the frequency values of the letters at each position of the input strings. Specifically, the frequency value $f_{j,a}$ for a letter $a \in \Sigma$ at a position $1 \leq j \leq m$ is computed as follows:

$$f_{j,a} := \frac{|\{s_i \in \Omega \mid s_i[j] = a\}|}{n} \tag{2.15}$$

A letter is then chosen for each $j = 1, \ldots, m$ based on these frequency values. How this is done exactly, depends on the specific solution construction function and will be explained in the following.

In function GenerateGreedySolution(C) of ADAPT_CMSA (see line 4 of Algorithm 2.1 on page 42) for each position j a deterministic choice of the letter with the lowest frequency value is made. Ties are broken randomly.

In contrast, in function ProbabilisticSolutionConstruction(C) of our standard CMSA_INT (see line 8 of Algorithm 1.1 on page 19), first, a value $r \in [0, 1]$ is drawn uniformly at random for each position $j = 1, \ldots, m$. If $r \leq d_{\text{rate}}$—where $0 \leq d_{\text{rate}} < 1$ is the determinism rate already known from other applications in this book—the letter with the lowest frequency value is chosen in a deterministic way. Otherwise, a letter is chosen randomly (roulette wheel selection) based on probabilities that are proportional to the inverse of the letter frequencies.

Finally, function ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}) of ADAPT_CMSA (see line 8 of Algorithm 2.1 on page 42) chooses for each position $j = 1, \ldots, m$ a letter as follows. First, the following values are defined:

$$\mathbf{q}_{j,a} \begin{cases} \frac{\alpha_{\text{bsf}}}{f_{j,a}} & \text{if } c_{j,a} \in S^{\text{bsf}} \\ \frac{(1-\alpha_{\text{bsf}})}{f_{j,a}} & \text{otherwise} \end{cases} \tag{2.16}$$

Subsequently, a letter is chosen for position j by roulette wheel selection based on the following probabilities:

$$\mathbf{p}(a) := \frac{q_{j,a}}{\sum_{b \in \Sigma} q_{j,b}} \quad \forall\, a \in \Sigma \tag{2.17}$$

All three functions, after finishing the construction of a solution in terms of a string (s), transform this solution into a solution S containing for each position j the corresponding solution component. This solution S is then returned to the respective CMSA algorithm.

2.4.4 Sub-instance Solving

The way of solving sub-instances C' in the case of the FFMS problem is exactly the same as the one described in Sect. 1.4.2.2 on page 25 in the context of the MDS problem. Therefore, for solving a sub-instance C' by means of the application of CPLEX in function SolveSubinstance(C', t_{ILP}) of Algorithm 1.1 (page 19) and of Algorithm 2.1 (page 42), the FFMS ILP model is extended by adding the following set of constraints:

$$x_{j,a} = 0 \quad \text{for all } c_{j,a} \in C \setminus C' \tag{2.18}$$

In this way, only solution components (resp. letter-position assignments) that form part of the sub-instance C' may be selected in order to appear in solutions. Finally, remember that function SolveSubinstance(C', t_{ILP}) returns the CPLEX solution in terms of a set S^{ILP} of solution components. Moreover, the application of CPLEX is subject to a time limit of t_{ILP} CPU seconds, which means that the solution S^{ILP} returned by the function SolveSubinstance(C', t_{ILP}) is not necessarily an optimal solution to the sub-instance C'.

2.4.5 Experimental Evaluation

The experimental evaluation for the FFMS problem encompasses the following algorithms:

1. Greedy: The heuristic obtained by the execution of the algorithm presented in Sect. 2.4.3 (previous section) in a deterministic way.
2. Cplex: Application of CPLEX 22.1 to each considered problem instance, utilizing the default parameter values of CPLEX.
3. Cmsa_Int: The standard CMSA algorithm, making use of the intuitive approach to defining the set of solution components.
4. Adapt_Cmsa: The self-adaptive CMSA approach from this chapter.

As before, CPLEX 22.1 is used—both in standalone mode (Cplex) and within the CMSA variants—in one-threaded mode. For conducting the experiments we used the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

2.4.5.1 FFMS Benchmark Set

For each combination of $n \in \{100, 200, 300, 400\}$ (number of input strings) and $m \in \{100, 500, 1000\}$ (length of the input strings) exactly 30 problem instances were generated uniformly at random over $\Sigma = \{A, C, T, G\}$. This alphabet of size four was chosen due to the applications of the FFMS in bio-informatics. Moreover, thresholds $t = 0.8m$ and $t = 0.85m$ will be considered for solving all these instances. However, as threshold $t = 0.8m$ proved to be more difficult for our algorithms, the result section below will only show the results obtained for this threshold. In total, the generated benchmark set contains 360 problem instances.

2.4.5.2 Parameter Tuning

Both Cmsa_Int and Adapt_Cmsa were tuned with the parameter tuning tool `irace`; see Sect. 1.2.1 on page 12 for a description of `irace`. For this purpose, an additional tuning instance was generated for each combination of n and m, which makes a total of 12 tuning instances, considered both for thresholds $t = 0.8m$ and $t = 0.85m$. The budget of `irace`—that is, the number of algorithm runs allowed

Table 2.3 Parameters, domains, and tuning results for the FFMS problem

Parameter	Domain	CMSA_INT	ADAPT_CMSA
n_a	$\{1, \ldots, 50\}$	27	n.a.
$age_{\max}$	$\{1, \ldots, 10\}$	9	n.a.
d_{rate}	$[0.0, 0.99]$	0.81	n.a.
t_{ILP}	$\{1, \ldots, 50\}$	30	1
$\text{cplex}_{\text{emphasis}}$	{`true`, `false`}	`true`	`true`
$\text{cplex}_{\text{warmstart}}$	{`true`, `false`}	`true`	`false`
$\text{cplex}_{\text{abort}}$	{`true`, `false`}	`false`	n.a.
α^{LB}	$[0.6, 0.99]$	n.a.	0.64
α^{UB}	$[0.6, 0.99]$	n.a.	0.99
α_{red}	$[0.01, 0.1]$	n.a.	0.1
t_{prop}	$[0.1, 0.8]$	n.a.	0.42

for tuning—was set to 3000, and the computation time limit for the algorithms was set to 600 CPU seconds per problem instance. The tuning results are presented in Table 2.3. The most striking difference between CMSA_INT and ADAPT_CMSA is the computation time limit for CPLEX. While CMSA_INT uses a limit of 30 seconds (without enabling the abort feature), ADAPT_CMSA uses a limit of 1 second.

2.4.5.3 Results

The results are shown by means of box plots in Fig. 2.10. They show that ADAPT_CMSA outperforms CMSA_INT especially in the context of short input strings, that is, for instances with $m = 100$ and also (to a lesser extent) with $m = 500$. On the contrary, CMSA_INT seems to have an advantage over ADAPT_CMSA in the context of longer input strings ($m = 1000$). These findings are also confirmed by the CD plots provided in Fig. 2.11. In fact, they show that overall ADAPT_CMSA outperforms CMSA_GEN with statistical significance on this benchmark set. However, Fig. 2.11d shows that CMSA_GEN outperforms ADAPT_CMSA with statistical significance on the subset of benchmark instances with $m = 1000$.

As in the case of the MPIDS problem (see Sect. 2.3.4.3), we plotted STNWeb graphics of the obtained results; see Sect. 1.2.2 on page 13 for a description of the STNWeb tool and the type of graphics that are produced. Figure 2.12 shows the STN (complete vs. partitioned) for the first problem instance with $n = 100$ input strings and an input string length of $m = 1000$ (longest). The complete STN (Fig. 2.12a) shows that the ADAPT_CMSA search trajectories are much longer. This is mainly because ADAPT_CMSA does smaller steps than CMSA_INT in the search space (due to a lower CPU time limit for CPLEX in comparison to CMSA_INT). The partitioned STN (Fig. 2.12b) indicates a property of the FFMS problem that we already observed in the case of the MPIDS problem: Trajectory overlaps are only found at the start of algorithm trajectories, both concerning trajectories of

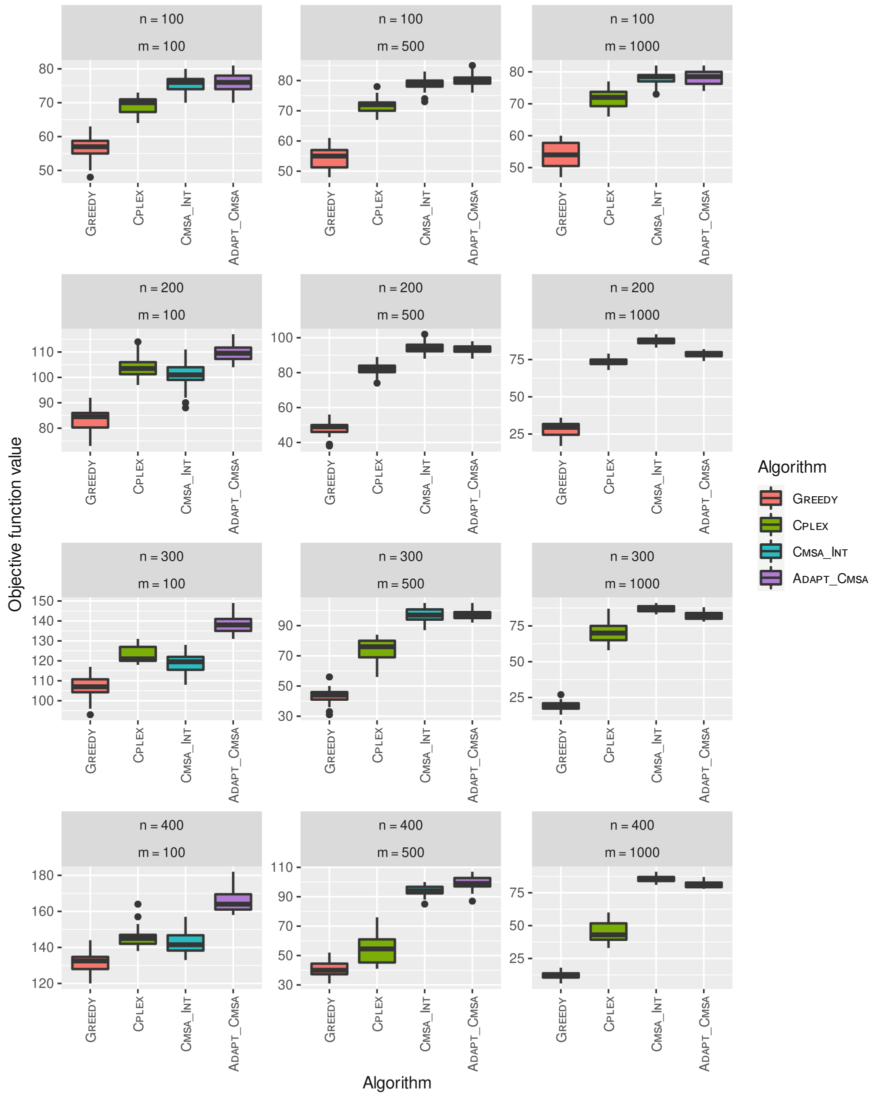


Fig. 2.10 Results for the FFMS problem (threshold $t = 0.8m$)

different algorithms and between trajectories of the same algorithm. This indicates that different good solutions to an FFMS instance might have quite different structures. This is indeed confirmed by the scatter plots in Fig. 2.13. They show for each pair of same-quality solutions from the search trajectories of CMSA_INT and ADAPT_CMSA their difference (in terms of the number of solution string positions that are different in both solutions). More specifically, it can be observed that better same-quality solution pairs are generally characterized by a larger difference.

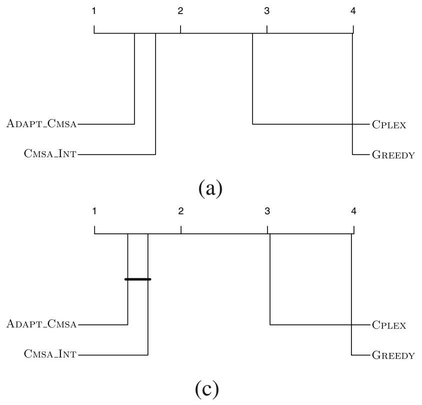


Fig. 2.11 Critical Difference (CD) plots concerning the FFMS results. (**a**) All problem instances. (**b**) Instances with $m = 100$. (**c**) Instances with $m = 500$. (**d**) Instances with $m = 1000$

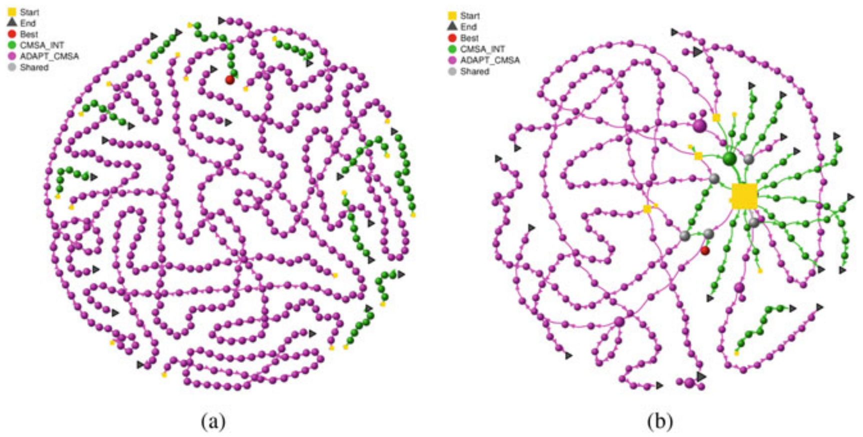


Fig. 2.12 STNWeb graphics. (**a**) and (**b**) show 10 runs of Cmsa_Int and Adapt_Cmsa for the first problem instance with $n = 100$ and $m = 1000$ (few but long input strings). While (**a**) shows the complete STN, (**b**) shows the same STN after partitioning

2.5 Conclusions

In this chapter, it was shown that a self-adaptive variant of CMSA called Adapt_Cmsa can be very useful for solving certain combinatorial optimization problems. This holds especially for large-scale problem instances where standard versions of CMSA might have problems adjusting the parameters such that the resulting sub-instances can still be solved by CPLEX within reasonable computation times. The increased adaptability of Adapt_Cmsa is achieved by a self-adaptive way of changing two parameters during the execution of the algorithm: (1) the number of solutions constructed per iteration, and (2) the value of a parameter that

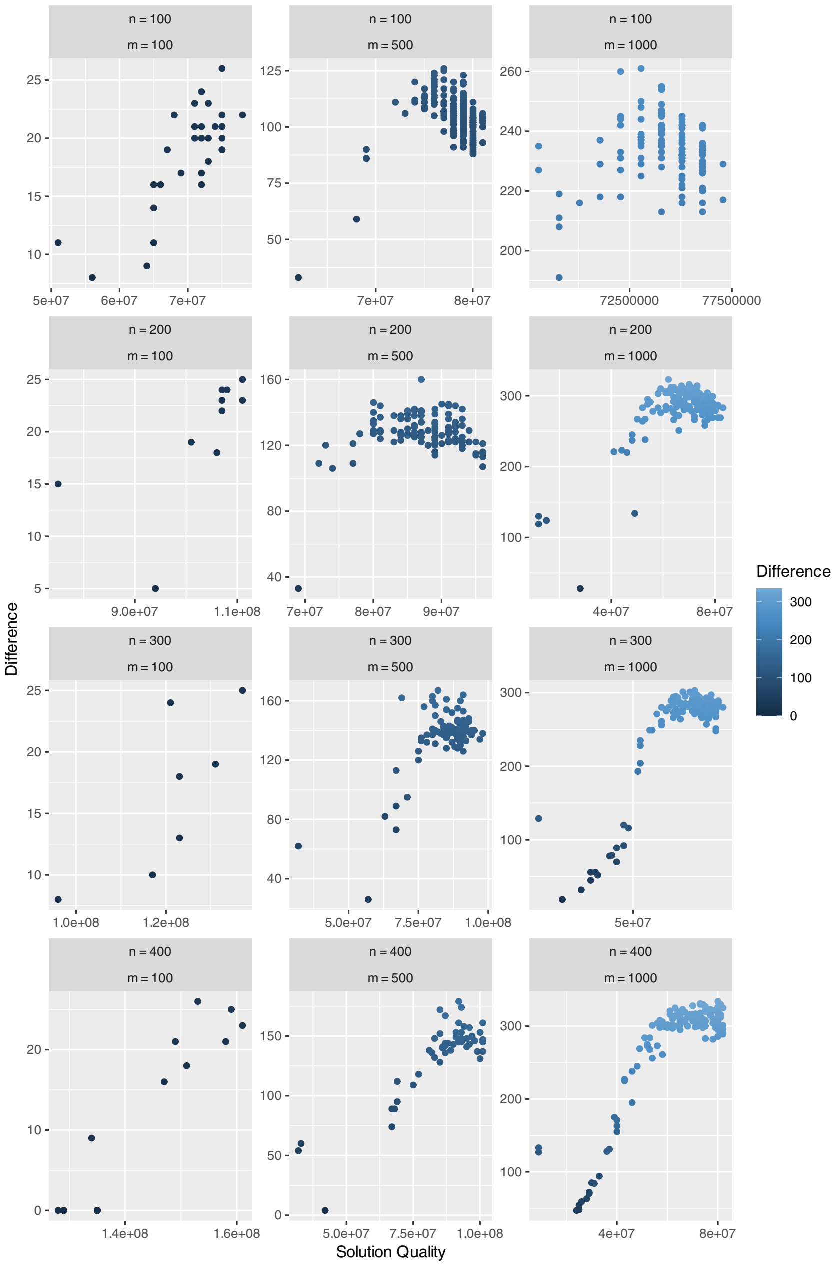


Fig. 2.13 Differences between FFMS solutions of the same quality. The x-axes of all plots indicate the solution quality (that is, the objective function values), while the y-axes show the differences between solutions of the same quality from the considered search trajectories

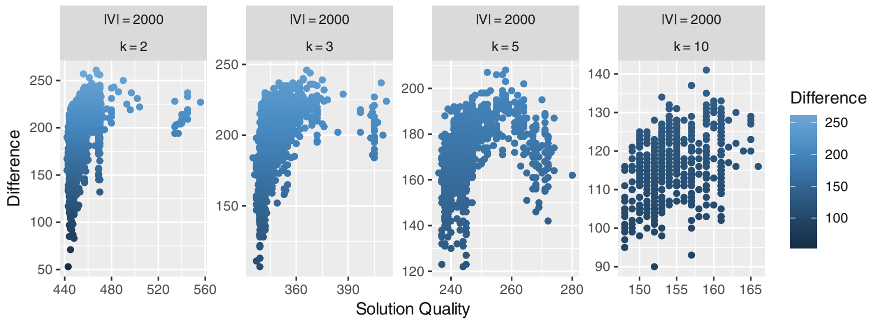


Fig. 2.14 Differences between MDS solutions of the same quality

biases the construction of new solutions towards the best-so-far solution. However, note that biasing solution constructions towards the best-so-far solution may also sometimes result in a disadvantage of ADAPT_CMSA in comparison to standard CMSA variants. This is the case for problems in which occasionally larger jumps in the search space are necessary in order to escape from basins of attraction of sub-optimal solutions. In fact, we also tried to apply ADAPT_CMSA to the MDS problem, which was considered in Chap. 1. However, in the case of the MDS problem, ADAPT_CMSA resulted consistently worse than both CMSA_GEN and CMSA_INT. Remember that in the case of the MDS problem, STNWeb graphics showed overlap at the end of algorithm trajectories (see, for example, Fig. 1.14b on page 34). This is very much in contrast to what happens in the case of the MPIDS and FFMS problems. The difference between the STNWeb graphics also results in very different scatter plots regarding the difference between pairs of same-quality solutions; see Fig. 2.14. Instead of a growing difference between pairs of same-quality solutions when solutions become better, the opposite happens in the case of the MDS problem. The difference between same-quality solutions decreases. In summary, these studies show that no algorithm variant is better for all possible problems, which is very much in line with the so-called *no free lunch theorems* [19].

References

1. Akbay, M.A., Blum, C.: Application of CMSA to the minimum positive influence dominating set problem. In: Artificial Intelligence Research and Development, pp. 17–26. IOS Press, Amsterdam, Netherlands (2021)
2. Akbay, M.A., Kalayci, C.B., Blum, C.: Application of CMSA to the electric vehicle routing problem with time windows, simultaneous pickup and deliveries, and partial vehicle charging. In: Metaheuristics International Conference, pp. 1–16. Springer (2022)
3. Akbay, M.A., Kalayci, C.B., Blum, C.: Application of Adapt-CMSA to the two-echelon electric vehicle routing problem with simultaneous pickup and deliveries. In: European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar), pp. 16–33. Springer (2023)

4. Akbay, M.A., López Serrano, A., Blum, C.: A self-adaptive variant of CMSA: application to the minimum positive influence dominating set problem. International Journal of Computational Intelligence Systems **15**(1), 44 (2022)
5. Blum, C., Festa, P.: A hybrid ant colony optimization algorithm for the far from most string problem. In: Proceedings of EvoCOP 2014 – European Conference on Evolutionary Computation in Combinatorial Optimization, pp. 1–12. Springer (2014)
6. Blum, C., Pinacho-Davidson, P.: Application of negative learning ant colony optimization to the far from most string problem. In: Proceedings of EvoCOP – European Conference on Evolutionary Computation in Combinatorial Optimization, no. 13987 in Lecture Notes in Computer Science, pp. 82–97. Springer (2023)
7. Bouamama, S., Blum, C.: An improved greedy heuristic for the minimum positive influence dominating set problem in social networks. Algorithms **14**(3), 79 (2021)
8. Djukanović, M., Kartelj, A., Blum, C.: Self-adaptive CMSA for solving the multidimensional multi-way number partitioning problem. Expert Systems with Applications p. 120762 (2023)
9. Fournier, A.K., Hall, E., Ricke, P., Storey, B.: Alcohol and the social network: Online social networking sites and college students' perceived drinking norms. Psychology of Popular Media Culture **2**(2), 86 (2013)
10. Gallardo, J.E., Cotta, C.: A GRASP-based memetic algorithm with path relinking for the far from most string problem. Engineering Applications of Artificial Intelligence **41**, 183–194 (2015)
11. Günneç, D., Raghavan, S., Zhang, R.: Least-cost influence maximization on social networks. INFORMS Journal on Computing **32**(2), 289–302 (2020)
12. Long, C., Wong, R.C.W.: Minimizing seed set for viral marketing. In: 2011 IEEE 11th International Conference on Data Mining, pp. 427–436. IEEE Press (2011)
13. Mousavi, S.R.: A hybridization of constructive beam search with local search for far from most strings problem. International Journal of Computer and Information Engineering **4**(8), 1200–1208 (2010)
14. Rad, A.A., Benyoucef, M.: Towards detecting influential users in social networks. In: International Conference on E-Technologies, pp. 227–240. Springer (2011)
15. Sun, R., Wu, J., Jin, C., Wang, Y., Zhou, W., Yin, M.: An efficient local search algorithm for minimum positive influence dominating set problem. Computers & Operations Research **154**, 106197 (2023)
16. Tatsis, V.A., Parsopoulos, K.E.: Dynamic parameter adaptation in metaheuristics using gradient approximation and line search. Applied Soft Computing **74**, 368–384 (2019)
17. Wang, F., Camacho, E., Xu, K.: Positive influence dominating set in online social networks. In: International Conference on Combinatorial Optimization and Applications, pp. 313–321. Springer (2009)
18. Wang, G.: Domination problems in social networks. Ph.D. thesis, University of Southern Queensland (2014)
19. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997)

Chapter 3
Adding Learning to CMSA

Abstract CMSA is undeniably an algorithm whose efficacy can, to some extent, be attributed to its inherent simplicity. As demonstrated in previous chapters of this book, unsophisticated variants of CMSA are capable of yielding highly satisfactory results. Nevertheless, such basic CMSA variants can, of course, undergo enhancement through the incorporation of supplementary algorithmic components. One avenue for refining barebone CMSA variants involves introducing a learning component into the solution construction mechanism. This addition enables the solution construction mechanism to generate solutions of improving quality over time. In this chapter, we will show how this can be done in the context of two combinatorial optimization problems that were already used for the illustration of other CMSA variants in previous chapters. In particular, applications to the Minimum Dominating Set (MDS) problem and the Far From Most String (FFMS) problem are presented.

3.1 Introduction

One of the possible disadvantages of standard CMSA is the fact that the solution construction mechanism generates, at each iteration, solutions from the same probability distribution as in all previous iterations. This means that the average quality of the solutions generated by the solution construction mechanism will remain stationary over time. Therefore, it is reasonable to believe that adding a learning mechanism to the solution construction process, causing the construction of improving solutions over time, might improve the overall search and optimization capability of CMSA.

In this chapter, we introduce a specific approach for incorporating a learning mechanism into standard CMSA, which was detailed in Chap. 1 of this book. Specifically, we will elaborate on the integration of CMSA with the so-called *bacterial algorithm (BA)*, an evolutionary algorithm where the crossover operator is inspired by observed bacterial processes. It is important to note that the choice of the bacterial algorithm is just one of many options for introducing learning to CMSA, and this application should be regarded as an illustrative example. Generally, BA

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

algorithms draw inspiration from the survival dynamics of bacterial populations and their evolution, including the development of resistance to antibiotics. It is noteworthy that bacterial populations, upon developing such resistance, become impervious to the effects of antibiotics. While this resistance is advantageous from the bacteria's perspective, it poses significant challenges in the medical field. Infections caused by antibiotic-resistant microorganisms often defy conventional treatment, leading to prolonged illness and an increased risk of mortality. In essence, antibiotic resistance represents a form of drug resistance where a microorganism can withstand exposure to an antibiotic [5].

Recent research has explored the application of bacterial behavior in diverse contexts, including its utilization in group formation for designing students' activities. The fundamental concept revolves around students collaborating in a group to enhance their academic performance [2].

The developed CMSA algorithm with learning is labeled LEARN_CMSA. We demonstrate the usefulness of LEARN_CMSA in the context of the Minimum Dominating Set (MDS) problem which is already known from Chap. 1 of this book. Moreover, we show its application to the Far From Most String (FFMS) problem known from Chap. 2. However, before presenting these applications, we first describe the BA algorithm and LEARN_CMSA in general terms.

3.2 The Bacterial Algorithm

The BA algorithm was first described in [2, 6]. In this work, we present a simplification of the original algorithm. Before delving into the algorithm description, the interested reader should note, however, that we believe that nearly any population-based metaheuristic could be used for the same purpose. In other words, we do not believe that the success of LEARN_CMSA, which will be demonstrated further below, is due to specific algorithmic features of the BA algorithm. In contrast, we believe that the success of LEARN_CMSA is due to the way it is interleaved with CMSA.

Bacteria, being microorganisms or microscopic life forms, share this classification with viruses, algae, fungi, and protozoa. Essentially, bacteria are unicellular organisms of minuscule size with the ability to thrive in diverse environments such as oceans, terrestrial habitats, outer space, and even within the human intestine. The interaction between humans and bacteria is intricate; at times, bacterial behavior proves beneficial, even essential, to human well-being, while in other instances, it may instigate perilous diseases and health complications. Since the discovery of penicillin by Alexander Fleming in 1929 [4], antibiotics have played a crucial role in treating diseases caused by bacteria and other microorganisms. However, a significant challenge arises when bacteria are frequently exposed to the same type of antibiotics, leading to the development of defense mechanisms to counteract the antibiotics' effects. This pivotal survival mechanism for bacteria involves communication within the population, functioning as a collaborative mechanism that involves the transfer of DNA among bacteria. Consequently, more robust bacteria can pass on their traits to weaker ones, enabling them to acquire the ability to resist the common adversary: antibiotics.

A significant distinction between higher organisms and bacteria lies in the mechanism of genetic reproduction and recombination. Notably, populations of superior organisms exhibit genetic variability through a vertical process—offspring are generated as part of a new generation through sexual interaction between parents. In contrast, genetic diversity within bacteria populations can occur through a horizontal process, wherein genetic material is exchanged among individuals without necessitating the creation of a new individual. Consequently, within the bacterial context, it is more fitting to refer to donors and receptors rather than parents and offspring. In bacteria, reproduction is achieved through cell division, a process of replication resulting in a new bacterial generation containing identical genetic material. This process may be susceptible to errors during replication or influenced by external factors, such as mutagens, potentially impacting the outcome.

As previously discussed, the emergence of antibiotic resistance poses a significant concern for human health. Conversely, for bacteria, this development signifies an evolutionary enhancement that augments their ability to survive. Viewing this bacterial behavior from an optimization perspective reveals it as a valuable source of inspiration, as exemplified in [2, 6]. This is especially true for the process of *horizontal transfer* of DNA material, wherein genetic material is shared among fellow bacteria belonging to the same generation.

The pseudo-code of our variant of the bacterial algorithm (BA) is provided in Algorithm 3.1. The algorithm takes as input values for the following five parameters (see line 1):

1. p_{size}: population size.
2. pr_{heur}: rate of initial solutions generated by a probabilistic heuristic
3. d_{rate}: determinism rate used by the probabilistic heuristic.
4. pr_{con}: the mutation probability during the conjugation phase.
5. pr_{reg}: the mutation probability during the regeneration phase.

At the onset of the algorithm, the best-so-far solution (S^{bsf}) is initialized as the empty set. Subsequently, the initial population of solutions, comprising p_{size} solutions, is generated using the function GenerateInitialPopulation($p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}}$). In this process, a solution is created with a probability of pr_{heur} through a randomized heuristic, which is problem-specific. Otherwise, the solution is generated uniformly at random. Note that each solution corresponds to a bacterium. However, in an attempt to describe the algorithm in metaphor-free language, we will not use the term bacterium from here on.

The initial step in each iteration involves determining the iteration-best solution S^{ib} from the current population (refer to line 5), facilitating the update of the best-so-far solution if indicated (line 6). Following that, the two principal procedures of the BA—conjugation and regeneration—are performed. Both procedures commence similarly (refer to lines 7 and 8 for conjugation, and lines 11 and 12 for regeneration). Specifically, the current population P undergoes division into

Algorithm 3.1: Pseudo-code of the bacterial algorithm (BA)

```
 1: input: parameter values for $p_{\text{size}}$, $pr_{\text{heur}}$, $d_{\text{rate}}$, $pr_{\text{con}}$, $pr_{\text{reg}}$
 2: $S^{\text{bsf}} := \emptyset$
 3: $P :=$ GenerateInitialPopulation$(p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}})$
 4: while CPU time limit not reached do
 5:     $S^{\text{ib}} := \text{argmin}\{f(S) \mid S \in P\}$
 6:     if $f(S^{\text{ib}}) < f(S^{\text{bsf}})$ then $S^{\text{bsf}} := S^{\text{ib}}$
        CONJUGATION PHASE
 7:     $\text{Atb}_{\text{level}} :=$ DetermineSeparationLevel$(P)$
 8:     $(P_{\text{donor}}, P_{\text{receptor}}) :=$ Classification$(\text{Atb}_{\text{level}}, P)$
 9:     $P_{\text{receptor}} :=$ Conjugation$(P_{\text{donor}}, P_{\text{receptor}}, pr_{\text{con}})$
10:     $P := P_{\text{donor}} \cup P_{\text{receptor}}$
        REGENERATION PHASE
11:     $\text{Atb}_{\text{level}} :=$ DetermineSeparationLevel$(P)$
12:     $(P_{\text{donor}}, P_{\text{receptor}}) :=$ Classification$(\text{Atb}_{\text{level}}, P)$
13:     $P_{\text{receptor}} :=$ Regeneration$(P_{\text{donor}}, pr_{\text{reg}})$
14:     $P := P_{\text{donor}} \cup P_{\text{receptor}}$
15: end while
16: output: $S^{\text{bsf}}$
```

two segments: donor solutions (P_{donor}) and receptor solutions (P_{receptor}). This division is initiated by determining a separator level ($\text{Atb}_{\text{level}}$) through the function DetermineSeparatorLevel(P).

Two pairs of solutions, denoted as (S_i, S_j) and (S_k, S_l), are randomly selected from the current population P. Subsequently, the superior solution from each pair is determined. Let $S_1 := \text{argmin}\{f(S_i), f(S_j)\}$ and $S_2 := \text{argmin}\{f(S_k), f(S_l)\}$. Additionally, designate S_{max} as the solution with the lower quality between S_1 and S_2, i.e., $S_{\text{max}} := \text{argmax}\{f(S_1), f(S_2)\}$. Accordingly, $\text{Atb}_{\text{level}}$ is defined as $f(S_{\text{max}})$. This employs a cost-effective approach to select a solution with a fitness value close to the median of the population.[1] Subsequently, $\text{Atb}_{\text{level}}$ is employed to partition the current population into two sub-populations: P_{donor}, consisting of donor solutions, and P_{receptor}, comprising recipient solutions. In particular, solutions in P_{receptor} exhibit an objective function value worse than $\text{Atb}_{\text{level}}$, while donor solutions have an objective function value better or equal to $\text{Atb}_{\text{level}}$.

The Conjugation Phase

In the *conjugation phase* of BA, all receptor solutions from P_{receptor} receive a random piece of genetic material from some randomly chosen donor solution from P_{donor}. As in nature, this operation may suffer a corruption in the genetic transcription (mutation). Thus, mutation is applied with a mutation probability of pr_{con}.

[1] Note that our description of this process assumes a minimization problem to be considered for optimization. In the context of a maximization problem, obvious adjustments must be made.

Algorithm 3.2: Pseudo-code of LEARN_CMSA

```
1: **input 1:** complete set of solution components $C$
2: **input 2:** values for CMSA parameters $n_a$, $age_{\max}$, and $t_{\text{ILP}}$
3: **input 3:** values for BA parameters $p_{\text{size}}$, $pr_{\text{heur}}$, $d_{\text{rate}}$, $pr_{\text{con}}$, $pr_{\text{reg}}$
4: **input 4:** values for CMSA/BA interplay parameters $b_{\text{iter}}$, $r_{\text{inject}}$
5: $S^{\text{bsf}} := \emptyset$
6: $C' := \emptyset$
7: $age[c] := 0$ for all $c \in C$
8: $P$ := GenerateInitialPopulation($p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}}$)
9: **while** CPU time limit not reached **do**
10:    $P$ := Execute_BA_Algorithm($P, b_{\text{iter}}, p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}}, pr_{\text{con}}, pr_{\text{reg}}$)
11:    $T$ := Extract_From($P, n_a$)
12:    **for** all $S \in T$ **do**
13:       **for** all $c \in S$ **and** $c \notin C'$ **do**
14:          $age[c] := 0$
15:          $C' := C' \cup \{c\}$
16:       **end for**
17:    **end for**
18:    $S^{\text{ILP}}$ := SolveSubinstance($C', t_{\text{ILP}}$)
19:    **if** $f(S^{\text{ILP}}) < f(S^{\text{bsf}})$ **then** $S^{\text{bsf}} := S^{\text{ILP}}$ **end if**
20:    Adapt($C', S^{\text{ILP}}, age_{\max}$)
21:    $P$ := InjectSolverSolution($P, S^{\text{bsf}}, r_{\text{inject}}$)
22: **end while**
23: **output:** $S^{\text{bsf}}$
```

The Regeneration Phase

In the *regeneration phase* of BA, after classifying the members of the current population into donors P_{donor} and receptors P_{receptor}, all solutions from P_{receptor} are exchanged with clones of randomly chosen donor solutions after applying mutation with probability pr_{reg}.

These steps are iterated until a computation time limit is reached. Upon termination, the best-so-far solution S^{bsf} is yielded as the output.

3.3 The LEARN_CMSA Algorithm: A General Description

The pseudo-code of LEARN_CMSA is provided in Algorithm 3.2. Note that this pseudo-code is an extension of the one of standard CMSA presented in Algorithm 1.1 on page 19. Alongside the previously specified CMSA and BA parameters, it requires input for the following two parameters that govern the interaction between CMSA and BA:

1. b_{iter}: number of BA iterations executed in function Execute_BA_Algorithm(P, b_{iter}, p_{size}, pr_{heur}, d_{rate}, pr_{con}, pr_{reg}) at each CMSA iteration; see line 10.
2. r_{inject}: the rate of injection of the solution returned by the ILP solver (S^{bsf}) into the current BA population in function InjectSolverSolution(P, S^{bsf}, r_{inject}); see line 21.

> Differences Between LEARN_CMSA and Standard CMSA

1. After the initialization of the CMSA parameters in lines 5–7, the initial population P of the BA algorithm is generated in line 8. This is done as previously explained in Sect. 3.2.
2. At the onset of every CMSA iteration, the function Execute_BA_Algorithm(P, b_{iter}, p_{size}, pr_{heur}, d_{rate}, pr_{con}, pr_{reg}) carries out b_{iter} iterations of the BA algorithm, following the same methodology described in Sect. 3.2. The output of this function is the current population P of the BA algorithm.
3. Rather than employing a probabilistic, constructive heuristic for producing new solutions at each iteration, LEARN_CMSA utilizes function Extract_From(P, n_{a}) (refer to line 11) to extract n_{a} solutions from the current BA population P, which are then stored in set T. Specifically, T comprises the best solution from P in addition to $n_{\text{a}} - 1$ randomly selected donor solutions from P. It is worth noting that, for this purpose, the separator level ($\text{Atb}_{\text{level}}$) is determined, and the population P is partitioned into donors and receptors, as described in Sect. 3.2.
4. As a last step of every LEARN_CMSA iteration, the solution S^{bsf} is employed to substitute $\lfloor r_{\text{inject}} \cdot |P_{\text{receptor}}| \rfloor$ randomly chosen receptor solutions within P. This action is executed through the InjectSolverSolution(P, S^{bsf}, r_{inject}) function, located at line 21.

In this hybridization approach of CMSA and BA, both memory mechanisms—the sub-instance C' of CMSA and the population P of BA—mutually influence each other. Specifically, a set of donor solutions from P is incorporated into C' during each iteration, and CMSA influences BA by introducing S^{bsf} into the BA population P.

3.4 Application to the MDS Problem

The first application of LEARN_CMSA that will be presented is the one to the Minimum Domination Set (MDS) problem which was already introduced in Sect. 1.4 on page 20. Concerning the complete set C of solution components for the LEARN_CMSA approach, we decided for the intuitive approach in which C contains a solution component c_i for each vertex $v_i \in V$ of the input graph

$G = (V, E)$. Moreover, solving sub-instances in function SolveSubinstance(C', t_{ILP})—see line 18 of Algorithm 3.2—works exactly in the same way as described in Sect. 1.4.2.2 on page 25. In the following, the remaining algorithmic components of BA and LEARN_CMSA will be outlined.

3.4.1 Generating the Initial Population

Function GenerateInitialPopulation($p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}}$)—see line 4 of Algorithm 3.1, respectively line 8 of Algorithm 3.2—generates the initial population of BA and of the BA-part of LEARN_CMSA. In particular, this function generates p_{size} solutions in the following way. The generation of each solution works as follows:

1. A number $r \in [0, 1]$ is drawn uniformly at random. In case $r \leq pr_{\text{heur}}$, the solution is generated by the randomized heuristic described in Sect. 1.4.1.1 on page 22. This randomized heuristic requires values for parameters d_{rate} (determinism rate) and l_{size} (candidate list size). In an attempt to reduce the number of parameters to be tuned, in Sect. 3.4.4.2 we will only consider d_{rate}, which is the more important parameter among these two. In contrast, l_{size} is set to a fixed value of 10 both for BA and LEARN_CMSA.
2. In case r, the random number drawn uniformly at random in the first step, is greater than pr_{heur}, a solution is randomly constructed in the following way. First, all nodes of the graph are sorted randomly. Subsequently, the resulting ordered list of nodes is traversed sequentially, adding a node to the solution if it covers at least one node which is still uncovered so far. This process stops once all nodes of the graph are covered.

3.4.2 Implementation of Conjugation

Remember that in the conjugation phase of BA, each receptor solution $S_r \in P_{\text{receptor}}$ receives a piece of genetic material from a randomly chosen donor solution from P_{donor}. Moreover, mutation is applied to the result with a mutation probability pr_{con}. Subsequently, we will explain how this step is implemented in the case of the application to the MDS problem.

Given a receptor solution S_r and a randomly chosen donor solution S_d, first, they are transformed into solutions s_r and s_d containing the corresponding nodes of the input graph. Then, a set $\tilde{V}$ is defined containing the union of the sets of nodes present in s_r and s_d. Each $v \in \tilde{V}$ is assigned a value

$$\delta_v := |N[v]| + r_v \quad , \tag{3.1}$$

where $N[v]$ is the closed neighborhood of v in G, that is, $N[v] := N(v) \cup \{v\}$, and r_v is a random value chosen from the normal distribution $\mathcal{N}(0, \sigma_{\text{pert}}^2)$, that is, a normal distribution with zero as mean and σ_{pert} as standard deviation. Note that σ_{pert} is an important parameter of both BA and LEARN_CMSA which will be tuned in Sect. 3.4.4.2. Next, all nodes from $\tilde{V}$ are ordered according to decreasing δ_v-values. Finally, a new solution s_{con} is generated by sequentially traversing this list and adding each node that covers at least one of the nodes of the input graph that are still uncovered. The process stops once the whole graph is covered.

Application of Mutation

Finally, mutation is applied to the new solution s_{con} as follows. First, let $\tilde{V}$ be defined as the set of nodes of input graph G which are not in s_{con}. Then, for each node v of s_{con} it is decided with a probability pr_{con} if it is removed from s_{con} and added to $\tilde{V}$. This results in a partial solution s_{con}^p. Then, each $v \in \tilde{V}$ is assigned a value

$$\gamma_v := \left|\left\{v' \in N[v] \mid v' \text{ still uncovered by } s_{\text{con}}^p\right\}\right| + r_v \quad . \tag{3.2}$$

In other words, γ_v is the sum of the number of still uncovered neighbors of v (concerning the partial solution s_{con}^p) and r_v, which is—as above—a random value chosen from the normal distribution $\mathcal{N}(0, \sigma_{\text{pert}}^2)$. Next, all nodes from $\tilde{V}$ are ordered according to decreasing γ_v-values. A mutated solution s_{mut} is then generated by sequentially traversing this list and adding each node that covers at least one of the nodes of the input graph that are still uncovered. The process stops once the whole graph is covered. The corresponding CMSA-solution S_{mut} then replaces the receptor solution S_r in population P.

3.4.3 Implementation of Regeneration

Remember that, after the conjugation phase (see lines 7–10 of Algorithm 3.1 on page 74), the resulting population P is again divided into donor solutions P_{donor} and receptor solutions P_{receptor}. In the regeneration phase, each receptor solution $S_r \in P_{\text{receptor}}$ is replaced in the following way. First, a donor solution $S_d \in P_{\text{donor}}$ is chosen uniformly at random. Next, a clone S_d^c of S_d is produced. After that, mutation is applied to S_d^c in the same way as described above in the conjugation phase. The only difference is that pr_{reg}, instead of pr_{con}, is now used as the probability to reduce S_d^c.

3.4.4 Experimental Evaluation

The experimental evaluation of BA and LEARN_CMSA for the MDS problem encompasses the following algorithms:

1. CPLEX: Application of CPLEX 22.1 to each considered problem instance, utilizing the default parameter values of CPLEX.
2. CMSA_INT: The standard CMSA algorithm, making use of the intuitive approach to defining the set of solution components. This algorithm was described in Sect. 1.4 on page 20.
3. BA: The pure bacterial algorithm from this section.
4. LEARN_CMSA: The CMSA approach extended with a learning mechanism (based on the bacterial algorithm) presented in this section.

As before, CPLEX 22.1 is used—both in standalone mode (CPLEX) and within LEARN_CMSA—in one-threaded mode. For conducting the experiments we used the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

3.4.4.1 Benchmark Instances

The same benchmark instances as those already introduced in Chap. 1 of this book were used for the experimental evaluation, that is, 480 Erdös-Rényi graphs, 480 Watts-Strogatz graphs, and 480 Barabási-Albert graphs. In particular, for each of the three considered graph models, this benchmark set consists of 30 graphs for each combination of $|V| \in \{500, 1000, 1500, 2000\}$ and four different graph densities. Remember that the graph density is controlled in Erdös-Rényi graphs by the edge probability (p), in Watts-Strogatz graphs by a parameter k, and in Barabási-Albert graphs by a parameter m.

3.4.4.2 Algorithm Tuning

Both BA and LEARN_CMSA underwent parameter tuning with the `irace` tool, which was already used in all other experimental evaluations presented in this book. The interested reader may find a description of `irace` in Sect. 1.2.1 on page 12. In particular, `irace` was applied with a budget of 3000 algorithm runs exactly once for the tuning of each of the two algorithms. The same tuning instances were used for this purpose as the ones described in Sect. 1.4.3.2 on page 27. Moreover, the same CPU time limits were used as those described in Sect. 1.4.3.2, that is, 150 CPU seconds was used for all graphs with $|V| = 500$, 300 CPU seconds for all graphs with $|V| = 1000$, 450 CPU seconds for all graphs with $|V| = 1500$ and 600 CPU seconds for all graphs with $|V| = 2000$.

Table 3.1 Parameters, domains, and tuning results for the MDS problem

Parameter	Domain	CMSA_INT	BA	LEARN_CMSA
n_a	$\{1,\ldots,50\}$	4	n.a.	30
age_max	$\{1,\ldots,10\}$	3	n.a.	2
d_rate	[0.0, 0.99]	0.29	0.8	0.65
l_size	$\{3,\ldots,50\}$	35	n.a.	15
t_ILP	$\{1,\ldots,20\}$	13	n.a.	4
$\text{cplex}_\text{emphasis}$	{`true`, `false`}	`true`	n.a.	`true`
$\text{cplex}_\text{warmstart}$	{`true`, `false`}	`false`	n.a.	`false`
$\text{cplex}_\text{abort}$	{`true`, `false`}	`false`	n.a.	`false`
p_size	$\{10,\ldots,1000\}$	n.a.	21	207
pr_heur	[0.0, 1.0]	n.a.	0.43	0.43
pr_con	[0.0, 0.5]	n.a.	0.44	0.06
pr_reg	[0.0, 0.5]	n.a.	0.02	0.1
σ_pert	[1.0, 10.0]	n.a.	1.09	3.26
r_inject	[0.01, 0.99]	n.a.	n.a.	0.12
b_iter	$\{1,\ldots,100\}$	n.a.	n.a.	7

Table 3.1 shows both the parameters involved in the different algorithms together with their domains, and the tuning results. Note that the tuning results of CMSA_INT are also provided in this table. They were copied from Table 1.2 on page 27. The first eight parameters in this table are the usual CMSA-related parameters.[2] The next five parameters are specific to BA, and the BA-related parts of LEARN_CMSA. Hereby, note that the four parameters are generic ones, while the fifth one (σ_pert) is specific to our application to the MDS problem. Finally, the last two parameters in the table determine the interplay between BA and CMSA in LEARN_CMSA.

The following parameter settings are noteworthy. First, the number of solutions that are fed into the sub-instance at each iteration (n_a) is much higher in LEARN_CMSA than in CMSA_INT. In addition to this, the CPU time limit for CPLEX (for solving sub-instances) is much lower. This indicates that sub-instances in LEARN_CMSA are built based on better solutions than in CMSA_INT. Otherwise, sub-instances in LEARN_CMSA would be too large to be solved by CPLEX within four CPU seconds. Another noteworthy difference concerns the setting of the population size of BA in comparison to the one in LEARN_CMSA. While BA makes use of a small population size of 21 individuals, the BA-part in LEARN_CMSA requires a significantly larger population size of 207 individuals. This might be due to the need for maintaining a certain diversity in the BA population of LEARN_CMSA to be able to feed the sub-instance with both high-quality solutions but also diverse ones.

[2] Remember that a description of the CPLEX parameters $\text{cplex}_\text{emphasis}$, $\text{cplex}_\text{warmstart}$, and $\text{cplex}_\text{abort}$ that are used within CMSA_INT and LEARN_CMSA is provided on page 23.

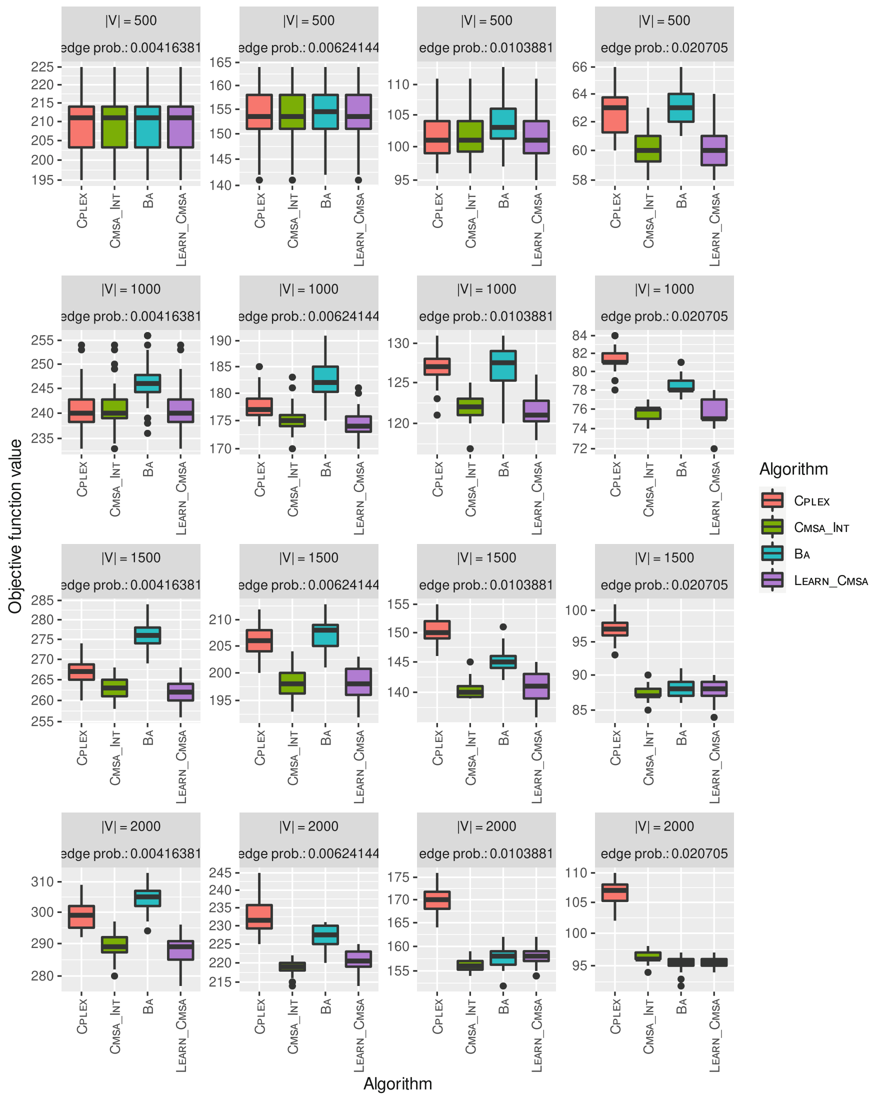


Fig. 3.1 LEARN_CMSA results for Erdös-Rényi graphs

3.4.4.3 Results

All four algorithmic techniques (CPLEX, CMSA_INT, BA and LEARN_CMSA) were applied exactly once to each of the problem instances from the benchmark set. The computation time limit for CMSA_INT, BA and LEARN_CMSA was the same as the one used for tuning (see previous section). The results are shown in the form of box plots in Fig. 3.1 concerning Erdös-Rényi graphs and in Fig. 3.2 concerning

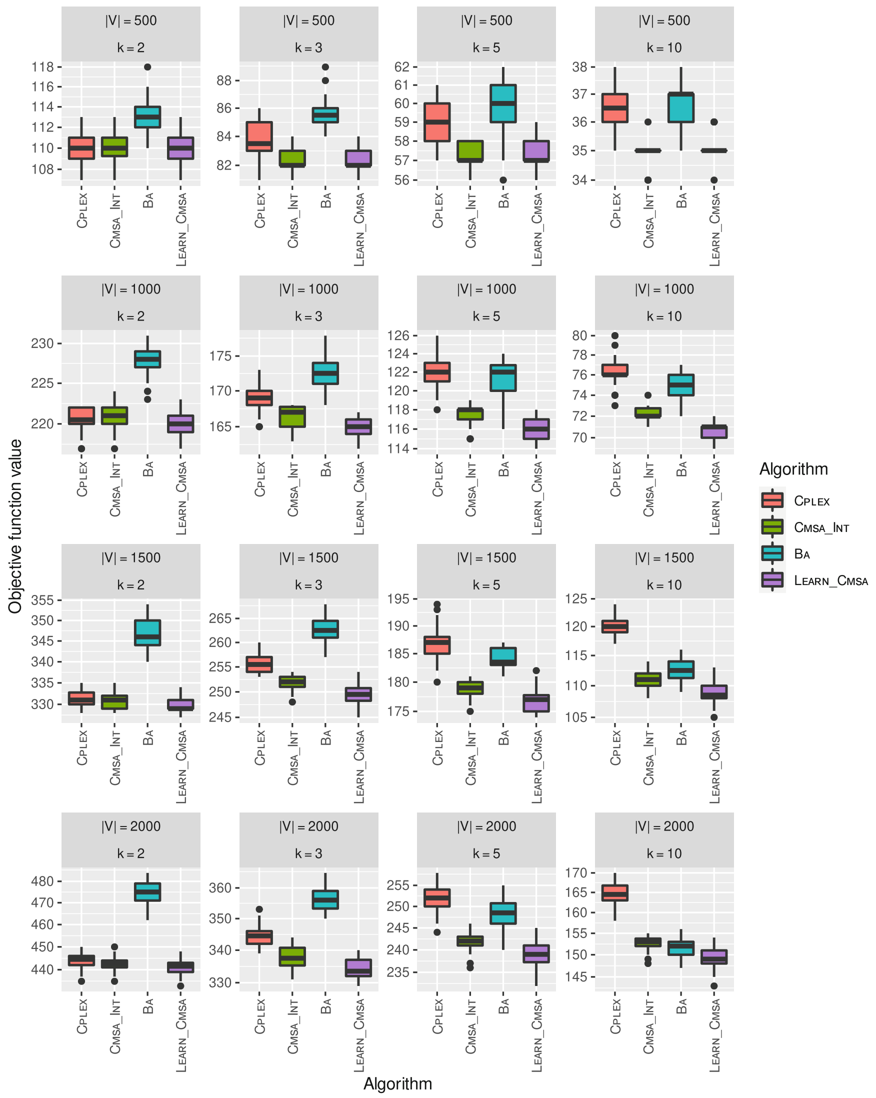


Fig. 3.2 LEARN_CMSA results for Watts-Strogatz graphs

Watts-Strogatz graphs. Results for Barabási-Albert graphs are not shown because no significant difference between CMSA_INT and LEARN_CMSA could be detected. Both graphics contain a 4 × 4 grid of box plots, where the rows present the results (from top to bottom) for graphs of increasing size, and the columns (from left to right) present the results for graphs of increasing density.

To support the result analysis with claims regarding statistical significance, separate CD plots are presented for the graphs of each of the two network models in Figs. 3.3 and 3.4. Refer to Sect. 1.2.3 on page 16 for a comprehensive explanation

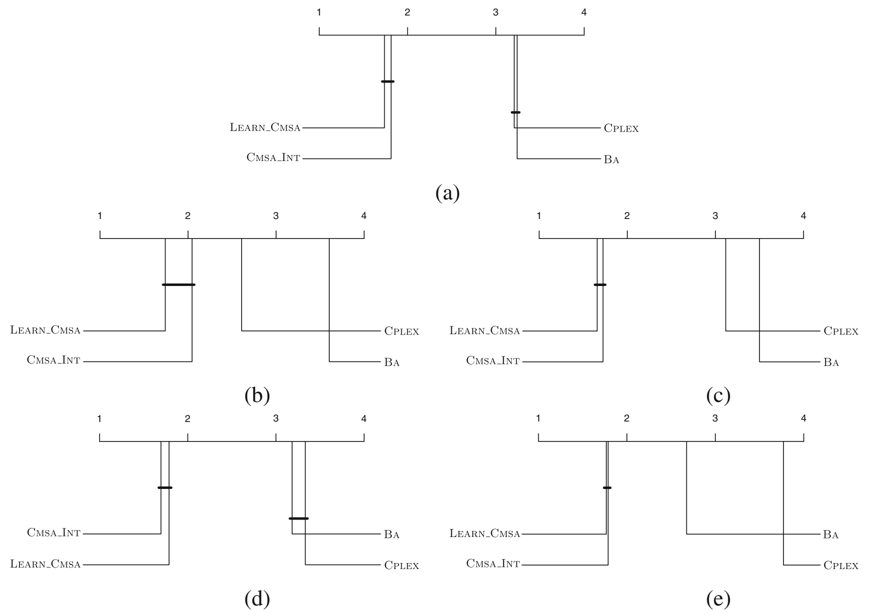


Fig. 3.3 Critical difference (CD) plots concerning Erdös-Rényi graphs. (**a**) All graphs. (**b**) Density $p = 0.00624144$. (**c**) Density $p = 0.00416381$. (**d**) Density $p = 0.0103881$. (**e**) Density $p = 0.020705$

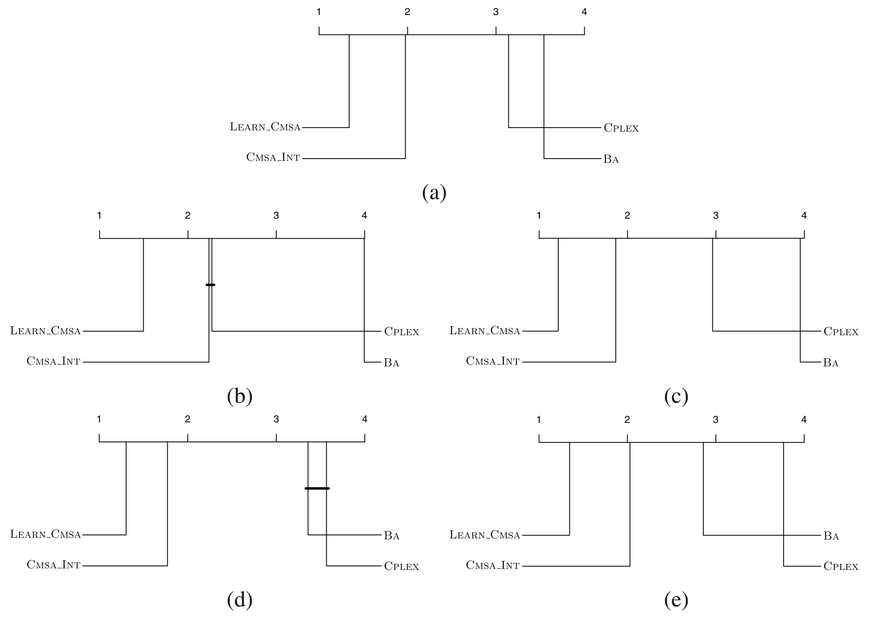


Fig. 3.4 Critical difference (CD) plots concerning Watts-Strogatz graphs. (**a**) All graphs. (**b**) Density $k = 2$. (**c**) Density $k = 3$. (**d**) Density $k = 5$. (**e**) Density $k = 10$

of CD plots. Each CD plot figure comprises five graphics. The first one at the top provides statistical information across the entire set of graphs for the respective network model. The remaining four CD plot graphics offer statistical insights into all graphs of a specific density.

Main Observations Concerning the MDS Results

1. As a stand-alone algorithm, BA is clearly inferior to both CMSA variants. Especially in the context of sparse problem instances (left-most columns in both box plot figures), BA is not able to compete with the other algorithms. However, its relative performance is improving with growing graph density. It even outperforms CPLEX for the densest graphs of both types.
2. Even though BA is inferior as a stand-alone approach, its learning mechanism clearly adds value to LEARN_CMSA. While LEARN_CMSA is only slightly better than CMSA_INT in the context of Erdös-Rényi graphs, LEARN_CMSA outperforms CMSA_INT with statistical significance for Watts-Strogatz graphs.
3. In the context of Erdös-Rényi graphs, the advantage of LEARN_CMSA over CMSA_INT increases with decreasing graph density. This is different for Watts-Strogatz graphs, where LEARN_CMSA exhibits a more consistent superiority over CMSA_INT over the whole range of tested graph densities.

As in all other experimental evaluations described in this book, STN graphics were produced to further analyze the results; see Sect. 1.2.2 on page 13 for a description of the STNWeb tool that was used to produce these graphics. Figure 3.5a shows STN graphics (in addition to graphics providing information about the evolution of the algorithms over time) for two problem instances. In particular, the graphics in Fig. 3.5a, c and e deal with the first Watts-Strogatz graph with $n = 1500$ nodes and a rather high density ($k = 5$), while the remaining graphics in Fig. 3.5 (right column of graphics) refer to the first Erdös-Rényi graph with $n = 1000$ nodes and a lower density determined by the edge probability $p = 0.00614144$. While the two complete STN graphics actually look rather similar (see Fig. 3.5a and b), the STN graphics after search space partitioning reveal the following differences. In the case of the Watts-Strogatz graph (left column of graphics), it can be observed that after overlaps in the search trajectories rather at the beginning of the search process, the trajectories of CMSA_INT seem to move to a common attractor in the search space, while the ones of LEARN_CMSA, which is the best algorithm for this problem instance, seem to be able to identify solutions of very good quality with a different structure. Also for the BA trajectories, there does not seem to exist a common attractor. The fact of not finding overlaps between BA trajectories and trajectories of the CMSA variants after the initial stages of the search process is also partially explained by the graphic in Fig. 3.5e, which shows that—after the initial stages of the search process, that is, a few seconds in terms of computation

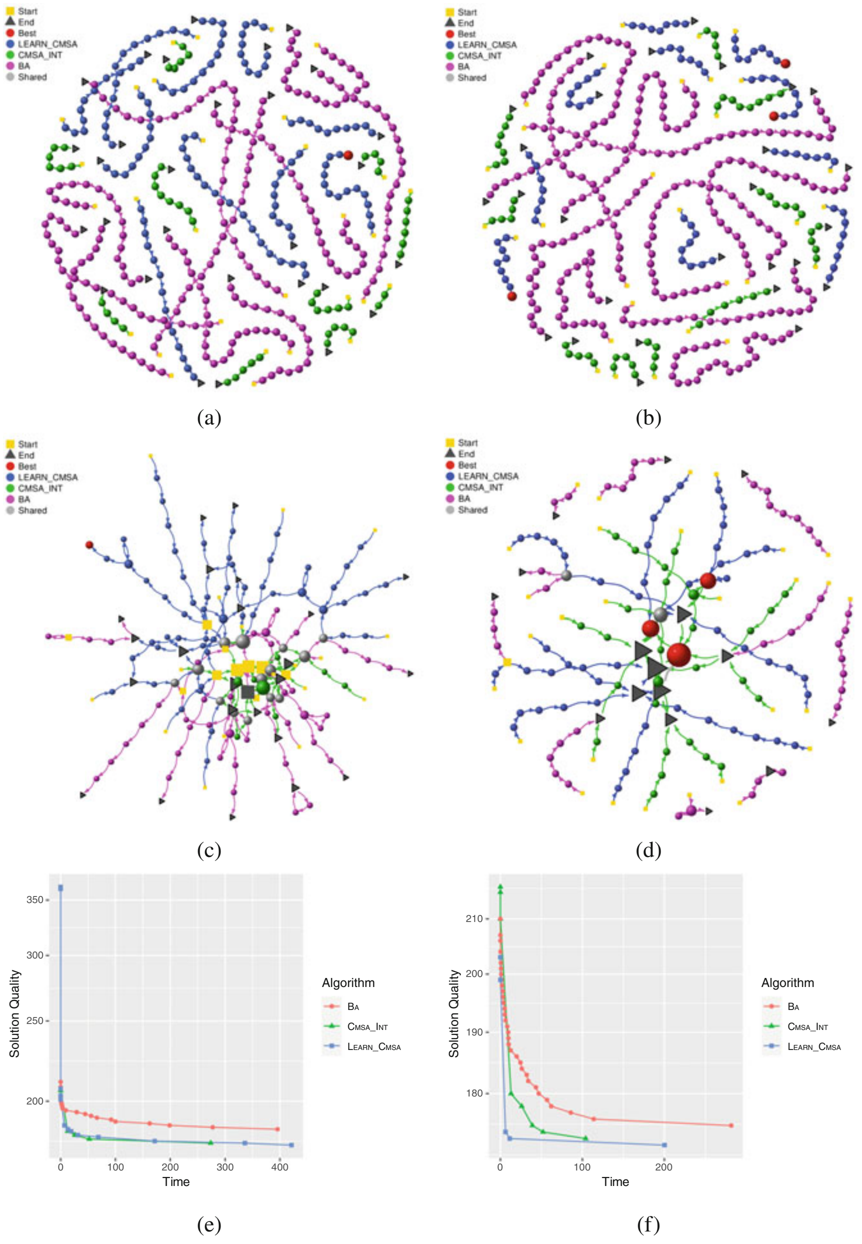


Fig. 3.5 STN graphics and algorithm evolution concerning the MDS problem. (**a**, **b**) Complete STN. (**c**, **d**) STN (after partitioning). (**e**, **f**) Algorithm evolution

time—the CMSA trajectories quickly advance to solutions of a quality which is never reached by any BA trajectory.

On the other side, the trajectories of both CMSA variants are clearly attracted by the same area of the search space in the case of the Erdös-Rényi graph; see the right column of graphics in Fig. 3.5. In fact, even though only two LEARN_CMSA trajectories can find solutions with the best quality (see Fig. 3.5b) there are CMSA_INT trajectories that end up in very similar solutions; see, for example, the CMSA_INT trajectories which pass through red dots in the STN graphic after search space partitioning; see Fig. 3.5d.

3.5 Application to the FFMS Problem

The second application of LEARN_CMSA presented in the following is the one to the Far From Most String (FFMS) problem which was already introduced in Sect. 2.4 starting on page 59. Concerning the complete set C of solution components for the LEARN_CMSA approach, we decided also here for the following intuitive definition (already introduced in Sect. 2.4.2 on page 62): each combination of a position j in a solution string (where $j = 1, \dots, m$) and a letter $a \in \Sigma$ is a solution component $c_{j,a}$. That is, $C := \{c_{j,a} \mid j = 1, \dots, m \text{ and } a \in \Sigma\}$. Any feasible solution S is a subset of C such that for each position $j = 1, \dots, m$, S contains exactly one of the solution components from $C_j := \{c_{j,a} \mid a \in \Sigma\}$. Note that, given a feasible solution S, a solution s in string form can be derived in a well-defined way by placing character a at position j of s for each solution component $c_{j,a} \in S$. The same holds the other way around.

Moreover, solving sub-instances in function SolveSubinstance(C', t_{ILP})—see line 18 of Algorithm 3.2—works exactly in the same way as described in Sect. 2.4.4 on page 63. In the following, the remaining algorithmic components of BA and LEARN_CMSA will be outlined.

3.5.1 Generating the Initial Population

Function GenerateInitialPopulation(p_{size}, pr_{heur}, d_{rate})—see line 3 of Algorithm 3.1, respectively line 8 of Algorithm 3.2—is responsible for creating the initial population of BA and the BA-part of LEARN_CMSA. Specifically, this function produces p_{size} solutions in the following manner. The generation process for each solution is outlined as follows:

1. A number $r \in [0, 1]$ is drawn uniformly at random. In case $r \leq pr_{\text{heur}}$, the solution is generated by the randomized heuristic described in Sect. 2.4.3 on page 62. This randomized heuristic requires a value for parameter d_{rate} (determinism rate), which will be determined by parameter tuning.

2. In case r, the random number drawn uniformly at random in the first step, is greater than pr_{heur}, a solution is randomly constructed by choosing for each of the m positions a letter from Σ with uniform probability.

3.5.2 Implementation of Conjugation

Recall that during the conjugation phase of BA, each receptor solution $S_r \in P_{\text{receptor}}$ acquires a genetic fragment from a randomly selected donor solution in P_{donor}. Additionally, mutation is applied to the outcome with a mutation probability denoted as pr_{con}. Below, we will elaborate on the implementation of this step in the context of the application to the FFMS problem.

Given a receptor solution S_r and a randomly chosen donor solution S_d, first, both are converted into their corresponding string forms s_r and s_d, and the receptor solution s_r is cloned, resulting in a cloned receptor solution s_r^c. Then, two indexes i and k are chosen uniformly at random such that $1 \leq i < k \leq m$. Subsequently, for each l from i to k—that is, for all $l \in \{i, i+1, \ldots, k-1, k\}$—the following is done: mutation is applied with a probability pr_{con} by placing a random character from Σ at position l of s_r^c. Otherwise, the new letter at position l of solution s_r^c is the letter at position l of the donor solution s_d, that is, $s_r^c[l] := s_d[l]$. Afterwards, the new solution s_r^c is transformed into the corresponding set S_r^c of solution components. Moreover, S_r^c replaces the original receptor solution S_r in the population P.

3.5.3 Implementation of Regeneration

Keep in mind that following the conjugation phase (refer to lines 7–10 of Algorithm 3.1 on page 74), the resultant population P is once again partitioned into donor solutions P_{donor} and receptor solutions P_{receptor}. In the regeneration phase, each receptor solution $S_r \in P_{\text{receptor}}$ is substituted in population P as follows. First, a donor solution $S_d \in P_{\text{donor}}$ is chosen uniformly at random. Next, a clone S_d^c of S_d is produced and transformed into its corresponding string form s_d^c. After that, mutation is applied to each position l of s_d^c with probability pr_{reg}, that is, with probability pr_{reg} the letter at position l of s_d^c is replaced with a random letter from Σ. After that, s_d^c is transformed back to a corresponding solution S_d^c in the form of solution components. Finally, S_d^c replaces S_d in population P.

3.5.4 Experimental Evaluation

The experimental evaluation of BA and LEARN_CMSA for the FFMS problem considers the following list of approaches:

1. CMSA_INT: Standard CMSA using the intuitive approach for the definition of the set of solution components. This algorithm was described in Sect. 1.4 on page 20.
2. ADAPT_CMSA: The adaptive variant of CMSA whose application to the FFMS problem was described in Sect. 2.4 of this book. Note that ADAPT_CMSA replaces the standalone application of CPLEX for this experimental evaluation. In Sect. 2.4 the superiority of ADAPT_CMSA over the standalone application of CPLEX in the context of the FFMS problem was shown.
3. BA: The pure bacterial algorithm from this section.
4. LEARN_CMSA: The CMSA approach extended with a learning mechanism (based on the bacterial algorithm) presented in this section.

As in all cases showcased in this book, CPLEX 22.1 is used in one-threaded mode in all considered CMSA variants. The experiments were conducted on the already described IIIA-CSIC in-house high-performance computing cluster, consisting of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

3.5.4.1 Benchmark Instances

For the experiments, we used the same benchmark instances as those already used for the experimental evaluation of ADAPT_CMSA in Sect. 2.4 of Chap. 2. These benchmark instances were described in Sect. 2.4.5.1 on page 64. The only difference is that, in addition to considering threshold $t = 0.8m$, all 360 problem instances are also solved with a threshold of $t = 0.85m$. This threshold has also been used in the related literature; see, for example, [3].

3.5.4.2 Parameter Tuning

Both BA and LEARN_CMSA underwent parameter tuning using the `irace` tool, which has been employed in all other experimental evaluations detailed in this book. The interested reader may find a description of `irace` in Sect. 1.2.1 on page 12. In particular, `irace` was applied with a budget of 3000 algorithm runs exactly once for the tuning of each of the two algorithms. The same tuning instances were used for this purpose as the ones described in Sect. 2.4.5.2 on page 64. Shortly, we have 12 tuning instances, all considered both for threshold $t = 0.8m$ and $t = 0.85m$. Moreover, the computation time limit for the algorithms was set to 600 CPU seconds per problem instance.

Table 3.2 presents the parameters and their respective domains for the various algorithms, along with the tuning results. Please be aware that the tuning outcomes for CMSA_INT and ADAPT_CMSA can also be found in this table. These results have been extracted from Table 2.3 on page 65. The initial seven parameters in this table pertain to standard CMSA. The subsequent four parameters are exclusive to BA, also covering the BA-related aspects of LEARN_CMSA. Lastly, the last

Table 3.2 Parameters, domains, and tuning results for the FFMS problem

Parameter	Domain	CMSA_INT	BA	LEARN_CMSA	ADAPT_CMSA
n_a	$\{1, \ldots, 50\}$	27	n.a.	32	n.a.
$age_{\max}$	$\{1, \ldots, 10\}$	9	n.a.	5	n.a.
d_{rate}	[0.0, 0.99]	0.81	0.85	0.49	n.a.
t_{ILP}	$\{1, \ldots, 20\}$	30	n.a.	48	1
cplex$_{\text{emphasis}}$	{true, false}	true	n.a.	true	true
cplex$_{\text{warmstart}}$	{true, false}	true	n.a.	true	true
cplex$_{\text{abort}}$	{true, false}	false	n.a.	false	n.a.
α^{LB}	[0.6, 0.99]	n.a.	n.a.	n.a.	0.64
α^{UB}	[0.6, 0.99]	n.a.	n.a.	n.a.	0.99
α_{red}	[0.01, 0.1]	n.a.	n.a.	n.a.	0.1
t_{prop}	[0.1, 0.8]	n.a.	n.a.	n.a.	0.42
p_{size}	$\{10, \ldots, 1000\}$	n.a.	581	219	n.a.
pr_{heur}	[0.0, 1.0]	n.a.	0.21	0.99	n.a.
pr_{con}	[0.0, 0.5]	n.a.	0.0	0.14	n.a.
pr_{reg}	[0.0, 0.5]	n.a.	0.01	0.14	n.a.
r_{inject}	[0.01, 0.99]	n.a.	n.a.	0.29	n.a.
b_{iter}	$\{1, \ldots, 100\}$	n.a.	n.a.	39	n.a.

two parameters in the table dictate the interaction between BA and CMSA within LEARN_CMSA.

As in the case of the parameter tuning results for the MDS problem presented in Sect. 3.4.4.2, the number of solutions fed into the sub-instance at each iteration (n_a) is higher in LEARN_CMSA than in CMSA_INT. This indicates that sub-instances in LEARN_CMSA are built based on better solutions than in CMSA_INT. However, there are also noteworthy differences to the parameter settings of BA and LEARN_CMSA in the context of the MDS problem. A striking difference is that BA requires a rather large population size for the FFMS problem ($p_{\text{size}} = 581$), while the opposite was the case for the MDS problem. On the other side, the size of the BA population within LEARN_CMSA is nearly the same in both applications (207 individuals in the case of the MDS problem vs. 219 individuals for the FFMS problem).

3.5.4.3 Results

The results are shown employing box plots in Fig. 3.6 (concerning threshold $t = 0.8m$) and in Fig. 3.7 (concerning threshold $t = 0.85$). To summarize the results and to provide a statistical basis for the comparison, corresponding CD plots are shown in Fig. 3.8 (concerning threshold $t = 0.8m$) and in Fig. 3.9 (concerning threshold $t = 0.85m$). Remember that a general description of the nature of CD plots and the information they provide was given in Sect. 1.2.3 in Chap. 1 of this book.

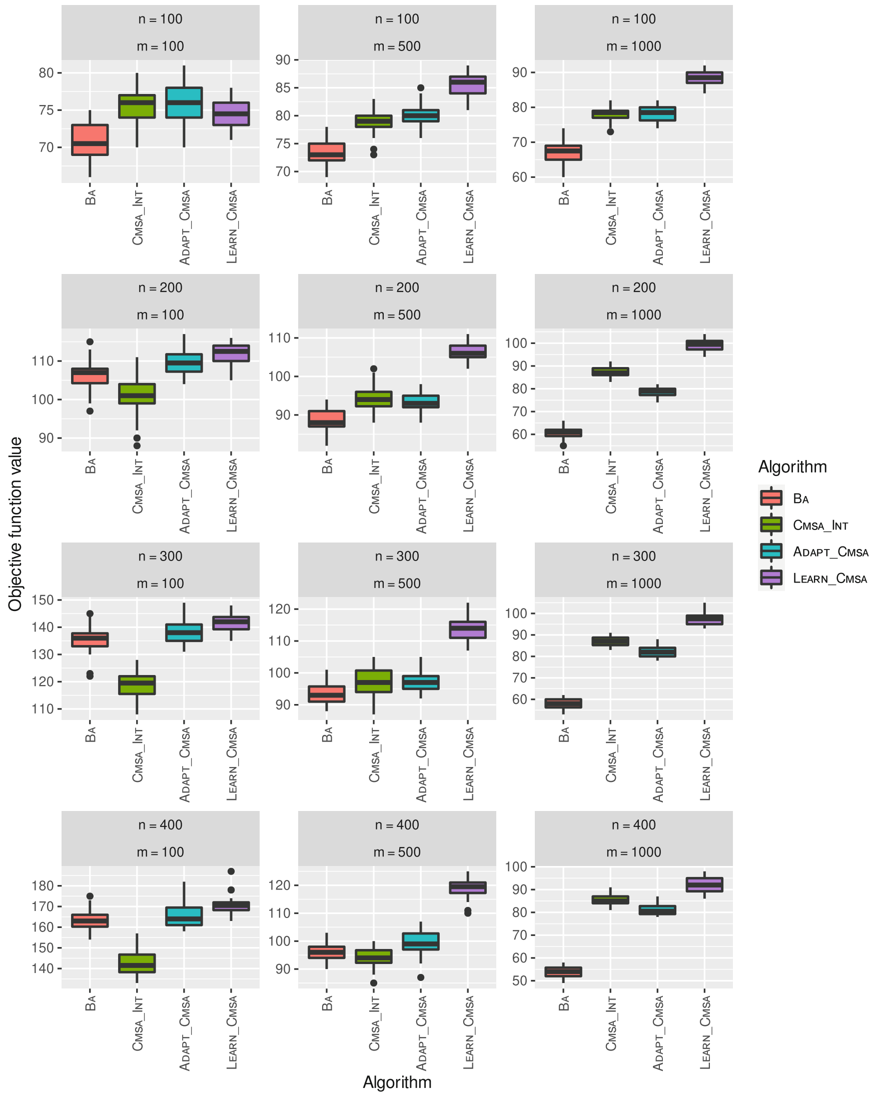


Fig. 3.6 LEARN_CMSA results for the FFMS problem, $t = 0.8m$

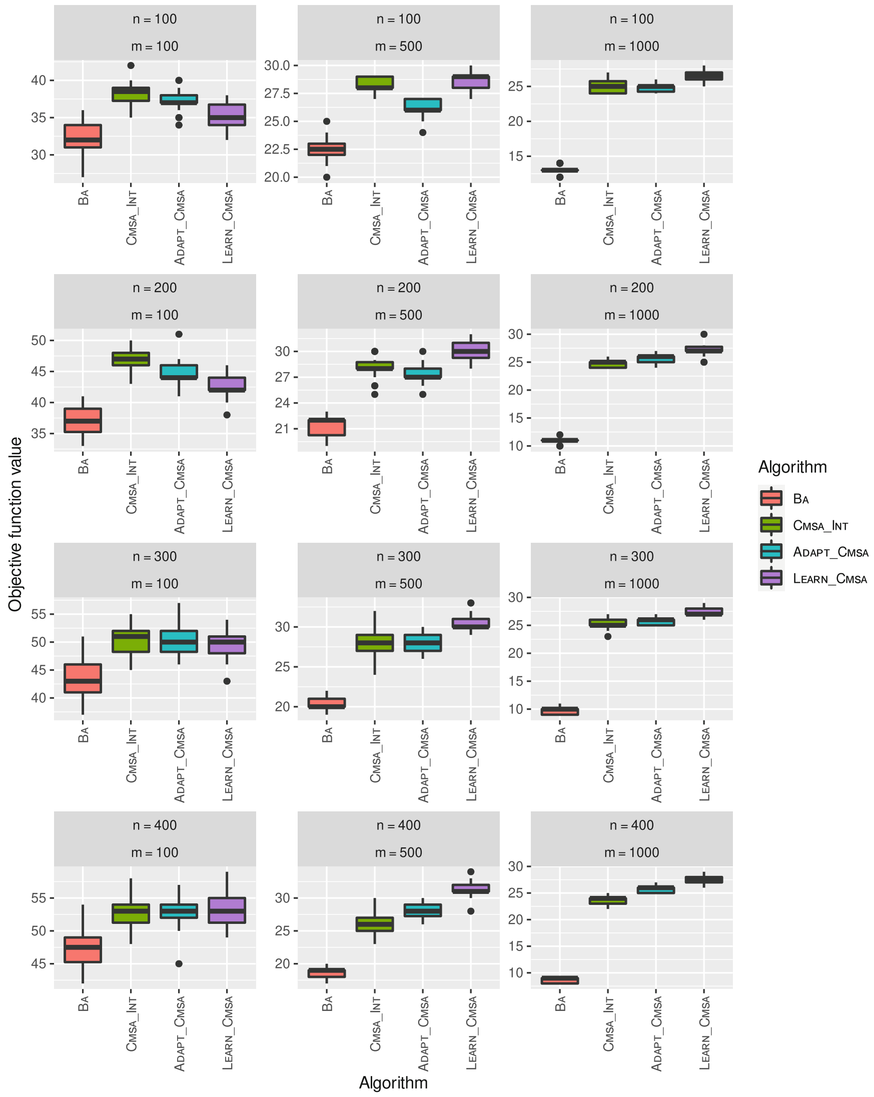


Fig. 3.7 LEARN_CMSA results for the FFMS problem, $t = 0.85m$

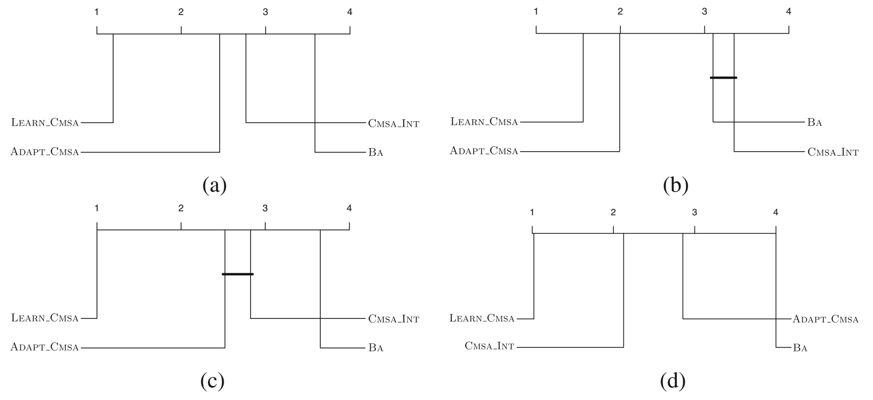


Fig. 3.8 Critical difference (CD) plots concerning FFMS instances with $t = 0.8m$. (**a**) All instances. (**b**) Instances with $m = 100$. (**c**) Instances with $m = 500$. (**d**) Instances with $m = 1000$

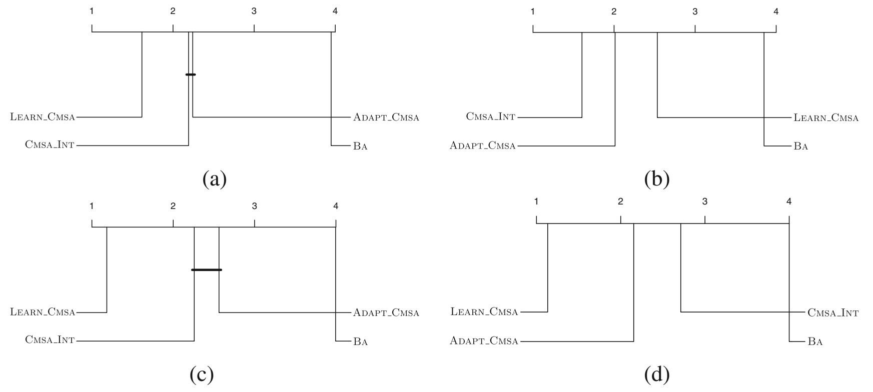


Fig. 3.9 Critical difference (CD) plots concerning FFMS instances with $t = 0.85m$. (**a**) All instances. (**b**) Instances with $m = 100$. (**c**) Instances with $m = 500$. (**d**) Instances with $m = 1000$

Main Observations Concerning the FFMS Results

- Both for thresholds $t = 0.8m$ and $t = 0.85m$, LEARN_CMSA outperforms all other competitors with statistical significance; see Figs. 3.8a and 3.9a. In particular, LEARN_CMSA outperforms both competing CMSA variants: CMSA_INT and ADAPT_CMSA. Moreover, it outperforms the pure bacterial algorithm (BA).
- The relative performance of LEARN_CMSA is better in the context of threshold $t = 0.8m$, where the algorithm can outperform the other approaches with statistical significance even for all three subsets of problem instances of different input string length (see Fig. 3.8b–d). In contrast to this, CMSA_INT and ADAPT_CMSA perform better than LEARN_CMSA for problem instances with threshold $t = 0.85m$ and a short input string length of $m = 100$ (see Fig. 3.9b).

- Even though BA clearly is the worst-performing algorithm in this comparison, its learning component is again—as in the case of the MDS problem—a powerful addition to the standard CMSA algorithm (CMSA_INT).

3.6 Conclusions and Possible Research Directions

Adding a learning component to the solution construction process of CMSA certainly has a very high potential, as shown by the two example applications presented in this chapter. However, there is also a wide range of options for designing such learning components and their interaction with CMSA. The negative aspect of the specific algorithm showcased in this chapter is the dependence on a rather high number of parameters. Moreover, in the proposed approach both CMSA and the learning component (that is, bacterial algorithm) maintain their identity, resulting in a rather low level of integration. In particular, they communicate by mutually feeding their best solutions into the memory mechanism of the other approach. Given the current success of using machine learning (ML) techniques to improve optimization algorithms (see, for example, [1, 7]) it might be possible to design a machine learning component well-integrated into CMSA supporting the goal of shifting the construction of feasible solutions to better parts of the search space over time.

References

1. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. European Journal of Operational Research **290**(2), 405–421 (2021)
2. Contreras A., R., Hernández P., V., Pinacho-Davidson, P., Pinninghoff J., M.A.: A bacteria-based metaheuristic as a tool for group formation. In: International Work-Conference on the Interplay Between Natural and Artificial Computation, pp. 443–451. Springer (2022)
3. Ferone, D., Festa, P., Resende, M.G.: Hybridizations of GRASP with path relinking for the far from most string problem. International Transactions in Operational Research **23**(3), 481–506 (2016)
4. Fleming, A.: On the antibacterial action of cultures of a penicillium, with special reference to their use in the isolation of B. influenzae. British Journal of Experimental Pathology **10**(3), 226 (1929)
5. Odonkor, S.T., Addo, K.K.: Bacteria resistance to antibiotics: Recent trends and challenges. International Journal of Biological & Medical Research **2**(4), 1204–1210 (2011)
6. Pinninghoff J., M.A., Orellana M., J., Contreras A., R.: Bacterial resistance algorithm. an application to CVRP. In: From Bioinspired Systems and Biomedical Applications to Machine Learning: 8th International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2019, Almería, Spain, June 3–7, 2019, Proceedings, Part II 8, pp. 204–211. Springer (2019)
7. Weiner, J., Ernst, A.T., Li, X., Sun, Y.: Ranking constraint relaxations for mixed integer programs using a machine learning approach. EURO Journal on Computational Optimization **11**, 100061 (2023)

Chapter 4
Replacing Hard Mathematical Models with Set Covering Formulations

Abstract Many packing, routing, and knapsack problems can be expressed both in terms of standard assignment-type integer linear programming models and in terms of set-covering-based models. Black-box solvers such as CPLEX and Gurobi find it generally very hard to solve assignment-type mathematical models of these problems. Therefore, the Operations Research community has developed specific exact and heuristic techniques that exploit set-covering-based models. In this chapter, it is shown that integer linear programming models based on set covering can also be very useful for their use within CMSA. In particular, this is shown by applications of CMSA to the Variable-Sized Bin Packing (VSBP) problem and to the Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickups and Deliveries (EVRP-TW-SPD). In both applications, CMSA based on a set covering model significantly outperforms CMSA when using an assignment-type model. Moreover, state-of-the-art results are obtained for both considered optimization problems.

4.1 Introduction

In this chapter, we focus on employing CMSA for combinatorial optimization problems that involve partitioning a finite set of items into distinct subsets. This category encompasses various significant problems, including bin packing problems, multiple knapsack problems, assembly line balancing, and vehicle routing problems, among others. Traditional black-box solvers like CPLEX and Gurobi often face considerable challenges in solving standard assignment-type ILP models for these problems. Consequently, the Operations Research community has devised specialized exact techniques based on set covering models to address these challenges. Set covering models prove particularly beneficial in the realm of column generation methods, as illustrated in works such as [4, 8, 19]. In addition, the transformation of vehicle routing and packing problems to set covering problems has also been exploited in the context of heuristic methods; see, for example, [5, 15, 17]. This chapter will demonstrate the effectiveness of such models within CMSA algorithms, especially in the context of solving sub-instances. In particular, this

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

will be done for two different optimization problems. The first problem, called the Variable-Sized Bin Packing (VSBP) problem is from the bin packing field, while the second one—called Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickups and Deliveries (EVRP-TW-SPD)—is from the field of electric vehicle routing. The substitution of standard assignment-type ILP models with set covering formulations often leads to highly efficient techniques. Notably, these set-covering-based CMSA techniques, unlike column generation methods, are relatively straightforward to implement.

In fact, in the context of both considered problems, our best-performing CMSA variants achieve state-of-the-art results. In the case of the VSBP problem, new best-known solutions are found in 68 out of 150 cases. Furthermore, in the context of the EVRP-TW-SPD problem, no other heuristic optimization method has been proposed so far.

4.2 Application to Variable-Sized Bin Packing

The Variable-Sized Bin Packing (VSBP) problem can be formally characterized as follows: Given is a set $S_{\text{items}} = \{1, \ldots, n\}$ comprising n items, each denoted as $i \in S_{\text{items}}$ with a positive weight w_i, and a set $B = \{1, \ldots, m\}$ consisting of m bin types. Each bin type $k \in B$ is defined by a positive capacity W_k and a cost C_k. Without loss of generality, it is assumed that the capacities satisfy $W_1 < \ldots < W_m$. The objective of the VSBP problem is to efficiently pack the n items into bins, with the aim of minimizing the total cost of the utilized bins. Importantly, there are no restrictions on how many times a bin type may be employed. The VSBP problem can be mathematically modeled as an assignment-type ILP, as detailed in [11]. This model, henceforth denoted by $\texttt{ILP}_{\text{std}}^{\text{VSBP}}$, can be stated as follows.

$$\min \quad \sum_{j=1}^{n} \sum_{k=1}^{m} C_k \cdot y_{jk} \tag{4.1}$$

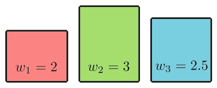
$$\text{subject to} \quad \sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i = 1, \ldots, n \tag{4.2}$$

$$\sum_{k=1}^{m} y_{jk} \leq 1 \quad \text{for } j = 1, \ldots, n \tag{4.3}$$

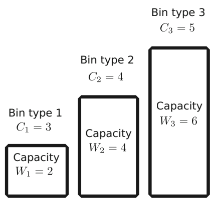
$$\sum_{i=1}^{n} w_i \cdot x_{ij} \leq \sum_{k=1}^{m} W_k \cdot y_{jk} \quad \text{for } j = 1, \ldots, n \tag{4.4}$$

$$x_{ij} \in \{0, 1\} \quad \text{for } i, j = 1, \ldots, n$$

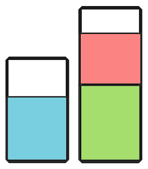
$$y_{jk} \in \{0, 1\} \quad \text{for } j = 1, \ldots, n \text{ and } k = 1, \ldots, m$$



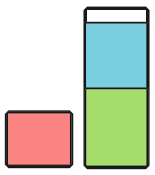
(a) Three items



(b) Three bin types



(c) A valid solution with cost 4 + 5 = 9



(d) Optimal solution with cost 3 + 5 = 8

Fig. 4.1 Illustrating an instance with $n = 3$ items and $m = 3$ distinct bin types: The weights of the items are presented in (**a**), while the capacities and costs of the bin types are outlined in (**b**). In (**c**), a valid solution is depicted, wherein item 3 is allocated to a bin of type 2 (incurring a cost of 4), while items 1 and 2 are assigned to a bin of type 3 (with a cost of 5). Consequently, the solution in (**c**) attains a value of $4 + 5 = 9$. Alternatively, (**d**) showcases the optimal solution, where item 1 is assigned to a bin of type 1 (incurring a cost of 3), and items 2 and 3 are allocated to a bin of type 3 (with a cost of 5). Thus, the cost of the optimal solution is $3 + 5 = 8$

This ILP model employs two sets of binary variables. When x_{ij} is set to 1, it signifies that item i is placed in bin j. Similarly, a setting of $y_{jk} = 1$ indicates that bin j is assigned bin type k. It is worth noting that the number of used bins can be effectively limited to n, which represents the number of items. Constraints (4.2) ensure that each item is assigned to precisely one bin, while constraints (4.3) enforce that each utilized bin is associated with exactly one bin type. Additionally, constraints (4.4) guarantee the adherence to bin capacities. Importantly, the VSBP problem is classified as NP-hard due to its nature as a generalization of the one-dimensional bin packing problem. For an illustrative instance of a small VSBP problem, refer to Fig. 4.1.

4.2.1 Short Literature Review Concerning the VSBP Problem

This short review is directed toward the original version of the VSBP problem, where, as previously indicated, the number of available bins per bin type is unlimited. Lower bounds and heuristic methods for a more generalized version of the VSBP problem, featuring explicit limits on the number of bins per bin type, were derived in [7]. Haouari and Serairi [11] introduced a variety of greedy heuristics and a genetic algorithm for the VSBP problem. Additionally, [12] proposed a sophisticated variable neighborhood search (VNS) algorithm, and the results reported in their work have remained uncontested so far. Instead of improving those results, subsequent research efforts have shifted towards exploring the VSBP problem with additional constraints. Recent papers include the VSBP problem with time windows [10] and the VSBP problem with conflicts [9].

4.2.2 *Set-Covering Based ILP Model of the VSBP Problem*

An alternative ILP model based on set-covering for the VSBP problem can be formulated as follows. Consider $\mathcal{B}$ as the set comprising all potential bins with an assigned set of items. The weight w_b of a bin $b \in \mathcal{B}$ is defined as the total weight of the items assigned to that specific bin. Furthermore, the cost c_b of a bin $b \in \mathcal{B}$ is determined by the cost of the lowest-cost bin type capable of accommodating all items assigned to bin b. Lastly, let $\mathcal{B}_i \subset \mathcal{B}$ represent the set of bins containing item i. Given these definitions, the set-covering-based ILP model for the VSBP problem, hereinafter referred to as $\texttt{ILP}^{\text{VSBP}}_{\text{setcov}}$, can be expressed as follows.

$$\min \quad \sum_{b \in \mathcal{B}} c_b \cdot x_b \tag{4.5}$$

$$\text{subject to} \quad \sum_{b \in \mathcal{B}_i} x_b \geq 1 \quad \text{for } i = 1, \ldots, n \tag{4.6}$$

$$x_b \in \{0, 1\} \quad \text{for all } b \in \mathcal{B}$$

It is worth noting that an exact correspondence between $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$ and $\texttt{ILP}^{\text{VSBP}}_{\text{setcov}}$ could be achieved by substituting the "$\geq$" symbol in constraints (4.6) with the equality symbol ("="). However, any optimal solution derived with the "$\geq$" symbol can readily be transformed into an optimal solution of the model with the "=" symbol by eliminating duplicate items from all bins except one. Additionally, as per [4], the linear programming relaxation of the model using the "$\geq$" symbol is numerically more stable and, consequently, easier to solve. This enhances the feasibility of solving the ILP using solvers such as CPLEX or Gurobi.

4.2.3 *Application of Standard CMSA to the VBSP Problem*

First, we applied the standard CMSA from Sect. 1.3.1 on page 18 to the VSBP problem. As we will use the generic variant of defining the solution components (see below), this algorithm will henceforth be called CMSA_GEN. As ILP model for solving sub-instances, CMSA_GEN uses the one from Sect. 4.2, that is, model

$\texttt{ILP}^{\text{VSBP}}_{\text{std}}$. Concerning the set C of solution components, for each binary variable of the ILP model exactly two solution components are introduced: one component that corresponds to setting the variable to zero, and another component that refers to setting the variable to one. In the case of model $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$, this means that C consists of solution components cx^0_{ij} and cx^1_{ij} for all binary variables x_{ij} $(i, j, = 1, \ldots, n)$, and of solution components cy^0_{jk} and cy^1_{jk} for all binary variables y_{jk} $(j = 1, \ldots, n;\ k = 1, \ldots, m)$. Hereby, cx^0_{ij}, for example, corresponds to $x_{ij} = 0$, while cx^1_{ij} corresponds to $x_{ij} = 1$. Moreover, $C = \{cx^0_{11}, \ldots, cx^0_{nn}, cx^1_{11}, \ldots, cx^1_{nn}, cy^0_{11}, \ldots, cy^0_{nm}, cy^1_{11}, \ldots, cy^1_{nm}\}$ is the complete set of $2n^2 + 2nm$ solution components. Any valid solution S is a subset of C with $|S| = n^2 + nm$ because, for each binary variable, a solution S contains exactly one of the two corresponding solution components.

4.2.3.1 Probabilistic Construction of VSBP Solutions

For the following discussion, a bin $b \subseteq \{1, \ldots, n\}$ is a set of items. Moreover, a bin b is always characterized by three well-defined measures:

1. b_{load}: The load of a bin b is the sum of the weights of the items it contains, expressed as $b_{\text{load}} := \sum_{i \in b} w_i$.
2. b_{type}: The type of a bin b is identified as the lowest-cost bin type capable of accommodating the load of the bin. This is formally defined as $b_{\text{type}} := k$ such that $C_k < C_r$ for all $r \in \{1, \ldots, m\}$ with $W_r \geq b_{\text{load}}$.
3. b_{cost}: The cost of a bin is equivalent to the cost of its type, specified as $b_{\text{cost}} := C_{b_{\text{type}}}$.
4. b_{ratio}: The ratio between the cost and the load of a bin is expressed as $b_{\text{ratio}} := \frac{b_{\text{cost}}}{b_{\text{load}}}$.

In addition, let $\max_{\text{load}}$ be defined as the maximum capacity of all bin types, that is, $\max_{\text{load}} := \max\{W_j \mid j = 1, \ldots, m\}$. For the probabilistic construction of a solution, the following simple procedure is applied; see also Algorithm 4.1. First, the n items are randomly ordered; see line 3. Then, the set of bins (B) is initialized by placing the first item from the list in a new bin, whose load, type, cost, and ratio are determined as defined above. Then, in the pre-determined order, the remaining items are placed into bins. In particular, in probability, among all options to place an item, the one resulting in a bin with a lower ratio is preferred over the others. The placement of an item into a bin is done in function ChooseOption(O, d_{rate}, l_{size}) (see line 16), where O is the current set of options. The working of this function is as follows. First, a number z is chosen uniformly at random from [0, 1]. In case $z \leq d_{\text{rate}}$, the chosen option is the one with the lowest ratio. Otherwise, the $\min\{|O|, l_{\text{size}}\}$ options with the lowest ratios are pre-selected from O, and the chosen option is randomly determined among those. When all items are placed into bins, the set of bins is sorted by bin ratio (from small to large); see function

Algorithm 4.1: Probabilistic construction of a valid VSBP solution

```
1: input: values for solution construction parameters $d_{\text{rate}}$, $l_{\text{size}}$
2: Let $(i_1, \ldots, i_n)$ be a randomly ordered list of all $n$ items
3: $b := \{i_1\}$
4: $B := \{b\}$
5: for $l := 2, \ldots, n$ do
6:    $i := i_l$
7:    $O := \emptyset$
8:    for $b \in B$ do
9:       if $b_{\text{load}} + w_i \leq \max_{\text{load}}$ then
10:         $b^e := b \cup \{i\}$
11:         $O := O \cup \{b^e\}$
12:      end if
13:   end for
14:   if $O$ is non-empty then
15:      $b^e :=$ ChooseOption($O, d_{\text{rate}}, l_{\text{size}}$)
16:      $B := B \setminus \{b\} \cup \{b^e\}$
17:   else
18:      $b := \{i\}$
19:      $B := B \cup \{b\}$
20:   end if
21: end for
22: Sort($B$)
23: $S :=$ ExtractSolutionComponents($B$)
24: output: $S$
```

Sort(B) in line 22 of Algorithm 4.1. As a tie-breaking criterion, we utilized the smallest item index of a bin (preferring smaller ones). Finally, the constructed solution is transformed into the corresponding set S of solution components in function ExtractSolutionComponents(B); line 23. In the case of the set of solution components outlined above, this works as follows. If the first bin (after sorting B) is of type k, then cy^1_{1k} is added to S. Moreover, all cy^0_{1r} (with $r \neq k \in \{1, \ldots, m\}$) are added to S. The same is done for all other bins in B. Similarly, for each item i in the first bin (after sorting B), component cx^1_{i1} is added to S. Moreover, all cx^0_{ir} (with $r = 2, \ldots, n$) are added to S. The same is done for the items of all other bins from B.

4.2.3.2 Sub-instance Generation and Solving

During the *solve* stage of CMSA_GEN, first, a reduced problem instance is generated on the basis of C'. This is accomplished by introducing the following constraints to the $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$ model for all $i = 1, \ldots, n$, before employing an ILP solver:

1. For all $i, j = 1, \ldots, n$:
   - If $cx^0_{ij} \in C'$ and $cx^1_{ij} \notin C'$: add constraint $x_{ij} = 0$ to $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$
   - If $cx^0_{ij} \notin C'$ and $cx^1_{ij} \in C'$: add constraint $x_{ij} = 1$ to $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$

2. For all $j = 1, \ldots, n$ and $k = 1, \ldots, m$:
   - If $cy^0_{jk} \in C'$ and $cy^1_{jk} \notin C'$: add constraint $y_{jk} = 0$ to $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$
   - If $cy^0_{jk} \notin C'$ and $cy^1_{jk} \in C'$: add constraint $y_{jk} = 1$ to $\texttt{ILP}^{\text{VSBP}}_{\text{std}}$

To clarify, if a sub-instance C' exclusively contains one of the two solution components associated with a variable, the variable's value can be fixed accordingly. This implies that as more constraints of this nature are incorporated into the original ILP model, the search space for the ILP solver to navigate in solving the sub-instance becomes progressively reduced.

4.2.4 Application of Set-Covering Based CMSA to the VSBP Problem

In addition to applying CMSA_GEN to the VSBP problem, we also apply a CMSA variant that makes use of the set-covering model $\texttt{ILP}^{\text{VSBP}}_{\text{setcov}}$ from Sect. 4.2.2 for solving sub-instances. This CMSA variant is henceforth labeled CMSA_SETCOV.

Solution Components of CMSA_SETCOV

The entire set of solution components C comprises a component c^b for each valid bin b in $\mathcal{B}$ (refer to Sect. 4.2.2). Formally, this can be expressed as $C := \{c^b \mid b \in \mathcal{B}\}$. Any subset $S \subset C$, wherein each item $i \in \{1, \ldots, n\}$ appears in precisely one bin b such that $c^b \in S$, constitutes a valid solution for the addressed VSBP problem instance.

The probabilistic construction of solutions in CMSA_SETCOV works exactly in the same way as outlined in Sect. 4.2.3.1. The only difference lies in the implementation of function ExtractSolutionComponents(B) that assembles the solution components corresponding to a set of bins B. Here, this function simply adds for each $b \in B$ the corresponding solution component c^b to S.

Sub-instance Solving in CMSA_SETCOV

The ILP model solved in the *solve step* of this CMSA variant is obtained by exchanging $\mathcal{B}$ in model $\texttt{ILP}^{\text{VSBP}}_{\text{setcov}}$ by C', that is, by replacing the set of all possible valid bins with the set of those bins that form part of the current sub-instance C'. The solution S obtained from the ILP solver after t_{ILP} CPU seconds is then checked for duplicate occurrences of items. If this happens, duplicate items are randomly removed from the bins in S until each item appears exactly once in the bins of S.

4.2.5 *Experimental Evaluation*

The experiments were conducted on the IIIA-CSIC in-house cluster, already described in Sect. 1.2.1 on page 12, which consists of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM. For solving the corresponding sub-instances in both variants of CMSA, we employed CPLEX version 22.1 in single-threaded mode.

4.2.5.1 VSBP Problem Instances

In the utilized set of benchmark instances (from [12]), each instance has $m = 7$ bin types. The corresponding bin capacities are defined as $W_1 = 70$, $W_2 = 100$, $W_3 = 130$, $W_4 = 160$, $W_5 = 190$, $W_6 = 220$, and $W_7 = 250$. Additionally, item weights are randomly selected from the range $[1, 250]$. Notably, this benchmark set comprises three distinct classes of instances. Specifically, class B1 features a linear bin cost function $C_i = W_i$ $(i = 1, \dots, 7)$, class B2 is characterized by a concave cost function $C_i = \lceil 10\sqrt{W_i} \rceil$ $(i = 1, \dots, 7)$, and class B3 exhibits a convex cost function $C_i = \lceil 0.1 {W_i}^{3/2} \rceil$ $(i = 1, \dots, 7)$. For each combination of $n \in \{100, 200, 500, 1000, 2000\}$ (number of items) and bin cost function class, there are 10 problem instances, resulting in a total of 150 problem instances. It is important to note that optimal solutions for these instances are unknown.

4.2.5.2 Parameter Tuning

Both CMSA_GEN and CMSA_SETCOV require effective parameter values to work at their best. Specifically, the same eight parameters presented in the left-most column of Table 4.1 must be fine-tuned for both algorithms. The value domains for these parameters are outlined in the second column of the same table. The parameter tuning process was executed using the `irace` tool [14], which is described in Sect. 1.2.1 on page 12. Each variant of the algorithm underwent tuning independently for each instance class, with a budget of 2000 algorithm runs. Additionally, each run was constrained to 150 CPU seconds, in accordance with [12]. The resulting parameter values, as presented in Table 4.1, were utilized for the final experiments. The following parameter settings deserve some attention. First, observe that the computation time allotted to CPLEX for solving sub-instances in CMSA_GEN is very low for instances of classes B1 and B2. This suggests that sub-instance solving is not very useful when using the original ILP model.

Table 4.1 Parameter settings for CMSA_GEN and CMSA_SETCOV concerning the three classes of VSBP problem instances

		Class B1		Class B2		Class B3	
Parameter	Domain	CMSA_GEN	CMSA_SETCOV	CMSA_GEN	CMSA_SETCOV	CMSA_GEN	CMSA_SETCOV
t_{ILP}	(0.3, 30.0)	0.48	15.21	0.36	23.12	7.02	28.07
l_{size}	[1, 10]	1	10	1	5	1	8
d_{rate}	(0.0, 0.99)	0.59	0.7	0.96	0.64	0.89	0.97
n_{a}	[2, 50]	50	13	46	50	50	46
age_{max}	[1, 10]	8	3	3	1	3	1
$\text{cplex}_{\text{emphasis}}$	{true, false}	false	false	true	false	false	false
$\text{cplex}_{\text{warmstart}}$	{true, false}	false	false	false	true	true	false
$\text{cplex}_{\text{abort}}$	{true, false}	true	false	true	false	true	false

Moreover, the abort feature (CPLEX) is not used by CMSA_SETCOV, even though the computation time assigned for each application of CPLEX is rather high (see settings of parameter t_{ILP}). This suggests that sub-instances are quickly solved to optimality in CMSA_SETCOV.

4.2.5.3 Numerical Results

Tables 4.2 (instances of class B1), 4.3 (instances of class B2), and 4.4 (instances of class B3) present the best-known solutions from the literature (extracted from [12]), alongside the results obtained using CMSA_GEN and CMSA_SETCOV. The first two columns in these tables denote the number of items (n) and the instance number (#), respectively. Each table row corresponds to a specific problem instance, and the results of the three algorithms are provided in terms of the best solution identified across 10 runs, the average of the best solutions from these 10 runs, and the average times at which these solutions were discovered within the 150 CPU seconds time limit per run. Note that results are indicated in bold font if they correspond to the value of best-known solutions. Moreover, a gray background means that the corresponding best-known solution was improved.

Table 4.2 Results for the 50 instances of class B1 (linear cost function)

		Best	CMSA_GEN			CMSA_SETCOV		
n	#	known	Best	Avg	Avg. time	Best	Avg	Avg. time
100	1	12,700	12,760	12,776.0	44.82	**12,700**	12,700.0	0.12
	2	12,140	12,210	12,232.0	44.11	**12,140**	12,140.0	4.06
	3	13,620	13,670	13,686.0	49.34	**13,620**	13,620.0	0.11
	4	12,550	12,620	12,632.0	74.98	**12,550**	12,550.0	0.09
	5	10,630	10,690	10,706.0	66.26	**10,630**	10,630.0	0.59
	6	11,130	11,190	11,207.0	67.30	**11,130**	11,130.0	12.89
	7	13,020	13,080	13,086.0	64.28	**13,020**	13,020.0	0.06
	8	12,180	12,260	12,269.0	63.62	**12,170**	12,170.0	6.25
	9	11,090	11,180	11,187.0	41.38	**11,090**	11,090.0	1.09
	10	12,800	12,870	12,879.0	57.16	**12,800**	12,800.0	0.11
200	1	25,430	25,640	25,659.0	76.60	**25,430**	25,430.0	1.12
	2	26,300	26,400	26,439.0	70.39	**26,300**	26,300.0	0.08
	3	27,770	**27,770**	27,790.0	58.89	**27,770**	27,770.0	0.03
	4	24,300	24,490	24,507.0	70.86	**24,290**	24,290.0	5.88
	5	25,820	25,940	25,968.0	59.44	**25,820**	25,820.0	0.03
	6	23,820	23,980	24,007.0	60.76	**23,810**	23,810.0	1.05
	7	28,590	28,600	28,624.0	55.83	**28,590**	28,590.0	0.03
	8	25,900	26,040	26,067.0	83.77	**25,900**	25,900.0	0.21
	9	24,890	25,070	25,098.0	48.45	**24,890**	24,890.0	0.22
	10	25,760	25,850	25,868.0	79.21	**25,760**	25,760.0	0.05
500	1	61,770	62,350	62,424.0	95.47	**61,750**	61,758.0	20.56
	2	62,090	62,560	62,628.0	59.03	**62,070**	62,070.0	11.65
	3	66,770	67,320	67,371.0	60.61	**66,760**	66,763.0	31.65
	4	63,970	64,360	64,410.0	75.02	**63,970**	63,970.0	2.32
	5	62,150	62,670	62,698.0	80.46	**62,150**	62,150.0	0.44
	6	61,130	61,670	61,702.0	51.70	**61,090**	61,090.0	24.31
	7	63,340	63,930	63,991.0	64.20	**63,320**	63,320.0	2.14
	8	63,250	63,760	63,809.0	97.81	**63,210**	63,210.0	4.72
	9	61,170	61,740	61,777.0	78.08	**61,120**	61,120.0	36.41
	10	62,000	62,540	62,562.0	69.21	**61,990**	61,990.0	12.14
1000	1	126,610	127,620	127,674.0	73.63	**126,490**	126,496.0	52.58
	2	123,250	124,370	124,466.0	69.56	**123,120**	123,120.0	13.62
	3	123,070	124,320	124,390.0	54.98	**123,020**	123,029.0	36.97
	4	127,370	128,510	128,570.0	66.89	**127,360**	127,360.0	24.74
	5	127,710	128,990	129,036.0	74.28	**127,660**	127,660.0	23.94
	6	125,580	126,640	126,766.0	52.26	**125,520**	125,522.0	56.93
	7	128,260	129,290	129,380.0	62.90	**128,260**	128,260.0	1.49
	8	130,410	131,450	131,513.0	65.99	**130,410**	130,410.0	5.11
	9	125,680	126,910	126,970.0	68.94	**125,630**	125,635.0	28.76
	10	129,400	130,380	130,480.0	62.26	**129,380**	129,380.0	7.77

(continued)

Table 4.2 (continued)

		Best	CMSA_GEN			CMSA_SETCOV		
n	#	known	Best	Avg	Avg. time	Best	Avg	Avg. time
2000	1	254,330	256,380	256,455.0	59.79	**254,290**	254,290.0	44.52
	2	257,370	259,590	259,723.0	73.83	**257,330**	257,330.0	47.29
	3	251,880	254,100	254,180.0	80.85	**251,770**	251,770.0	57.39
	4	248,520	250,610	250,653.0	68.64	**248,470**	248,470.0	26.36
	5	245,110	247,810	247,905.0	46.48	**245,060**	245,066.0	88.49
	6	250,930	253,500	253,571.0	69.33	**250,870**	250,878.0	40.58
	7	258,700	261,140	261,198.0	53.64	**258,680**	258,690.0	62.45
	8	256,950	259,330	259,412.0	49.21	256,970	256,978.0	71.54
	9	258,480	260,720	260,866.0	56.24	**258,450**	258,458.0	66.39
	10	255,750	257,670	257,802.0	71.50	**255,750**	255,750.0	6.92

Main Observations Concerning the VSBP Problem Results

- First, CMSA_SETCOV clearly outperforms CMSA_GEN. Only in two out of 150 cases, CMSA_GEN is able to find a solution of the same quality as the one found by CMSA_SETCOV. Moreover, while CMSA_SETCOV finds the best-known solutions for the small problem instances with $n \in \{100, 200\}$ items, it clearly outperforms the current state of the art in the context of larger problem instances (especially for $n \in \{1000, 2000\}$) as indicated by the number of new best-known solutions.
- For 68 out of 150 problem instances, CMSA_SETCOV successfully discovers new best-known solutions. Specifically, it identifies 27 new best-known solutions for the 50 B1 instances, 26 new solutions for the 50 B2 instances, and 15 new solutions for the 50 B3 instances.
- Only in 7 out of 150 cases, the best solution found by CMSA_SETCOV is slightly worse than the best-known solution.

In order to test the statistical relevance of these results, so-called CD plots were produced; see Sect. 1.2.3 on page 16 for a description of CD plots. Note that, in the context of these plots, we added the results of the VNS approach from [12] and of the GA approach from [7]. The plot from Fig. 4.2a shows that—from a global point of view—CMSA_SETCOV outperforms all other algorithms with statistical significance. When considering the instances from the three classes separately, no statistically significant difference between CMSA_SETCOV and VNS can be detected in the context of the B3 class (see Fig. 4.2d).

Table 4.3 Results for the 50 instances of class B2 (concave cost function)

		Best	CMSA_GEN			CMSA_SETCOV		
n	#	known	Best	Avg	Avg. time	Best	Avg	Avg. time
100	1	8890	8920	8923.9	63.88	**8890**	8890.0	0.04
	2	7832	7864	7870.9	72.02	**7832**	7832.0	2.53
	3	8516	**8516**	8545.1	97.94	**8516**	8516.0	0.04
	4	8591	8611	8619.5	78.30	**8591**	8591.0	0.02
	5	8474	8496	8502.0	68.44	**8474**	8474.0	0.28
	6	7538	7571	7578.0	53.49	**7538**	7538.0	3.59
	7	7876	7890	7898.0	47.32	**7876**	7876.0	0.03
	8	8116	8148	8158.9	58.86	**8116**	8116.0	0.70
	9	8392	8409	8414.9	77.77	**8392**	8392.0	0.43
	10	9127	9137	9139.0	53.29	**9127**	9127.0	0.03
200	1	17,307	17,445	17,453.1	80.09	**17,307**	17,307.0	4.53
	2	16,391	16,533	16,562.1	84.47	**16,391**	16,394.3	62.26
	3	16,637	16,687	16,708.7	63.75	**16,629**	16,629.8	38.82
	4	15,864	15,925	15,953.5	77.29	**15,864**	15,864.0	0.30
	5	17,699	17,729	17,735.5	84.51	**17,699**	17,699.0	0.04
	6	15,457	15,541	15,572.4	73.06	**15,457**	15,457.0	3.03
	7	16,203	16,329	16,343.8	85.28	**16,203**	16,203.0	0.08
	8	15,353	15,490	15,503.7	82.03	**15,353**	15,353.0	2.49
	9	15,860	16,062	16,072.4	78.29	**15,860**	15,860.0	0.11
	10	15,292	15,437	15,447.9	81.29	**15,292**	15,292.0	0.14
500	1	39,307	39,555	39,609.4	81.73	**39,305**	39,305.6	46.36
	2	40,767	41,027	41,085.9	74.65	**40,765**	40,765.0	9.64
	3	39,963	40,273	40,312.7	79.02	**39,963**	39,963.0	2.19
	4	38,945	39,391	39,439.8	76.98	**38,934**	38,934.7	32.24
	5	39,785	40,122	40,166.6	81.12	**39,775**	39,775.0	1.70
	6	43,096	43,213	43,288.1	99.02	**43,096**	43,096.0	10.50
	7	41,307	41,541	41,574.1	78.85	**41,306**	41,306.0	7.00
	8	39,756	40,049	40,080.4	70.89	**39,738**	39,741.4	49.30
	9	40,166	40,424	40,480.0	73.82	**40,154**	40,154.0	26.21
	10	41,046	41,324	41,352.8	63.15	**41,046**	41,046.0	0.28
1000	1	81,458	81,899	81,988.2	90.45	**81,447**	81,447.0	0.82
	2	78,523	79,309	79,380.2	65.58	**78,515**	78,518.5	100.57
	3	81,544	82,216	82,319.7	65.93	**81,525**	81,528.5	101.22
	4	80,265	80,813	80,920.6	65.70	**80,255**	80,259.4	48.64
	5	81,076	81,795	81,844.5	83.11	**81,066**	81,070.0	55.43
	6	81,333	81,879	82,000.9	89.12	81,343	81,348.0	44.53
	7	81,200	81,555	81,715.0	90.93	**81,199**	81,199.0	2.76
	8	80,899	81,658	81,747.2	106.88	**80,849**	80,856.2	28.34
	9	78,381	79,037	79,160.4	67.42	**78,365**	78,374.5	106.69
	10	84,535	84,853	84,931.2	56.74	**84,503**	84,504.8	71.25

(continued)

Table 4.3 (continued)

n	#	Best known	CMSA_GEN Best	CMSA_GEN Avg	CMSA_GEN Avg. time	CMSA_SETCOV Best	CMSA_SETCOV Avg	CMSA_SETCOV Avg. time
2000	1	160,446	161,725	161,903.4	92.55	**160,287**	160,290.5	107.32
	2	162,193	163,211	163,370.1	62.43	**162,193**	162,197.0	46.56
	3	161,879	163,086	163,230.0	91.45	**161,857**	161,861.3	93.70
	4	161,128	161,945	162,080.0	62.24	**161,064**	161,064.7	46.03
	5	164,625	165,530	165,647.8	74.98	**164,605**	164,612.9	95.17
	6	159,107	160,338	160,419.9	70.71	**159,096**	159,096.0	40.79
	7	162,445	163,422	163,544.7	88.91	**162,391**	162,391.0	12.01
	8	159,878	161,171	161,238.5	81.00	**159,869**	159,878.2	47.41
	9	161,694	162,638	162,748.0	72.75	**161,683**	161,683.0	45.61
	10	153,403	154,768	154,908.7	71.44	**153,266**	153,267.0	57.66

4.2.5.4 Performance Difference Between the Two VSBP ILP Models

Finally, we aim to show why CMSA_SETCOV outperforms CMSA_GEN so clearly. For this purpose, we generate sub-instances of different sizes, translate them both into models $\texttt{ILP}_{\text{std}}^{\text{VSBP}}$ and $\texttt{ILP}_{\text{setcov}}^{\text{VSBP}}$, and solve them with CPLEX. In particular, we generated sub-instances by probabilistically constructing $n_a \in \{2, 5, 10, 20, 50\}$ solutions and by merging their solution components. This was done for the first B1 instance with $n \in \{100, 200, 500, 1000, 2000\}$ items. Figures 4.3, 4.4, 4.5, 4.6, and 4.7 show radar charts that present the obtained results in the five different cases. Each radar chart provides four different measures, averaged over 10 runs:

1. The number of variables in the models of the sub-instances (top).
2. The relative MIP gap after the termination of CPLEX (right).
3. The computation time required by CPLEX (bottom).
4. The absolute improvement when comparing the result of solving the sub-instance with the best individual solution that was used to generate the sub-instance.

It is important to mention that the time limit for CPLEX was consistently set to 20 CPU seconds for solving these sub-instances. Within this context, a model is considered promising when there is a substantial improvement (left), and the number of variables (top), the relative MIP gap (right), and the required time (bottom) are all low. The presented radar plots affirm that this holds true for the $\texttt{ILP}_{\text{setcov}}^{\text{VSBP}}$ model, while conversely, the situation is reversed for the $\texttt{ILP}_{\text{std}}^{\text{VSBP}}$ model. It is evident that this trend becomes more accentuated as the size of the problem instances increases.

Table 4.4 Results for the 50 instances of class B3 (convex cost function)

n	#	Best	CMSA_GEN			CMSA_SETCOV		
		known	Best	Avg	Avg. time	Best	Avg	Avg. time
100	1	19,364	19,393	19,414.8	79.77	**19,364**	19,364.0	0.02
	2	19,000	19,016	19,025.1	32.03	**19,000**	19,000.0	0.03
	3	18,272	18,280	18,298.5	41.17	**18,272**	18,272.0	0.01
	4	19,016	19,029	19,054.3	48.91	**19,016**	19,016.0	0.02
	5	16,612	16,648	16,653.2	71.84	**16,612**	16,612.0	0.04
	6	18,632	18,649	18,662.9	60.51	**18,632**	18,632.0	0.06
	7	18,682	18,708	18,726.0	92.06	**18,682**	18,682.0	0.04
	8	19,517	19,520	19,540.0	78.12	**19,517**	19,517.0	0.02
	9	17,950	17,968	17,983.9	92.37	**17,950**	17,950.0	0.03
	10	17,127	17,135	17,150.3	59.36	**17,127**	17,127.0	0.02
200	1	35,423	35,600	35,636.6	56.79	**35,423**	35,423.0	0.79
	2	36,362	36,638	36,699.6	70.19	**36,362**	36,362.0	6.80
	3	33,390	33,620	33,648.2	81.67	**33,390**	33,390.0	0.30
	4	34,327	34,529	34,572.1	76.59	**34,327**	34,327.0	0.13
	5	38,055	38,231	38,265.5	76.52	**38,055**	38,055.0	0.09
	6	35,009	35,194	35,228.9	42.86	**35,009**	35,009.0	0.10
	7	38,175	38,313	38,353.4	69.01	**38,175**	38,175.0	0.47
	8	36,003	36,154	36,180.4	80.27	**36,003**	36,003.0	0.09
	9	32,700	32,883	32,922.5	61.19	**32,700**	32,700.0	1.75
	10	36,998	37,124	37,236.7	57.45	**36,998**	36,998.0	0.18
500	1	94,768	95,293	95,363.9	114.88	**94,768**	94,768.0	1.11
	2	97,983	98,455	98,506.4	71.85	**97,983**	97,983.0	0.84
	3	95,832	96,365	96,537.1	90.70	**95,832**	95,832.0	1.22
	4	91,068	91,598	91,723.3	96.05	**91,068**	91,068.0	0.45
	5	87,676	88,479	88,509.7	85.86	**87,676**	87,676.0	0.63
	6	83,124	83,926	84,065.9	63.61	**83,124**	83,124.0	21.24
	7	90,407	91,061	91,124.8	32.83	**90,407**	90,407.0	0.97
	8	87,059	87,844	87,883.4	57.71	**87,059**	87,059.0	7.37
	9	87,398	88,012	88,148.4	105.40	**87,398**	87,398.0	2.50
	10	90,541	91,097	91,226.8	86.41	90,543	90,543.0	23.88
1000	1	176,950	178,524	178,603.4	64.64	176,953	176,955.3	98.43
	2	180,993	182,387	182,477.0	73.19	**180,989**	180,990.6	73.80
	3	182,758	184,448	184,540.3	52.71	**182,754**	182,754.6	58.44
	4	180,859	182,523	182,593.6	71.58	**180,857**	180,859.3	58.48
	5	179,158	180,722	180,766.4	98.23	**179,154**	179,154.8	61.79
	6	188,838	190,405	190,462.0	74.14	188,839	188,840.2	34.81
	7	178,185	179,873	179,929.9	69.07	**178,183**	178,189.7	89.80
	8	177,461	179,120	179,188.0	88.13	**177,459**	177,459.0	14.70
	9	181,005	182,688	182,735.9	67.75	**181,001**	181,007.7	69.93
	10	176,902	178,608	178,668.8	90.00	**176,902**	176,917.2	49.65

(continued)

Table 4.4 (continued)

n	#	Best known	CMSA_GEN Best	CMSA_GEN Avg	CMSA_GEN Avg. time	CMSA_SETCOV Best	CMSA_SETCOV Avg	CMSA_SETCOV Avg. time
2000	1	356,244	360,181	360,299.1	68.42	**356,235**	356,236.6	85.25
	2	369,839	373,372	373,580.8	57.24	**369,811**	369,831.7	68.47
	3	364,550	368,194	368,358.8	80.45	**364,527**	364,562.3	72.10
	4	356,984	360,844	360,947.8	88.17	**356,958**	356,969.8	102.76
	5	365,557	369,038	369,166.6	79.64	365,564	365,576.4	106.84
	6	365,142	368,915	369,041.4	78.61	**365,116**	365,120.8	56.37
	7	360,824	364,687	364,771.3	67.67	**360,816**	360,838.2	90.21
	8	371,799	375,086	375,321.6	64.33	**371,779**	371,798.6	96.17
	9	355,723	359,485	359,606.1	73.22	355,726	355,745.3	104.49
	10	357,058	361,201	361,305.5	75.07	**357,036**	357,040.1	68.10

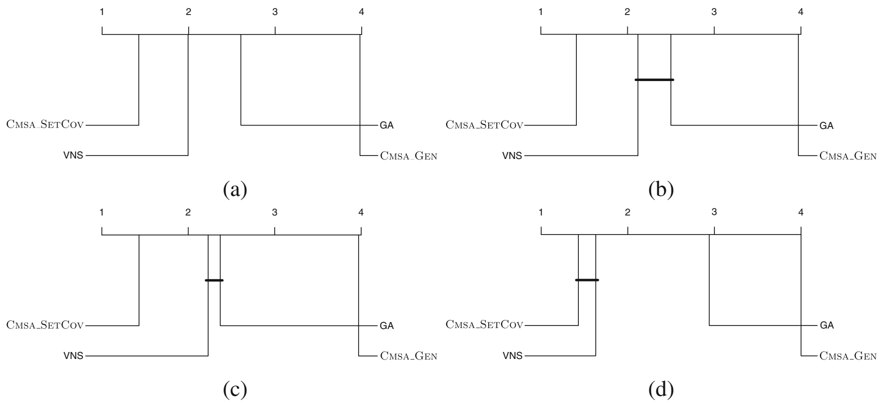


Fig. 4.2 Critical difference (CD) plots showing the statistical significance of the VSBP problem results. (**a**) All 150 problem instances. (**b**) 50 B1 instances. (**c**) 50 B2 instances. (**d**) 50 B3 instances

4.2.5.5 STNWeb Graphics Concerning the VSBP Results

As in the case of all previous experimental evaluations presented in this book, we plotted STNWeb graphics of the obtained VSBP results; see Sect. 1.2.2 on page 13 for a description of the STNWeb tool and the type of graphics that are produced. Figure 4.8 shows the typical case of the first problem instance with 500 items from the B1 class. The complete STN in Fig. 4.8a indicates that all 10 runs of CMSA_GEN reach solutions of the same quality, the best ones found between CMSA_GEN and CMSA_SETCOV. However, all 10 solutions are different to each other. The STN after search space partitioning (see Fig. 4.8b) shows that all of these best solutions are very much related to each other. Observe that they all can be found in the same area of the search space. We assume that good solutions of the same quality often have only small differences with each other. The STN after

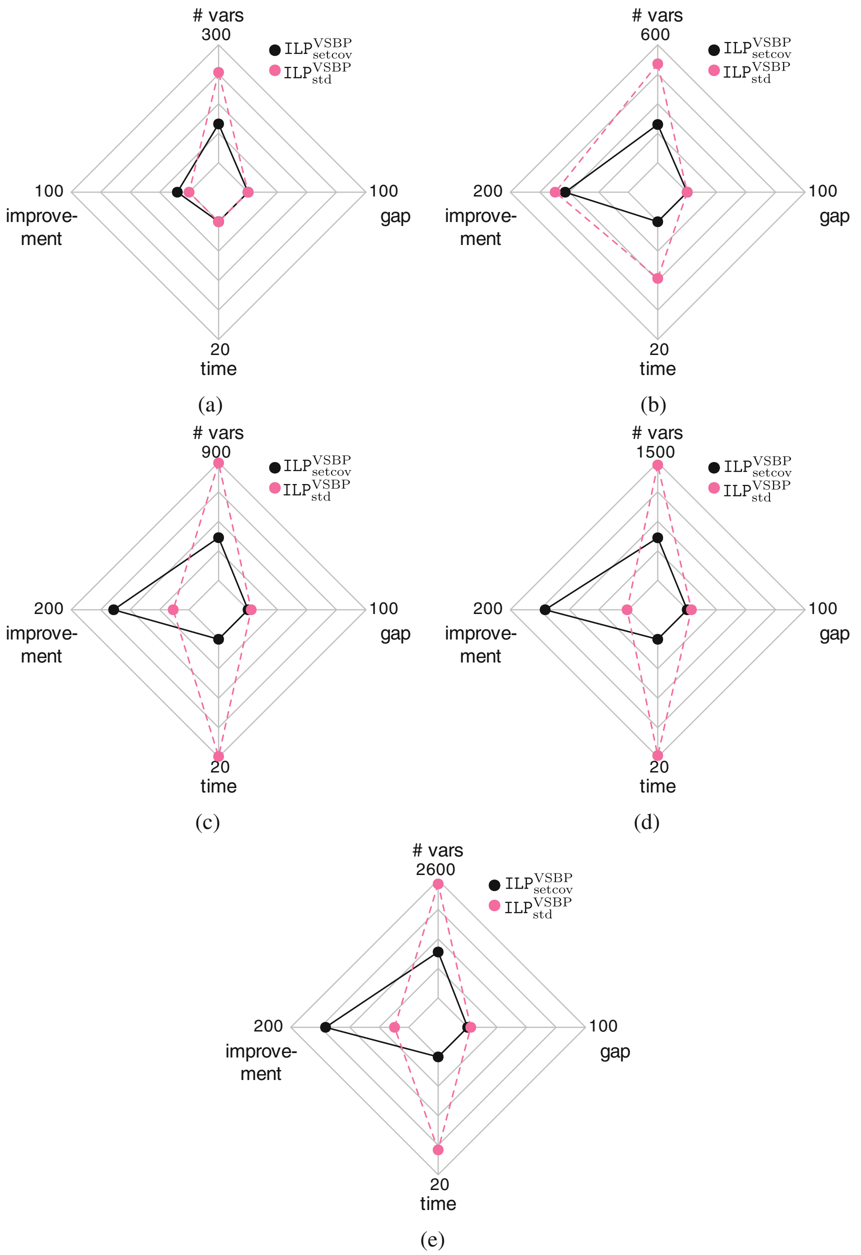


Fig. 4.3 Radar charts concerning the comparison of the two ILP models applied to a VSBP problem instance (B1 class) with 100 items. (**a**) Two solution constructions. (**b**) Five solution constructions. (**c**) 10 solution constructions. (**d**) 20 solution constructions. (**e**) 50 solution constructions

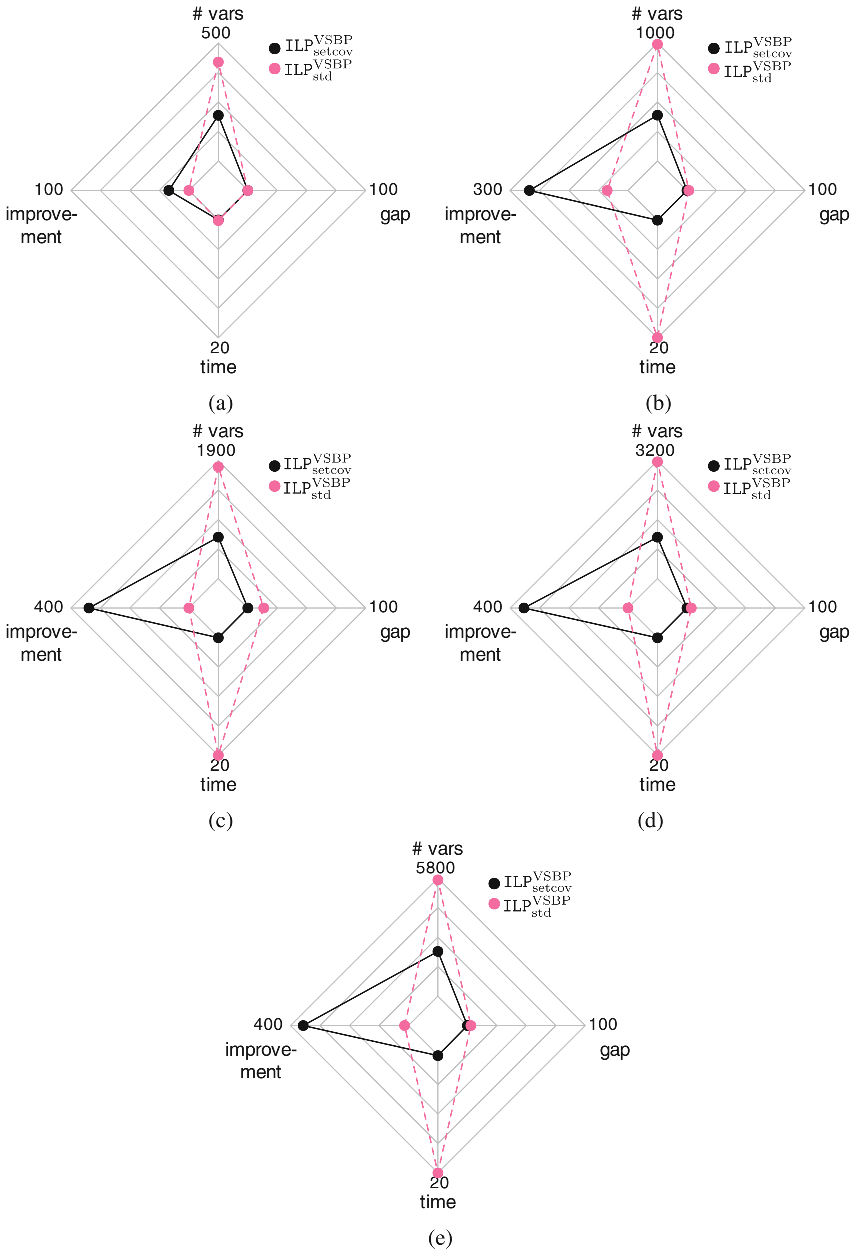


Fig. 4.4 Radar charts concerning the comparison of the two ILP models applied to a VSBP problem instance (B1 class) with 200 items. (**a**) Two solution constructions. (**b**) Five solution constructions. (**c**) 10 solution constructions. (**d**) 20 solution constructions. (**e**) 50 solution constructions

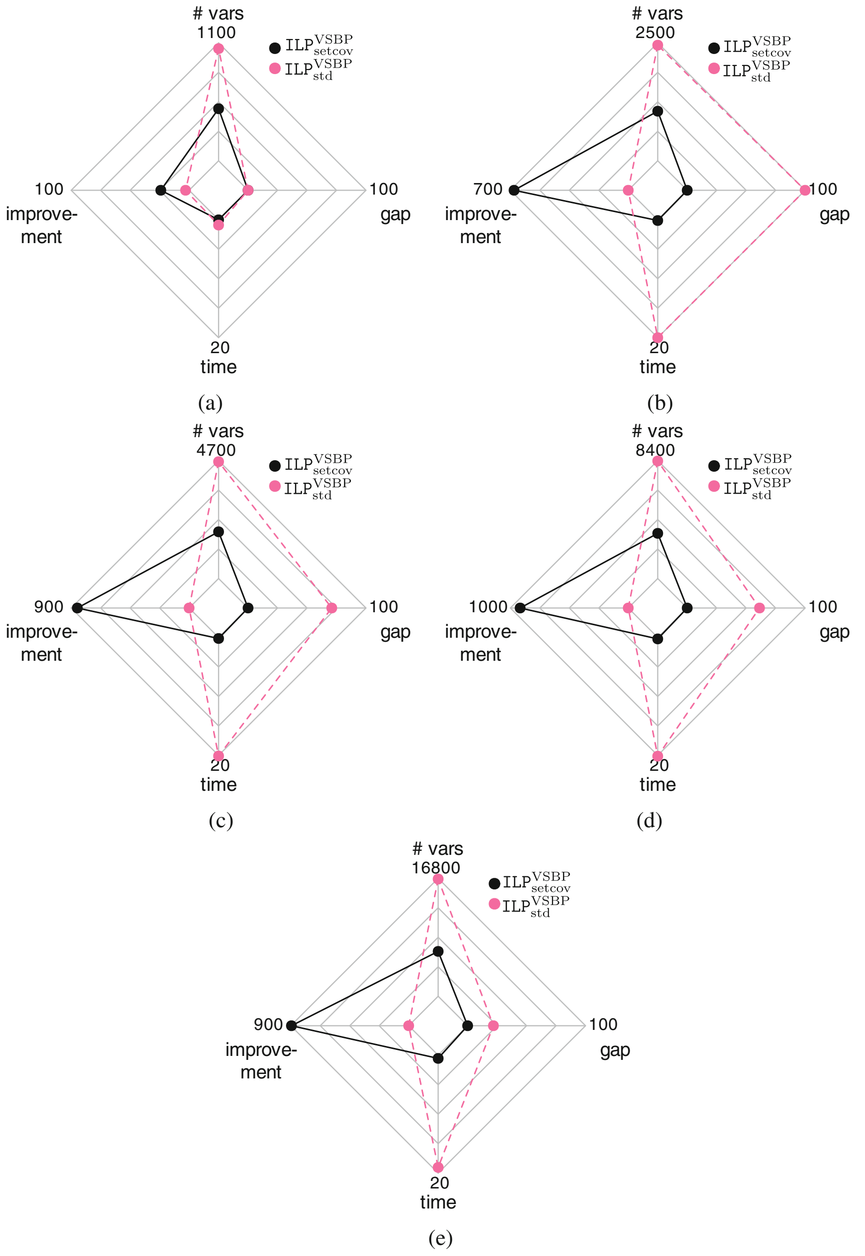
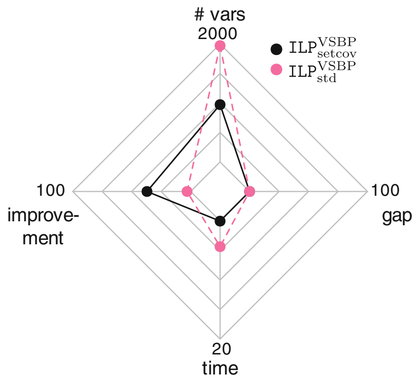
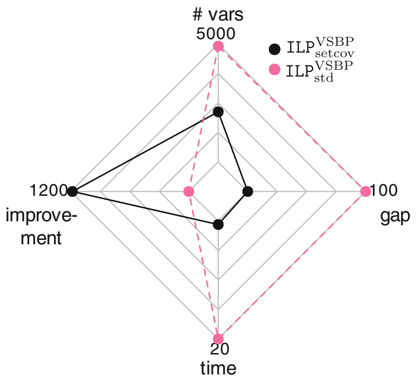


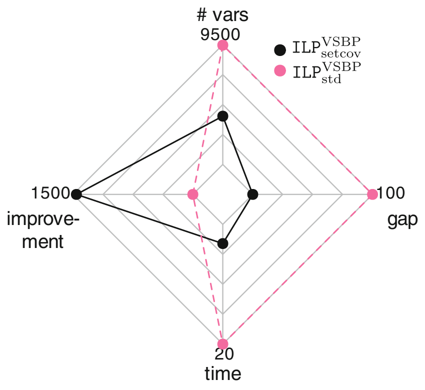
Fig. 4.5 Radar charts concerning the comparison of the two ILP models applied to a VSBP problem instance (B1 class) with 500 items. (**a**) Two solution constructions. (**b**) Five solution constructions. (**c**) 10 solution constructions. (**d**) 20 solution constructions. (**e**) 50 solution constructions



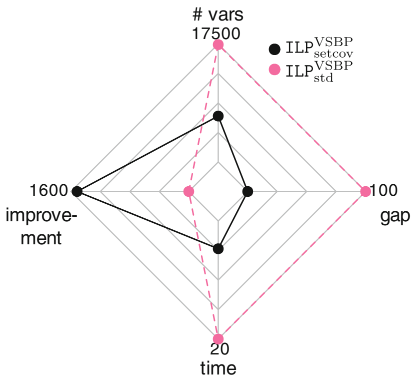
(a)



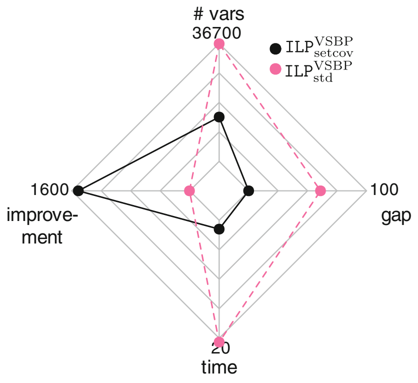
(b)



(c)



(d)



(e)

Fig. 4.6 Radar charts concerning the comparison of the two ILP models applied to a VSBP problem instance (B1 class) with 1000 items. (**a**) Two solution constructions. (**b**) Five solution constructions. (**c**) 10 solution constructions. (**d**) 20 solution constructions. (**e**) 50 solution constructions

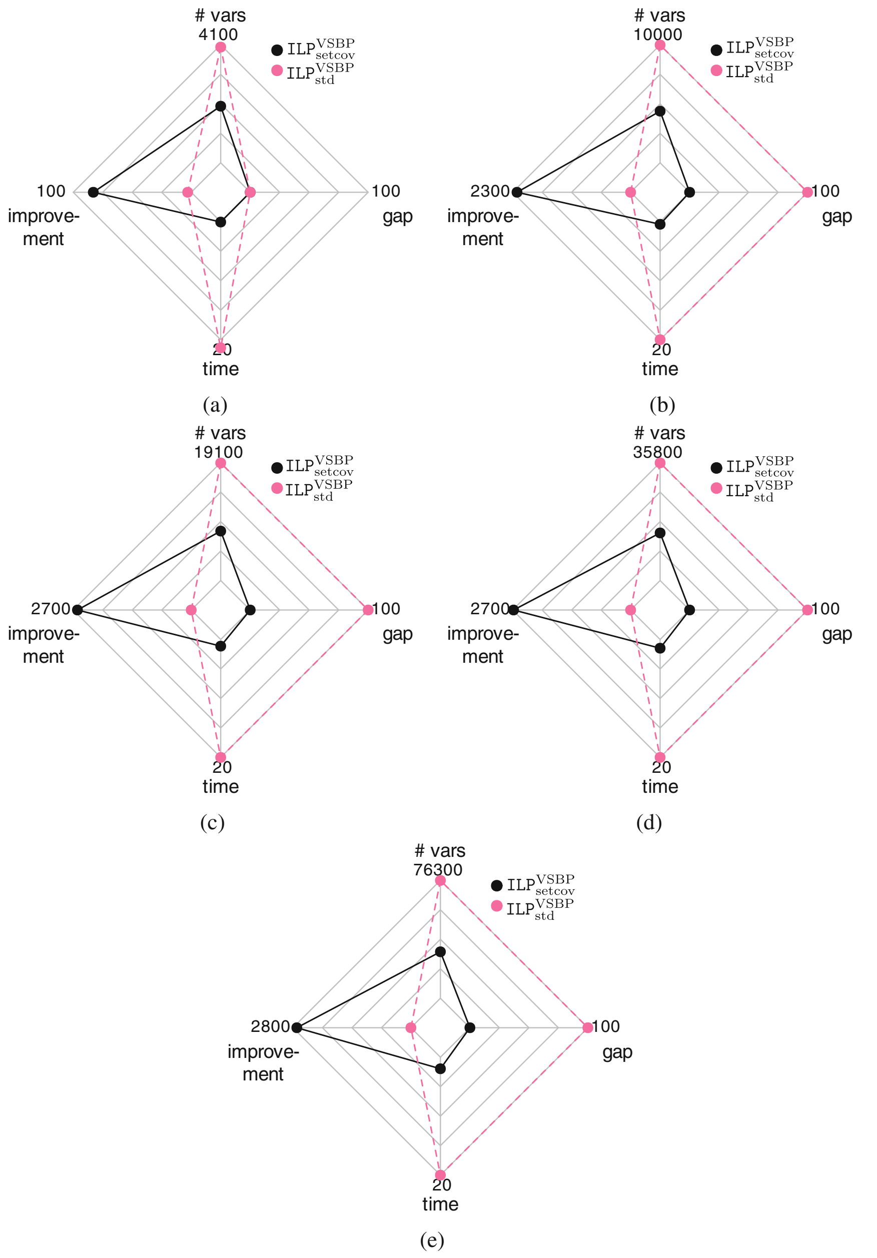


Fig. 4.7 Radar charts concerning the comparison of the two ILP models applied to a VSBP problem instance (B1 class) with 2000 items. (**a**) Two solution constructions. (**b**) Five solution constructions. (**c**) 10 solution constructions. (**d**) 20 solution constructions. (**e**) 50 solution constructions

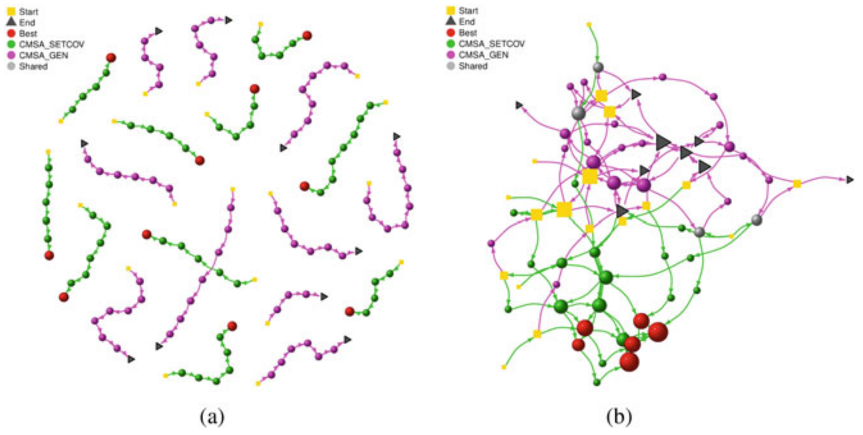


Fig. 4.8 STNWeb graphics. (**a**) and (**b**) show 10 runs of CMSA_GEN and CMSA_SETCOV for the first problem instance with 500 items from the B1 class. While (**a**) shows the complete STN, (**b**) shows the same STN after partitioning

search space partitioning also shows that at the beginning of the search trajectories of CMSA_GEN and CMSA_SETCOV there are overlaps. However, the trajectories of CMSA_GEN simply stop much earlier than the ones of CMSA_SETCOV, because the algorithm is not able to find any better solutions within the given CPU time.

4.3 Application to an Electric Vehicle Routing Problem

As mentioned earlier, another extensive category of problems where standard ILP models can be substituted with set-covering-based ILP models is vehicle routing. In recent years, due to increasing environmental concerns, numerous researchers have directed their attention to vehicle routing problems (VRPs) that involve electric vehicles, commonly known as electric vehicle routing problems (EVRPs). In this section, we aim to provide a second example showcasing the advantages of employing set-covering-based ILP models within CMSA algorithms. Specifically, we explore the Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickups and Deliveries (EVRP-TW-SPD). We will apply an enhanced ADAPT_CMSA algorithm (see also Sect. 2.1 on page 42) in two variants: The first one uses a standard assignment-type ILP model for solving sub-instances, while the second one makes use of a set-covering formulation of the problem.

The standard assignment-type ILP model for the EVRP-TW-SPD builds upon the model designed for the EVRP-TW-PR problem, as proposed in [13]. Notably, the EVRP-TW-PR model is itself a modified version of the model proposed earlier for the EVRP-TW problem in [22]. In the case of the EVRP-TW-SPD, we extend these models further to incorporate SPD constraints. When addressing SPD constraints within the context of vehicle routing problems, it is crucial to recognize that each customer's demand may encompass two distinct requirements: (1) delivering goods to the demand point, referred to as the "delivery demand", and (2) collecting goods from the demand point, termed the "pickup demand". Fulfilling both demands concurrently is essential when a vehicle visits a particular customer.

To align with the conventions established in the existing literature, we adhere to their notation. Specifically, the EVRP-TW-SPD problem entails a set of n customers, denoted as $V = \{1, \dots, n\}$, and a set of charging stations designated as F. To accommodate multiple visits to any charging station, we introduce a set $F^{'}$ that encompasses multiple instances of each charging station from F. The depot is represented by nodes 0 and $n + 1$, where node 0 serves as the starting point and node $n + 1$ will function as the endpoint for each route. It is essential to note that both 0 and $n + 1$ refer to the same, single depot. The set $V^{'} = V \cup F^{'}$ contains all customers and dummy charging stations, with the subscripts 0, $n + 1$, or both indicating the inclusion of the respective instances of the depot. Utilizing these notations, we define the following sets:

1. $F_0^{'} := F^{'} \cup \{0\}$
2. $V_0^{'} := V^{'} \cup \{0\}$
3. $V_{n+1}^{'} := V^{'} \cup \{n + 1\}$
4. $V_{0,n+1}^{'} := V^{'} \cup \{0\} \cup \{n + 1\}$

In accordance with established sets and notations, the EVRP-TW-SPD is defined on a complete, directed graph $G(V_{0,n+1}^{'}, A)$. Set $A = \{(i, j) | i, j \in V_{0,n+1}^{'}, i \neq j\}$ contains all possible arcs. Each arc (i, j) is characterized by a corresponding distance d_{ij} and travel time t_{ij}. The energy consumed per unit distance traveled by an electric vehicle (EV) is represented by a constant h. A fleet of electric vehicles, all possessing identical loading capacity EV_{cap} and battery capacity B_{cap}, is stationed at a depot to meet the simultaneous delivery demand $q_i > 0$ and pickup demand $p_i > 0$ of each $i \in V$. Each vertex $i \in V_{0,n+1}^{'}$ is permitted to be visited only within a designated time window $[e_i, l_i]$, indicating the earliest and latest possible visiting times. Additionally, each customer $i \in V$ has a service time s_i, representing the duration an electric vehicle spends at a customer location. When an EV visits a charging station, its battery undergoes charging at a constant rate of $g > 0$.

The ILP model for the problem incorporates the following decision variables. The binary variable x_{ij} is assigned a value of 1 if the arc a_{ij} is part of a vehicle's route and 0 otherwise. The initiation time of service for each customer visited by the electric vehicle is stored by the decision variable τ_i. Additionally, to keep tabs on the state of charge of the battery upon arrival and departure at each vertex $i \in V_{0,n+1}^{'}$, the decision variables y_i and Y_i are utilized, respectively. Furthermore, the variables u_{ij} and v_{ij} represent the remaining cargo to be delivered to the customers along the route and the amount of cargo already collected (picked up) at previously visited customers, respectively. The technical description of the ILP model, denoted as $\texttt{ILP}_{\text{std}}^{\text{EVRP}}$, is as follows.

$$\min \quad \sum_{i \in V_0^{'}, j \in V_{n+1}^{'}} d_{ij} x_{ij} + \sum_{j \in V_{n+1}^{'}} M x_{0j} \tag{4.7}$$

$$\text{subj. to} \quad \sum_{j \in V_{n+1}^{'}, i \neq j} x_{ij} = 1 \quad \forall i \in V \tag{4.8}$$

$$\sum_{j \in V_{n+1}^{'}, i \neq j} x_{ij} \leq 1 \quad \forall i \in F^{'} \tag{4.9}$$

$$\sum_{i \in V_0^{'}, i \neq j} x_{ij} - \sum_{i \in V_{n+1}^{'}, i \neq j} x_{ji} = 0 \quad \forall j \in V^{'} \tag{4.10}$$

$$\tau_i + (t_{ij} + s_i) x_{ij} - l_0 (1 - x_{ij}) \leq \tau_j \quad \forall i \in V_0, j \in V_{n+1}^{'}, i \neq j \tag{4.11}$$

$$\tau_i + t_{ij} x_{ij} + g(Y_i - y_i) - (l_0 + g B_{\text{cap}})(1 - x_{ij}) \leq \tau_j \quad \forall i \in F^{'}, j \in V_{n+1}^{'}, i \neq j \tag{4.12}$$

$$e_j \leq \tau_j \leq l_j \quad \forall j \in V_{0,n+1}^{'} \tag{4.13}$$

$$0 \leq u_{0j} \leq EV_{\text{cap}} \quad \forall j \in V_{n+1}^{'} \tag{4.14}$$

$$v_{0j} = 0 \quad \forall j \in V_{n+1}^{'} \tag{4.15}$$

$$\sum_{i \in V_0^{'}, i \neq j} u_{ij} - \sum_{i \in V_{n+1}^{'}, i \neq j} u_{ji} = q_j \quad \forall j \in V^{'} \tag{4.16}$$

$$\sum_{i \in V_{n+1}^{'}, i \neq j} v_{ji} - \sum_{i \in V_0^{'}, i \neq j} v_{ij} = p_j \quad \forall j \in V^{'} \tag{4.17}$$

$$u_{ij} + v_{ij} \leq EV_{\text{cap}} x_{ij} \quad \forall i \in V_0^{'}, j \in V_{n+1}^{'}, i \neq j \tag{4.18}$$

$$0 \leq y_j \leq y_i - (h d_{ij}) x_{ij} + B_{\text{cap}} (1 - x_{ij}) \quad \forall i \in V, \forall j \in V_{n+1}^{'}, i \neq j \tag{4.19}$$

$$0 \leq y_j \leq Y_i - (h d_{ij}) x_{ij} + B_{\text{cap}} (1 - x_{ij}) \quad \forall i \in F_0^{'}, \forall j \in V_{n+1}^{'}, i \neq j \tag{4.20}$$

$$y_i \leq Y_i \leq B_{\text{cap}} \quad \forall i \in F_0^{'} \tag{4.21}$$

$$x_{ij} \in 0, 1 \quad \forall i \in V_0^{'}, j \in V_{n+1}^{'}, i \neq j \tag{4.22}$$

The distance-based objective function, originally proposed in [13], is extended to give priority to solutions employing fewer vehicles, even if the total distance traveled in such solutions surpasses that of other solutions. This extension involves introducing an additional cost parameter $M > 0$ per utilized vehicle. It is important to note that the number of vehicles used in a solution corresponds to the variables on outgoing arcs of the depot (0) with a value of 1. In this context, the objective function (4.7) aims to minimize the total travel and vehicle costs. Constraints (4.8) ensure that each customer is visited by an electric vehicle, while constraints (4.9) allow vehicles to visit a charging station when necessary. Constraints (4.10) ensure that each vehicle visiting a particular node must also depart from the corresponding node. The arrival and departure times are calculated using constraints (4.11) and (4.12), which take into account service and battery charging times. Constraints (4.13) permit vehicles to visit each node within the corresponding time windows while preventing sub-tours. Constraints (4.14)–(4.18) ensure simultaneous fulfillment of delivery and pickup demands for customers. Finally, constraints (4.19)–(4.21) are associated with the battery state of charge. For an illustrative instance along with a solution, refer to Fig. 4.9.

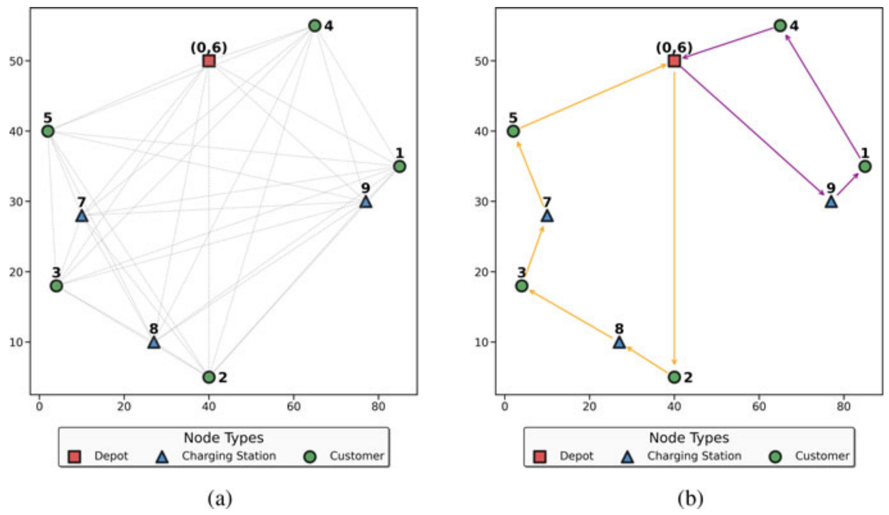


Fig. 4.9 Visualization of an (**a**) EVRP instance and (**b**) its solution: Figure (**a**) displays a map featuring the positions of a depot, five customers, and three charging stations, represented by Cartesian coordinates. The fully connected graph, denoted by gray dashed lines, contains a connection between every pair of nodes. Figure (**b**) depicts a valid solution for the specified instance on the same map. It consists of two separate tours as shown by arrows of distinct colors. Both routes commence and conclude at the depot, traversing different customers and charging stations along the way

4.3.1 Short Literature Review Concerning the EVRP-TW-SPD

Addressing increasing environmental concerns and the ensuing demand for alternative fuel sources in logistics, recent research has concentrated on formulating routing strategies that optimize the transportation of goods while accounting for the limited driving range and en-route charging requirements associated with EVs. These challenges are commonly denoted as EVRPs, or more broadly, Green Vehicle Routing Problems. Comprehensive surveys of recent research on EVRPs are available in [3, 16]. Given that our primary focus is on the methodology for solving specific types of problems rather than particular problem variants, we recommend interested readers consult these survey papers for more in-depth references. Instead, we highlight the distinctions between the EVRP-TW-SPD and existing problems in the literature. Beyond incorporating time window constraints, our problem also addresses simultaneous pickup and delivery (SPD) constraints related to customer deliveries, a consideration commonly linked to reverse logistics. Despite the crucial role of reverse logistics in advancing sustainability, the number of publications exploring variants of the EVRP-SPD remains limited. To date, only [24] have explored SPD constraints within the realm of EVRPs. Notably, in conventional EVRP models, the assumption is that EV batteries are fully charged upon visiting a charging station. In contrast, the problem considered here embraces a more realistic scenario by allowing for partial recharging.

4.3.2 Set-Covering Based ILP Model of the EVRP-TW-SPD

Typically, assignment-type ILP models like the one outlined for the EVRP-TW-SPD face challenges in generating effective lower bounds, as noted in prior studies such as [2]. Furthermore, experiments detailed in [1] revealed the difficulty of CPLEX in identifying feasible solutions for the corresponding model within reasonable execution times, even when dealing with small-sized sub-instances of the original problem instances.

Similar to the approach outlined for the VSBP problem in Sect. 4.2.2 on page 98, the EVRP-TW-SPD can be expressed through a set-covering-based ILP as follows. Let $\mathcal{T}$ represent the set of all possible (and feasible) tours, where a tour is defined as the journey of a single vehicle leaving from and returning to the depot. Each tour $T_r \in \mathcal{T}$ is assessed based on the total distance traveled d_r, which is the sum of distances for all arcs along the tour. Lastly, let $\mathcal{T}_i \subset \mathcal{T}$ be the set of tours that cater to customer $i \in V$. With these definitions, the set-covering-based ILP model for the EVRP-TW-SPD, hereafter referred to as $\texttt{ILP}^{\text{EVRP}}_{\text{setcov}}$, can be formulated as follows.

$$\min \quad \sum_{T_r \in \mathcal{T}} d_r x_r + M \sum_{T_r \in \mathcal{T}} x_r \tag{4.23}$$

$$\text{subject to} \quad \sum_{T_r \in \mathcal{T}_i} x_r \geq 1 \quad \forall\, i \in V \tag{4.24}$$

$$x_r \in \{0, 1\} \quad \forall\, T_r \in \mathcal{T}$$

The objective function aims to minimize the overall travel and vehicle costs, while constraints (4.24) guarantee that each customer is visited at least once. It is worth noting that the set-covering-based formulation is typically employed as a post-optimization method in the VRP literature, as seen in [20]. In contrast, our findings demonstrate that CMSA serves as a viable algorithmic framework for iteratively applying both heuristics and exact components.

4.3.3 *Application of* ADAPT_CMSA *to the EVRP-TW-SPD*

Similar to the approach taken for the VSBP problem, we initially develop a version of CMSA based on the assignment-type ILP model—specifically, model $\texttt{ILP}^{\text{EVRP}}_{\text{std}}$—for the EVRP-TW-SPD. However, in contrast to the VSBP problem, where CMSA_GEN was used as CMSA variant, in the case of the EVRP-TW-SPD we use ADAPT_CMSA. Within the context of ADAPT_CMSA, the complete set C of solution components includes a component c_{ij} for each arc a_{ij} from $A = \{(i, j) | i, j \in V'_{0,n+1}, i \neq j\}$. Consider the following illustration: The vector $\mathbf{I}$ encompasses all the node indexes for a small-scale problem instance featuring three charging stations and five customers, with nodes indexed as 0 and 6 representing the depot.

$$\mathbf{I} = (\underbrace{0,}_{\text{depot}} \; \underbrace{1, 2, 3, 4, 5,}_{\text{customers}} \; \underbrace{6,}_{\text{depot}} \; \underbrace{7, 8, 9}_{\text{charging stations}} \;)$$

Now, let us examine a solution comprising two tours, T_1 and T_2, where T_1 =<0-9-1-4-6> and T_2 =<0-2-8-3-7-5-6>. In the context of ADAPT_CMSA, this solution is expressed as $S = \{c_{0,9}, c_{9,1}, c_{1,4}, c_{4,6}, c_{0,2}, c_{2,8}, c_{8,3}, c_{3,7}, c_{7,5}, c_{5,6}\}$. In other words, a solution S in ADAPT_CMSA is maintained in terms of the list of solution components that represent the arcs utilized in any of the tours within S.

4.3.4 *The* ADAPT_CMSA *Algorithm*

The pseudo-code outlined in Algorithm 4.2 is shared between ADAPT_CMSA and ADAPT_CMSA_SETCOV. Although the pseudo-code for ADAPT_CMSA was previously presented in Algorithm 2.1 on page 42, we now provide the specific pseudo-code for our implementation targeting the EVRP-TW-SPD. This is

Algorithm 4.2: Pseudo-code of ADAPT_CMSA for the EVRP-TW-SPD

```
1: input 1: Complete set of solution components $C$
2: input 1: Values for ADAPT_CMSA parameters $t_{\text{prop}}$, $t_{\text{ILP}}$, $n^{\text{init}}$, $n^{\text{inc}}$
3: input 2: Values for solution construction parameters $\alpha^{\text{LB}}$, $\alpha^{\text{UB}}$, $\alpha_{\text{red}}$, $l_{\text{size}}^{\text{init}}$, $l_{\text{size}}^{\text{inc}}$
4: $S^{\text{bsf}} :=$ GenerateGreedySolution$(C)$
5: $\alpha_{\text{bsf}} := \alpha^{\text{UB}}$, $C' := S^{\text{bsf}}$
6: Initialize$(n_{\text{a}}, l_{\text{size}})$
7: while CPU time limit not reached do
8:   for $i := 1, \ldots, n_{\text{a}}$ do
9:     $S :=$ ProbabilisticSolutionConstruction$(C, S^{\text{bsf}}, \alpha_{\text{bsf}}, l_{\text{size}})$
10:    LocalSearch1$(S)$
11:    $C' := C' \cup S$
12:  end for
13:  $(S^{\text{ILP}}, t_{\text{solve}}) :=$ SolveSubinstance$(C', t_{\text{ILP}})$ {This function returns two objects: (1) the
     obtained solution $(S^{\text{ILP}})$, (2) the required computation time $(t_{\text{solve}})$}
14:  LocalSearch2$(S^{\text{ILP}})$
15:  if $t_{\text{solve}} < t_{\text{prop}} \cdot t_{\text{ILP}}$ and $\alpha_{\text{bsf}} > \alpha^{\text{LB}}$ then $\alpha_{\text{bsf}} := \alpha_{\text{bsf}} - \alpha_{\text{red}}$ end if
16:  if $f(S^{\text{ILP}}) < f(S^{\text{bsf}})$ then
17:    $S^{\text{bsf}} := S^{\text{ILP}}$
18:    Initialize$(n_{\text{a}}, l_{\text{size}})$
19:  else
20:    if $f(S^{\text{ILP}}) > f(S^{\text{bsf}})$ then
21:      if $n_a = n^{\text{init}}$ then $\alpha_{\text{bsf}} := \min\{\alpha_{\text{bsf}} + \frac{\alpha_{\text{red}}}{10}, \alpha^{\text{UB}}\}$ else Initialize$(n_{\text{a}}, l_{\text{size}})$ end if
22:    else
23:      Increment$(n_{\text{a}}, l_{\text{size}})$
24:    end if
25:  end if
26:  $C' := S^{\text{bsf}}$
27: end while
28: output: $S^{\text{bsf}}$
```

necessary because, in addition to problem-specific local search procedures, we have introduced a generalization to ADAPT_CMSA. The changes in Algorithm 4.2 compared to Algorithm 2.1 are highlighted in blue and will be elaborated upon in detail in the following. However, for gaining a general understanding of the ADAPT_CMSA algorithm we recommend reading Sect. 2.2 of Chap. 2 starting on page 43 before proceeding in this section.

Recall that in ADAPT_CMSA, the number of solution constructions per iteration (n_{a}) was dynamically adjusted as follows: Initially, at the algorithm's start, n_{a} was initialized to 1. Additionally, whenever the solver solution (S^{ILP}) proved strictly superior to the best-so-far solution (S^{bsf}), n_{a} was re-initialized to 1. Furthermore, when the quality of S^{ILP} equaled the quality of S^{bsf}, n_{a} was increased by one. In Algorithm 4.2, this process is generalized by introducing an initial value n^{init} for n_{a} and an increment value n^{inc} for n_{a}, both being tunable parameters of the algorithm. Alongside n_{a}, another parameter (l_{size}), used to adjust the level of greediness for solution construction, is introduced. This parameter, which is called the candidate list size, undergoes a self-adaptive treatment identical to n_{a}. In other words, this parameter is (re)initiated or incremented under the same conditions as n_{a}. In order to do so, Algorithm 4.2 employs two additional tunable parameters: $l_{\mathrm{size}}^{\mathrm{init}}$ and $l_{\mathrm{size}}^{\mathrm{inc}}$. The generalizations described in this paragraph can be observed (in blue color) in lines 6, 18, 21, and 23 of Algorithm 4.2.

The second modification from the original ADAPT_CMSA version in Chap. 2 involves the incorporation of problem-specific local search procedures. Specifically, following the construction of a solution S through the function call in line 9 of Algorithm 4.2, a local search procedure is invoked (refer to function LocalSearch1(S) in line 10), during which each tour of S undergoes a dedicated local search process. Common intra-route operators such as *relocation*, *swap*, and *two_opt* are sequentially applied for this purpose. Additionally, a best-improvement strategy is adopted within the context of the applied operators. The *relocation* operator systematically extracts each node from its current position within a route and relocates it to an alternative position within the same route. Conversely, the *swap* operator involves interchanging the positions of a pair of selected nodes within the same route. Finally, the *two_opt* neighborhood explores all feasible combinations of selecting two non-adjacent nodes in the same route and then reverses the arrangement of the nodes situated between the chosen pair of nodes.

The second local search procedure is employed to enhance the solver solution S^{ILP} in each iteration; refer to the function LocalSearch2(S^{ILP}) in line 14 of the algorithm. Specifically, this local search procedure leverages inter-tour neighborhoods such as *exchange (1,1)* and *shift (1,0)*. The *exchange (1,1)* neighborhood explores all possible two-customer swaps not within the same tour, while the *shift (1,0)* neighborhood assesses each option for removing a customer from its current tour and placing it at any feasible location in other tours. Similar to LocalSearch1(S), the operators employed by LocalSearch2(S) adhere to the best-improvement search strategy.

4.3.4.1 Constructing Solutions to the EVRP-TW-SPD

Solutions must be constructed in ADAPT_CMSA and ADAPT_CMSA_SETCOV deterministically at the start of the algorithm—function GenerateGreedySolution(C) in line 4 of Algorithm 4.2—and probabilistically at each algorithm iteration, see function ProbabilisticSolutionConstruction(C, S^{bsf}, α_{bsf}, l_{size}) in line 9 of the algorithm. While at the start of the algorithm, a so-called insertion heuristic is invoked to produce an initial, feasible solution, probabilistic solution constructions during algorithm iterations make either use of the insertion algorithm or a version of the Clarke & Wright (C&W) savings algorithm [6] adapted to the EVRP-TW-SPD. The choice between these two methods is done uniformly at random. Both heuristics exclusively generate feasible solutions. In the following, both solution construction techniques are described.

Probabilistic C&W Savings Algorithm In line with the original C&W approach, our algorithm variant commences by creating a set of direct routes, labeled as $R = \{(0 - i - (n+1)) \mid i \in V\}$. Subsequently, the algorithm initializes a savings list L comprising pairs of nodes (i, j), with i and j representing customers and charging stations. The savings value σ_{ij} for each pair is calculated using the following equation:

$$\sigma_{ij} := d_{0i} + d_{0j} - \lambda d_{ij} + \mu |d_{0i} - d_{0j}| \tag{4.25}$$

In this context, λ and μ serve as the so-called *route shape* and *asymmetry scaling* parameters, respectively. The route shape parameter λ emphasizes the selection of nodes based on their distance from each other [23], while the parameter μ scales the asymmetry between nodes i and j [18]. Effective values for these parameters are determined through a parameter tuning process outlined in Sect. 4.3.6.2. It is crucial to note that L exclusively contains pairs of nodes (i, j) that meet the following two criteria:

1. Nodes i and j belong to different tours.
2. Both i and j are adjacent to the depot in the tour to which they belong.

Furthermore, the construction of a solution is not solely influenced by the savings values of node pairs (i, j) but also by whether or not arc a_{ij} appears in the current best-so-far solution S^{bsf}. For this purpose, an additional value, q_{ij}, is computed for each entry $(i, j) \in L$:

$$q_{ij} := \begin{cases} (\sigma_{ij} + 1) \cdot \alpha_{\text{bsf}} & \text{if } c_{ij} \in S^{\text{bsf}} \\ (\sigma_{ij} + 1) \cdot (1 - \alpha_{\text{bsf}}) & \text{otherwise} \end{cases} \tag{4.26}$$

The algorithm executes the subsequent series of steps until the savings list L becomes empty.

1. Following the computation of q_{ij} for all entries in L, the list is arranged in non-increasing order based on the q_{ij} values. Subsequently, a reduced list L_r is generated, comprising the initial l_{size} elements from L, where l_{size} is a tunable parameter of the algorithm.
2. Next, an entry (i, j) is selected from L_r based on the following probabilities:

$$\mathbf{p}(ij) := \frac{q_{ij}}{\sum_{(i',j') \in L_r} q_{i'j'}} \quad \forall\, (i, j) \in L_r \tag{4.27}$$

   It is important to observe that the probability of choosing arcs that are part of the best-so-far solution S^{bsf} increases with an increasing value of α_{bsf}, where $0 \leq \alpha^{\text{LB}} \leq \alpha_{\text{bsf}} \leq \alpha^{\text{UB}} \leq 1$.

3. Subsequently, the tours associated with nodes i and j are merged. The merging process falls into one of the following four potential cases, depending on the direct connection of nodes i and j to the depot:

   (Case 1)
   - $T_1 : \{0 - i - \cdots - n+1\}, T_2 : \{0 - j - \cdots - n+1\}$
   - Merging: Reverse T_1, $\mathsf{rev}(T_1)$, and concatenate with T_2
   - Result: $T_m : \{0 - \cdots - i - j \cdots - n+1\}$

   (Case 2)
   - $T_1 : \{0 - i - \cdots - n+1\}, T_2 : \{0 - \cdots - j - n+1\}$
   - Merging: Reverse both T_1 and T_2, $\mathsf{rev}(T_1)$, $\mathsf{rev}(T_2)$, and concatenate
   - Result: $T_m : \{0 - \cdots - i - j \cdots - n+1\}$

   (Case 3)
   - $T_1 : \{0 - \cdots - i - n+1\}, T_2 : \{0 - j - \cdots - n+1\}$
   - Merging: Concatenate T_1 and T_2
   - Result: $T_m : \{0 - \cdots - i - j \cdots - n+1\}$

   (Case 4)
   - $T_1 : \{0 - \cdots - i - n+1\}, T_2 : \{0 - \cdots - j - n+1\}$
   - Merging: Reverse T_2, $\mathsf{rev}(T_2)$, and concatenate with T_1
   - Result: $T_m : \{0 - \cdots - i - j \cdots - n+1\}$

   Depending on the positions of nodes i and j within the tour, it may be necessary to reverse one or both of the selected tours to establish a direct connection from i to j. In such instances, the reversed version of tour T_1 is denoted as $\mathsf{rev}(T_1)$. Subsequently, the feasibility of the combined tour T_m is assessed with regard to vehicle loading capacity and time windows. If the resulting route violates vehicle capacity and/or time window constraints, it is considered infeasible and excluded from the savings list. A new candidate is then chosen using the previously outlined procedure. If the merged tour is infeasible due to battery constraints, a charging station is introduced into the tour. Determining the optimal location for the charging station involves identifying the first node in the tour where the electric vehicle arrives with a negative battery level. Subsequently, a charging station is inserted between this node and the preceding one. After determining the insertion position, the charging station that minimally increases the overall tour distance is selected and placed accordingly. If the tour remains infeasible, the same procedure is applied to the preceding arcs. In cases where infeasibility persists despite attempts to insert charging stations, the merged tour is discarded, and the associated nodes are removed from the savings list. The next candidate pair of nodes is then selected from the savings list, following the aforementioned procedure. This tour merging process is iteratively executed until the savings list is depleted. Once the merging phase concludes, some of the initially added charging stations may become unnecessary. Consequently, redundant charging stations are identified and subsequently removed from the constructed tours.
4. Finally, the savings list L is updated as described above.

In the final step, the ultimate set of tours undergoes a transformation into its corresponding set of solution components.

Probabilistic Insertion Algorithm Our second constructive heuristic works by sequentially incorporating customers into available tours until all customers have been visited. The initial customer to be included in the tour is selected based on either the distance from the depot or the latest feasible visiting time. Specifically, the initial tour is formed by inserting the customer with the greatest distance from the depot or the earliest deadline. Following this, we generate a cost list L containing all potential insertion points for each unvisited customer along with their associated costs. The cost of inserting a customer at a specific point is determined using the following equation, which calculates the cost cost_{jik} of inserting customer i between nodes j and k.

$$\text{cost}_{jik} = d_{ji} + d_{ik} - d_{jk} \tag{4.28}$$

Subsequently, a q_{jik} value is calculated for each entry $(j, i, k) \in L$ as follows:

$$q_{jik} := \begin{cases} (\text{cost}_{jik} + 1) \cdot (1 - \alpha_{\text{bsf}})(1 - \alpha_{\text{bsf}}) & \text{if } c_{ji} \in S^{\text{bsf}} \text{ and } c_{ik} \in S^{\text{bsf}} \\ (\text{cost}_{jik} + 1) \cdot \alpha_{\text{bsf}} & \text{if } c_{ji} \notin S^{\text{bsf}} \text{ and } c_{ik} \notin S^{\text{bsf}} \\ (\text{cost}_{jik} + 1) \cdot \alpha_{\text{bsf}}(1 - \alpha_{\text{bsf}}) & \text{otherwise} \end{cases} \tag{4.29}$$

Following that, the choice of an entry (j, i, k) from L is executed based on the probabilities computed using Eq. (4.29). If the vehicle's capacity allows, the customer is incorporated into the appropriate location within the tour. Moreover, if the insertion proves to be infeasible due to battery constraints, a charging station is introduced into the tour using the method outlined in the C&W savings algorithm. In instances where inserting a customer leads to the vehicle surpassing its load or battery capacity (even after charging station insertion), or results in a time window violation, a new tour is initiated, encompassing only the specific customer.

After inserting all of the customers and a complete solution is derived, the obtained set of tours is transformed into the corresponding set S of solution components.

4.3.4.2 Sub-instance Solving

The incumbent sub-instance C' is solved at each iteration of ADAPT_CMSA by first generating a corresponding ILP model based on model $\texttt{ILP}^{\text{EVRP}}_{\text{std}}$ and then solving the model with a CPU time limit of t_{ILP} seconds with CPLEX in function SolveSubinstance(C', t_{ILP}). In order to generate this model, the following constraints are added to $\texttt{ILP}^{\text{EVRP}}_{\text{std}}$:

$$x_{ij} = 0 \quad \text{for all } c_{ij} \in C \setminus C' \tag{4.30}$$

Put differently, if an arc a_{ij} has not been employed in any of the solutions that were merged into C', the utilization of this arc is prohibited by setting the value of x_{ij} to zero.

4.3.5 *The* ADAPT_CMSA_SETCOV *Algorithm*

The ILP model for solving sub-instances in ADAPT_CMSA_SETCOV is model $\texttt{ILP}_{\text{setcov}}^{\text{EVRP}}$ from Sect. 4.3.2 on page 119. Remember that the complete set of solution components C, in the case of ADAPT_CMSA_SETCOV, consists of a component c_r for each valid tour $T_r \in \mathcal{T}$ (see Sect. 4.3.2), that is, $C := \{c_r \mid T_r \in \mathcal{T}\}$. Any subset $S \subset C$ such that each customer $i \in V$ is served by exactly one tour of S is a valid solution to the EVRP-TW-SPD problem instance.

The probabilistic solution construction process in ADAPT_CMSA_SETCOV operates in an identical manner as in ADAPT_CMSA. The distinction lies in the fact that the solutions generated comprise solution components that directly correspond to tours, as opposed to arcs as observed in the case of ADAPT_CMSA.

Another distinction lies in the utilization of the ILP model for solving sub-instances, as previously mentioned. Specifically, when dealing with a sub-instance C', the corresponding ILP model is derived by substituting every occurrence of $\mathcal{T}$ in $\texttt{ILP}_{\text{setcov}}^{\text{EVRP}}$ with C'. This implies that the model exclusively considers tours present in sub-instance C' as eligible tours. Following each execution of CPLEX, once the solver solution S^{ILP} is attained, a check for duplicate customer occurrences is conducted before returning the solution to the main algorithmic level of ADAPT_CMSA_SETCOV. All redundant customers are identified, and subsequently, the advantage of removing each redundant customer—directly tied to the distance between the respective customer and its adjacent nodes—is calculated. Following this, redundant customers are systematically eliminated, starting with the one offering the greatest benefit, until each customer is allocated to a single tour exclusively.

4.3.6 *Experimental Evaluation*

The experiments were carried out on the same high-performance computing cluster used for all preceding experiments in this book, namely, the in-house computing cluster of the IIIA-CSIC. The cluster is equipped with machines featuring Intel® Xeon® 5670 CPUs, each possessing 12 cores clocked at 2.933 GHz and a minimum of 32 GB of RAM. Additionally, the application to sub-instances in both CMSA variants employed CPLEX version 20.1 in single-threaded mode. Furthermore, the ILP models representing complete problem instances were tackled using CPLEX version 20.1 in standalone mode.

4.3.6.1 Problem Instances for the EVRP-SPD-TW

A benchmark set of problem instances was generated on the basis of the EVRP-TW instances introduced in [22]. The resulting dataset comprises a total of 92 instances, categorized into 36 small-sized instances and 56 large-sized instances. Small-sized instances involve 5, 10, and 15 customers, while large-sized instances consist of 100 customers and 21 charging stations. These instances are classified into three distinct groups based on the spatial distribution of customer locations: clustered instances (indicated by the prefix "c" in the instance name), randomly distributed instances (prefix "r"), and instances with a hybrid of random and clustered distributions (prefix "rc"). Each group further encompasses two sub-classes (type1 and type2) that differentiate instances based on factors such as time windows, vehicle load, and battery capacity.

Initially, it was necessary to segregate the delivery demand into pickup and delivery demands because each customer in the original instances had only one demand information. To derive delivery and pickup demands from the original demand, we employed the method outlined in [21]. This approach involves calculating a ratio $\rho_i = \min\{\frac{x_i}{y_i}, \frac{y_i}{x_i}\}$ for each customer $i \in V$, where (x_i, y_i) represents the Cartesian coordinates of customer i. Subsequently, the delivery demand q_i was determined by multiplying the original demand δ_i by ρ_i, while the pickup demand p_i was obtained by subtracting q_i from δ_i.

4.3.6.2 Parameter Tuning

Similar to the approach taken for the VSBP problem, we utilized the scientific tuning software irace [14] to determine effective parameter values for ADAPT_CMSA and ADAPT_CMSA_SETCOV; see Sect. 1.2.1 on page 12 for a description. The tuning process involved six instances: `r107`, `r205`, `rc101`, `rc104`, `rc105`, and `rc205`. The budget allocated for irace—indicating the maximum number of algorithm runs permitted for tuning—was set to 2500, with a fixed time limit of 900 CPU seconds per instance. Additionally, the precision of irace was set to two decimal places for numerical parameters. Table 4.5 details the parameters, their domains, and the final

Table 4.5 Parameters, their domains, and the chosen values as determined by irace

Parameter	Domain	ADAPT_CMSA	ADAPT_CMSA_SETCOV
t_{ILP}	$\{5, 7, 10, 15, 20, 25, 30, 35, 40\}$	40	20
α^{LB}	$[0.6, 0.99]$	0.92	0.75
α^{UB}	$[0.6, 0.99]$	0.98	0.86
α_{red}	$[0.01, 0.1]$	0.07	0.07
t_{prop}	$[0.1, 0.8]$	0.17	0.23
n^{init}	$\{1, 3, 5, 10, 50, 100, 200, 300, 500\}$	1	10
n^{inc}	$\{1, 3, 5, 10, 50, 100, 200, 300, 400\}$	1	50
$l_{\text{size}}^{\text{init}}$	$\{3, 5, 10, 15, 20, 50, 100, 200\}$	100	10
$l_{\text{size}}^{\text{inc}}$	$\{3, 5, 10, 15, 20, 50, 100, 200\}$	15	20
λ	$[1, 2]$	1.99	1.38
μ	$[0, 1]$	0.23	0.58

values chosen for the experimentation. In this context, remember that the first group of five parameters in this table are the usual ADAPT_CMSA parameters. The second group consisting of four parameters are the ones resulting from the generalization of ADAPT_CMSA. Finally, the last two parameters are the route shape parameter (λ) and the asymmetric scaling parameter (μ) from the C&W solution construction heuristic. Note also that, due to an already elevated number of algorithm parameters, in the case of this experimental evaluation CPLEX was always applied with the default parameter setting.

It is noteworthy that the values obtained for n^{init} and n^{inc} are notably smaller in the context of ADAPT_CMSA in comparison to those for ADAPT_CMSA_SETCOV. One plausible explanation for this discrepancy is that the ILP model employed within ADAPT_CMSA poses an obstacle to the algorithm's success. It appears that the sub-instances need to be kept as small as possible to enable CPLEX to generate valid solutions within the constrained time when solving these sub-instances. This interpretation is corroborated by the values obtained for t_{ILP}. In the case of ADAPT_CMSA, the time limit for CPLEX to solve the ILP models in each iteration is approximately twice as high. In contrast, the value of the $l_{\text{size}}^{\text{init}}$ parameter determined for ADAPT_CMSA is considerably higher than that determined for ADAPT_CMSA_SETCOV. A higher $l_{\text{size}}^{\text{init}}$ could be perceived as a diversification mechanism, serving as compensation for dealing with small sub-instances.

4.3.6.3 Results

This section presents a comprehensive experimental evaluation of the two CMSA variants across all instances detailed in Sect. 4.3.6.1. The numerical outcomes for small-sized instances can be found in Tables 4.6, 4.7, and 4.8, while results for larger-sized instances are displayed in Tables 4.9, 4.10, and 4.11. To gauge the algorithms' effectiveness in handling small instances, we compared ADAPT_CMSA and ADAPT_CMSA_SETCOV with the standalone application of CPLEX, denoted henceforth as CPLEX. However, given CPLEX's limitation in handling large instances, our comparison for such scenarios is restricted to the two CMSA variants. Both CMSA variants were executed with a computation time limit of 150 CPU seconds for small instances and 900 CPU seconds for large instances. In contrast, CPLEX was granted a computation time limit of 2 hours for small-size instances. Each algorithm underwent 10 runs for each problem instance. It is worth noting that, to calculate objective function values, we set the cost of each vehicle used in a solution to 1000, i.e., $M = 1000$.[1]

In each result table, the initial column presents the instance names, while the columns labeled 'm' indicate the number of vehicles employed in the best solution discovered by the respective algorithm across 10 runs. For both ADAPT_CMSA and ADAPT_CMSA_SETCOV, the 'best' columns showcase the objective function

[1] Remember that M is the cost given in the ILP models to the use of a vehicle.

Table 4.6 Results for small-sized EVRP-TW-SPD instances with 5 customers

Instance	CPLEX			ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	m	Best	gap (%)	m	Best	Avg.	Time	m	Best	Avg.	Time
`c101C5`	**2**	**2257.75**	0	**2**	**2257.75**	2257.75	0.03	**2**	**2257.75**	2257.75	0.017
`c103C5`	**1**	**1175.37**	0	**1**	**1175.37**	1175.37	0.77	**1**	**1175.37**	1175.37	0.975
`c206C5`	**1**	**1242.56**	0	**1**	**1242.56**	1242.56	0.04	**1**	**1242.56**	1242.56	0.006
`c208C5`	**1**	**1158.48**	0	**1**	**1158.48**	1158.48	0.01	**1**	**1158.48**	1158.48	0.001
`r104C5`	**2**	**2136.69**	0	**2**	**2136.69**	2136.69	0.10	**2**	**2136.69**	2136.69	0.011
`r105C5`	**2**	**2156.08**	0	**2**	**2156.08**	2156.08	0.01	**2**	**2156.08**	2156.08	0.001
`r202C5`	**1**	**1128.78**	0	**1**	**1128.78**	1128.78	12.81	**1**	**1128.78**	1128.78	0.001
`r203C5`	**1**	**1179.06**	0	**1**	**1179.06**	1179.06	0.32	**1**	**1179.06**	1179.06	0.068
`rc105C5`	**2**	**2233.77**	0	**2**	**2233.77**	2233.77	0.13	**2**	**2233.77**	2233.77	0.061
`rc108C5`	**2**	**2253.93**	0	**2**	**2253.93**	2253.93	0.01	**2**	**2253.93**	2253.93	0.003
`rc204C5`	**1**	**1176.39**	0	**1**	**1176.39**	1176.39	0.10	**1**	**1176.39**	1176.39	0.015
`rc208C5`	**1**	**1167.98**	0	**1**	**1167.98**	1167.98	0.74	**1**	**1167.98**	1167.98	0.037
`average`	**1.42**	**1605.57**	0	**1.42**	**1605.57**	1605.57	1.25	**1.42**	**1605.57**	1605.57	0.100

Table 4.7 Results for small-sized EVRP-TW-SPD instances with 10 customers

Instance	CPLEX			ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	*m*	Best	Gap (%)	*m*	Best	Avg.	Time	*m*	Best	Avg.	Time
`c101C10`	**3**	**3388.25**	0	**3**	**3388.25**	3388.25	0.40	**3**	**3388.25**	3388.55	0.46
`c104C10`	**2**	**2273.93**	0	**2**	**2273.93**	2273.93	0.68	**2**	**2273.93**	2273.93	41.31
`c202C10`	**1**	**1304.06**	0	**1**	**1304.06**	1304.06	0.64	**1**	**1304.06**	1304.06	0.14
`c205C10`	**2**	**2228.28**	0	**2**	**2228.28**	2228.28	15.40	**2**	**2228.28**	2228.28	0.05
`r102C10`	**3**	**3249.19**	0	**3**	**3249.19**	3249.19	25.93	**3**	**3249.19**	3249.19	0.01
`r103C10`	**2**	**2206.12**	0	**2**	**2206.12**	2206.12	7.50	**2**	**2206.12**	2206.12	31.33
`r201C10`	**1**	**1241.51**	0	**1**	**1241.51**	1241.51	32.32	**1**	**1241.51**	1241.51	22.73
`r203C10`	**1**	**1218.21**	0	**1**	**1218.21**	1218.21	20.02	**1**	**1218.21**	1218.21	38.67
`rc102C10`	**4**	**4423.51**	0	**4**	**4423.51**	4423.51	0.34	**4**	**4423.51**	4423.51	0.02
`rc108C10`	**3**	**3345.93**	0	**3**	**3345.93**	3345.93	20.80	**3**	**3345.93**	3345.93	0.02
`rc201C10`	**1**	**1412.86**	16.64	**1**	**1412.86**	1412.86	2.80	**1**	**1412.86**	1412.86	1.33
`rc205C10`	**2**	**2325.98**	0	**2**	**2325.98**	2325.98	0.19	**2**	**2325.98**	2325.98	0.64
`average`	**2.08**	**2384.82**	1.66	**2.08**	**2384.82**	2384.82	10.59	**2.08**	**2384.82**	2384.84	11.392

Table 4.8 Results for small-sized EVRP-TW-SPD instances with 15 customers

Instance	CPLEX			ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	*m*	Best	Gap (%)	*m*	Best	Avg.	Time	*m*	Best	Avg.	Time
`c103C15`	**3**	**3348.46**	7.3	**3**	**3348.46**	3348.47	68.59	**3**	**3348.46**	3348.46	5.35
`c106C15`	**3**	**3275.13**	0	**3**	**3275.13**	3275.13	6.49	**3**	**3275.13**	3275.13	46.03
`c202C15`	**2**	**2383.62**	0	**2**	**2383.62**	2383.62	18.12	**2**	**2383.62**	2393.39	7.50
`c208C15`	**2**	**2300.55**	0	**2**	**2300.55**	2300.55	1.55	**2**	**2300.55**	2300.55	12.23
`r102C15`	**5**	**5412.78**	20.6	**5**	**5412.78**	5412.78	2.59	**5**	**5412.78**	5412.78	0.12
`r105C15`	**4**	**4336.15**	0	**4**	**4336.15**	4336.15	1.47	**4**	**4336.15**	4336.15	1.22
`r202C15`	2	2361.51	27.3	2	2358.00	2364.90	32.32	**1**	**1507.32**	1677.46	64.29
`r209C15`	**1**	**1313.24**	0	**1**	**1313.24**	1313.24	17.41	**1**	**1313.24**	1313.24	15.62
`rc103C15`	**4**	**4397.67**	0	**4**	**4397.67**	4397.67	0.34	**4**	**4397.67**	4397.67	0.30
`rc108C15`	**3**	**3370.25**	0	**3**	**3370.25**	3370.25	58.85	**3**	**3370.25**	3370.25	0.34
`rc202C15`	**2**	**2394.39**	0	**2**	**2394.39**	2394.39	0.68	**2**	**2394.39**	2394.39	27.98
`rc204C15`	**1**	1403.38	28.7	**1**	**1382.22**	1385.72	68.09	**1**	**1382.22**	1382.55	71.87
`average`	2.67	3024.76	6.99	2.67	3022.71	3023.57	23.04	**2.58**	**2951.82**	2966.83	21.070

Table 4.9 Results for large-sized clustered EVRP-TW-SPD instances

Instance	ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	*m*	Best	Avg.	Time	*m*	Best	Avg.	Time
c101	**12**	13,043.40	13,043.42	385.13	**12**	13,057.80	13,063.54	292.56
c102	**11**	12,056.80	12,920.23	560.77	**11**	12,073.10	12,944.34	468.81
c103	11	12,004.70	12,026.90	452.13	**10**	11,134.90	11,917.80	718.09
c104	**10**	10,872.80	11,353.78	629.96	**10**	10,870.70	10,876.49	608.43
c105	**11**	12,023.80	12,341.60	562.10	**11**	12,034.10	12,068.86	582.74
c106	**11**	12,013.10	12,438.06	652.00	**11**	12,025.70	12,059.29	434.80
c107	**11**	12,006.40	12,023.97	538.41	**11**	12,026.70	12,046.38	393.01
c108	11	11,994.70	12,016.10	579.51	**10**	11,025.80	11,822.60	556.58
c109	**10**	11,042.20	11,885.30	714.89	**10**	10,941.00	11,180.77	746.17
c201	**4**	4629.95	4629.95	37.59	**4**	4678.37	4703.43	390.96
c202	**4**	4629.95	4629.95	273.58	**4**	4664.26	4706.94	394.84
c203	**4**	4632.27	4690.06	740.49	**4**	4641.45	4734.31	497.60
c204	**4**	4633.08	4665.78	801.76	**4**	4660.64	4737.07	716.94
c205	**4**	4629.95	4629.95	76.87	**4**	4629.95	4629.95	125.43
c206	**4**	4629.95	4629.95	213.28	**4**	4629.95	4629.95	203.04
c207	**4**	4629.95	4629.95	255.85	**4**	4629.95	4635.27	260.34
c208	**4**	4629.95	4629.95	284.78	**4**	4629.95	4629.95	261.72
average	7.65	8476.64	8657.94	456.42	**7.53**	8373.78	8552.17	450.12

values of the best solutions derived from 10 runs. Additionally, the ‘avg.’ columns display the average objective function values over the best solutions from each of the 10 runs. Furthermore, the ‘time’ columns reveal the average computation times of the two CMSA variants to find the best solutions in each run. Lastly, the ‘gap(%)’ columns indicate the percentage difference between the optimal solutions attained and the best lower bounds achieved by CPLEX. It is essential to note that a gap value of zero implies that CPLEX has identified (and proved) an optimal solution.

> Main Observations for Small-Sized Problem Instances

- CPLEX solved 31 problem instances to optimality. For the remaining five problem instances (rc201C10, c103C15, r102C15, r202C15, and rc204C15), it provided feasible solutions.
- Both CMSA variants were able to find the optimal solutions computed by CPLEX.
- In the case of the r202C15 instance, ADAPT_CMSA and ADAPT_CMSA_SETCOV were able to improve over the solution obtained by CPLEX by 0.15% and 36.17%, respectively.
- Furthermore, ADAPT_CMSA and ADAPT_CMSA_SETCOV improved the solution obtained by CPLEX by 1.51% in the case of the rc204C15 instance.

Table 4.10 Results for large-sized random EVRP-TW-SPD instances

Instance	ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	*m*	Best	Avg.	Time	*m*	Best	Avg.	Time
r101	**18**	19,633.80	19,939.79	653.66	**18**	19,640.60	19,661.15	678.64
r102	17	18,470.80	19,292.16	707.40	**16**	17,474.10	17,696.35	798.71
r103	15	16,296.50	17,050.75	711.23	**14**	15,280.30	15,306.17	639.83
r104	13	14,141.10	14,255.53	616.85	**12**	13,084.30	13,111.31	766.15
r105	15	16,389.20	17,212.83	680.24	**14**	15,471.30	16,346.10	611.98
r106	15	16,292.00	16,836.67	701.75	**14**	15,314.80	15,441.68	746.22
r107	13	14,168.90	15,016.67	680.99	**12**	13,140.10	13,669.50	783.82
r108	12	13,079.80	13,531.30	667.03	**11**	12,073.70	12,998.17	742.49
r109	14	15,237.30	15,674.51	759.64	**13**	14,220.80	14,468.12	744.57
r110	13	14,170.20	14,905.73	528.09	**12**	13,114.30	13,544.76	770.70
r111	**12**	13,144.20	14,584.19	696.71	**12**	13,148.80	13,965.98	716.12
r112	**12**	13,155.60	14,053.56	471.58	**12**	13,044.10	13,078.65	850.31
r201	**4**	5192.33	5216.92	720.44	**4**	5276.75	5363.04	187.38
r202	**3**	4250.70	5020.88	688.65	**3**	4193.33	4940.22	876.26
r203	**3**	3942.74	4352.52	868.05	**3**	3985.02	4060.18	822.50
r204	**3**	3820.72	3854.31	787.19	**3**	3793.76	3827.81	876.38
r205	**3**	4055.28	4124.64	731.94	**3**	4065.06	4126.45	360.47
r206	**3**	3978.10	4065.05	756.40	**3**	3991.44	4047.16	784.07
r207	**3**	3878.91	3910.07	659.85	**3**	3881.97	3918.29	878.20
r208	**3**	3791.27	3829.39	849.73	**3**	3732.80	3776.20	895.79
r209	**3**	3975.64	4015.90	665.83	**3**	3933.55	3977.24	765.83
r210	**3**	3920.37	3984.78	755.82	**3**	3926.79	3961.42	740.36
r211	**3**	3814.42	3893.38	825.95	**3**	3824.47	3857.62	799.45
average	8.83	9947.82	10,374.85	703.70	**8.43**	9548.35	9788.85	732.01

- Both CMSA variants demand significantly less computation time compared to CPLEX. To elaborate, while CPLEX took an average of 2965.35 seconds to identify its best solutions (please note that this information is not presented in the result tables), ADAPT_CMSA achieved this in 23.04 seconds, and ADAPT_CMSA_SETCOV accomplished it in a mere 21.07 seconds.

In summary, it could be said that the small-sized problem instances are not enough of a challenge to show many differences between the two CMSA variants. Therefore, we now turn to the analysis of the results concerning large-sized problem instances. Note that, in this case, the numerical results from Tables 4.9, 4.10, and 4.11 are accompanied by critical difference (CD) plots that provide information about the statistical significance of the results; see Sect. 1.2.3 on page 16 for a general description of CD plots. Note that, apart from the results of ADAPT_CMSA and ADAPT_CMSA_SETCOV, these CD plots also consider the results of probabilistic versions of the C&W savings heuristic (pC&W) and the insertion heuristic (pSI) which were applied in a multi-start fashion with the same

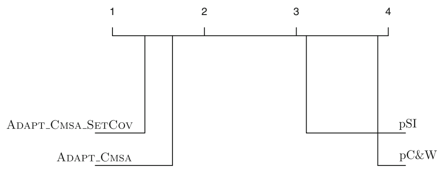
Table 4.11 Results for large-sized random clustered EVRP-TW-SPD instances

Instance	ADAPT_CMSA				ADAPT_CMSA_SETCOV			
Name	*m*	Best	Avg.	Time	*m*	Best	Avg.	Time
`rc101`	**16**	17,667.70	18,513.67	718.11	**16**	17,696.20	17,741.63	629.24
`rc102`	16	17,576.80	17,909.78	558.28	**15**	16,558.20	16,628.46	601.70
`rc103`	14	15,366.90	16,245.06	764.45	**13**	14,358.20	14,999.16	691.74
`rc104`	13	14,270.50	14,315.17	637.05	**12**	13,222.50	13,261.65	698.43
`rc105`	15	16,500.90	16,933.42	652.30	**14**	15,470.70	15,820.90	639.84
`rc106`	14	15,432.50	16,069.05	632.16	**13**	14,448.30	15,249.17	578.17
`rc107`	13	14,313.40	14,437.31	769.62	**12**	13,276.50	13,386.51	738.62
`rc108`	**12**	13,226.00	13,891.71	620.56	**12**	13,184.70	13,214.50	717.49
`rc201`	**4**	5504.77	5819.06	703.93	**4**	5617.75	5786.48	322.85
`rc202`	**4**	5324.64	5442.41	593.36	**4**	5436.63	5541.80	196.92
`rc203`	**4**	5109.88	5177.69	644.21	**4**	5086.44	5159.15	757.20
`rc204`	**3**	4036.49	4525.03	745.44	**3**	3962.88	4252.28	899.13
`rc205`	**4**	5260.14	5338.50	607.45	**4**	5285.41	5375.54	314.93
`rc206`	**4**	5234.55	5289.90	670.19	**4**	5210.57	5275.72	562.86
`rc207`	**3**	4150.60	4930.81	694.79	**3**	4197.81	4650.01	864.38
`rc208`	**3**	3977.50	4046.09	762.57	**3**	3920.17	4002.79	750.41
`average`	8.88	10,184.58	10,555.29	673.40	**8.50**	9808.31	10,021.61	622.74

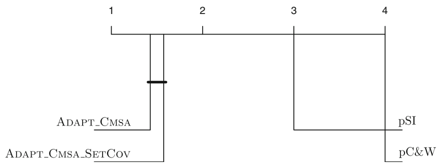
computation time limits as the CMSA variants. Their numerical results are not given in the Tables, first, for space reasons, and second, because they were much worse than those of the CMSA variants.

Main Observations Concerning Large-Sized Problem Instances

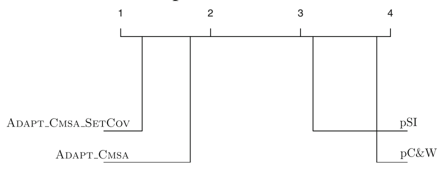
- First, the CD plot from Fig. 4.10a shows that, overall, ADAPT_CMSA_SETCOV outperforms ADAPT_CMSA with statistical significance.
- This claim also holds true for the random and the random-clustered instances; see the CD plots in Fig. 4.10c and d.
- However, from the results presented in Table 4.9, it seems difficult to come to a definite conclusion for clustered-type instances. ADAPT_CMSA_SETCOV seems to provide a slightly better performance both in terms of best and average results. Nevertheless, Fig. 4.10 shows no significant difference between the performance of ADAPT_CMSA_SETCOV and ADAPT_CMSA on clustered instances.
- It can also be observed that the performance of ADAPT_CMSA_SETCOV decreases in the context of instances with a long scheduling horizon (`c2*` `r2*` and `rc2*`); see Fig. 4.10f. Solutions for those instances include fewer routes and hence more customers per route when compared to the solutions for the instances with short scheduling horizons (`c1*` `r1*` and `rc1*`).
- Finally, both CMSA variants significantly outperform the probabilistic multi-start versions of the construction heuristics (pC&W and pSI) in all cases.



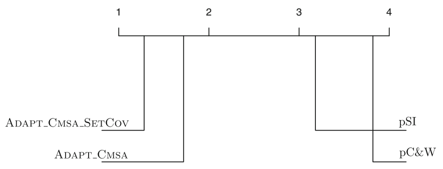
(a) All problem instances



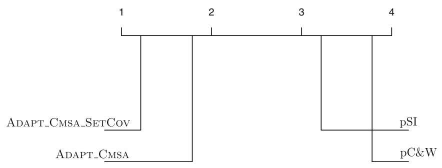
(b) Clustered instances



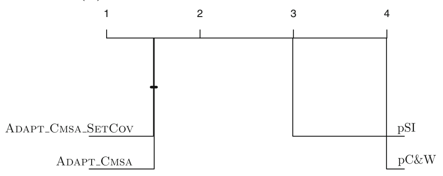
(c) Random instances



(d) Random-clustered instances



(e) Instances `r1*`, `c1*` and `rc1*`



(f) Instances `r2*`, `c2*` and `rc2*`

Fig. 4.10 Critical Difference (CD) plots concerning the results for large-sized EVRP-TW-SPD instances. The results in (**a**) consider all instances together, while the remaining plots display the results for instance subsets

In summary, both variants of Adapt-CMSA show a very satisfactory performance both in the context of small and large problem instances. Moreover, Adapt_Cmsa_SetCov shows superiority over Adapt_Cmsa, particularly in the context of random and random-clustered instances.

4.3.6.4 Performance Difference Between the Two EVRP-TW-SPD ILP Models

Finally, akin to the VSBP problem case, we aim to demonstrate the reasons for the superior performance of Adapt_Cmsa_SetCov over Adapt_Cmsa. To illustrate this, we once again create sub-instances of varying sizes, convert them into models $\texttt{ILP}^{\text{EVRP}}_{\text{std}}$ and $\texttt{ILP}^{\text{EVRP}}_{\text{setcov}}$, and subsequently solve them using CPLEX.

Specifically, we created 10 sub-instances through a probabilistic process involving the construction of 100, resp. 500, solutions for a small problem instance and 50, resp. 100, solutions for a large problem instance. In particular, instance `r202C15` was used for representing a small problem instance (with 15 customers), while `c101` with 100 customers was used for representing a large problem instance. In Fig. 4.11, radar charts illustrate the results obtained in these four scenarios.

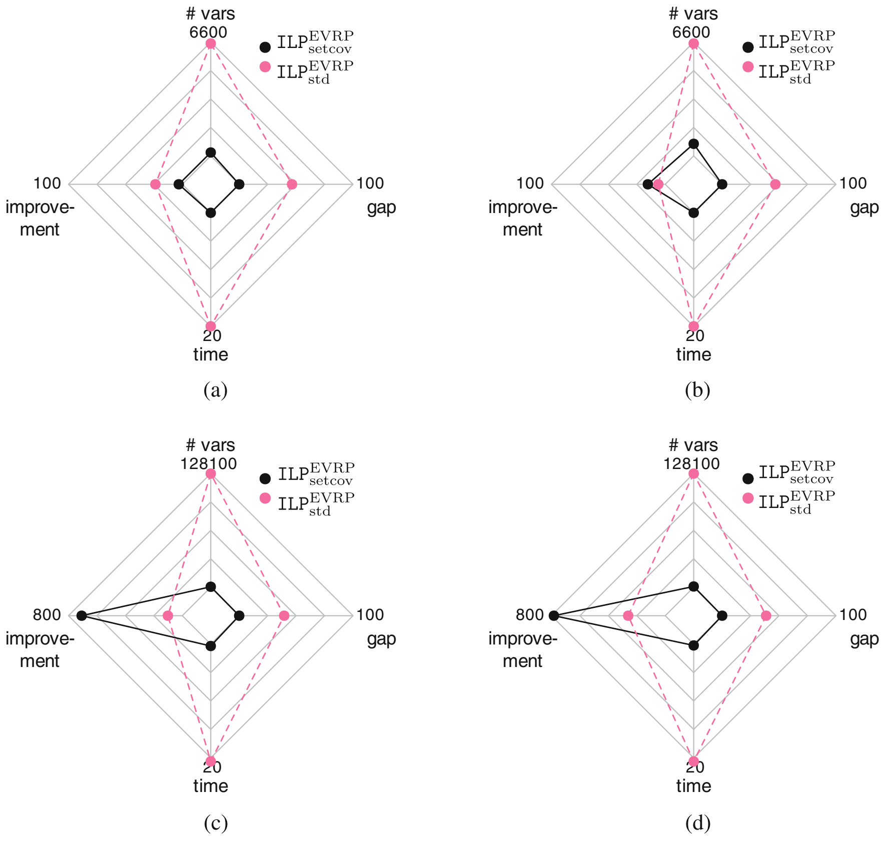


Fig. 4.11 Radar charts concerning the comparison of the two ILP models for the EVRP-TW-SPD problem applied to a small problem instance with 15 customers (see (**a**) and (**b**)), and to a large problem instance with 100 customers (see (**c**) and (**d**)). (**a**) Instance `r202C15` (15 customers), 100 solutions. (**b**) Instance `r202C15` (15 customers), 500 solutions. (**c**) Instance `c101` (100 customers), 50 solutions. (**d**) Instance `c101` (100 customers), 100 solutions

Remember that each radar plot shows four measures, averaged over the 10 sub-instances:

1. The number of variables in the ILP models of the sub-instances (top).
2. The relative MIP gap after CPLEX termination (right).
3. The computation time required by CPLEX (bottom).
4. The absolute improvement when comparing the result of solving the sub-instance with the best individual solution used to generate it.

It is important to note that the time limit for CPLEX was consistently set to 20 CPU seconds in all cases.

A promising ILP model would be expected to exhibit a substantial improvement with low values for the number of variables, the relative MIP gap, and the required time. The radar charts concerning the large problem instance (see Fig. 4.11c and

d) indicate that this is the case for model $\texttt{ILP}^{\text{EVRP}}_{\text{setcov}}$, while the opposite is actually the case for model $\texttt{ILP}^{\text{EVRP}}_{\text{std}}$. Especially the case of the small problem instance considering the lower number of solution constructions (see Fig. 4.11a) indicates that sub-instances must not be too small. Otherwise, there might not be many improvements to be found in the context of model $\texttt{ILP}^{\text{EVRP}}_{\text{setcov}}$.

4.3.6.5 STNWeb Graphics Concerning the EVRP-TW-SPD Results

Additionally, STNWeb graphics were produced for some examples of the obtained EVRP-TW-SPD results; see Sect. 1.2.2 on page 13 for a description of the STNWeb tool and the type of graphics that are produced. Figure 4.12 shows two typical cases.

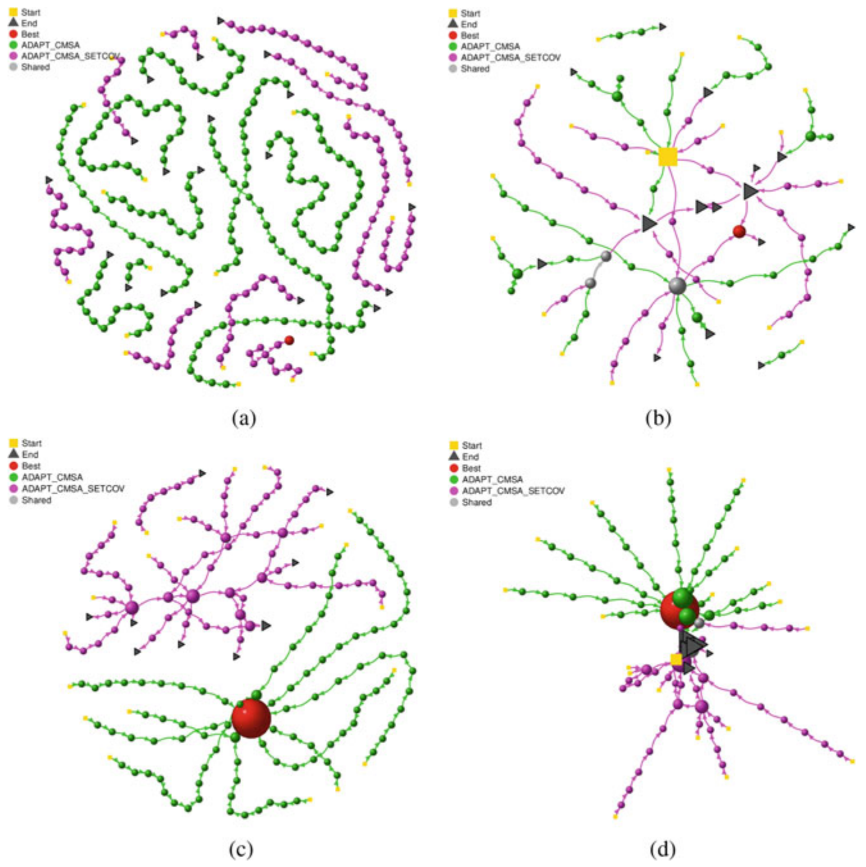


Fig. 4.12 STNWeb graphics. (**a**) and (**b**) show 10 runs of ADAPT_CMSA and ADAPT_CMSA_SETCOV for the random-clustered instance `rc106`, while (**c**) and (**d**) correspond to the clustered instance `c201`. While (**a**) and (**c**) depict the complete STNs, (**b**) and (**d**) show the same STNs after search space partitioning

The first one, consisting of the complete STN in Fig. 4.12a and the STN after search space partitioning in Fig. 4.12b, shows the case of a random-clustered instance (`rc106`) for which ADAPT_CMSA_SETCOV works better than ADAPT_CMSA. While the complete STN does not show any trajectory overlaps, the STN after search space partitioning clearly shows that the ADAPT_CMSA_SETCOV trajectories end up in the same area of the search space, while the ADAPT_CMSA trajectories have some overlaps with the ADAPT_CMSA_SETCOV trajectories, especially in early, resp. intermediate stages, of the search process. However, they simply stop earlier, before reaching the area with the best solutions.

The second example shows the complete STN (Fig. 4.12c) and the STN after search space partitioning (Fig. 4.12d) of a clustered instance (`c201`), for which ADAPT_CMSA works better than ADAPT_CMSA_SETCOV. The complete STN shows that all ADAPT_CMSA trajectories converge to the same best-found (and possibly optimal) solution. The STN after search space partitioning shows that ADAPT_CMSA is actually attracted by the same area of the search space. However, the algorithm is not able to find the very best solutions in that area.

4.4 Conclusions and Future Research Directions

In this chapter, we have explored the application of various CMSA variants to tackle two NP-hard combinatorial optimization problems. The first one was the Variable-Sized Bin Packing problem, followed by the Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickup and Delivery. Both optimization problems share a common characteristic: They can be formulated using an assignment-type integer linear program as well as a set-covering-based integer linear program. Both models were employed in identical CMSA algorithms for solving sub-instances at each iteration.

The results unequivocally demonstrate the superior performance of CMSA variants utilizing set-covering-based models over those employing standard assignment-type models. From our perspective, CMSA algorithms prove to be an ideal algorithmic framework for leveraging set-covering-based models in solving optimization problems of this nature. This preference arises because CMSA algorithms are less intricate and more straightforward to implement compared, for example, to column-generation approaches. Furthermore, CMSA algorithms possess the capability to explore search spaces, distinguishing them from simpler heuristic methods found in the literature designed to take profit from set-covering-based models.

At least two possible lines for future work might be envisaged. One line consists of the consolidation of the findings outlined in this chapter in the context of additional combinatorial optimization problems that can be modeled by set-covering-based models. Another possible line of work consists of the improvement of the CMSA algorithms presented in this work. In the context of the VSBP problem, for example, only the first solution construction approach that came to mind was implemented. Adding additional greedy heuristics for the construction step of CMSA could help to generate potentially different bins that, in combination with other bins, could help to find even better solutions. In this way, the obtained results might be improved in the few cases in which the proposed algorithm is not able to compete with the state-of-the-art variable neighborhood search technique from the literature. Also in the case of the electric vehicle routing problem, we see potential for improvement by adding additional solution construction techniques.

References

1. Akbay, M.A., Kalayci, C.B., Blum, C.: Application of CMSA to the electric vehicle routing problem with time windows, simultaneous pickup and deliveries, and partial vehicle charging. In: L. Di Gaspero, P. Festa, A. Nakib, M. Pavone (eds.) Proceedings of MIC 2022 – Metaheuristics International Conference, pp. 1–16. Springer International Publishing, Cham (2023)
2. Angelelli, E., Mansini, R.: The vehicle routing problem with time windows and simultaneous pick-up and delivery. In: Quantitative approaches to distribution logistics and supply chain management, pp. 249–267. Springer (2002)
3. Asghari, M., Mirzapour Al-e-hashem, S.M.J.: Green vehicle routing problem: A state-of-the-art review. International Journal of Production Economics **231**, 107899 (2021)
4. Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W., Vance, P.H.: Branch-and-price: Column generation for solving huge integer programs. Operations Research **46**(3), 316–329 (1998)
5. Cacchiani, V., Hemmelmayr, V.C., Tricoire, F.: A set-covering based heuristic algorithm for the periodic vehicle routing problem. Discrete Applied Mathematics **163**, 53–64 (2014)
6. Clarke, G., Wright, J.W.: Scheduling of vehicles from a central depot to a number of delivery points. Operations Research **12**(4), 568–581 (1964)
7. Crainic, T.G., Perboli, G., Rei, W., Tadei, R.: Efficient lower bounds and heuristics for the variable cost and size bin packing problem. Computers & Operations Research **38**(11), 1474–1482 (2011)
8. Desrochers, M., Soumis, F.: A column generation approach to the urban transit crew scheduling problem. Transportation Science **23**(1), 1–13 (1989)
9. Ekici, A.: A large neighborhood search algorithm and lower bounds for the variable-sized bin packing problem with conflicts. European Journal of Operational Research **308**(3), 1007–1020 (2023)
10. Fleszar, K.: A new MILP model and fast heuristics for the variable-sized bin packing problem with time windows. Computers & Industrial Engineering **175**, 108849 (2023)
11. Haouari, M., Serairi, M.: Heuristics for the variable sized bin-packing problem. Computers & Operations Research **36**(10), 2877–2884 (2009)
12. Hemmelmayr, V., Schmid, V., Blum, C.: Variable neighbourhood search for the variable sized bin packing problem. Computers & Operations Research **39**(5), 1097–1108 (2012)
13. Keskin, M., Çatay, B.: Partial recharge strategies for the electric vehicle routing problem with time windows. Transportation Research Part C: Emerging Technologies **65**, 111–127 (2016)
14. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
15. Machado, A.M., Mauri, G.R., Boeres, M.C.S., de Alvarenga Rosa, R.: A new hybrid matheuristic of GRASP and VNS based on constructive heuristics, set-covering and set-partitioning formulations applied to the capacitated vehicle routing problem. Expert Systems with Applications **184**, 115556 (2021)

16. Moghdani, R., Salimifard, K., Demir, E., Benyettou, A.: The green vehicle routing problem: A systematic literature review. Journal of Cleaner Production **279**, 123691 (2021)
17. Monaci, M., Toth, P.: A set-covering-based heuristic approach for bin-packing problems. INFORMS Journal on Computing **18**(1), 71–85 (2006)
18. Paessens, H.: The savings algorithm for the vehicle routing problem. European Journal of Operational Research **34**(3), 336–344 (1988)
19. Parker, M., Ryan, J.: A column generation algorithm for bandwidth packing. Telecommunication Systems **2**(1), 185–195 (1993)
20. Rochat, Y., Taillard, É.D.: Probabilistic diversification and intensification in local search for vehicle routing. Journal of Heuristics **1**(1), 147–167 (1995)
21. Salhi, S., Nagy, G.: A cluster insertion heuristic for single and multiple depot vehicle routing problems with backhauling. Journal of the Operational Research Society **50**(10), 1034–1042 (1999)
22. Schneider, M., Stenger, A., Goeke, D.: The electric vehicle-routing problem with time windows and recharging stations. Transportation Science **48**(4), 500–520 (2014)
23. Yellow, P.: A computational modification to the savings method of vehicle scheduling. Journal of the Operational Research Society **21**(2), 281–283 (1970)
24. Yilmaz, Y., Kalayci, C.B.: Variable neighborhood search algorithms to solve the electric vehicle routing problem with simultaneous pickup and delivery. Mathematics **10**(17), 3108 (2022)

Chapter 5
Application of CMSA in the Presence of Non-binary Variables

Abstract Up to this point, the applications of CMSA discussed in this book, as well as those found in the related literature, have focused on addressing combinatorial optimization problems that can be expressed through binary integer linear programming (ILP) formulations. Such problems represent an ideal scenario for CMSA, as sub-instances can be easily defined by fixing specific decision variables to certain values or excluding them altogether from the models. However, when confronted with a problem expressed through a more general ILP that incorporates discrete decision variables with non-binary domains, a notable challenge emerges. Unlike constraint programming solvers, for example, ILP solvers cannot handle non-contiguous domains, making it impossible to simply eliminate certain values from these domains. In this chapter, we present an illustration of CMSA applied to a combinatorial optimization problem naturally formulated as a non-binary ILP. Specifically, we make use of the Bounded Knapsack Problem with Conflicts.

5.1 Introduction

In this book, the following combinatorial optimization problems have been discussed so far: (1) the Minimum Dominating Set (MDS) problem in Chaps. 1 and 3, (2) the Far From Most String (FFMS) problem in Chaps. 2 and 3, (3) the Minimum Positive Influence Dominating Set (MPIDS) problem in Chap. 2, (4) the Variable-Sized Bin Packing (VSBP) problem in Chap. 4 and (5) the Electric Vehicle Routing Problem with Time Windows and Simultaneous Pickup and Delivery (EVRP-TW-SPD) in Chap. 4. All these problems share an important property. They are naturally expressed in terms of binary ILPs, that is, ILPs that only consist of discrete decision variables with binary domains. Moreover, sub-instances to these problems can always be expressed through ILP models in which some decision variables are left free, and others are fixed to one of the two values from their domains. Note that whenever one value is removed from the domain of a binary decision variable, this decision variable can immediately be fixed to the remaining value. In the context of applications to problems such as the Multi-Dimensional Knapsack Problem

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3_5

(see [1]) binary decision variables might even be removed from the corresponding ILP models.

Now consider that we would like to apply CMSA to a combinatorial optimization problem that is naturally expressed through a non-binary ILP model, that is, an ILP model in which (at least some of) the variables have discrete, non-binary domains. In this case, we might run into the following problem.

Application of CMSA to Non-binary Problem: Potential Problem

Imagine that the considered discrete optimization problem includes a decision variable x_i with domain $D_i := \{1, 2, 3\}$. Moreover, imagine that we apply standard CMSA using the generic way of defining the solution components, resulting in a complete set C of solution components containing a solution component c_i^j for each variable x_i and domain value $j \in D_i$. Further, imagine that we are in the first iteration of CMSA and that $n_a = 2$ solutions are constructed. One in which $x_i = 1$, and another one in which $x_i = 3$. That is, the allowed values for x_i for solving the corresponding sub-instance are $\{1, 3\}$. However, this reduced domain is non-contiguous, and **ILP solvers are not able to handle non-contiguous variable domains**.

When dealing with a mixed problem, which includes both binary and non-binary variables, one can consider the following differentiation. If the count of non-binary variables is relatively small compared to the number of binary variables, it may be reasonable to disregard the non-binary variables and rely exclusively on the binary decision variables for the functioning of CMSA. However, if the number of non-binary variables is substantial, this approach becomes impractical. Under such circumstances, one might contemplate the binarization of the ILP, and that is precisely the methodology elucidated in this chapter. Specifically, we illustrate the implementation of standard CMSA on a combinatorial optimization problem termed the Bounded Knapsack Problem with Conflicts (BKPWC). This variant represents an extension of the widely recognized Bounded Knapsack Problem (BKP) [9].

5.2 The Bounded Knapsack Problem with Conflicts

The Bounded Knapsack Problem (BKP) was first presented in [9] as an extension of the classical 0-1 Knapsack Problem (0-1 KP) [7], and later treated, for example, in [5, 6]. Hereby, a knapsack with capacity C_{cap} has to be filled with items from a finite set of n items. Each item j has a profit p_j and a weight w_j. Moreover, $m_j > 0$ copies of each item j are maximally available. To obtain a candidate solution, it has to be decided how many (of maximally m_j) copies of each item j are to be placed into the knapsack. Note that this last aspect is the extension in comparison to the classical 0-1 KP, where each item may, or may not, be placed in the knapsack.

Finally, a candidate solution is feasible if the sum of the weights of all item copies added to the knapsack is maximally C_{cap}, which is the capacity of the knapsack. The objective function value of a feasible solution is calculated as the sum of the profits of all item copies added to the knapsack.

In this chapter, we consider an extension of the BKP called the BKP with Conflicts (BKPWC). This problem variant is inspired by a recent trend in the packing literature that concerns the consideration of conflicts between items. Examples of related problems include the Bin Packing Problem with Conflicts [4], the Quadratic Multi-knapsack Problem with Conflicts and Balance Constraints [8], and the Variable-Sized Bin Packing Problem with Conflicts and Item Fragmentation [3]. In the context of the BKPWC, when two items i and j are in conflict, the knapsack may only contain copies of one of the two items. To model item conflicts, we introduce a so-called conflict graph $G = (V, E)$ where an edge (i, j) exists in E if, and only if, items i and j are in conflict.

The BKPWC can be modelled as an ILP in the following way.

$$\max \quad \sum_{j=1}^{n} p_j \cdot x_j \tag{5.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j \cdot x_j \leq C_{\text{cap}} \tag{5.2}$$

$$x_j \leq M \cdot y_j \quad \forall\, j = 1, \ldots, n \tag{5.3}$$

$$y_i + y_j \leq 1 \quad \forall\, (i, j) \in E \tag{5.4}$$

$$y_j \in \{0, 1\}$$

$$x_j \in \{0, \ldots, m_j\}$$

Note that two sets of variables are utilized in this model. The integer variables $0 \leq x_j \leq m_j$ indicate the number of copies of each item that are placed into the knapsack. Moreover, the binary variables $0 \leq y_j \leq 1$ are so-called indicator variables. Constraints (5.3) force a variable y_j to assume value one if at least one copy of item j is placed into the knapsack. Hereby, M is constant set to the maximum of all m_j-values, that is, $M := \max\{m_j \mid j = 1, \ldots, n\}$. Finally, constraint (5.2) is the capacity constraint, while constraints (5.4) are the conflict constraints.

5.2.1 Converting the BKPWC ILP to a Binary Program

One of the most popular methods for converting a general ILP into a binary ILP is the following one; see also page 60 of [2]. Let x_j be a general discrete variable with domain $D_j := \{d_1, d_2, \ldots, d_{k-1}, d_k\}$ containing k values. Hereby, each value

$d_r \in D_j$ is an integer value, that is, $d_r \in \mathbb{Z}$. Without loss of generality, let us assume that $d_1 \leq d_2 \leq \ldots \leq d_{k-1} \leq d_k$. Moreover, note that the values in D_j may be non-contiguous, that is, it does not necessarily hold that $d_{r+1} = d_r + 1$ for all $r \in \{1, \ldots, k-1\}$. Then, the discrete variable x_j can be replaced in the given ILP by a set of k binary variables $\{x_{j,r} \mid r = 1, \ldots, k\}$. Moreover, each occurrence of x_j in the given ILP is replaced by the following sum:

$$\sum_{r=1}^{k} d_r \cdot x_{j,r} \tag{5.5}$$

In addition, the following constraint is added to the ILP:

$$\sum_{r=1}^{k} x_{j,r} = 1 \tag{5.6}$$

This constraint enforces that the value of exactly one of the binary variables that are used to replace the discrete variable x_j is set to 1.

Making use of these rules, the ILP model of the BKPWC can be converted to the following binary ILP:

$$\max \quad \sum_{j=1}^{n} \left(p_j \cdot \sum_{r=0}^{m_j} x_{j,r} \right) \tag{5.7}$$

$$\text{subject to} \quad \sum_{j=1}^{n} \left(w_j \cdot \sum_{r=0}^{m_j} x_{j,r} \right) \leq C_{\text{cap}} \tag{5.8}$$

$$\sum_{r=0}^{m_j} x_{j,r} \leq M \cdot y_j \quad \forall\, j = 1, \ldots, n \tag{5.9}$$

$$y_i + y_j \leq 1 \quad \forall\, (i, j) \in E \tag{5.10}$$

$$\sum_{r=0}^{m_j} x_{j,r} = 1 \quad \forall\, j = 1, \ldots, n \tag{5.11}$$

$$y_j \in \{0, 1\}$$

$$x_{j,r} \in \{0, 1\} \quad \forall\, j = 1, \ldots, n \text{ and } r = 0, \ldots, m_j$$

5.3 Application of CMSA to the BKPWC

Next, the application of standard CMSA from Sect. 1.3.1 of Chap. 1 to the BKPWC is described. Our implementation will be based on the intuitively defined set of solution components C which consists of a solution component $c_{j,r}$ for each combination of a decision variable x_j with a domain value $r \in D_j = \{0, \ldots, m_j\}$. In other words, when a component $c_{j,r}$ is present in sub-instance C' this means that domain value r for variable x_j is an allowed choice in the ILP model that corresponds to sub-instance C'. In the following, we describe, first, how solutions are generated in a probabilistic way at each iteration of CMSA_INT. Second, the way of generating the ILP model corresponding to sub-instance C' is described.

5.3.1 Probabilistic Solution Construction

The probabilistic construction of a valid BKPWC solution is pseudo-coded in Algorithm 5.1. First, we assume—without loss of generality—that the items are ordered such that $\frac{p_j}{w_j} \geq \frac{p_{j+1}}{w_{j+1}}$ for all $j = 1, \ldots, n-1$. The construction of a solution consists of deciding, for each item, the number of item copies to be placed into the knapsack. After initializing the solution S to the empty set, and the remaining capacity (s_{cap}) to the total capacity of the knapsack (C_{cap}), the first item to be handled is chosen uniformly at random; see line 4 of Algorithm 5.1. After selecting the number $l \in \{0, \ldots, m_j\}$ of copies of item j to be placed in the knapsack in function SelectNumberOfCopies(j, m_j, s_{cap}) (explained below) and adding the corresponding solution component $c_{j,l}$ to solution S, set I is updated by removing

Algorithm 5.1: BKPWC solution construction

```
1: input: the items $j = 1, \ldots, n$
2: $S := \emptyset$, $s_{\text{cap}} := C_{\text{cap}}$
3: $I := \{1, \ldots, n\}$ {Set of all items}
4: Choose $j \in I$ uniformly at random
5: l := SelectNumberOfCopies($j, m_j, s_{\text{cap}}$)
6: $S := S \cup \{c_{j,l}\}$, $s_{\text{cap}} := s_{\text{cap}} - l \cdot w_j$
7: $I := I \setminus \{j\} \setminus \{k \in \{1, \ldots, n\} \mid (i,k) \in E$ or $w_k > s_{\text{cap}}\}$
8: while $I \neq \emptyset$ do
9:     $j := \min\{i \mid i \in I\}$
10:    l := SelectNumberOfCopies($j, m_j, s_{\text{cap}}$)
11:    $S := S \cup \{c_{j,l}\}$, $s_{\text{cap}} := s_{\text{cap}} - l \cdot w_j$
12:    $I := I \setminus \{j\} \setminus \{k \in \{1, \ldots, n\} \mid (i,k) \in E$ or $w_k > s_{\text{cap}}\}$
13: end while
14: for $j \in \{1, \ldots, n\}$ s.t. no component $c_{j,*} \in S$ exists do
15:    $S := S \cup \{c_{j,0}\}$
16: end for
17: output: $S$
```

item j, all items that are in conflict with j, and all items of which not even a single copy would fit into the incumbent partial solution; see line 7. Then, at each further step of the solution construction procedure, first, the lowest-index item is chosen from I. Subsequently, the number of copies of j to be placed in the knapsack is selected (line 10), the corresponding solution component is added to S and I is updated as described above. This procedure ends as soon as I is empty. Finally, for all untreated items j, solution component $c_{j,0}$ is added to S, which indicates that S does not contain a single copy of these items; see lines 14–16.

The remaining aspect of the solution construction is the choice of the number of copies of selected items in function SelectNumberOfCopies(j, m_j, s_{cap}) which is executed in lines 5 and 10 of Algorithm 5.1. First, the maximum number of copies ($n_{\max}$) of j that would fit into S is determined. More specifically, $n_{\max}$ is set to the largest integer value such that $n_{\max} \cdot w_j \leq s_{\text{cap}}$. Moreover, $n_{\max}$ is possibly further reduced due to the maximum number of available copies of j: $n_{\max} := \min\{n_{\max}, m_j\}$. Then, a value v is sampled from a normal distribution $\mathcal{N}(0, \sigma^2)$, that is, a normal distribution with zero as mean and σ as standard deviation. In case $v < 0$, v is multiplied with -1 in order to obtain a positive value. Finally, l (the number of copies of item j to be placed in the knapsack) is defined as follows:

$$l := n_{\max} - \lfloor v \rfloor \ , \tag{5.12}$$

where $\lfloor v \rfloor$ refers to the integer part of v. Note that, in this way of selecting l, the highest chance is given to the maximum number of copies. Obviously, this chance decreases with an increasing value of σ, which is a parameter of our algorithm. This way of selecting l is reasonable as it was shown by studying optimal BKPWC solutions, that the number of copies of most of the selected items is actually the maximum possible one.

5.3.2 Sub-instance Solving

Sub-instance C' is, at each iteration of CMSA_INT, tackled by solving an ILP model on the basis of the binarized ILP model for the BKPWC presented in Sect. 5.2.1. In fact, the ILP model corresponding to a sub-instance C' is obtained from the ILP model in Sect. 5.2.1 by simply removing all terms including a variable $x_{j,r}$ whose solution component $c_{j,r}$ is not found in C'. This is for the following reasons. If a solution component $c_{j,r}$ does not form part of C', choosing r copies of item j for being placed into the knapsack is not an option. Therefore, variable $x_{j,r}$ must be fixed to zero, which—in the case of this ILP—can be handled by removing all terms involving $x_{j,r}$ from the ILP.

5.4 Experimental Evaluation

The following algorithms are included in the experimental evaluation presented in this section:

1. GREEDY: A greedy heuristic obtained by executing the approach from Algorithm 5.1 in a deterministic way by choosing the next item always as the lowest-index item not treated yet, and by always selecting the maximum number of item copies that fit into the incumbent partial solution.
2. CPLEX: Application of CPLEX 22.1 to each considered problem instance with the default parameter values of CPLEX. The employed ILP model is the natural one with integer variables from page 143.
3. CPLEX_BIN: Application of CPLEX 22.1 to each considered problem instance with the default parameter values of CPLEX. The employed ILP model is the binarized one exclusively utilizing binary variables from Sect. 5.2.1.
4. CMSA: The standard CMSA algorithm, based on the intuitive way of defining the set of solution components as outlined in this chapter.

Note that CPLEX 22.1 is used—both in standalone mode (CPLEX and CPLEX_BIN) and within CMSA—in single-threaded mode. For conducting the experiments we used the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

5.4.1 Problem Instances

A large set of 1800 problem instances was generated as suggested in [9]. In particular, these instances differ in the number of items ($n \in \{500, 1000, 5000\}$), the knapsack capacity of an instance ($\text{cap}_{\text{type}} \in \{10, 20, 30, 40, 50\}$), and in the number of conflicts ($\text{conf}_{\text{type}} \in \{1, 3, 5, 10\}$). Note that both cap_{type} and $\text{conf}_{\text{type}}$ refer to percentages. For each combination of these three instance parameters, 30 problem instances were randomly generated in the following way.

For each item j, first, a profit p_j is randomly selected from $\{1, 2, \ldots, 99, 100\}$. Second, a non-correlated weight w_j is chosen from the same range, that is, from $\{1, 2, \ldots, 99, 100\}$. Next, a maximal number of item copies m_j is selected uniformly at random from $\{5, 6, 7, 8, 9, 10\}$. Then, the knapsack capacity of item j is fixed as follows:

$$C_{\text{cap}} := \left\lfloor \frac{\left(\sum_{j=1}^{n} m_j \cdot w_j\right) \cdot \text{cap}_{\text{type}}}{100} \right\rfloor \tag{5.13}$$

Finally, of all possible conflicts between two items i and j, $\text{conf}_{\text{type}}\%$ are randomly selected and added to the problem instance.

5.4.2 Parameter Tuning

As for all other experimental studies described in this book, the `irace` tool—generally described in Sect. 1.2.1 on page 12—was used for tuning the parameters of CMSA. The following is the list of parameters considered for parameter tuning:

- n_{a}: The number of solution constructions per CMSA iteration.
- age_{max}: The maximum age solution components may reach before being removed from the sub-instance C'.
- t_{ILP}: The CPU time limit (in seconds) for the application of CPLEX for solving a sub-instance C'.
- $\text{cplex}_{\text{emphasis}}$: Use of heuristic emphasis in CPLEX.
- $\text{cplex}_{\text{warmstart}}$: Use of warm start in CPLEX.
- $\text{cplex}_{\text{abort}}$: Aborting CPLEX whenever then best-so-far solution S^{ILP} is improved.
- σ: The standard deviation of the normal distribution involved in the choice of the number of copies of an item to be placed into the knapsack. This parameter was described in Sect. 5.3.1.

CMSA was tuned depending on problem instance size in terms of the number of items. In other words, CMSA was tuned for items with 500, 1000, and 5000 items separately. As tuning instances, additional problem instances were generated. More specifically, for each combination of n, cap_{type}, and $\text{conf}_{\text{type}}$, exactly one tuning instance was generated. This makes a total of 20 tuning instances per tuning run. As a computation time limit, 100 CPU seconds was used for all instances with $n = 500$ items, 250 CPU seconds for all instances with $n = 1000$ items, and 600 CPU seconds for all instances with $n = 5000$ items. Note also that `irace` was given a budget of 2000 algorithm runs for each tuning application.

The results of the tuning process are provided in Table 5.1, together with the parameter domains considered. The following observations are worth to be discussed. First, the value of σ is very low in all cases. This indicates that the chance of choosing a number of copies of an item to be placed in the knapsack which is different to the maximum possible one should be very low. Another interesting observation concerns the number of solution constructions per iteration. While this number is rather high for instances with 500 and 1000 items, it drops significantly in the case of instances with 5000 items. This happens, most probably, because CPLEX reaches its limits when problem instances of that size are considered. As a

Table 5.1 Parameters, domains and tuning results of CMSA for the BKPCW

Parameter	Domain	500 items	1000 items	5000 items
n_a	$\{1, \ldots, 200\}$	172	169	14
age_{max}	$\{1, \ldots, 10\}$	6	1	3
t_{ILP}	$\{1, \ldots, 20\}$	5	7	13
$\text{cplex}_{\text{emphasis}}$	{true, false}	false	true	false
$\text{cplex}_{\text{warmstart}}$	{true, false}	true	true	true
$\text{cplex}_{\text{abort}}$	{true, false}	false	true	false
σ	$[1, 10]$	1.12	1.07	1.59

final comment, it always seems beneficial to make use of the warm-start feature of CPLEX for solving sub-instances.

5.4.3 Results

All four algorithmic techniques (GREEDY, CPLEX, CPLEX_BIN and CMSA) were applied exactly once to each of the problem instances from the benchmark set. The computation time limit for CPLEX, CPLEX_BIN and CMSA was the same as the one used for tuning (see previous section). The results are shown in the form of box plots in Figs. 5.1, 5.2, and 5.3. Note that there is exactly one graphic for each problem instance size. Each of these graphics contains a 4×5 grid of box plots. Hereby, the rows present the results (from top to bottom) for problem instances with an increasing knapsack capacity, and the columns (from left to right) present the results for problem instances with an increasing number of conflicts.

To be able to support the analysis of the results with claims about their statistical significance, CD plots are provided as in all other experimental evaluations presented in this book. These plots are provided in Fig. 5.4. In particular, the plot in Fig. 5.4a contains statistics for all problem instances together. The next three plots (Fig. 5.4b–d) provide information for the problem instances grouped by size (number of items). Finally, Fig. 5.4e–h show the results for the benchmark set grouped by the number of conflicts.

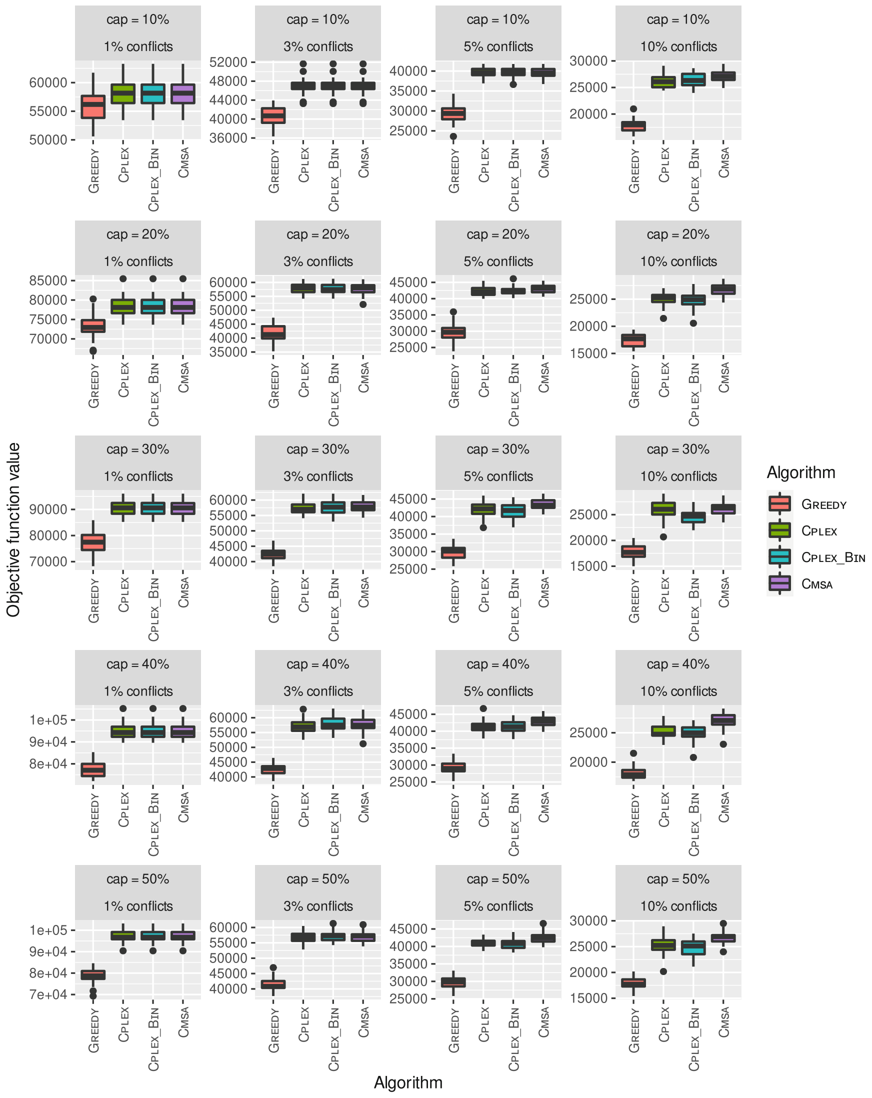


Fig. 5.1 BKPWC results for instances with 500 items

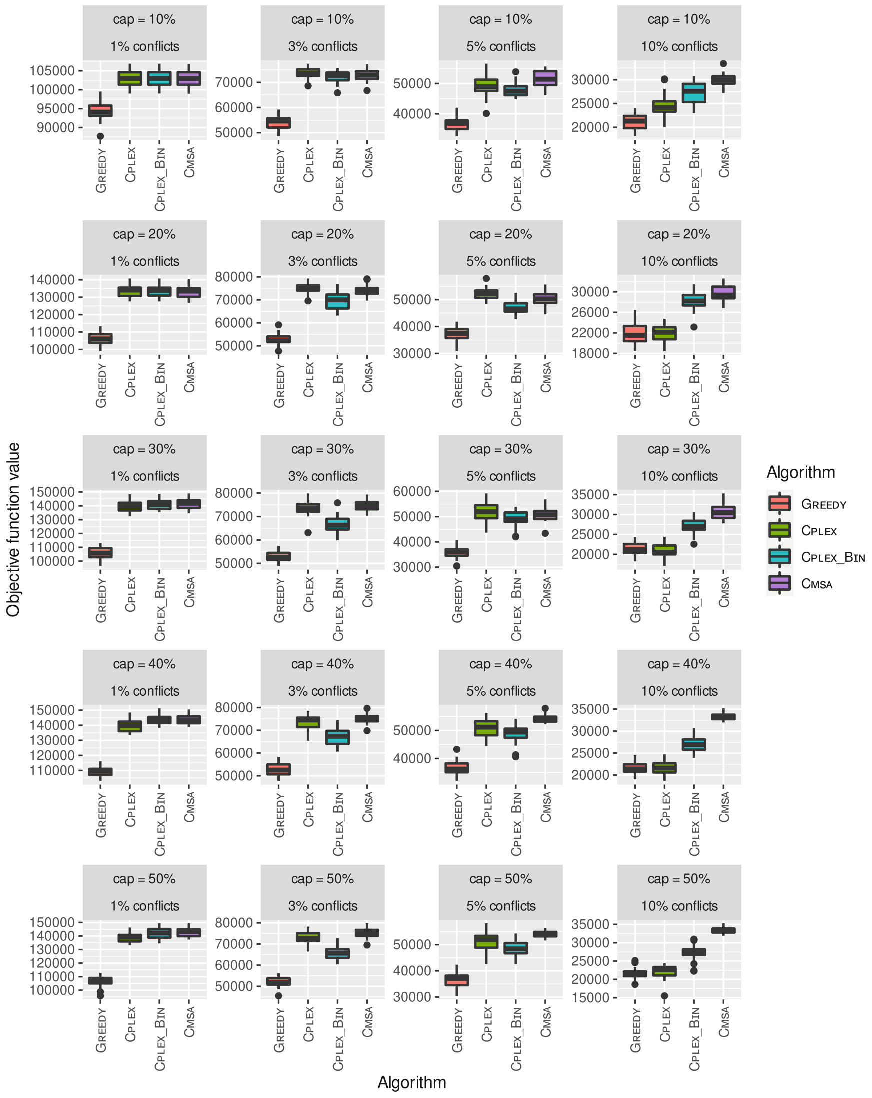


Fig. 5.2 BKPWC results for instances with 1000 items

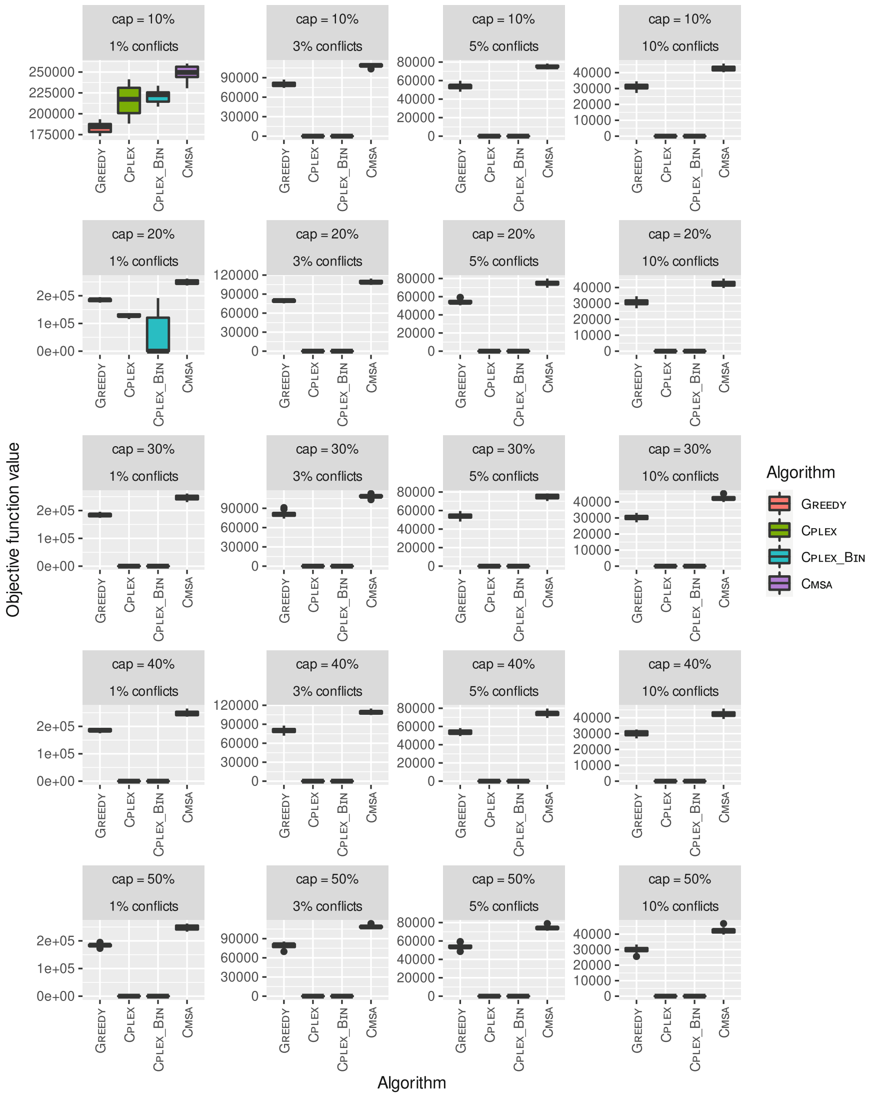


Fig. 5.3 BKPWC results for instances with 5000 items

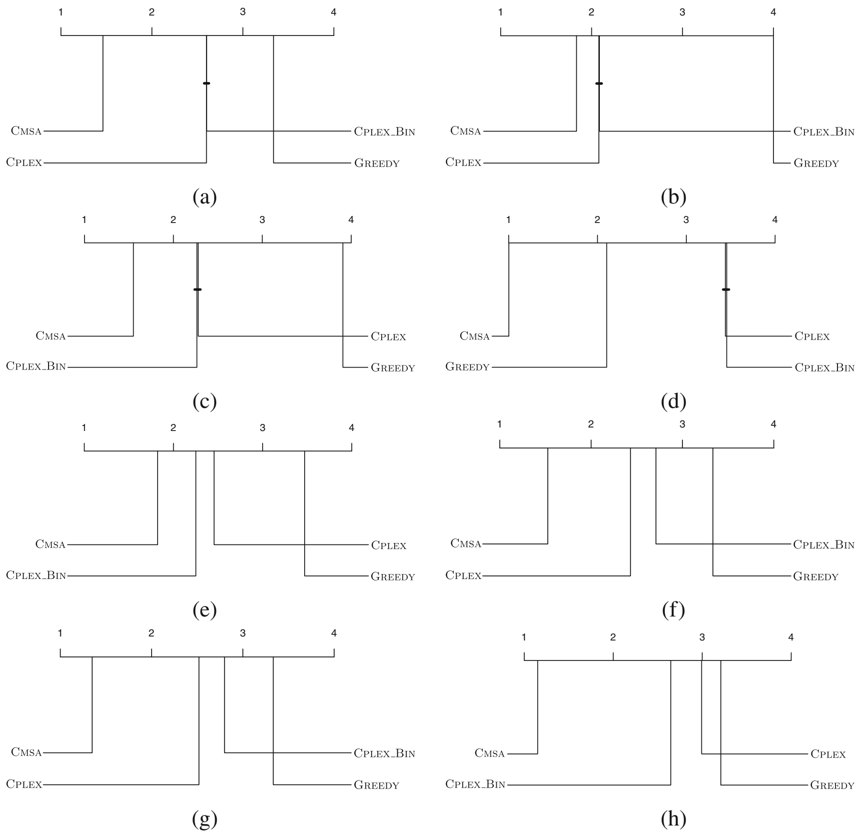


Fig. 5.4 Critical Difference (CD) plots concerning BKPCW results. (**a**) All instances. (**b**) Instances with 500 items. (**c**) Instances with 1000 items. (**d**) Instances with 5000 items. (**e**) Instances with $\text{conf}_{\text{type}} = 1$. (**f**) Instances with $\text{conf}_{\text{type}} = 3$. (**g**) Instances with $\text{conf}_{\text{type}} = 5$. (**h**) Instances with $\text{conf}_{\text{type}} = 10$

Main Observations Concerning the BKPWC Results

1. First, and most importantly, CMSA generally outperforms both CPLEX variants (CPLEX and CPLEX_BIN) and GREEDY.
2. Again—as, for example, in the case of the MDS problem presented in Chap. 1—the comparison with the CPLEX variants shows exactly the pattern that one would expect from the comparison of a hybrid technique with an exact technique: when problem instances are rather easy to be solved with CPLEX, CMSA performs on a comparable level. In contrast, when problem instances become large and/or difficult to solve, CMSA clearly outperforms both CPLEX variants.
3. In general, problem instances become more difficult for both CPLEX variants with growing size, with growing capacity, and with a growing number of conflicts. In fact, for most of the problem instances with 5000 items, both CPLEX variants can only find the trivial solution obtained by choosing to place no item copies at all into the knapsack.[1]
4. The CD plots for sub-groups of the benchmark set allow to make the following interesting observation. When grouping by instance size—see Fig. 5.4b–d—the performances of the two CPLEX variants are basically indistinguishable. However, when grouping the instances with respect to the number of conflicts (see Fig. 5.4e–h) it shows that CPLEX_BIN outperforms CPLEX with statistical significance for instances with the lowest and the highest number of conflicts, while the opposite is the case for instances with the two medium levels of conflicts. However, at this moment we do not have any explanation for this phenomenon.

5.5 Conclusions and Further Research Directions

As this chapter has shown, there is no problem in applying CMSA to general ILPs. And in the example case that was studied—that is, the Bounded Knapsack Problem With Conflicts—this certainly makes sense. However, note that the transformation of an ILP to a binary problem increases both the number of variables and the number of constraints. Depending on the problem, respectively the considered problem instance, this increase might be significant. Imagine problem instances of the studied knapsack problem with a much higher upper bound for the number of item copies, for example. Therefore, such a transformation might not always be that well-behaved as shown in this chapter. In other words, the algorithm designer has to decide from case to case whether a transformation to a binary problem with the

[1] This can be seen by the fact that the solutions generated by the CPLEX variants have value zero.

subsequent application of CMSA makes sense. Moreover, many practically relevant optimization problems can be expressed in terms of mixed ILP models, involving both integer and binary variables, possibly in addition to continuous variables. In those cases, CMSA might be applied, for example, solely considering the binary variables. However, this strongly depends on the number of binary variables in comparison to the integer variables. In general, the application of CMSA to mixed ILPs might be a promising line for future research.

References

1. Blum, C., Ochoa, G.: A comparative analysis of two matheuristics by means of merged local optima networks. European Journal of Operational Research **290**(1), 36–56 (2021)
2. Chen, D.S., Batson, R.G., Dang, Y.: Applied integer programming: modeling and solution. John Wiley & Sons (2011)
3. Ekici, A.: Variable-sized bin packing problem with conflicts and item fragmentation. Computers & Industrial Engineering **163**, 107844 (2022)
4. Gendreau, M., Laporte, G., Semet, F.: Heuristics and lower bounds for the bin packing problem with conflicts. Computers & Operations Research **31**(3), 347–358 (2004)
5. Kellerer, H., Pferschy, U., Pisinger, D.: The Bounded Knapsack Problem, pp. 185–209. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
6. Li, Y., He, Y., Li, H., Guo, X., Li, Z.: A binary particle swarm optimization for solving the bounded knapsack problem. In: H. Peng, C. Deng, Z. Wu, Y. Liu (eds.) Computational Intelligence and Intelligent Systems, pp. 50–60. Springer Singapore, Singapore (2019)
7. Martello, S., Toth, P.: A new algorithm for the 0-1 knapsack problem. Management Science **34**(5), 633–644 (1988)
8. Olivier, P., Lodi, A., Pesant, G.: The quadratic multiknapsack problem with conflicts and balance constraints. INFORMS Journal on Computing **33**(3), 949–962 (2021)
9. Pisinger, D.: A minimal algorithm for the bounded knapsack problem. INFORMS Journal on Computing **12**(1), 75–82 (2000)

Chapter 6
Additional Research Lines Concerning CMSA

Abstract The main chapters of this book were devoted to significant research endeavors within the CMSA framework. Conversely, this concluding chapter provides a brief overview of research directions related to CMSA that have either received limited exploration so far or are presently under investigation. Specifically, it introduces a general CMSA approach for binary integer linear programming models. In this context, CMSA is employed to address integer linear programming models without any identification of the modeled problem. Furthermore, the chapter discusses a study where a metaheuristic is utilized instead of an integer linear programming solver for sub-instance solving. Following an examination that elucidates the relationship between large neighborhood search and CMSA, the chapter wraps up by underscoring promising avenues for future research.

6.1 A Problem-Agnostic CMSA for Binary Problems

One potential drawback of CMSA lies in the necessity for a problem-specific approach to probabilistically generate solutions during the solution construction step for the optimization problem under consideration. However, sometimes a well-working heuristic might not be available. Therefore, the authors of [5] attempted to develop a problem-agnostic CMSA for the application to general binary ILPs (BIPs) that can be expressed in the following way:

$$\min\{\mathbf{c}^T\mathbf{x} : A\mathbf{x} \leq \mathbf{b}, x_j \in \{0,1\}\ \forall j = 1,\ldots,n\} \tag{6.1}$$

where A is an $m \times n$ matrix, $\mathbf{b}$ is the right-hand-side vector of size m, $\mathbf{c}$ is a cost vector, and $\mathbf{x}$ is the vector of n binary decision variables. Note that m is the number of constraints of this BIP. This type of problem is generic enough to model a wide range of combinatorial optimization problems, including all optimization problems considered in this book. In addition, a myriad of applications are listed, for example, in the MIPLIB 2010 and 2017 collections of problem instances [12, 16].

For the application of a problem-agnostic CMSA to such a general BIP, the main challenge is to find a way for the fast production of (possibly) feasible solutions,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3_6

without knowing the exact nature of the problem and without any knowledge about the structure of feasible solutions. In this context, note that the necessity of quickly identifying feasible solutions to general ILPs is important also for ILP solvers such as CPLEX and Gurobi. Therefore, the research community on mathematical programming has put quite some effort into this issue. Over the years, several methods have been proposed to try to produce feasible solutions to general ILPs. One of the best-known approaches is the so-called *feasibility pump* [3, 10, 11]. However, the authors of [5] decided for a faster mechanism based on linear relaxation solving and on a simple constraint programming (CP) tool, as explained below.

6.1.1 *Application of* CMSA_GEN

The authors applied an extension of standard CMSA based on the generic way of defining the solution components (CMSA_GEN). Remember that this CMSA variant was outlined in Sect. 1.4.2 of Chap. 1 of this book. For this CMSA variant applied to BIPs, the set C of generic solution components will contain for each binary variable x_i two solution components:

1. Component c_j^0: corresponding to $x_j = 0$.
2. Component c_j^1: corresponding to $x_j = 1$.

A solution to the given BIP is any binary vector $\mathbf{s}$ that fulfills the constraints from Eq. (6.1). In case a solution $\mathbf{s}$ to the problem is characterized by $s_i = 0$ (which means that x_i is set to zero), the corresponding CMSA-solution S (defined as a set of solution components) contains component c_i^0. Similarly, if $s_i = 1$ in a solution $\mathbf{s}$, then S contains component c_i^1. The complete, generic set of solution components C is therefore defined as follows:

$$C := \{c_1^0, \ldots, c_n^0, c_1^1, \ldots, c_n^1\} \ , \tag{6.2}$$

where n is the number of binary variables in the BIP. Note that a feasible CMSA-solution S contains exactly n solution components: $\{c_j^* \mid j = 1, \ldots, n\}$, that is, one component per variable of the BIP.

As a final technical clarification, it is important to note that, for the rest of this section, we extend the objective function of the addressed BIP by considering that $f(\emptyset) := \infty$. Moreover, in the following, we will use both the vector notation of a solution ($\mathbf{s}$) and the CMSA-based notation of a solution in terms of a set of solution components (S) interchangeably. It is also important to remark that the optimization goal for the function f is considered to be minimization.

6.1.1.1 Before the Start of CMSA_GEN

Before starting with the first CMSA_GEN iteration, the node heuristic of CPLEX is used to obtain the first feasible solution. If, in this way, a feasible solution can be obtained, it is stored in $\mathbf{s}^{\mathrm{bsf}}$, which is the vector version of the best-so-far CMSA-solution S^{bsf}. Otherwise, S^{bsf} is set to $\emptyset$, as usual, and the LP relaxation of the given BIP is solved. However, in order not to spend too much computation time on this step, a computation time limit of t^{LP} seconds is applied. After this, the possibly optimal solution of the LP relaxation is stored in vector $\mathbf{x}^{\mathrm{LP}}$.

6.1.1.2 Solution Construction

Whenever function ProbabilisticSolutionGeneration(C) is called (see line 8 of Algorithm 1.1 on page 19 of Chap. 1), the following is done. First, a so-called sampling vector $\mathbf{x}^{\mathrm{samp}}$ for sampling new (possibly feasible) solutions by randomized rounding is generated. If $S^{\mathrm{bsf}} \neq \emptyset$, $\mathbf{x}^{\mathrm{samp}}$ is generated based on the vector-version $\mathbf{s}^{\mathrm{bsf}}$ of S^{bsf} and a so-called *determinism rate* $0 < d_{\mathrm{rate}} < 0.5$ as follows:

$$x_j^{\mathrm{samp}} = \begin{cases} d_{\mathrm{rate}} & \text{if } s_j^{\mathrm{bsf}} = 0 \\ 1 - d_{\mathrm{rate}} & \text{if } s_j^{\mathrm{bsf}} = 1 \end{cases}$$

for all $j = 1, \ldots, n$. In case $S^{\mathrm{bsf}} = \emptyset$, $\mathbf{x}_{\mathrm{samp}}$ is—for all $j = 1, \ldots, n$—generated on the basis of $\mathbf{x}^{\mathrm{LP}}$:

$$x_j^{\mathrm{samp}} = \begin{cases} x_j^{\mathrm{LP}} & \text{if } d_{\mathrm{rate}} \leq x_j^{\mathrm{LP}} \leq 1 - d_{\mathrm{rate}} \\ d_{\mathrm{rate}} & \text{if } x_j^{\mathrm{LP}} < d_{\mathrm{rate}} \\ 1 - d_{\mathrm{rate}} & \text{if } x_j^{\mathrm{LP}} > 1 - d_{\mathrm{rate}} \end{cases}$$

After generating $\mathbf{x}_{\mathrm{samp}}$, a possibly infeasible binary solutions $\mathbf{s}$ is generated from $\mathbf{x}_{\mathrm{samp}}$ by randomized rounding. Note that this is done in the order $j = 1, \ldots, n$. Finally, $\mathbf{s}$ is translated in set-form S and returned to CMSA_GEN.

CP-Support During Solution Construction Optionally, the algorithm proposed in [5] makes use of the Constraint Propagation engine `cprop` that implements ideas from [1, 20] for the construction of solutions.[1]

The support of CP is utilized in two distinct manners. Firstly, it involves processing all constraints, detecting implications derived from the constraint set, and preprocessing the problem to maintain the corresponding variables fixed throughout the search process. Secondly, it alters the solution construction mechanism as

[1] The used CP tool can be obtained at https://github.com/h-g-s/cprop.

follows: instead of sequentially deriving values for variables in the order of $j = 1, \ldots, n$, a random order π is selected for each solution construction. This means that at step j, instead of determining a value for variable x_j, a value for variable $x_{\pi(j)}$ is determined. Subsequently, after selecting a value for variable $x_{\pi(j)}$, the CP tool verifies if this assignment yields an infeasible solution. If so, variable $x_{\pi(j)}$ is fixed to the alternative value. If, once again, the CP tool concludes that this configuration cannot result in a feasible solution, the standard solution construction progresses as outlined before—that is, without CP support—is utilized to finalize the construction of an unfeasible solution. Conversely, if a feasible value can be selected for the current variable, CP might suggest potential implications involving further (so far unfixed) variables that consequently need to be fixed to certain values. All such implications are addressed before handling the next non-fixed variable according to π.[2]

6.1.1.3 Extension of the Standard Algorithm

Instead of utilizing fixed values for parameters d_{rate} and t_{ILP}, the approach outlined in [5] proposes the following strategy. Both parameters have an associated lower and upper bound. At the initiation of CMSA_GEN, the values for d_{rate} and t_{ILP} are initialized to their lower bounds. If an iteration leads to an improvement of the best-so-far solution, the values of d_{rate} and t_{ILP} revert to their respective lower bounds. Conversely, if there is no improvement, the values of d_{rate} and t_{ILP} are incremented by a factor determined by subtracting the lower bound from the upper bound and dividing the result by 5.0. In addition, whenever the value of d_{rate} or t_{ILP} surpasses its upper bound, it is reset to the lower bound value. According to the authors, this methodology draws inspiration from the approach used to manage neighborhood size in variable neighborhood search (VNS) algorithms [15].

6.1.2 Experimental Evaluation

Two CMSA variants were tested in [5]. The first one, CMSA_GEN, does not employ CP support for solution construction, while the second one—CMSA_GEN_CP—does make use of CP. Moreover, CPLEX was applied to all problem instances with two settings. CPLEX-OPT makes use of the default settings, while CPLEX-HEUR utilizes the highest level of heuristic emphasis. CPLEX 12.7 was employed for these experiments in 2019. However, they were already performed on the IIIA-CSIC in-house high-performance computing cluster of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM.

[2] Note that, after fixing a value for $x_{\pi(j)}$, the value of $x_{\pi(j+1)}$ might already be fixed due to one of the implications dealt with earlier.

Table 6.1 Characteristics of the six BIP instances for which results are shown in Fig. 6.1

BIP instance name	# Cols/Vars	# Rows	Opt. Val.	MIPLIB status 2019
`air04`	8904	823	56137.0	Easy
`opm2-z12-s14`	10,800	319,508	−64291.0	Hard
`protfold`	1835	2112	−31.0	Hard
`rmine14`	32,205	268,535	Unknown	Open
`t1717`	73,885	551	Unknown	Open
`rflcs-2048-3n-div-8`	5461	7,480,548	Unknown	n.a.

6.1.2.1 Benchmark Instances

30 problem instances (27 from MIPLIB 2010 and three additional ones from the authors' research) were chosen for the experimentation in [5]. We only show graphical results for six of these instances. Their names and characteristics are provided in Table 6.1. In particular, after the first table column with the instance names, two columns provide the number of columns (corresponding to the number of binary variables) and the number of rows, respectively, of the matrix A from the BIP model; see Eq. (6.1) on page 157. Next, row number four presents the value of an optimal solution (if known), and the last table column shows the MIPLIB status in 2019 (ranging from 'easy' to 'open'). Note that the last instance (`rflcs-2048-3n-div-8`) is a difficult instance of the so-called repetition-free longest common subsequence (RFLCS) problem from the authors' research. The hardness of this instance is due to a massive amount of constraints.

6.1.2.2 Results

Instead of performing a full parameter tuning, the authors of [5] designed four different settings, and determined the best setting for each problem instance. We refer the interested reader to [5] for more information.

All four approaches (CMSA_GEN, CMSA_GEN_CP, CPLEX-OPT, and CPLEX-HEUR) were applied with a computation time limit of 1000 CPU seconds to each problem instance. However, as the CMSA variants are stochastic algorithms, they are applied 10 times to each instance, while the two CPLEX variants are applied exactly once to each instance. A sample of the obtained results is presented for the six problem instances from Table 6.1 in Fig. 6.1. It contains for each problem instance a graphic that shows the evolution (in terms of the objective function value of the best-found solutions) over time. Note that, in the case of the CMSA variants, the graphics show the average behavior over 10 runs, together with a confidence ribbon.

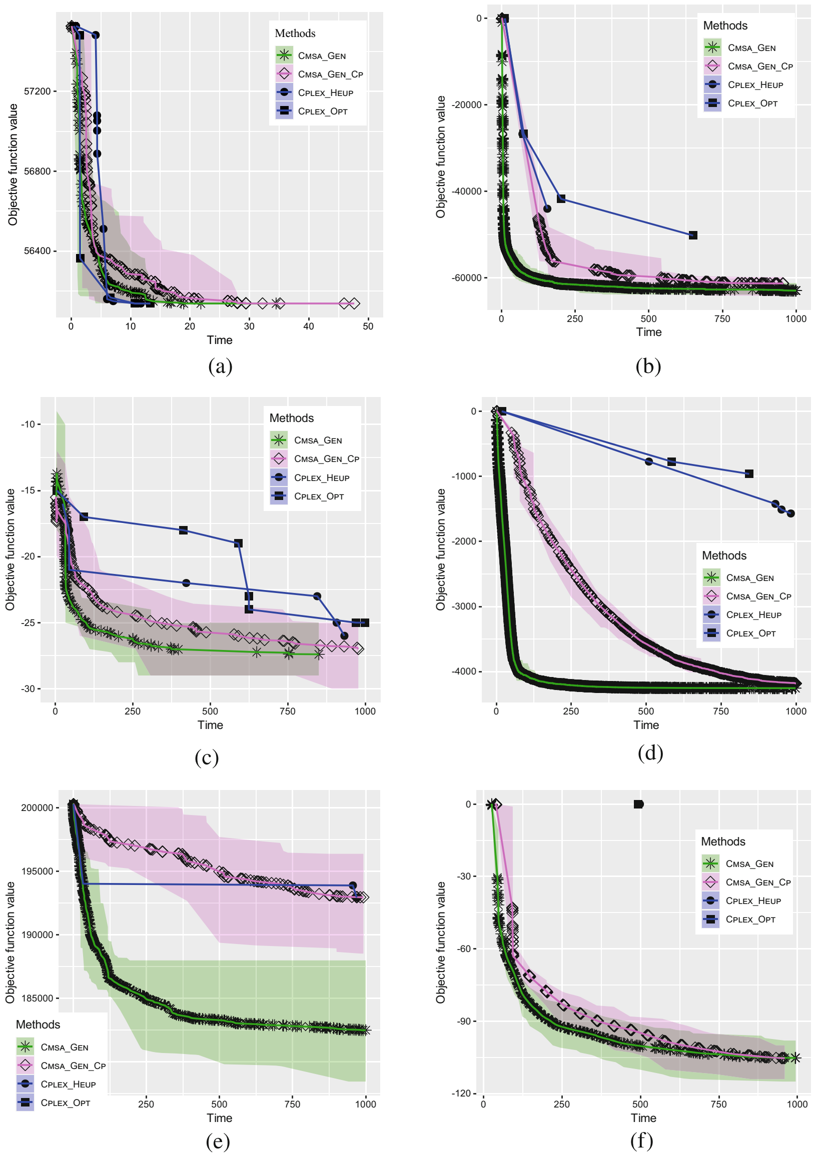


Fig. 6.1 Anytime performance of CMSA_GEN, CMSA_GEN_CP, CPLEX-HEUR and CPLEX-OPT for six exemplary BIP instances from MIPLIB 2010. The mean performance of the CMSA variants, together with the confidence ribbon (based on 10 independent runs) is provided. (**a**) Instance `air04`. (**b**) Instance `opm2-z12-s14`. (**c**) Instance `protfold`. (**d**) Instance `rmine14`. (**e**) Instance `t1717`. (**f**) Instance `rflcs-2048-3n-div-8`

Observations Concerning the BIP Results

- For instances categorized as 'easy', both CPLEX variants are usually faster than the CMSA variants in reaching optimal solutions. An example is shown in Fig. 6.1a (instance `air04`).
- Often, CPLEX-HEUR obtains better solutions than CPLEX-OPT, which is to be expected, because the focus of CPLEX-HEUR is on quickly finding good solutions, while the focus of CPLEX-OPT is on proving optimality earlier.
- For problem instances categorized as 'hard' and 'open', both CMSA variants often show a clear advantage over the two CPLEX versions in the sense that (1) good solutions are found much earlier in the search process, and (2) the solutions found at the end of a run are generally much better than those found by the CPLEX versions. Examples are shown in Fig. 6.1b–d.
- Concerning a comparison between the two CMSA variants, it can be observed that the standard variant (CMSA_GEN) is often faster than the CMSA variant with CP support (CMSA_GEN_CP). This is because CP support comes with a cost. This is most strikingly seen in the example of Fig. 6.1e. However, on the other side, CP support helps to find feasible solutions in the context of moderately constrained problems.
- Finally, one of the disadvantages of both CMSA variants is shown in the context of very constrained problems. For such problems, the CMSA variants often do not even find a single feasible solution.

6.1.3 Discussion

The work on a problem-agnostic CMSA version for BIP problems is certainly only a first step along this avenue of research. In particular, the ability of the algorithm to quickly find feasible solutions must be improved in the context of highly constrained problems. For this purpose, the hybrid biased random key genetic algorithm (BRKGA) from [2] might be used, for example.

6.2 Applying a Metaheuristic in the CMSA Framework

In [19], the authors presented a variant of CMSA in which the use of an exact solver for solving sub-instances was replaced by the use of a metaheuristic. The authors were able to show in the context of the so-called weighted independent domination (WID) problem that their metaheuristic-based CMSA outperformed the standalone application of the metaheuristic. In other words, this paper provided evidence for the ability of CMSA to improve approximate techniques such as metaheuristics when applied within the CMSA framework. In the following, we shortly describe the application from [19] and replicate some of the obtained results.

6.2.1 The Weighted Independent Domination (WID) Problem

The WID problem is an NP-hard combinatorial optimization problem first introduced in [7]. For the description of the problem, the following graph-theoretical concepts are required, in addition to the ones already used in this book. In particular, given an undirected graph $G = (V, E)$, an edge $e \in E$ is called *incident* to a node $v \in V$, in case v is one of the two endpoints of e. Furthermore, the set of edges incident to a node $v \in V$ is denoted by $\delta(v)$. Remember from Chap. 1 that a subset $D \subseteq V$ of the nodes is called a *dominating set* if every node $v \in V \setminus D$ is adjacent to at least one node from D, that is if for every node $v \in V \setminus D$ there exists at least one node $u \in D$ such that $v \in N(u)$.

Independent Sets and Their Relation to Dominating Sets

- Unlike a dominating set, an *independent set* $I \subseteq V$ has the property that no pair $v \neq v' \in I$ of vertices are connected by an edge in the graph G.
- Hereby, an independent set $I \subseteq V$ is labeled as a *maximal independent set* if the addition of any node from $V \setminus I$ would result in the loss of the independent set property.
- It is worth noting that every maximal independent set is a dominating set. Consequently, a maximal independent set is commonly referred to as an *independent dominating set*.
- Vice versa, a subset $D \subseteq V$ is an independent dominating set if D is a maximal independent set.

Finally, given an independent dominating set $D \in V$, for all $v \in V \setminus D$ we define the *D-restricted neighborhood* $N(v \mid D)$ as $N(v \mid D) := N(v) \cap D$, that is, the neighborhood of v is restricted to all its neighbors that are in D.

In the WID problem, we are presented with an undirected graph $G = (V, E)$ along with weights assigned to both nodes and edges. To elaborate, for each vertex $v \in V$ and each edge $e \in E$, we are provided with a non-negative integer weight $w(v) \geq 0$ and $w(e) \geq 0$, respectively. The objective of the WID problem is to identify an independent dominating set D within graph G that minimizes a cost function defined as follows:

$$f(D) := \sum_{u \in D} w(u) + \sum_{v \in V \setminus D} \min\{w(v, u) \mid u \in N(v \mid D)\} \tag{6.3}$$

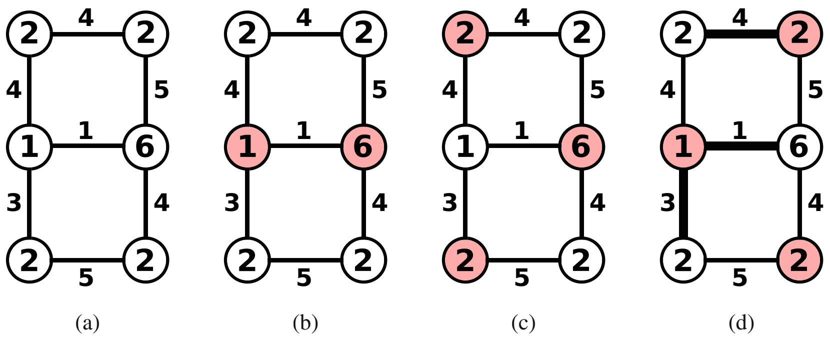


Fig. 6.2 (**a**) shows an undirected graph with node and edge weights. (**b**) a minimum dominating set (weights are irrelevant). (**c**) a maximum independent set (weights are irrelevant). (**d**) an optimal WID solution. Bold edges contribute to the objective function value which is $2+1+2+4+1+3 = 13$. The first 3 numbers are the node weights and the remaining three numbers are the contributing edge weights

In other words, the objective function value of an independent dominating set D is computed by summing the weights of the nodes within D, in addition to the weights of the minimum-weight edges linking nodes outside D to nodes inside D.

As an example consider the graphics in Fig. 6.2. In particular, the graphic in Fig. 6.2a shows a simple example graph. The weights assigned to nodes are displayed within the nodes themselves, while the weights assigned to edges are shown adjacent to the respective edges. Figure 6.2b shows an optimal MDS (minimum dominating set) solution of this graph. (Remember that node and edge weights are irrelevant for the MDS problem.) However, note that the node-set from Fig. 6.2b is not an independent set, because the two nodes of the set are connected by an edge of the graph. Figure 6.2c shows an optimal MIS (maximum independent set) solution, a problem for which the node and edge weights are again irrelevant. Finally, the graphic of Fig. 6.2d shows the optimal WID solution for this simple example graph. Those nodes that do not form part of the solution (top left, middle right, and bottom left) are linked by means of the minimum-weight edges—indicated in bold—to nodes that form part of the solution.

6.2.1.1 ILP Model of the WID Problem

The authors of [19] developed three different ILP models for the WID problem. Here we only present the computationally best one. But keep in mind that even this ILP model, which is the best one out of three different models, cannot be used within CMSA for solving sub-instances because it can only be applied to very small problem instances.

The ILP model makes use of two sets of binary variables. For each node $v \in V$ it uses a binary variable x_v. Moreover, for each edge $e \in E$ the model uses a binary variable z_e. Hereby, x_v indicates if v is chosen for the solution, while z_e indicates if $e \in E$ is selected for connecting a non-chosen node to a chosen one.

$$\min \quad \sum_{v \in V} x_v w(v) + \sum_{e \in E} z_e w(e) \tag{6.4}$$

$$\text{subject to} \quad x_v + x_u \leq 1 \quad \forall\, e = (u, v) \in E \tag{6.5}$$

$$x_v + \sum_{u \in N(v)} x_u \geq 1 \quad \forall\, v \in V \tag{6.6}$$

$$x_v + x_u \geq z_e \quad \forall\, e = (u, v) \in E \tag{6.7}$$

$$x_v + \sum_{e \in \delta(v)} z_e \geq 1 \quad \forall\, v \in V \tag{6.8}$$

$$x_v \in \{0, 1\} \quad \forall\, v \in V$$

$$z_e \in \{0, 1\} \quad \forall\, e \in E$$

Constraints (6.5) are the independent set constraints, that is, they make sure that adjacent nodes can not both form part of the solution. Furthermore, constraints (6.6) represent the dominating set constraints. These constraints guarantee that for every node $v \in V$, either the node itself or at least one of its neighbors is included in the solution. Next, constraints (6.7) have the following function. When both x_v and x_u—concerning an edge $e = (u, v) \in E$—are set to zero, the respective constraint (6.7) forces variable z_e to take value zero, which means that an edge that connects two non-selected nodes can not be chosen for the solution. If, for any edge $e = (u, v) \in E$, both x_v and x_u are assigned zero, the associated constraint (6.7) forces variable z_e to assume the value zero as well. This indicates that an edge linking two unselected nodes cannot be included in the solution. Finally, constraints (6.8) ensure that each node $v \in V$ that does not form part of the solution is connected by an edge to a node that forms part of the solution.

6.2.2 A Greedy Heuristic for the WID Problem

Two different greedy heuristics were developed in [19]. However, here we only describe the better one—henceforth simply called GREEDY—which is used within the metaheuristic for the WID problem whose description is provided in the next section.

The pseudo-code of GREEDY is given in Algorithm 6.1. In general terms, GREEDY commences with an empty partial solution $S = \emptyset$, and iteratively adds precisely one node from the remaining graph $G' = (V', E')$ (as explained further down) to the partial solution S at each construction step. Hereby, S being a partial solution means that S is an independent set, but not yet a dominating set. However,

Algorithm 6.1: Greedy heuristic (GREEDY) for the WID problem

```
1: input: a undirected graph $G = (V, E)$ with node and edge weights
2: $S := \emptyset$
3: $G' := G$
4: while $V' \neq \emptyset$ do
5:     $v^* := \text{argmin}\left\{f^{\text{aux}}(S \cup \{v\}) \mid v \in V'\right\}$ {Ties are randomly resolved}
6:     $S := S \cup \{v^*\}$
7:     Remove from $G'$ all nodes from $N[v \mid G']$ and their incident edges
8: end while
9: output: An independent dominating set $S$ of $G$
```

it can be extended to be a dominating set. At the start of GREEDY, the *remaining graph* G' is a copy of G; see line 3.

For describing how a node from G' is chosen at each construction step, the following notations are required. First, the maximum weight of any edge in E is denoted by $w_{\max}$. An *auxiliary objective function value* $f^{\text{aux}}(S)$ is defined for any (partial) solution S as follows.

$$f^{\text{aux}}(S) := \sum_{v \in V} c(v \mid S) \ , \tag{6.9}$$

where $c(v \mid S)$ is called the *contribution* of node v concerning partial solution S. These contributions are defined as follows:

1. If $v \in S$: $c(v \mid S) := w(v)$
2. If $v \notin S$ and $N(v) \cap S = \emptyset$: $c(v \mid S) := w_{\max}$
3. If $v \notin S$ and $N(v) \cap S \neq \emptyset$: $c(v \mid S) := \min\{w(e) \mid e = (v, u), u \in S\}$

Note that in the case of S being a complete solution, it holds that $f(S) = f^{\text{aux}}(S)$. GREEDY chooses, at each construction step, the node $v^* \in V'$ such that its addition to the current partial solution S leads to the least increase of the auxiliary objective function value; see line 5 of Algorithm 6.1.

After adding node $v^* \in V'$ to S, all nodes from $N[v^* \mid G']$—that is, from the closed neighborhood of v^* in G'—are removed from V'. Moreover, all their incident edges are removed from E'. In this way, only those nodes that maintain the property of S being an independent set may be added to S in subsequent construction steps.

6.2.3 A PBIG Metaheuristic for the WID Problem

Based on the greedy heuristic outlined in the previous section, the authors of [19] devised a so-called population-based iterated greedy (PBIG) metaheuristic for solving the WID problem. Hereby, a PBIG algorithm is a simple extension of the well-known iterated greedy (IG) algorithm [21] towards working with populations of solutions.

Algorithm 6.2: PBIG for the WID problem

```
1: input: an input graph $G = (E, V)$, values for parameters $p_{\text{size}}, L, U, d_{\text{rate}}, l_{\text{size}}$
2: $\mathcal{P}$ := Generate_Initial_Population($p_{\text{size}}, d_{\text{rate}}, l_{\text{size}}$)
3: while termination condition not satisfied do
4:     $\mathcal{P}_{\text{new}} := \emptyset$
5:     for each candidate solution $S \in \mathcal{P}$ do
6:         $\hat{S}$ := Destroy_Partially($S$)
7:         $S'$ := Reconstruct($\hat{S}, d_{\text{rate}}, l_{\text{size}}$)
8:         Adapt_Destruction_Rate($S, S'$)
9:         $\mathcal{P}_{\text{new}} := \mathcal{P}_{\text{new}} \cup \{S'\}$
10:    end for
11:    $\mathcal{P}$ := Accept($\mathcal{P}, \mathcal{P}_{\text{new}}$)
12: end while
13: output: best solution from $\mathcal{P}$
```

The pseudo-code of PBIG is provided in Algorithm 6.2. The algorithm starts by generating the initial population of p_{size} solutions in function Generate_Initial_Population($p_{\text{size}}, d_{\text{rate}}, l_{\text{size}}$). For this purpose, GREEDY is applied p_{size} times in a probabilistic way by using two parameters ($d_{\text{rate}} \in [0, 1]$ and $l_{\text{size}} \in \mathbb{Z}^+$) in the following way. At each construction step of GREEDY, first, a random value $0 \leq r \leq 1$ is chosen uniformly at random. In case $r \leq d_{\text{rate}}$, the best node (v^*, see line 5 of Algorithm 6.1) is deterministically chosen. Otherwise—that is, in case $r > d_{\text{rate}}$—the best $\min\{|V'|, l_{\text{size}}\}$ nodes from V' are considered and one of them is chosen uniformly at random.

At each iteration of PBIG, the following is done regarding each solution S of the incumbent population $\mathcal{P}$. First, S is partially destroyed in function Destroy_Partially(S). For this purpose, solution S maintains an individual destruction rate $\text{dest}^S_{\text{rate}}$ whose value is dynamically updated and may move between a lower bound L and an upper bound U, which are parameters of PBIG. That is, the values of L and U must be fixed before running PBIG such that $0 \leq L \leq U \leq 1$. To partially destroy solution S, $\max\{3, \lfloor \text{dest}^S_{\text{rate}} \cdot |S| \rfloor\}$ randomly chosen vertices from S are removed. This results in a partial solution $\hat{S}$.

The partial solution $\hat{S}$ obtained by the destruction procedure outlined above is then subject to probabilistic reconstruction in function Reconstruct($\hat{S}, d_{\text{rate}}, l_{\text{size}}$), resulting in a new complete solution S' which is stored in set $\mathcal{P}_{\text{new}}$. This is done by the procedure already used for the probabilistic construction of the solutions of the initial population above. Moreover, the individual destruction rate $\text{dest}^{S'}_{\text{rate}}$ is initialized to the lower bound L.

In other words, each solution $S \in \mathcal{P}$ gives rise to a new solution S'. As a last step, the individual destruction rate $\text{dest}^S_{\text{rate}}$ of the solution S is updated in function Adapt_Destruction_Rate(S, S') based on S' as follows: if $f(S') < f(S)$, $\text{dest}^S_{\text{rate}}$ is set to the lower bound L. Otherwise, an amount of $\text{dest}^{\text{inc}}_{\text{rate}}$ is added to $\text{dest}^S_{\text{rate}}$. If this causes that $\text{dest}^S_{\text{rate}} > U$, $\text{dest}^S_{\text{rate}}$ is re-initialized to the lower bound L.

Finally, the last step of each iteration of PBIG consists in the selection of the best p_{size} solutions from $\mathcal{P} \cup \mathcal{P}_{\text{new}}$ and replacing the solutions in $\mathcal{P}$ with these p_{size} solutions.

The key idea behind any PBIG algorithm is to combine the exploration capabilities of a population-based approach with the exploitation capabilities of a greedy heuristic. By maintaining a population of solutions and iteratively improving them through partial destruction and probabilistic reconstruction, PBIG aims to efficiently explore the search space while focusing on promising regions that contain high-quality solutions. Overall, PBIG algorithms tend to provide a balance between exploration and exploitation, making them effective for solving various combinatorial optimization problems. The specific implementation details and parameter settings may vary depending on the problem being addressed.

6.2.4 Using PBIG *for Solving Sub-instances in CMSA*

In [19], the authors used standard CMSA based on the intuitive way of defining the solution components (CMSA_INT) that was introduced in Sect. 1.3.1 of Chap. 1 of this book. In particular, for each node $v_i \in V$ of the input graph $G = (V, E)$ the complete set C of solution components contains a solution component c_i. As PBIG is used instead of CPLEX for solving sub-instances at each algorithm iteration, the resulting approach is henceforth called CMSA_PBIG.

The probabilistic construction of solutions in CMSA_INT works in the same way as explained above for the construction of the solutions of the initial population in PBIG. The only difference is that the determinism rate (d_{rate}) and the candidate list size (l_{size}) are now called $d_{\text{rate}}^{\text{CMSA}}$ and $l_{\text{size}}^{\text{CMSA}}$ to differentiate them from the d_{rate} and l_{size} parameters of PBIG.

Sub-instances C' of CMSA_PBIG, which are sets of solution components corresponding to nodes of the WID input graph, are solved—as already mentioned above—by PBIG. However, note that for solving a sub-instance C', all actions of PBIG are restricted to the selection of nodes corresponding to solution components in C'. Moreover, note that PBIG is applied to each sub-instance without being warm-started, that is, the best-so-far solution of CMSA_PBIG is not used to influence the generation of the initial PBIG iteration. This is because, in preliminary experiments, this was shown to be counterproductive.

6.2.5 Experimental Evaluation

As the aim of this section is the reproduction of some of the results from [19], we only included PBIG and CMSA_PBIG into the comparison. As in all other cases presented in this book, the IIIA-CSIC in-house high-performance computing cluster

Table 6.2 Parameters, domains, and tuning results for the WID problem

Parameter	Domain	PBIG	CMSA_PBIG
p_{size}	$\{2, \ldots, 200\}$	184	43
L	$[0.05, 0.95]$	0.51	0.38
U	$[0.05, 0.95]$	0.79	0.73
$\text{dest}_{\text{rate}}^{\text{inc}}$	$[0.01, 0.1]$	0.05	0.1
d_{rate}	$[0.0, 0.99]$	0.25	0.04
l_{size}	$\{3, \ldots, 50\}$	23	15
n_{a}	$\{2, \ldots, 50\}$	n.a.	40
$age_{\max}$	$\{1, \ldots, 50\}$	n.a.	1
$d_{\text{rate}}^{\text{CMSA}}$	$[0.0, 0.99]$	n.a.	0.41
$l_{\text{size}}^{\text{CMSA}}$	$\{3, \ldots, 50\}$	n.a.	30
t_{ILP}	$\{1, \ldots, 50\}$	n.a.	23

of machines equipped with Intel® Xeon® 5670 CPUs having 12 cores of 2.933 GHz and at least 32 GB of RAM was used for conducting all the experiments.

6.2.5.1 Problem Instances

Instead of using the problem instances from the original paper [19], we decided to produce random graphs with the Erdös-Rényi model [9], which requires the number of nodes (n) and the probability of an edge existing between any pair of nodes (p) as input. In particular, we generated 30 graphs for each combination of $|V| \in \{500, 1000, 1500, 2000\}$ and three different graph densities $p \in \{0.05, 0.15, 0.25\}$. In total, this benchmark set consists of 360 graphs. As in Chap. 1 in the context of the MDS problem, we used the implementation of the Erdös-Rényi model from the `igraph` library for this purpose.[3]

6.2.5.2 Parameter Tuning

As usual in this book, the `irace` tool was utilized for tuning the parameters of PBIG and CMSA_PBIG. Both algorithms were tuned exactly once for the entire benchmark set, which is a difference to [19] where parameter tuning was more fine-grained. For parameter tuning, additional problem instances were generated. More specifically, for each combination of $|V|$ and graph density, exactly one tuning instance was generated. This makes a total of 12 tuning instances. As computation time limit, $|V|$ CPU seconds were given, that is, the more nodes a graph has, the longer the allowed running time. Finally, `irace` was given a budget of 3000 algorithm runs.

Table 6.2 shows both the parameters involved in the two algorithms together with their domains, and the tuning results. The first six parameters in this table

[3] https://igraph.org/.

are the parameters of PBIG. The next five parameters are the usual CMSA-related parameters.[4]

The following parameter settings are noteworthy. The determinism rate for solution construction in the CMSA-part of CMSA_PBIG (as shown by the value of parameter $d_{\text{rate}}^{\text{CMSA}}$) is very different to the determinism rate for solution reconstruction in the PBIG-part of CMSA_PBIG (see the value of parameter d_{rate}). While the value is rather high for the construction of solutions that are merged into the sub-instance of CMSA_PBIG, the value is very low (0.04) for the application of PBIG for solving the sub-instance at each iteration. A possible interpretation is that for CMSA_PBIG it seems important to equip the sub-instances with seemingly good solution components, while the focus of PBIG for solving the sub-instances is clearly on exploration.

6.2.5.3 Results

Both algorithms (PBIG and CMSA_PBIG) were applied exactly once to each of the problem instances from the benchmark set. The computation time limit was the same as the one used for tuning (see previous section). The results are shown in the form of box plots in Fig. 6.3, which contains a 3×4 grid of box plots. Hereby, the rows present the results (from top to bottom) for problem instances with an increasing graph size (in terms of the number of nodes), and the columns (from left to right) present the results for problem instances with an increasing density. To be able to support the analysis of the results with claims about their statistical significance, a CD plot (see Fig. 6.4) is provided, as in all other experimental evaluations presented in this book. In particular, the plot in Fig. 6.4 contains statistics over the whole set of problem instances.

> Main Observations Concerning the WID Problem Results

1. First, and most importantly, CMSA_PBIG outperforms PBIG with statistical significance.
2. Second, the box plots in Fig. 6.3 show that the improvement of CMSA_PBIG over PBIG can be seen for all graph sizes and densities.
3. Third, the improvement of CMSA_PBIG over PBIG seems to grow with an increasing graph density.

[4] Consider that parameter t_{ILP}, which received its name due to being the time limit for the ILP solver CPLEX, limits the computation time of PBIG at each iteration of CMSA_PBIG. However, for consistency reasons, we did not change the name of this parameter for the present application.

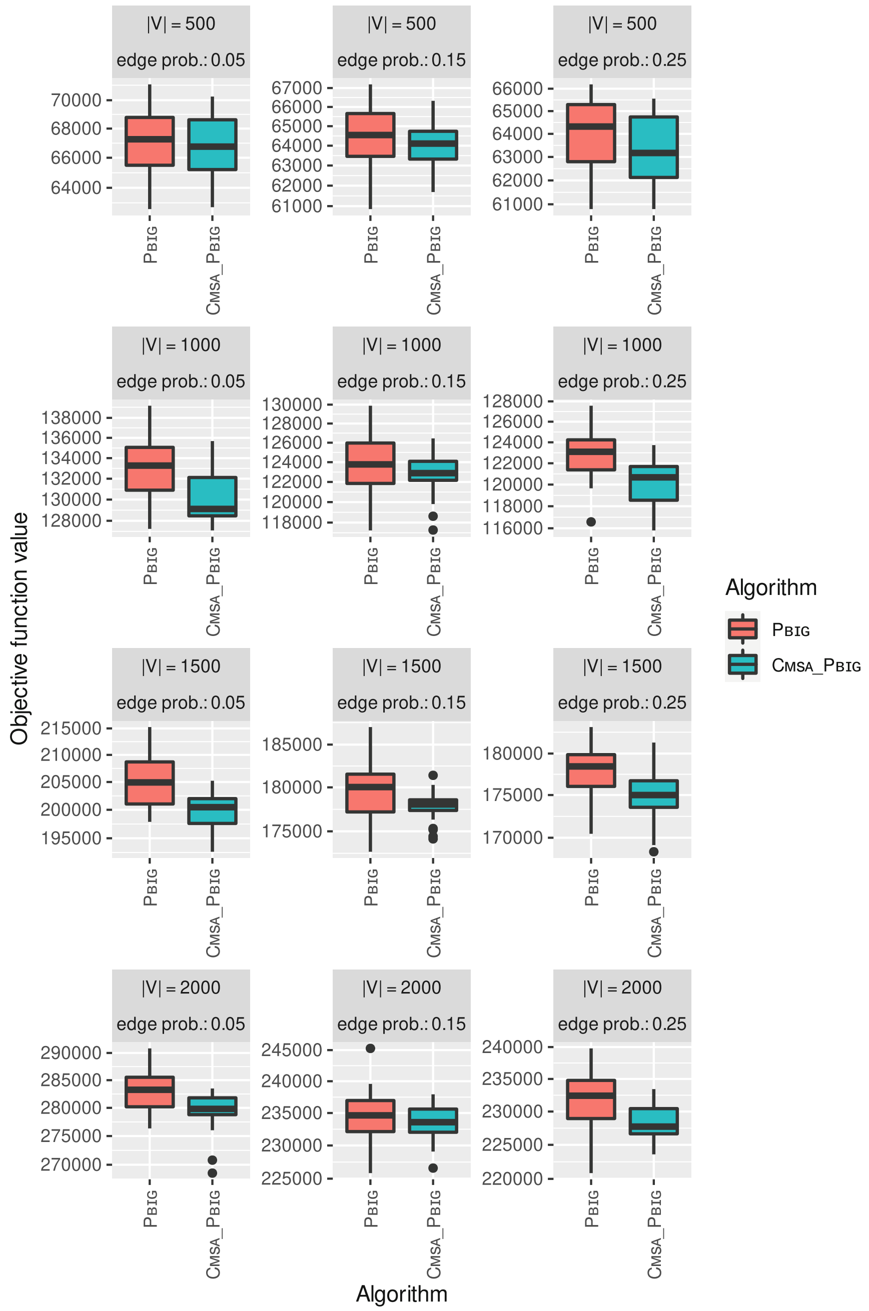


Fig. 6.3 Results of CMSA_PBIG and PBIG for the WID problem

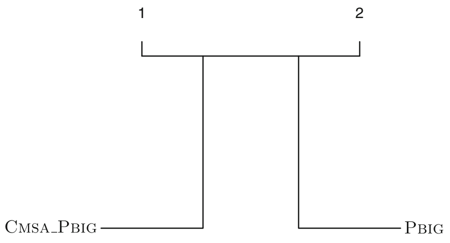


Fig. 6.4 Critical Difference (CD) plot concerning the WID problem results

6.2.6 *Discussion*

Even though in the context of the PBIG metaheuristics for solving the WID problem we were able to obtain an improvement when applying PBIG within the CMSA framework, this improvement is not something that can be generally expected when studying other metaheuristic implementations for other combinatorial optimization problems. As an example, note that we tried to apply the BA algorithms from Chap. 3 within the CMSA framework, both for the MDS and the FFMS problem. However, in both cases, the standalone application of BA obtained better results than the application of BA in the CMSA framework. We suspect that, in the case of the application of CMSA_PBIG for the WID problem, the CMSA framework might help PBIG to escape from the area of attraction of local optima. Escaping from local optimal seems less required in the case of BA when applied for the MDS and FFMS problems.

6.3 Relation Between CMSA and LNS

As a final topic in this concluding chapter, this section will be devoted to shed some light on the differences between CMSA and large neighborhood search (LNS), which is arguably the most well-known hybrid metaheuristic that has been developed so far. In both methods, sub-instances of the tackled problem instances are solved at each iteration. However, they differ in the way in which these sub-instances are obtained. The relation between CMSA and LNS has been studied in [6]. Here we provide a short overview of their main findings.

6.3.1 *Destruction-Based LNS*

The authors of [6] compared CMSA with destruction-based LNS, that is, with an LNS method that partially destroys the incumbent solution at each iteration, before the obtained partial solution is passed to CPLEX (or another exact solver) in order to compute the best valid solution that contains that partial solution (within a given time limit). For more information on LNS see Sect. 1.1.5 on page 10.

The pseudocode for a general LNS approach utilizing an ILP solver to solve the corresponding sub-instance at each iteration is outlined in Algorithm 6.3. As in the case of CMSA, we assume that solutions in this LNS algorithm are subsets from a complete set C of solution components. Initially, the starting solution S^{bsf} (also serving as the best-so-far solution) is generated using the function GenerateInitialSolution(C) (refer to line 2). Typically, a greedy heuristic is employed for this task. During each iteration, the following steps are executed. Firstly, a copy of the best-so-far solution S^{bsf} is partially destroyed; this is achieved through the function DestroyPartially(S^{bsf}, $\text{dest}_{\text{rate}}$) at line 5, where the extent of destruction is determined by a parameter $\text{dest}_{\text{rate}}$ known as the *destruction rate*. Various methods can be used for the partial destruction of a solution. The most basic approach, likely prevalent in many cases, involves random destruction. However, one might consider employing heuristically guided methods for partial destruction. Regardless, the resulting partial solution S' is then passed to the ILP solver through the function Reconstruct(S', t_{ILP}) at line 6. This function, besides S', takes a time limit t_{ILP} as input. The ILP solver is directed to only consider solutions containing S' for this operation, effectively constraining the sub-instance to solutions incorporating S'. The function returns S^{ILP}, the best valid solution found within t_{ILP} CPU seconds. Given the time constraint, it is important to note that S^{ILP} may not necessarily be an optimal solution to the sub-instance. Finally, the better solution between S^{ILP} and S^{bsf} is selected as the incumbent solution for the next iteration. While this selection process may appear stringent, other more probabilistic approaches for choosing between S^{ILP} and S^{bsf} are conceivable. Nevertheless, the LNS algorithm examined in [6] is equipped with a variable destruction rate $L \leq \text{dest}_{\text{rate}} \leq U$, managed akin to the neighborhood size in Variable Neighborhood Search (VNS) algorithms [14]. Specifically, if S^{ILP} is better than S^{bsf}, $\text{dest}_{\text{rate}}$ is reverted to the lower bound L. Otherwise, $\text{dest}_{\text{rate}}$ is incremented by $\text{dest}_{\text{rate}}^{\text{inc}}$, another parameter of the algorithm. If, post-increment, $\text{dest}_{\text{rate}}$ surpasses the upper bound U, it is reset to the lower bound L. Appropriate selection of L and U values enables the algorithm to escape from local minima.

6.3.2 Empirical Comparative Study

Given the description of destruction-based LNS above, it is clear that both CMSA and LNS solve sub-instances of the tackled problem instances at each iteration. The difference between the two approaches is found in the way in which these sub-instances are generated and maintained. While CMSA updates an initially empty sub-instance by adding those solution components that appear in constructed solutions, LNS obtains a new sub-instance at each iteration by the partial destruction of a copy of the best-so-far solution. This implies also a difference in the way in which sub-instances are solved. In the case of CMSA, as we have seen through

Algorithm 6.3: Destruction-based large neighborhood search (LNS)

```
 1: input: solution components ($C$), values for parameters $L$, $U$, $\text{dest}_{\text{rate}}^{\text{inc}}$, $t_{\text{ILP}}$
 2: $S^{\text{bsf}} := \mathsf{GenerateInitialSolution}(C)$
 3: $\text{dest}_{\text{rate}} := L$
 4: while CPU time limit not reached do
 5:    $S' := \mathsf{DestroyPartially}(S^{\text{bsf}}, \text{dest}_{\text{rate}})$
 6:    $S^{\text{ILP}} := \mathsf{Reconstruct}(S', t_{\text{ILP}})$
 7:    if $S^{\text{ILP}}$ is better than $S^{\text{bsf}}$ then
 8:       $S^{\text{bsf}} := S^{\text{ILP}}$
 9:       $\text{dest}_{\text{rate}} := L$
10:    else
11:       $\text{dest}_{\text{rate}} := \text{dest}_{\text{rate}} + \text{dest}_{\text{rate}}^{\text{inc}}$
12:       if $\text{dest}_{\text{rate}} > U$ then $\text{dest}_{\text{rate}} := L$
13:    end if
14: end while
15: output: $S^{\text{bsf}}$
```

a range of examples in this book, the ILP model of the problem to be solved is extended to allow only components from the current sub-instance to be included in solutions. In contrast, the ILP model in the case of LNS is extended to enforce the presence of the solution components in the incumbent partial solution to be present in any valid solution.

The question asked by the authors of [6] was the following one: is there a type of optimization problem for which CMSA generally outperforms LNS, and vice versa? They hypothesized that CMSA outperforms LNS when the number of components in valid solutions is rather low, and vice versa. To test this hypothesis, CMSA and LNS were both implemented for two different problems: (1) the multi-dimensional knapsack problem (MDKP) and (2) the minimum common string partition (MCSP) problem.

On the example of the MDKP it is easy to see why this problem was chosen. The MDKP, an NP-hard combinatorial optimization problem, has been extensively studied and falls under the category of subset selection problems. Additionally, it has served as a popular benchmark for testing new algorithmic approaches, as evidenced by previous research; see, for example, [8, 17]). The problem is formally defined as follows: Given a set C of n items and m different resources, each resource ($k = 1, \ldots, m$) has a specified quantity (referred to as *capacity*) $\text{cap}_k > 0$, and each item $c_i \in C$ ($i = 1, \ldots, n$) requires a certain amount (referred to as *resource consumption*) $r_{i,k} \geq 0$ from the k-th resource. Additionally, each item $c_i \in C$ is associated with a positive profit p_i. A subset $S \subseteq C$ is considered a feasible solution if, for each resource $k = 1, \ldots, m$, the total consumption over all selected items ($\sum_{c_i \in S} r_{i,k}$) does not surpass the resource capacity cap_k. Furthermore, a feasible solution S is deemed non-extensible if it is impossible to add any item $c_i \in C \setminus S$ to S without compromising its validity as a solution. The primary objective is to identify a feasible subset S that maximizes the total profit ($\sum_{c_i \in S} p_i$). The standard ILP formulation for the MDKP is outlined as follows:

$$\max \quad \sum_{c_i \in C} p_i \cdot x_i \tag{6.10}$$

$$\text{subject to} \quad \sum_{c_i \in C} r_{i,k} \cdot x_i \leq \text{cap}_k \quad \forall\, k = 1, \ldots, m \tag{6.11}$$

$$x_i \in \{0, 1\} \quad \forall\, c_i \in C$$

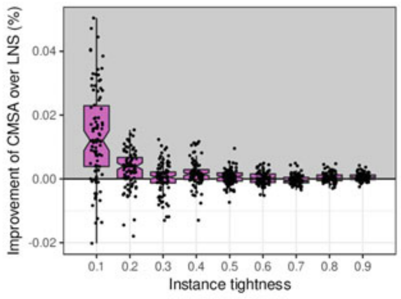
Note that this model is based on a binary variable for each item from C. The inequalities (6.11) limit the total consumption for each resource and are called *knapsack constraints*.

> The MDKP is Parametrizable

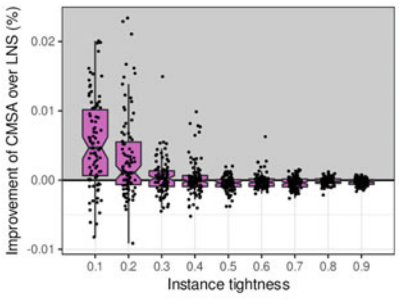
Note that when resource capacities are low, valid MDKP solutions contain few items and are, therefore, rather small. On the other side, the larger the resource capacities, the larger are valid solutions. In this sense, MDKP instances can be generated in a controlled and parameterized way to obtain problem instances from the whole range between instances with solutions containing very few items, and instances with solutions containing a lot of items.

The authors of [6] used the methodology described in [8, 13] for the generation of MDKP instances. In particular, five different values for n (the number of items) were considered: $n \in \{100, 500, 1000, 5000, 10{,}000\}$. Moreover, the number of resources (m) was fixed to 30. The *tightness* of a problem instance is determined by the resource capacities. The methodology from [8, 13] allows to determine the instance tightness through a parameter α which may take values between zero and one. The lower the value of α—that is, the tighter the generated problem instance—the smaller are the solutions, and vice versa. To generate instances over the whole tightness range, values $\alpha \in \{0.1, 0.2, \ldots, 0.8, 0.9\}$ were considered. Finally, the resource requirements $r_{i,j}$ were always chosen uniformly at random from $\{1, \ldots, 1000\}$. In total, 30 instances were generated for each combination of n and α, and the whole benchmark set consists of 1350 problem instances.

After tuning both CMSA and LNS for each combination of instance size (n) and instance tightness (α), both algorithms were applied exactly once to each problem instance with computation time limits depending on the instance size. Here we show only the most interesting results for instances with $n \in \{5000, 10{,}000\}$. Figure 6.5 shows the percentage improvement of CMSA over LNS for instances with $n = 5000$ items (Fig. 6.5a) and instances with $n = 10{,}000$ items (Fig. 6.5b). The x-axis of these box plots ranges over the whole instance tightness range, which increases from left to right. Each box is obtained from the results for 30 instances. Dots in the positive area (grey-shaded) are results for instances for which CMSA produced



(a)



(b)

Fig. 6.5 The percentage improvement of CMSA over LNS for instances with increasing resource capacity tightness. (**a**) Instances with $n = 5000$. (**b**) Instances with $n = 10{,}000$

a better result than LNS. Conversely, dots in the negative area represent instances for which LNS outperformed CMSA. In particular, note that these box plot graphics (empirically) confirm the author's hypothesis: CMSA works better than LNS for MDKP problem instances with rather small solutions. The same was shown in [6] in the context of the MCSP problem.

6.3.2.1 Discussion

The results from [6] presented above indicate that CMSA and LNS are somehow complementary. The question remains why CMSA has this apparent advantage over LNS for problems (or problem instances) for which solutions are rather small. The following intuition may eventually be validated. When solutions are small, LNS can not do large steps in the search space, because it always maintains a part of the incumbent solution. Therefore, if LNS starts with an initial solution that is far away from high-quality solutions, LNS might not be able to reach these solutions. This is because finding a feasible path to high-quality solutions might be rather unlikely to be found by LNS. On the other side, CMSA generates at each iteration several solutions in a probabilistic way. These solutions may, potentially, be located in any part of the search space. They are then merged into the sub-instance, which enables CMSA to do larger steps in the search space, even in the context of small solutions.

6.4 Future Work on CMSA

At this moment, the author sees at least three promising lines of future work on CMSA. The first one concerns an extension of the work done to develop a problem-agnostic CMSA for binary optimization problems; see Sect. 6.1. To achieve this, the way of generating feasible solutions for highly constrained problems must be improved. A problem-agnostic CMSA variant that reliably outperforms high-performance ILP solvers such as CPLEX and Gurobi would be very valuable as an easy-to-use tool for benchmarking new, hand-crafted optimization algorithms for specific optimization problems. So far, the ILP solvers themselves are used for this purpose. However, they are generally easily beaten in the context of large enough (or difficult enough) problem instances. Not disposing of a problem-agnostic CMSA requires designing a CMSA for any particular problem by hand.[5] Even though this might not be difficult in many cases, this sets the bar rather high for the adoption of CMSA as a baseline algorithm.

A second avenue of promising research concerns the one of utilizing state-of-the-art art exact (or approximate) solvers for specific problems instead of black-box ILP solvers for solving sub-instances in CMSA. In [18], the authors showed that by applying the currently best MaxSAT solvers within the framework of negative learning ant colony optimization they were able to improve over the results of these solvers. We imagine that this could also be possible in the context of CMSA.

The third research line deals with the use of machine learning (ML) techniques to add a learning component to the solution construction mechanism of CMSA. Remember that, in Chap. 3 of this book, a learning mechanism for CMSA was proposed in which solutions to be merged into the incumbent sub-instance were generated by a metaheuristic applied in an intertwined way with CMSA. However, using ML, there are other options for adding a learning mechanism. Reinforcement learning (RL) [22]—in particular, algorithms known from the multi-armed bandit problem—might be used for learning to construct good solutions during the runtime of CMSA.

To summarize, promising research remains to be done in the context of the CMSA algorithm. The optimization group at the IIIA-CSIC in Bellaterra (Barcelona) will take on this endeavor during the coming years. Our hope is certainly also that some other research groups on metaheuristics and their hybrids will join this effort in the quest for increasingly efficient CMSA variants.

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, Technische Universität Berlin, Germany (2007)
2. Andrade, C.E., Ahmed, S., Nemhauser, G.L., Shao, Y.: A hybrid primal heuristic for finding feasible solutions to mixed integer programs. European Journal of Operational Research **263**(1), 62–71 (2017)
3. Berthold, T., Lodi, A., Salvagnin, D.: Ten years of feasibility pump, and counting. EURO Journal on Computational Optimization **7**(1), 1–14 (2019)
4. Blum, C.: Advocating CMSA as a baseline algorithm for algorithm comparison in combinatorial optimization. In: Proceedings of InCITe 2024 – 4th International Conference on Information Technology, Lecture Notes in Electrical Engineering. Springer Nature Singapore (2024). In press

[5] In fact, this is advocated, for example, in [4].

5. Blum, C., Gambini Santos, H.: Generic CP-supported CMSA for binary integer linear programs. In: M.J. Blesa Aguilera, C. Blum, H. Gambini Santos, P. Pinacho-Davidson, J. Godoy del Campo (eds.) Proceedings of 11th International Workshop on Hybrid Metaheuristics – HM 2019, pp. 1–15. Springer International Publishing, Cham (2019)
6. Blum, C., Ochoa, G.: A comparative analysis of two matheuristics by means of merged local optima networks. European Journal of Operational Research **290**(1), 36–56 (2021)
7. Chang, S.C., Liu, J.J., Wang, Y.L.: The weighted independent domination problem in series-parallel graphs. Intelligent Systems and Applications **274**, 77–84 (2015)
8. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. Discrete Applied Mathematics **49**(1), 189–212 (1994)
9. Erdös, P., Rényi, A.: On random graphs I. Publ. math. debrecen **6**(290-297), 18 (1959)
10. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming **104**, 91–104 (2005)
11. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Mathematical Programming Computation **1**(2-3), 201–222 (2009)
12. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. Mathematical Programming Computation **13**(3), 443–490 (2021)
13. Hanafi, S., Freville, A.: An efficient tabu search approach for the 0-1 multidimensional knapsack problem. European Journal of Operational Research **106**(2–3), 659–675 (1998)
14. Hansen, P., Mladenović, N.: Variable Neighborhood Search: Principles and Applications. European Journal of Operational Research **130**(3), 449–467 (2001)
15. Hansen, P., Mladenović, N., Brimberg, J., Pérez, J.A.M.: Variable neighborhood search. Springer (2019)
16. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H.D., Ralphs, T.K., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010: Mixed integer programming library version 5. Mathematical Programming Computation **3**, 103–163 (2011)
17. Leung, S.C.H., Zhang, D., Zhou, C., Wu, T.: A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack problem. Computers and Operations Research **39**(1), 64–73 (2012)
18. Nurcahyadi, T., Blum, C., Manyà, F.: Negative learning ant colony optimization for MaxSAT. International Journal of Computational Intelligence Systems **15**(1), 71 (2022)
19. Pinacho Davidson, P., Blum, C., Lozano, J.A.: The weighted independent domination problem: Integer linear programming models and metaheuristic approaches. European Journal of Operational Research **265**(3), 860–871 (2018)
20. Sandholm, T., Shields, R.: Nogood learning for mixed integer programming. Tech. Rep. CMU-CS-06-155, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (2006)
21. Stützle, T., Ruiz, R.: Iterated Greedy, pp. 1–31. Springer International Publishing, Cham (2018)
22. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press (2018)

Appendix A
C++ Program Code: CMSA Applied to the MDS Problem

The following program code in C++ is the one of CMSA_INT applied to the minimum dominating set (MDS) problem as described in Chap. 1. This program code was used for the experimentation. We provide this code as an example for the simplicity of CMSA.

CMSA for the MDS Problem

```
/***********************************************************
               cmsa.cpp  -  description
                 -------------------
    begin          : Wed Nov 30 2022
    copyright      : (C) 2022 by Christian Blum
    email          : christian.blum@iiia.csic.es
 **********************************************************/

/***********************************************************
 *   This program is free software; you can redistribute it *
 *   and/or modify it under the terms of the GNU General    *
 *   Public License as published by the Free Software       *
 *   Foundation; either version 2 of the License, or (at    *
 *   your option) any later version.                        *
 **********************************************************/

using namespace std;

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <cmath>
#include <vector>
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3

```
#include <list>
#include <set>
#include <map>
#include <iomanip>
#include <algorithm>
#include <sstream>
#include <limits>
#include <random>
#include <chrono>
#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

struct Option {
    int vertex;
    double value;
};

struct Solution {
    set<int> vertices;
    int score;
};

// CMSA PARAMETERS
double computation_time_limit = 1000.0;
double cplex_time_limit = 10.0;
double determinism_rate = 0.8;
int n_of_sols = 10;
int age_limit = 10;
int candidate_list_size = 10;
bool warm_start = false;
bool heuristic_emphasis = false;
bool cplex_abort = false;

// INSTANCE DATA
int n_of_vertices;
vector< set<int> > neigh;
string input_file;

/* function for making CPLEX abort a run when improving
   over the currently best solution */
ILOSOLVECALLBACK2(abortCallback, IloCplex::Aborter&, abo,
                  int&, curbest) {

    if (hasIncumbent()) {
        IloNum nv = getIncumbentObjValue();
        if (curbest > int(nv)) abo.abort();
    }
}

/* function for sorting a vector of options */
bool option_compare(const Option& o1, const Option& o2) {

    return (o1.value > o2.value);
}

/* function for producing a random integer between 0
   and max - 1 */
int produce_random_integer(int max, double& rnum) {

    int num = int(double(max)*rnum);
    if (num == max) num = num - 1;
    return num;
}

/* function that returns a random element from a set of int */
int get_random_element(const set<int>& s, double& rnum) {

    double r = produce_random_integer(int(s.size()), rnum);
    set<int>::iterator it = s.begin();
    advance(it, r);
    return *it;
}

/* function for reading command line parameter values */
void read_parameters(int argc, char **argv) {

    int iarg=1;
    while (iarg < argc) {
        if (strcmp(argv[iarg],"-i")==0) input_file = argv[++iarg
            ];
        else if (strcmp(argv[iarg],"-t")==0)
            computation_time_limit = atof(argv[++iarg]);
        else if (strcmp(argv[iarg],"-cpl_t")==0)
            cplex_time_limit = atof(argv[++iarg]);
        else if (strcmp(argv[iarg],"-drate")==0)
            determinism_rate = atof(argv[++iarg]);
        else if (strcmp(argv[iarg],"-nsols")==0)
            n_of_sols = atoi(argv[++iarg]);
        else if (strcmp(argv[iarg],"-max_age")==0)
            age_limit = atoi(argv[++iarg]);
        else if (strcmp(argv[iarg],"-lsize")==0)
            candidate_list_size = atoi(argv[++iarg]);
        else if (strcmp(argv[iarg],"-warm_start")==0) {
            int val = atoi(argv[++iarg]);
            if (val == 1) warm_start = true;
        }
        else if (strcmp(argv[iarg],"-h_emph")==0) {
            int val = atoi(argv[++iarg]);
            if (val == 1) heuristic_emphasis = true;
        }
        else if (strcmp(argv[iarg],"-cpl_abort")==0) {
            int val = atoi(argv[++iarg]);
            if (val == 1) cplex_abort = true;
        }
        iarg++;
    }
}
```

```
/* function for solving a sub-instance by calling CPLEX */
void run_cplex(Solution& cpl_sol, Solution& best_sol,
               vector<int>& age, double& r_limit) {

    IloEnv env;
    env.setOut(env.getNullStream());
    env.setWarning(env.getNullStream());
    cpl_sol.score = std::numeric_limits<int>::max();

    try {

        IloModel model(env);

        /* for each vertex of the input graph we introduce a
           binary variable */
        IloNumVarArray x(env, n_of_vertices, 0, 1, ILOINT);

        // preparing warm-start
        IloNumVarArray mipVar(env);
        IloNumArray mipVal(env);
        if (warm_start) {
            for (int i = 0; i < n_of_vertices; ++i) {
                mipVar.add(x[i]);
                if (int((best_sol.vertices).count(i)) > 0) mipVal
                    .add(1);
                else mipVal.add(0);
            }
        }
        // end preparing warm-start

        // generating the objective function
        IloExpr obj(env);
        for (int i = 0; i < n_of_vertices; ++i) obj += x[i];
        model.add(IloMinimize(env, obj));
        obj.end();

        // generating the constraints
        for (int i = 0; i < n_of_vertices; ++i) {
            IloExpr expr(env);
            expr += x[i];
            for (set<int>::iterator sit = neigh[i].begin(); sit
                != neigh[i].end(); ++sit) expr += x[*sit];
            model.add(expr >= 1);
            expr.end();

            /* the values of those variables whose vertices are
               not in the sub-instance are fixed to zero */
            if (age[i] == -1){
                IloExpr expr1(env);
                expr1 += x[i];
                model.add(expr1 == 0);
                expr1.end();
            }
        }
        IloCplex cpl(model);

        /* the aborter stops CPLEX once a better solution than
           "best_sol" is found */
        if (cplex_abort) {
            IloCplex::Aborter abo(env);
            cpl.use(abo);
            cpl.use(abortCallback(env, abo, best_sol.score));
        }
        if (warm_start) cpl.addMIPStart(mipVar, mipVal);
        cpl.setParam(IloCplex::TiLim, r_limit);
        cpl.setParam(IloCplex::EpGap, 0.0);
        cpl.setParam(IloCplex::EpAGap, 0.0);
        cpl.setParam(IloCplex::Threads, 1);
        if (heuristic_emphasis) cpl.setParam(IloCplex::Param::
            Emphasis::MIP, 5);
        cpl.setWarning(env.getNullStream());

        // calling CPLEX to solve the model
        cpl.solve();

        /* the following is done if CPLEX found at least one
           feasible solution */
        if (cpl.getStatus() == IloAlgorithm::Optimal
                or cpl.getStatus() == IloAlgorithm::Feasible) {
            cpl_sol.score = 0;
            IloNumArray x_val(env);
            cpl.getValues(x_val, x);
            for (int i = 0; i < n_of_vertices; ++i) {
                // ADAPT-STEP of CMSA
                if (age[i] >= 0) {
                    /* increment the age of all vertices in the
                       sub-instance */
                    age[i] += 1;
                    if (double(x_val[i]) > 0.8) {
                        cpl_sol.score += 1;
                        /* set the age of all vertices from the
                           best CPLEX solution to zero */
                        age[i] = 0;
                        (cpl_sol.vertices).insert(i);
                    }
                    /* remove all vertices whose age has reached
                       the age limit from the sub-instance */
                    if (age[i] >= age_limit) {
                        age[i] = -1;
                    }
                }
            }
        }
    }
    catch (IloException& e) {
        cerr << "Concert exception caught: " << e << endl;
    }
    env.end();
```

```
}

/* function that probabilistically generates a solution with a
    greedy bias */
void generate_solution(Solution& greedy_sol, vector<int>& age,
      default_random_engine& generator,
      uniform_real_distribution<double>& standard_distribution) {

    greedy_sol.score = 0;
    vector<bool> allready_covered(n_of_vertices, false);
    int num_nodes_uncovered = n_of_vertices;
    vector< set<int> > uncovered_neighbors = neigh;

    set<int> candidates;
    for (int i = 0; i < n_of_vertices; ++i) candidates.insert(i);

    while (num_nodes_uncovered > 0) {
        vector<Option> choice;
        double max_val = -1.0;
        set<int> max_vertices;

        /* generate all options for the extension of the current
           partial solution */
        for (set<int>::iterator cit = candidates.begin();
                 cit != candidates.end(); ++cit) {
            Option opt;
            opt.vertex = *cit;
            opt.value = double(uncovered_neighbors[*cit].size());
            if (opt.value >= max_val) {
                if (opt.value > max_val) {
                    max_val = opt.value;
                    max_vertices.clear();
                }
                max_vertices.insert(*cit);
            }
            choice.push_back(opt);
        }
        sort(choice.begin(), choice.end(), option_compare);

        int chosen_vertex;
        double dec = standard_distribution(generator);
        if (dec > determinism_rate) {
            int max = candidate_list_size;
            if (int(choice.size()) < candidate_list_size)
                max = int(choice.size());
            double rnum = standard_distribution(generator);
            int pos = produce_random_integer(max, rnum);
            chosen_vertex = choice[pos].vertex;
        }
        else {
            double rnum = standard_distribution(generator);
            chosen_vertex = get_random_element(max_vertices, rnum
                );
        }
        (greedy_sol.vertices).insert(chosen_vertex);

        /* MERGE-STEP of CMSA: if a vertex chosen for the current
           solution does not yet form part of the sub-instance,
           add it to the sub-instance by initializing its age
           value to zero */
        if (age[chosen_vertex] == -1) {
            age[chosen_vertex] = 0;
        }

        if (not allready_covered[chosen_vertex]) {
            allready_covered[chosen_vertex] = true;
            --num_nodes_uncovered;
        }
        greedy_sol.score += 1;

        num_nodes_uncovered -= int(uncovered_neighbors[
            chosen_vertex].size());
        set<int> to_cover = uncovered_neighbors[chosen_vertex];
        for (set<int>::iterator sit = to_cover.begin();
                  sit != to_cover.end(); ++sit) {
            allready_covered[*sit] = true;
            for (set<int>::iterator ssit = neigh[*sit].begin();
                ssit != neigh[*sit].end(); ssit++)
                    uncovered_neighbors[*ssit].erase(*sit);
        }
        uncovered_neighbors[chosen_vertex].clear();

        for (set<int>::iterator sit = neigh[chosen_vertex].begin
            (); sit != neigh[chosen_vertex].end(); sit++)
            uncovered_neighbors[*sit].erase(chosen_vertex);
        candidates.erase(chosen_vertex);
        set<int> to_delete;

        for (set<int>::iterator cit = candidates.begin(); cit !=
            candidates.end(); ++cit) {
            if (int(uncovered_neighbors[*cit].size()) == 0 and
                allready_covered[*cit]) to_delete.insert(*cit);
        }
        for (set<int>::iterator sit = to_delete.begin(); sit !=
            to_delete.end(); ++sit) candidates.erase(*sit);
    }
}


/**********
Main function
**********/

int main( int argc, char **argv ) {

    if (argc < 3) {
        cout << "Usage: ./cmsa -i <input_file> ..." << endl;
        exit(1);
    }
    else read_parameters(argc,argv);

    std::cout << std::setprecision(2) << std::fixed;

    // initializes the random number generator
    unsigned seed = std::chrono::system_clock::now().
        time_since_epoch().count();
    std::default_random_engine generator(seed);
    std::uniform_real_distribution<double>
        standard_distribution(0.0,1.0);

    ifstream indata;
    indata.open(input_file.c_str());
    if(!indata) {
        cout << "Error: file could not be opened" << endl;
    }

    // reading the problem instance file
    indata >> n_of_vertices;
    neigh = vector< set<int> >(n_of_vertices);
    int v1, v2;
    while (indata >> v1 >> v2) {
        neigh[v1].insert(v2);
        neigh[v2].insert(v1);
    }
    indata.close();

    /* "age" is an integer vector that contains the age of all
       vertices. An age of -1 means that the vertex does not form
       part of the sub-instance, while an age of >= 0 means that
       the vertex forms part of the sub-instance. */
    vector<int> age(n_of_vertices, -1);

    Solution best_sol;
    best_sol.score = std::numeric_limits<int>::max();

    // the computation time starts now
    clock_t start = clock();

    // variable ctime stores the current time that has passed
    double ctime = 0.0;

    bool stop = false;

    // main loop of the CMSA algorithm
    while (not stop and (ctime < computation_time_limit)) {

        // CONSTRUCT-STEP of CMSA: generate "n_of_sols" solutions
        for (int na = 0; na < n_of_sols; ++na) {
            Solution greedy_sol;
            generate_solution(greedy_sol, age, generator,
                standard_distribution);
            if (greedy_sol.score < best_sol.score) {
                best_sol = greedy_sol;
                clock_t current = clock();
                ctime = double(current - start) / CLOCKS_PER_SEC;
                cout << "best " << best_sol.score << "\ttime " <<
                     ctime << "\tgreedy" << endl;
            }
        }

        /* calculate the time "r_limit" given to CPLEX for the
        next application to the current sub-instance */
        clock_t current = clock();
        ctime = double(current - start) / CLOCKS_PER_SEC;
        double r_limit = computation_time_limit - ctime;
        if (r_limit > cplex_time_limit) r_limit =
            cplex_time_limit;

        /* if the remaining computation time is less than 0.1
        seconds, it does not make sense to call CPLEX.
        Variable 'stop' is set to true and the algorithm stops */
        if (r_limit < 0.1) stop = true;
        if (not stop) {
            Solution cpl_sol;
            /* SOLVE-STEP of CMSA: apply CPLEX to the current
               sub-instance */
            run_cplex(cpl_sol, best_sol, age, r_limit);
            if (cpl_sol.score < best_sol.score) {
                best_sol = cpl_sol;
                current = clock();
                ctime = double(current - start) / CLOCKS_PER_SEC;
                cout << "best " << best_sol.score << "\ttime " <<
                     ctime << "\tcplex" << endl;
            }
            current = clock();
            ctime = double(current - start) / CLOCKS_PER_SEC;
        }
    }
}
```

Index

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. Blum, *Construct, Merge, Solve & Adapt*, Computational Intelligence Methods and Applications, https://doi.org/10.1007/978-3-031-60103-3