Leveraging Machine Learning for Guiding Metaheuristics in Combinatorial Optimization

Jaume Reixach Pérez

Leveraging Machine Learning for Guiding Metaheuristics in Combinatorial Optimization

A dissertation presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

by

Jaume Reixach Pérez

Director:

Dr. Christian Blum

Institut d'Investigació en Intel·ligència Artificial, CSIC
Programa de Doctorat en Informàtica
Universitat Autònoma de Barcelona
November, 2025

ABSTRACT

Combinatorial optimization is a branch of mathematical optimization concerned with problems where the solution space is finite but often too large for exhaustive search. Such problems arise in diverse domains, including logistics, bioinformatics, and scheduling. Two main approaches exist for tackling them: exact methods, which guarantee optimal solutions but scale poorly, and approximate methods, which trade optimality for efficiency. Within approximate methods lies the concept of metaheuristics, which are general frameworks that can be adapted to multiple combinatorial optimization problems. Notable examples include Ant Colony Optimization, Genetic Algorithms, and Tabu Search.

Recently, the field of machine learning has attracted considerable attention, fueled by advances in computation and numerous breakthroughs. In particular, a rather novel and promising application is its integration into combinatorial optimization algorithms, particularly metaheuristics. In this context, learning can take place either offline, with models trained before the algorithm's execution, or online, with models updated dynamically during the search.

This thesis investigates both paradigms. On the offline side, it introduces an evolutionary framework for learning heuristic information to guide metaheuristics. The framework is employed in three settings: a genetic algorithm and a beam search applied to string-based optimization problems, and the Clarke and Wright heuristic for vehicle routing problems. On the online side, two novel variants of the hybrid metaheuristic Construct, Merge, Solve, and Adapt (CMSA) are proposed. Both integrate feedback from the exact solver used in the *solve* step of CMSA to improve solution construction, the first through a reinforcement learning–inspired mechanism and the second via deep learning, enabling richer adaptation during the search.

Keywords: *Metaheuristics*, *Machine Learning*, *Neural Network*, *Evolutionary Computation*, *Genetic Algorithm*, *Longest Common Subsequence Problem*, *Electric Vehicle Routing Problem*, *Beam Search*, *Construct Merge Solve and Adapt*, *CPLEX*, *Reinforcement Learning*, *Deep Learning*, *Minimum Dominating Set Problem*, *Far From Most String Problem*, *Maximum Independent Set Problem*.

RESUMEN

La optimización combinatoria es una rama de la optimización matemática que se ocupa de problemas cuyo espacio de soluciones es finito, pero a menudo demasiado grande para ser explorado exhaustivamente. Este tipo de problemas surge en dominios diversos, como la logística, la bioinformática y la planificación. Existen dos enfoques principales para abordarlos: los métodos exactos, que garantizan soluciones óptimas pero no escalan bien, y los métodos aproximados, que sacrifican optimalidad a cambio de eficiencia. Dentro de los métodos aproximados se encuentra el concepto de metaheurísticas, que son marcos generales adaptables a múltiples problemas de optimización combinatoria. Ejemplos notables incluyen la Optimización por Colonia de Hormigas, los Algoritmos Genéticos y la Búsqueda Tabú.

Recientemente, el campo del aprendizaje automático ha atraído una atención considerable, impulsado por los avances en computación y numerosos descubrimientos. En particular, una aplicación relativamente novedosa y prometedora es su integración en algoritmos de optimización combinatoria, especialmente en metaheurísticas. En este contexto, el aprendizaje puede producirse de manera *offline*, con modelos entrenados antes de la ejecución del algoritmo, o de manera *online*, con modelos que se actualizan dinámicamente durante la búsqueda.

Esta tesis investiga ambos paradigmas. En el ámbito offline, se introduce un marco evolutivo para aprender información heurística que guíe a las metaheurísticas. El marco se emplea en tres escenarios: un algoritmo genético y un algoritmo *beam search* aplicados a problemas de optimización sobre cadenas de caracteres, y la heurística de Clarke y Wright para problemas de ruteo de vehículos. En el ámbito online, se proponen dos nuevas variantes de la metaheurística híbrida Construir, Fusionar, Resolver, y Adaptar (CMSA). Ambas integran feedback del solucionador exacto utilizado en el paso *resolver* de CMSA para mejorar la construcción de soluciones: la primera mediante un mecanismo inspirado en el aprendizaje por refuerzo, y la segunda a través del aprendizaje profundo, lo que permite una adaptación más completa durante la búsqueda.

RESUM

L'optimització combinatòria és una branca de l'optimització matemàtica que s'ocupa de problemes l'espai de solucions dels quals és finit, però sovint massa gran per ser explorat exhaustivament. Aquest tipus de problemes apareix en àmbits diversos, com la logística, la bioinformàtica i la planificació. Hi ha dos enfocaments principals per abordar-los: els mètodes exactes, que garanteixen solucions òptimes però no escalen bé, i els mètodes aproximats, que sacrifiquen optimalitat a canvi d'eficiència. Dins dels mètodes aproximats hi trobem el concepte de metaheurístiques, que són marcs generals adaptables a múltiples problemes d'optimització combinatòria. Alguns exemples destacats inclouen l'Optimització per Colònia de Formigues, els Algorismes Genètics i la Cerca Tabú.

Recentment, el camp de l'aprenentatge automàtic ha atret una atenció considerable, impulsada pels avenços en computació i nombrosos descobriments. En particular, una aplicació relativament nova i prometedora és la seva integració en algorismes d'optimització combinatòria, especialment en metaheurístiques. En aquest context, l'aprenentatge pot produir-se de manera *offline*, amb models entrenats abans de l'execució de l'algorisme, o de manera *online*, amb models que s'actualitzen dinàmicament durant la cerca.

Aquesta tesi investiga ambdós paradigmes. En l'àmbit offline, s'introdueix un marc evolutiu per aprendre informació heurística que guiï les metaheurístiques. El marc s'empra en tres escenaris: un algorisme genètic i un algortme *beam search* aplicats a problemes d'optimització sobre cadenes de caràcters, i l'heurística de Clarke i Wright per a problemes de ruteig de vehicles. En l'àmbit online, es proposen dues noves variants de la metaheurística híbrida Construir, Fusionar, Resoldre, i Adaptar (CMSA). Totes dues integren feedback del solucionador exacte utilitzat en el pas *resoldre* de CMSA per millorar la construcció de solucions: la primera mitjançant un mecanisme inspirat en l'aprenentatge per reforç, i la segona a través de l'aprenentatge profund, que permet una adaptació més completa durant la cerca.

ACKNOWLEDGEMENT

The success of a PhD depends on many factors, such as motivation, perseverance, and proactivity. In my experience, however, the most important factor isn't entirely under your control: it's the advisor you are assigned. In my case, I was truly lucky. Christian has been exceptional not only scientifically but also personally. He was always available, offering invaluable advice and guidance on countless occasions. I want to thank him first and foremost, for how supported and confident he made me feel throughout these years, and for how easy and enjoyable it was to work alongside him.

Another key factor for success is external support, and in that regard, I have also been very fortunate. I want to thank my girlfriend, Aina, for her unconditional support and for being there through the many moments when I felt stuck and uncertain about what to do next. I am also deeply grateful to my friends, with whom I've shared many trips, laughs, and much-needed breaks from research. I want to thank my family and my family-in-law for their constant support, especially my dad, with whom I lived during my PhD, and who was always there when I needed him most, as well as my dog and my cat, whose quiet company and unconditional affection kept my spirits up throughout this journey.

Finally, I am thankful for the resources and the friendly atmosphere at the Artificial Intelligence Research Institute (IIIA-CSIC). Although I was not there very often, I truly enjoyed the company of my colleagues in the second-floor office. I am also grateful to the Spanish Ministry of Science and Innovation for the funding that allowed me to attend several conferences and share my work with the research community. I would also like to thank the IIIA-CSIC computing cluster, which worked tirelessly day and night, rarely complained, and only occasionally crashed.

Contents

List of Figures

List of Tables

List of Algorithms

List of Terms and Abbreviations

BRKGA Biased Random Key Genetic Algorithm. vi, xvii, 25, 26, 27, 31, 35, 39, 40, 41, 44, 46, 47, 53, 58, 61, 63, 64, 79, 85, 153

BS Beam Search. v, vi, ix, xvii, 7, 8, 10, 11, 13, 25, 27, 31, 32, 33, 34, 35, 37, 40, 41, 46, 47, 49, 50, 51, 55, 56, 57, 58, 59, 60, 61, 62, 64, 66, 68, 69, 70, 85, 153, 154

CD Critical Difference. x, 120, 121, 123, 127, 130, 133, 136

CEVRP Capacitated Electric Vehicle Routing Problem. ix, x, xiv, 74, 75, 77, 80, 81, 82, 84, 85, 86, 88, 89, 90, 91, 92, 154

CMSA Construct, Merge, Solve, and Adapt. vii, x, xiv, xv, xvii, 5, 6, 8, 9, 11, 12, 13, 14, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 112, 113, 114, 116, 117, 118, 119, 120, 121, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 137, 138, 139, 141, 142, 146, 147, 151, 154, 155, 156, 157

CVRP Capacitated Vehicle Routing Problem. 71, 73

DL Deep Learning. vii, x, xvii, 4, 12, 13, 14, 96, 105, 106, 107, 108, 109, 110, 112, 113, 114, 117, 119, 121, 125, 126, 127, 128, 131, 133, 137, 138, 139, 141, 142, 146, 147, 151, 155, 156, 157

EC Evolutionary Computation. xvii, 8, 9, 19, 20

EVRP Electric Vehicle Routing Problem. vi, 14, 23, 71, 74, 75, 153, 154

EVRP-RJ-RT Electric Vehicle Routing Problem with Road Junctions and Road Types. ix, x, xiv, 74, 77, 78, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 154

FFMS Far From Most String. vii, viii, x, xiv, xv, 12, 14, 96, 105, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 126, 127, 137, 138, 139, 140, 141, 142, 146, 147, 155

GA Genetic Algorithm. ix, 3, 7, 8, 10, 13, 19, 20, 153

ILP Integer Linear Programming. vii, viii, 3, 113, 117, 118, 125, 131

LCS Longest Common Subsequence. v, ix, xvii, 11, 13, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 40, 42, 43, 44, 47, 55, 56, 57, 61, 153

LCSqS Longest Common Square Subsequence. v, vi, ix, xii, xvii, 10, 13, 25, 26, 27, 28, 30, 31, 32, 34, 36, 38, 39, 40, 41, 43, 46, 47, 49, 51, 52, 53, 54, 61, 63, 65, 153, 156

LLM Large Language Model. 6, 8

MDS Minimum Dominating Set. vii, viii, x, xi, xiv, 12, 13, 14, 96, 105, 113, 118, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 137, 142, 143, 144, 145, 146, 147, 151, 152, 155, 156

MIS Maximum Independent Set. viii, x, xi, xv, 12, 13, 14, 96, 105, 113, 118, 129, 130, 131, 133, 134, 135, 136, 137, 142, 147, 148, 149, 150, 151, 152, 155, 156

ML Machine Learning. v, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 21, 22, 23, 44, 53, 95, 153, 154, 156

RKGA Random Key Genetic Algorithm. v, xvii, 19, 20, 21, 22, 39, 49, 58, 63

RL Reinforcement Learning. vii, xvii, 4, 7, 8, 9, 12, 13, 14, 96, 99, 100, 102, 103, 104, 105, 106, 107, 109, 113, 114, 117, 119, 120, 121, 124, 126, 127, 128, 131, 133, 137, 138, 139, 141, 142, 146, 147, 151, 155, 156, 157

RLCS Restricted Longest Common Subsequence. v, vi, ix, xiii, 11, 13, 25, 26, 28, 30, 31, 32, 34, 36, 38, 55, 56, 57, 58, 60, 61, 62, 64, 65, 66, 67, 68, 70, 85, 153, 156

UCB Upper Confidence Bound. vii, 104, 105

VRP Vehicle Routing Problem. vi, 7, 11, 71, 72, 73, 74, 75, 76, 78, 80, 82, 84, 86, 88, 90, 92, 157

INTRODUCTION

0.1 Background and Motivation

Optimization consists of seeking a best element, with respect to some criteria, from a set of available alternatives. It has a broad range of applications across various fields, enhancing efficiency, reducing costs, and maximizing desired outcomes. In business, optimization is used for job scheduling to minimize makespan, in manufacturing to reduce waste while maximizing production, and in logistics to plan the most efficient routes [89]. With its vast number of applications, optimization plays a crucial role in addressing complex real-world problems.

More formally, an optimization problem can be defined as follows:

Definition 0.1.1. *Let E be a set, $F \subseteq E$ a subset of E and $f : F \longrightarrow \mathbb{R}$ a function. The optimization problem associated with (E, F, f) consists of finding $S^* \in F$ such that $f(S^*) \geq f(S)\ \forall\ S \in F$ (maximization) or such that $f(S^*) \leq f(S)\ \forall\ S \in F$ (minimization).*

In this context, E is called the set of candidate solutions, F the set of feasible solutions, f the objective function and S^* a best solution to the optimization problem.

A particular type of optimization problems are the ones which have a finite, although often extremely large, set of candidate solutions. These are called combinatorial optimization problems and are the ones on which we focus in this thesis. Methods for approaching combinatorial optimization problems can be divided into exact and approximate. Examples of exact methods include Branch-and-Bound [83] and Dynamic Programming [10], which systematically explore the solution space to guarantee optimality. Exact methods can be useful for particularly structured problems or for problem instances up to a certain size, but they are often impractical. This is because many combinatorial optimization problems fall into the category of NP-Hard problems [50], meaning informally that they are at least as hard as the hardest problems in the NP class.

To understand this classification, it is essential to introduce the complexity classes P and NP, which are defined in terms of decision problems, problems with a yes-or-no answer. Many combinatorial optimization problems have a corresponding decision problem that asks whether a feasible solution exists within a given bound. For example, in the Traveling Salesman Problem (TSP) [71], the optimization version seeks the shortest tour that visits all cities exactly once, while the decision version asks whether a tour exists with a total length of at most k, for a particular value k.

The complexity class P consists of decision problems that can be solved in polynomial time by a deterministic algorithm, meaning that their computational complexity scales at most polynomially with input size. In contrast, NP includes problems for which a given solution can be verified in polynomial time, even if finding the solution itself may be computationally difficult. A problem is NP-hard if it is at least as difficult as the hardest problems in NP, meaning that solving it efficiently would imply an efficient solution for all problems in NP.

For many combinatorial optimization problems, their decision problem counterparts are used to establish NP-hardness. If the decision version of an optimization problem is NP-hard, then the optimization version is at least as hard. This is because an efficient algorithm for the optimization problem would allow one to efficiently determine the answer to its decision version. This classification implies that no known algorithm can solve NP-hard optimization problems in polynomial time with respect to input size unless P = NP, a fundamental open question in theoretical computer science.

In practice, solving NP-hard optimization problems exactly is in general computationally infeasible for large instances due to their exponential time complexity. As many relevant combinatorial optimization problems are NP-Hard, a lot of work has been done regarding the development of approximate algorithms, which focus on finding high-quality solutions within reasonable time limits rather than guaranteeing optimality. Metaheuristics are a particular class of approximate algorithms that provide general algorithmic frameworks adaptable to different optimization problems [18]. These strategies guide the search process to efficiently explore the solution space, aiming to find optimal or near-optimal solutions. They can range from simple local search procedures to more complex learning-based strategies and are typically non-deterministic, meaning they may produce different solutions in different runs.

While metaheuristics are designed to be broadly applicable, they can still incorporate domain-specific heuristics, allowing them to balance generality and problem-specific efficiency. A key feature of metaheuristics is their ability to avoid getting trapped in certain regions of the search space, often through mechanisms such as diversification and intensification strategies. Many modern metaheuristics also leverage search history, dynamically adjusting their strategies based on past exploration to improve performance. Metaheuristics have become widely used for solving complex real-world optimization problems where exact methods are impractical.

They can be broadly categorized into trajectory-based methods and population-based methods. Trajectory-based methods iteratively refine a single solution by exploring the search space in a guided manner. Examples include Iterated Local Search [85] and Tabu Search [53]. In contrast, population-based methods maintain and improve a set of candidate solutions at each iteration. Notable examples include Genetic Algorithms (GAs) [54] and Ant Colony Optimization (ACO) [14].

The development of metaheuristics has evolved over several decades, driven by the need for efficient methods to tackle complex optimization problems. Early heuristic algorithms emerged in the mid-20th century, often tailored to specific problems, but lacked the flexibility to be widely applied. The term metaheuristic, introduced by Glover in 1986 in the context of Tabu Search [53], formalized the idea of high-level strategies that guide heuristic search processes. Around the same time, other influential metaheuristics were developed, including Simulated Annealing in 1983 [76], inspired by thermodynamic annealing, and GAs in 1975 [61], which mimic evolutionary principles such as selection, crossover, and mutation. The 1990s saw significant growth in the field, with the introduction of ACO in 1992 [41], inspired by the foraging behavior of ants, and Particle Swarm Optimization (PSO) in 1995 [72], which modeled the collective behavior of bird and fish swarms. This part of the history of metaheuristics is known as the method-centric period [119], during which multiple methods were proposed, and the field of metaheuristics bloomed, which can be noted by the creation of the Metaheuristics International Conference and the Journal of Heuristics, both in 1995, which were the first outlets dedicated to publishing research in metaheuristics.

Since the 2000s, research in metaheuristics has focused on their hybridization. Researchers have focused on combining ideas from different frameworks into a single heuristic algorithm. Some combinations have become more popular than others, such as the use of GRASP to generate solutions that are then combined using path relinking [108], or the use of local search within GAs [95]. Moreover, these hybridizations are not restricted to combining components of a metaheuristic with components of another. An important type of hybridizations involves using exact methods, such as Integer Linear Programming (ILP) solvers, within metaheuristics [19].

Parallel to this work on hybridization, a significant number of so-called “novel” metaphor-based metaheuristics have been and are still being published. However, these approaches do not introduce real innovations; instead, they merely repackage concepts from existing metaheuristics under different metaphors. While this body of literature cannot be ignored, it does not constitute true scientific progress and only contributes to confusion in the field, see for example [6].

Machine Learning (ML) [12] is a subfield of Artificial Intelligence concerned with designing algorithms that can improve their performance on a task by learning from data, rather than relying exclusively on explicit programming. The origins of ML can be traced back to the mid-20th century, when early concepts such as perceptrons and nearest-neighbor methods were first proposed. The 1980s and 1990s brought important developments in statistical learning theory and kernel methods, culminating in algorithms such as Support Vector Machines that remain influential today. More recently, advances in computing power, data availability, and optimization techniques have fueled the rapid rise of Deep Learning (DL) [56], which has become the dominant paradigm in modern ML.

ML has achieved remarkable success across a wide range of application domains. In computer vision, deep convolutional networks (CNNs) [78] have enabled breakthroughs in image recognition, object detection, and medical imaging. In natural language processing, transformer-based architectures now power state-of-the-art systems for translation, text generation, and information retrieval. Reinforcement Learning (RL) [122] has been responsible for notable achievements such as AlphaGo [117], which demonstrated superhuman performance in the game of Go. Beyond these high-profile examples, ML has also been widely adopted in areas such as recommendation systems, fraud detection, autonomous driving, and scientific discovery, where the ability to identify patterns in large datasets is highly valuable.

The defining strength of ML lies in its adaptability: models can be trained on data to automatically discover representations and decision rules that generalize to unseen situations. This stands in contrast to manually crafted heuristics, which rely heavily on human intuition and domain expertise. As such, ML is increasingly being considered not only as a standalone field but also as a complementary tool for advancing other areas of computer science.

Among these diverse applications, one area that has recently gained particular traction is the hybridization of metaheuristics with ML, which is the research line explored in this thesis. This is driven by the goal of enhancing their efficiency, adaptability, and overall performance. Traditional metaheuristics depend on manually designed strategies for balancing exploration and exploitation. However, ML can improve these processes by adapting algorithmic components on the basis of patterns extracted from past searches or problem instances.

There are two primary approaches for incorporating ML into metaheuristics: online learning and offline learning [123]. In online learning, learning occurs during the execution of the metaheuristic, where the algorithm continuously refines its strategies based on real-time feedback from the search process. This allows for dynamic adjustments. Conversely, offline learning takes place before the optimization process begins. It leverages historical data or problem-specific knowledge to tune parameters, generate heuristics, or train surrogate models. The pre-trained knowledge is then applied during the optimization run.

By embedding ML into metaheuristics through these approaches, the algorithms gain an adaptive intelligence that reduces reliance on handcrafted rules, improving their ability to navigate complex search spaces.

This thesis presents an offline framework for learning search components of metaheuristics in its first part, and two online learning approaches for improving Construct, Merge, Solve, and Adapt (CMSA) [13], a hybrid metaheuristic from the literature, in the second.

0.2 Literature Overview of Machine Learning in Metaheuristics

The survey by Talbi [123] provides a taxonomy that separates the integration of ML in metaheuristics into three main levels: problem-level, high-level, and low-level. Each level solves a different bottleneck of metaheuristics: problem-level integrations reduce solution evaluation cost or allow modeling unknown objectives; high-level ones allow to choose or compose entire metaheuristics; and low-level ones improve the behavior of internal operators.

Furthermore, as already introduced, applications within each category can be distinguished based on the timing of the learning process. In offline learning, ML models are trained before the execution of the metaheuristic, while in online learning, the learning process occurs dynamically during the search, leveraging data collected in real-time.

At the problem level, ML is employed to model the optimization problem, with one notable application being the use of ML models as surrogates for objective functions [69], which can be beneficial when solution evaluations are computationally expensive. A representative example of this approach is provided in [44], in which the combinatorial optimization problem arising in partition-based ensemble learning is tackled. In this paradigm, a pool of base ML models is divided into disjoint subsets (partitions), and the final prediction is obtained by aggregating the outputs within each subset and then combining them. The optimization problem therefore consists of deciding how to form these subsets. Hence, in their setting, evaluating the quality of a candidate solution requires training and validating predictive models, a process that is computationally expensive and impractical for large numbers of evaluations. To mitigate this cost, the authors integrate a surrogate model within an evolutionary algorithm, allowing fitness values of candidate solutions to be estimated rather than computed exactly. The surrogate is periodically retrained during the search, ensuring that it reflects the regions of the search space currently explored by the algorithm. Their results show that this strategy substantially reduces the number of true evaluations required while maintaining solution quality, illustrating the effectiveness of surrogate assistance when evaluation costs dominate the optimization process.

The high-level category encompasses the application of ML for selecting or generating entire metaheuristics. A prominent example of high-level integration is the algorithm selection paradigm, where models are trained to decide which metaheuristic or algorithm to apply to a given problem instance. As surveyed by Kerschke et al. [73], this approach has been widely studied in combinatorial optimization domains such as SAT, CSP, and planning. The key idea is to extract features that characterize problem instances and then use ML models to estimate the performance of candidate algorithms. The selector then chooses, for each new instance, the algorithm expected to perform best. A well-known example is SATzilla [130], which employs supervised learning to select among a portfolio of SAT solvers. This strategy effectively automates the high-level decision of algorithm choice, shifting the burden from human expertise to data-driven models and enabling portfolios of complementary metaheuristics to be exploited more effectively.

Another more recent example of high-level integration can be found in [113], where the authors explore the use of Large Language Models (LLMs) to enhance metaheuristics. In their work, an LLM is leveraged to propose alternative heuristics for the *construct* step of CMSA. By generating novel heuristic variations, the LLM effectively influences the high-level structure and behavior of the metaheuristic, rather than simply guiding low-level operators or tuning parameters. Their results show that the heuristics proposed by the LLM can outperform expert-designed heuristics, demonstrating the potential of generative models to automate and improve the design of combinatorial optimization algorithms.

Finally, low-level integration, involves using ML to guide specific components of the search process, such as generating high-quality initial solutions [94] or tuning algorithm parameters [84]. The work presented in this thesis falls within this category. Regarding the timing of learning, the first part of the thesis belongs to the offline class, while the second belongs to the online one. We now provide an overview of the literature related to this category of low-level integration.

Following the classification presented in the already mentioned survey, the low-level ML applications can be classified as follows: initial solution generation, search operator design, search operator selection, and parameter tuning.

Regarding initial population generation, metaheuristics usually generate initial solutions randomly or by means of a simple constructive heuristic. Using ML for this purpose can provide initial solutions of higher quality or greater diversity. Although most work on end-to-end solution generation does not explicitly aim at initializing metaheuristics, the methods developed in this area can be naturally repurposed for that goal. For instance, Vinyals et al. [125] introduced Pointer Networks, a neural architecture that learns to generate complete solutions to problems such as the Traveling Salesman Problem (TSP). Similarly, Kool et al. [77] proposed an attention-based model trained with RL to output high-quality tours for routing problems. While these models are typically evaluated as standalone solvers, their ability to produce diverse and competitive solutions suggests that they could serve as powerful generators of initial populations for metaheuristics, potentially improving convergence speed and solution quality.

In search operator design, the ML model directly influences how the metaheuristic explores the solution space by leveraging learned information. Our contributions fall within this category. The offline framework introduced in the first part of the thesis is presented in the context of three algorithms: a GA, a Beam Search (BS), and the Clarke and Wright heuristic for the Vehicle Routing Problem (VRP). Though applicable to broader tasks such as solution initialization or parameter tuning, in the three cases it aids in defining search operators. For the GA, the ML model is trained to generate high-quality individuals, that serve to bias the search process by making individuals of the GA more similar to them. In the case of the BS, the ML model serves as the heuristic function, guiding the search process by selecting which nodes to expand. Lastly, in the Clarke and Wright heuristic, the ML model replaces the savings values computations which guide the search process by deciding which routes to merge at each step. The second part of the thesis introduces two online learning approaches for incorporating learning capabilities into the *construct* step of CMSA, one of its key search operators.

Several related works from the literature can be grouped along the same line of learning search operators, which can be further classified depending on whether the learning occurs offline or online.

In offline approaches, the ML model is trained prior to the execution of the metaheuristic and then integrated as a fixed component during search. A well-known example is DeepACO [131], where a graph neural network is trained to provide learned heuristic information to Ant Colony Optimization (ACO), effectively biasing the solution construction toward promising search space regions. Similarly, NeuroLKH [129] enhances the Lin-Kernighan-Helsgaun (LKH) heuristic for the Traveling Salesman Problem (TSP) by learning edge scores to guide local search, replacing handcrafted measures with predictions from a neural model. Both works are closely related to our own applications, where ML models are trained to provide problem-specific guidance: in the case of the GA, by biasing the generation of individuals toward high-quality solutions that act as heuristic information, and in the case of the Clarke and Wright algorithm, by replacing handcrafted savings values with learned estimates.

Another relevant line of research focuses on BS. In this context, Huber et al. [63] and Ettrich et al. [46] both train models offline to approximate the heuristic function that evaluates partial solutions, demonstrating substantial improvements over manually designed heuristics. These works directly parallel our integration of ML into BS, where the learned model likewise replaces the heuristic function guiding node expansion. The key difference lies in the training methodology: while the mentioned papers employ RL, our approach relies on the proposed evolutionary learning framework, offering an alternative and more general strategy for obtaining effective heuristic functions.

In the context of Evolutionary Computation (EC), [112] proposes a GA in which a graph neural network (GNN) [115] is trained offline to bias the generation of offspring solutions for the multi-hop influence maximization problem. This is closely related to our own use of a GA for training within the offline learning framework, as a GA is likewise employed as the training mechanism for the learned model. In a different and more innovative direction, Chacón Sartori et al. [114] explore the integration of LLMs into metaheuristics, leveraging their pre-trained knowledge to generate problem-specific guidance. This LLM-based approach illustrates a novel way of injecting external knowledge into metaheuristic operators, complementing more traditional ML-driven strategies.

These contributions illustrate the diversity of offline learning approaches, which are typically tailored to a specific algorithm or problem. In contrast, the offline framework introduced in this thesis aims at general applicability, offering a methodology that can be adapted across different metaheuristics and problems.

Online approaches, where the ML model is updated during the run of the metaheuristic, are far less common. Most existing works focus on operator selection rather than operator design. For example, multi-armed bandit strategies have been used in hyper-heuristics to adaptively select operators during search [25, 81]. Another example regards [27], where the selection of operators employed by an EC algorithm is learned online during its execution. While these methods demonstrate the potential of online learning, they address which operator to use, rather than how operators are designed.

Closer to operator design is the efficient active search framework [62], where a pretrained neural model for combinatorial optimization is fine-tuned at test time using feedback from the instance being solved. Although not strictly a metaheuristic, this approach exemplifies the potential of adapting constructive operators online.

The online learning mechanisms presented in the second part of this thesis fall squarely into this underexplored area. By continuously adapting the *construct* step of CMSA using feedback gathered during the run, our contributions represent an instance of online operator design, pushing beyond the prevalent focus on operator selection and highlighting new possibilities for hybrid metaheuristics with dynamic learning capabilities.

Next, search operator selection consists of methods that maintain a pool of available operators and use an ML model to decide which one to apply at each step. The two online methods mentioned earlier fall into this category. A more recent example is provided by Johnn et al. [70], who address the selection of Large Neighborhood Search (LNS) operators. In their framework, each destroy–repair operator pair is treated as an action in a Markov Decision Process (MDP) [122]. To model the state, they extract structural features from the current solution and represent them using a graph-based encoding. A deep RL agent is then trained offline to learn a policy that selects the most promising operator given the state of the search. This approach enables the algorithm to adaptively exploit different neighborhoods during execution, outperforming static or manually designed selection strategies.

Finally, parameter tuning is the last considered class of low-level integration. Broadly, three strategies can be distinguished. First, some approaches focus on identifying high-performing parameter configurations for a given problem class or subset of instances. Second, instance-specific models have been developed that extract descriptive features of problem instances and predict suitable parameter settings accordingly. Third, adaptive methods adjust parameters dynamically during the search itself. For example, Hutter et al. [64] employ a random forest model to predict the performance of candidate parameter configurations, enabling efficient offline tuning across instances. In contrast, Lessmann et al. [84] use regression models in an online setting to adapt parameters on the fly, tailoring the configuration to the course of a particular search run.

0.3 Thesis Contributions

This thesis contributes to the expanding body of research regarding the use of ML within combinatorial optimization algorithms, in particular metaheuristics. Two main research lines are explored, offline learning and online learning.

Regarding offline learning, the main contribution consists of the development of a general offline evolutionary framework for learning specific components of the search process. This framework consists of having an ML model that parametrizes some search component of a metaheuristic. This could be a parameter setting, a heuristic function that guides the search process, or even a set of initial solutions. Given a problem instance, the model takes extracted features from the instance as input and outputs the corresponding predicted search component. The training process employs a GA and a set of training and validation instances. Each individual of the GA represents a set of model parameters, so that the training consists of the execution of the GA. This framework is implemented into three algorithms applied to three different combinatorial optimization problems.

- The first application is done in the context of another GA. In particular, the ML model's role is to produce a promising individual for a given problem instance. This promising individual is used as search guidance by biasing each individual toward it before applying the decoder. This GA is applied to the Longest Common Square Subsequence (LCSqS) problem, a combinatorial optimization problem that given a set of input strings aims to find a common subsequence as long as possible that is the concatenation of some other string with itself. The results show that the ML-guided GA is statistically superior to the standard GA when tackling instances with non-uniformly generated strings. It is currently the state-of-the-art approach for the LCSqS problem.
- In the second application, the ML model parametrizes the heuristic function of BS. BS is a tree search algorithm that, starting at the root node, at each

iteration expands a certain number of nodes of the current depth level depending on a parameter. The nodes to be expanded are decided using a heuristic function, which indicates the quality of expanding a node. This BS is applied to another string-related combinatorial optimization problem known as the Restricted Longest Common Subsequence (RLCS) problem. Given two sets of strings, this problem aims to find the Longest Common Subsequence (LCS) of the first set that does not contain any string of the second set a subsequence itself. Again, the results show that the ML framework improves algorithmic performance. Also in this case, our best-performing algorithm variant is currently the state-of-the-art for the RLCS problem.

- The last application regards the Clarke and Wright heuristic, an algorithm used to obtain reasonably good solutions to VRPs. A VRP, in its simplest form, consists of a depot and a set of customers, each with a demand. The goal is to schedule a number of routes, each starting and ending at the depot, so that the demands of the customers are satisfied and so that the routing cost, which is usually energy or distance, is minimized. The Clarke and Wright heuristic starts with a trivial solution consisting of one route per customer, each going from the depot to the customer and back. It then iteratively merges the routes of this solution, at each step trying to perform the merge that leads to the highest saving in routing cost. The ML framework is used to replace the savings computations with a forward pass of a neural network, leading to improvements in the quality of the final solution.

In the context of online learning, our contribution consists of the development of two online learning mechanisms for improving the *construct* step of the CMSA metaheuristic. CMSA is a hybrid metaheuristic introduced in [21], which keeps a subinstance of the problem at hand to which it applies an exact solver at each iteration. The algorithm requires a problem-specific notion of solution components, a finite set $C = \{c_1, c_2, \ldots, c_n\}$ of elements so that every solution of the problem under consideration can be expressed as a subset of C. For example, in the case of the well-known Travelling Salesman Problem (TSP), a valid set of solution components is the set of edges of the complete input graph. The subinstance kept by CMSA is a subset of C and is modified throughout the execution of the algorithm with the goal that the exact solver can eventually find a good solution in it. CMSA consists of four iterated steps which give its name: the *construct*, *merge*, *solve*, and *adapt* steps. The *construct* step probabilistically generates a certain number of solutions, the *merge* step adds the solution component in the constructed solutions to the subinstance, the *solve* step applies the exact solver to the subinstance and finally, the *adapt* step removes the solution components of the subinstance that have not appeared in the solution of the exact solver by a certain number of iterations. In the standard CMSA the *construct* step generally employs a probabilistic greedy heuristic tailored to the problem at hand. Our two developments consist of replacing this method with ML-based approaches.

- The first proposed method is inspired by the multi-armed bandit problem, one of the simplest RL settings in which there is no concept of state and the agent simply iteratively performs an action from a finite set and observes a reward. In the *construct* step of CMSA actions correspond to solution components. The agent starts with an empty solution to which it iteratively adds available solution components until obtaining a complete solution. To decide within the available solution components, it keeps one value for each, which we denote as quality values. These quality values are initialized uniformly and are updated after every *solve* step, where the solution given by the exact solver is used as feedback. In particular, the solution components appearing in the solution given by the exact solver have their quality values increased, while the ones in the subinstance that were not selected by the exact solver have them decreased.

- The second approach extends the first one by taking into account the current partial solution under construction when selecting the next solution component to be added. This is done by leveraging DL. More particularly, a neural network is built that takes the partial solution as input and outputs one value for each solution component, which, as with the first approach, represent the quality of selecting each solution component and are used to derive sampling probabilities. As with the first method, the exact solver is used to perform the learning update, this time by performing a gradient descent step with the aim of leading the solutions constructed to the ones outputted by the exact solver.

These two ML-enhanced implementations of CMSA are evaluated on three combinatorial optimization problems: the Far From Most String (FFMS) problem, the Minimum Dominating Set (MDS) problem, and the Maximum Independent Set (MIS) problem. The FFMS problem involves finding a string that maximizes the number of input strings whose Hamming distance from it meets or exceeds a given threshold. The MDS and MIS problems, on the other hand, are graph-based.

In the MDS problem, the objective is to identify the smallest possible set of nodes such that every node in the graph either belongs to this set or has at least one neighbor in it. Conversely, the MIS problem seeks to determine the largest subset of nodes in which no two nodes are neighbors.

The first approach, referred to as RL-CMSA, outperforms the standard CMSA across all three problems. The second approach, DL-CMSA, named for its DL component, achieves superior results compared to RL-CMSA only in the MIS problem. Notably, incorporating DL to account for the partial solution under construction introduces computational overhead. This added complexity does not appear to be worth it for all problems.

0.4 Thesis Organization

The remainder of this thesis is divided into two parts: offline learning and online learning. As the names suggest, the first part focuses on developments related to the offline learning framework, while the second introduces the new CMSA variants that incorporate online learning mechanisms. The offline learning section is structured as follows.

- Chapter 1 introduces the offline learning framework within the broader context of learning a search component for a metaheuristic.
- Chapter 2 introduces the LCS, LCSqS, and RLCS problems, along with algorithmic components from the literature for the LCS problem. Specifically, it presents a Dynamic Programming algorithm and a BS approach, both of which are utilized by the GA for solving the LCSqS problem.
- Chapter 3 explores the application of the offline learning framework to identify promising individuals within the GA for the LCSqS problem. Furthermore, it includes an experimental evaluation demonstrating the advantages of the learning mechanism.
- Chapter 4 presents the application of the offline learning framework for learning the heuristic function of the BS for the RLCS problem. Additionally, it presents an experimental comparison between the BS using the learned heuristic function and the standard version.
- Chapter 5 presents the application of the offline learning framework to the Clarke and Wright heuristic. It presents an experimental evaluation on two

realistic variants of the Electric Vehicle Routing Problem (EVRP), which consider electric delivery vehicles.

The online learning part is organized as follows.

- Chapter 6 presents the standard CMSA.
- Chapter 7 introduces RL-CMSA, the learning variant of CMSA that incorporates a basic RL mechanism.
- Chapter 8 presents DL-CMSA, which implements a more detailed learning mechanism by incorporating partial solution information through DL.
- Chapter 9 experimentally evaluates the CMSA variants on three combinatorial optimization problems: the FFMS, the MDS, and the MIS problems. Moreover, an in-depth algorithmic analysis is performed to better understand the results obtained.

Finally, Chapter 10 concludes the thesis with a summary and sketches possible future research directions related to the research presented.

0.5 Publications

Most of the content presented in this thesis has already been published in the following publications.

1. **Jaume Reixach**, Christian Blum, Marko Djukanović and Günther R. Raidl, "A Biased Random Key Genetic Algorithm for Solving the Longest Common Square Subsequence Problem," in IEEE Transactions on Evolutionary Computation. `https://doi.org/10.1109/TEVC.2024.3413150` [JCR Impact Factor 2024: 12]

2. **Jaume Reixach**, Christian Blum, Marko Djukanović, Günther R. Raidl. (2024). A Neural Network Based Guidance for a BRKGA: An Application to the Longest Common Square Subsequence Problem. Evolutionary Computation in Combinatorial Optimization. EvoCOP 2024. Lecture Notes in Computer Science, vol 14632. Springer, Cham. `https://doi.org/10.1007/978-3-031-57712-3_1` [CORE B conference, won best paper award]

3. **Jaume Reixach**, Christian Blum (2024). Extending CMSA with Reinforcement Learning: Application to Minimum Dominating Set. Metaheuristics. MIC 2024. Lecture Notes in Computer Science, vol 14754. Springer, Cham. `https://doi.org/10.1007/978-3-031-62922-8_27`

4. Marko Djukanović, Aleksandar Kartelj, Tome Eftimov, **Jaume Reixach**, Christian Blum (2024). Efficient Search Algorithms for the Restricted Longest Common Subsequence Problem. Computational Science – ICCS 2024. ICCS 2024. Lecture Notes in Computer Science, vol 14836. Springer, Cham. https://doi.org/10.1007/978-3-031-63775-9_5

5. **Jaume Reixach**, Christian Blum. How to improve "construct, merge, solve and adapt"? Use reinforcement learning!. Annals of Operations Research (2024). https://doi.org/10.1007/s10479-024-06243-7. In press [JCR Impact Factor 2024: 4.5].

6. Christian Blum, **Jaume Reixach** (2025). The Hybrid Metaheuristic CMSA. In: Martí, R., Pardalos, P.M., Resende, M.G. (eds) Handbook of Heuristics. Springer, Cham. https://doi.org/10.1007/978-3-319-07153-4_79-1

7. Marko Djukanović, **Jaume Reixach**, Ana Nikolikj, Tome Eftimov, Aleksandar Kartelj, Christian Blum (2025). A learning search algorithm for the Restricted Longest Common Subsequence problem. Expert Systems with Applications, 127731. https://doi.org/10.1016/j.eswa.2025.127731 [JCR Impact Factor 2025: 7.5]

8. **Jaume Reixach**, Christian Blum (2025). Improving the CMSA Algorithm with Online Deep Learning. ECAI 2025. IOS Press https://doi.org/10.3233/FAIA251352 [CORE A conference]

9. (Under Review) **Jaume Reixach**, Mehmet Anıl Akbay, and Christian Blum (2025). Old Dog, New Tricks: Learning-Based Savings for the Clarke and Wright Algorithm. Submitted to a Q1 journal

Part I

Offline Learning

Chapter 1

The General Framework

This chapter introduces the general offline learning framework. We assume that we have a metaheuristic and a Machine Learning (ML) model that parametrizes one of its search components. For instance, this could consist of a neural network that parametrizes the initial population of a Genetic Algorithm (GA) or the probabilities of using each available neighborhood structure in a Variable Neighborhood Search (VNS).

As mentioned previously, the ML model is trained by a GA, the individuals of which represent values for its parameters. The particular GA employed is a Random Key Genetic Algorithm (RKGA), introduced in the following section.

1.1 Random Key Genetic Algorithm (RKGA)

Evolutionary Computation (EC) algorithms are metaheuristics inspired by the way nature adapts living organisms to their environment. They can be viewed as computational models that simulate evolutionary processes. These algorithms operate on a population of individuals, each representing a solution, which is iteratively modified using recombination and mutation operators. The goal is to progressively refine the population by favoring individuals associated with better solutions through a selection process.

Algorithm 1.1, adapted from [18], outlines the fundamental structure of EC algorithms when applied to optimization problems.

In each iteration, a set of new individuals is produced using recombination and mutation operators, followed by the application of a selection operator to form the next generation's population. In our context, each individual represents a set of parameter values for the ML model and selection is done by testing the quality of the parameters of the individuals on a set of training problem instances.

An RKGA is a particular EC algorithm first introduced in [8]. It belongs to the broader class of GAs [54, 61].

Like other EC algorithms, GAs operate on a population of individuals, each

Algorithm 1.1 The general structure of an EC algorithm

```
1: $P :=$ generate_initial_population()
2: evaluate($P$)
3: while termination conditions not met do
4:     $P' :=$ recombine($P$)
5:     $P'' :=$ mutate($P$)
6:     evaluate($P''$)
7:     $P :=$ select($P'' \cup P$)
8: end while
```

representing a solution to the optimization problem under consideration. These solutions are encoded as chromosomes, which are strings composed of multiple positions, known as genes. Each gene takes a value, called an allele, from a predefined set. The population undergoes an evolutionary process over multiple iterations, referred to as generations. In each generation, a new population is formed from the existing one through selection, mating and random mutation. The selection mechanism favors individuals with higher fitness values, aiming to enhance the overall quality of solutions over time.

In an RKGA, chromosomes are represented as vectors of real numbers within the interval $[0, 1]$. A key component of these algorithms is a decoder function, which converts each chromosome into a solution for the given optimization problem. Like any GA, an RKGA maintains a fixed-size population of individuals, the size of which is denoted as p_{size}. The initialization process assigns each allele in every chromosome a random value from the interval $[0, 1]$. Once initialized, the fitness of each individual is evaluated using the decoder function. The population is then divided into two groups: (i) the elite population, comprising the top p_e individuals and (ii) the non-elite population, which consists of the remaining $p_{\text{size}} - p_e$ individuals. The parameter p_e, known as the number of elites, is defined such that $p_e < p_{\text{size}} - p_e$. In addition, another parameter, p_m represents the number of mutants, where $p_m < p_{\text{size}} - p_e$. The three groups, elites, non-elites, and mutants are used to generate the next generation. The next population is constructed as follows: the elite individuals are carried over unchanged, while p_m new mutants are introduced, each generated randomly in the same manner as during initialization. The remaining $p_{\text{size}} - p_e - p_m$ individuals are produced through mating, where two parents are randomly selected from the population. For each gene, the offspring inherits the corresponding allele from either parent, chosen at random.

Algorithm 1.2 shows the main structure of an RKGA.

In this process, the decoder is employed within the evaluate() function, where

Algorithm 1.2 The general structure of an RKGA

```
Input: Values for parameters $p_{\text{size}}, p_e$, and $p_m$
1: $P$ := generate_initial_population()
2: evaluate($P$)
3: while termination conditions not met do
4:     $P'$ := new_empty_population()
5:     $P$ := sort_depending_on_fitness($P$)
6:     for $i \in \{1, \ldots, p_e\}$ do
7:         $P'_i := P_i$
8:     end for
9:     for $i \in \{1, \ldots, p_m\}$ do
10:        $P'_{p_e+i}$ := generate_random_individual()
11:    end for
12:    for $i \in \{1, \ldots, p_{\text{size}} - p_e - p_m\}$ do
13:        $P'_{p_e+p_m+i}$ := generate_offspring($P$)
14:    end for
15:    $P := P'$
16:    evaluate($P$)
17: end while
```

it decodes each individual and assigns them their corresponding fitness value. This step is essential to the algorithm as it is the only part that depends on the specific problem. The RKGA explores the hypercube $[0, 1]^n$, where n represents the number of genes in each chromosome. Through the use of the decoder, this search in the hypercube is effectively translated into a search within the feasible solution space of the optimization problem at hand.

1.2 The Training Procedure

Once the search component of the metaheuristic has been parameterized using an ML model, the only remaining task in the training process is to decide the number of genes of each chromosome and design function evaluate() of Algorithm 1.2, which assigns a fitness value to every individual. In the most straightforward way possible, we keep one gene for each parameter of the ML model, so that individuals directly represent parameter settings. As restricting parameters to the interval $[0, 1]$ might hinder the learning process, we decide to use the interval $[-1, 1]$ instead. Notice that the RKGA description given above still holds, as this is equivalent to the decoder applying $x \longmapsto 2x - 1$ to each gene. In general, a bigger interval could be considered as a larger range for the parameter values can improve the adaptability of the ML model employed. In our case, using $[-1, 1]$ already produced good results.

We opt to use two distinct sets of problem instances inside evaluate(), which we refer to as the training and validation set. These sets should consist of instances that are representative of the types of problem instances the metaheuristic will need to solve in practice. The training set is used to evaluate individuals, while the validation set serves as a means of detecting overfitting and determining when to terminate the training process.

Algorithm 1.3 presents the structure of the evaluate() function.

Algorithm 1.3 The evaluation function of the training RKGA

```
Input: Set of training and validation instances $I_{\text{training}}, I_{\text{validation}}$
Input: ML model $M$
Input: Population $P$
 1: for $i \in \{1, \dots, p_{\text{size}}\}$ do
 2:     Fit the model $M$ with the weights $P_i$
 3:     $v_{\text{training}} = 0$
 4:     for each training instance $I \in I_{\text{training}}$ do
 5:         $v_{\text{training}} = v_{\text{training}} + \text{evaluate\_training\_quality}(M, I)$
 6:     end for
 7:     $v_{\text{training}} = \frac{v_{\text{training}}}{\text{size}(I_{\text{training}})}$
 8:     Set the fitness of $P_i$ to $v_{\text{training}}$
 9:     if $v_{\text{training}}$ is the best training value so far then
10:         $v_{\text{validation}} = 0$
11:         for each validation instance $I \in I_{\text{validation}}$ do
12:             $v_{\text{validation}} = v_{\text{validation}} + \text{evaluate\_validation\_quality}(M, I)$
13:         end for
14:         $v_{\text{validation}} = \frac{v_{\text{validation}}}{\text{size}(I_{\text{validation}})}$
15:         Store $v_{\text{validation}}$
16:     end if
17: end for
```

As shown, the fitness of an individual is determined by the so-called training value v_{training}, which represents the average quality of the model with the weights of the individual evaluated on the training instances. If a new best individual is identified, its parameters are again evaluated by computing the so-called validation value $v_{\text{validation}}$ in order to check for overfitting. This value is similarly the average quality of the model with the individual's weights, but in this case, over the validation instances. This latter value can be used to decide when to stop the training process as its decrease might indicate overfitting. In our first two applications, presented in Chapters 3 and 4, we decided to employ early stopping, meaning that we stopped the training process whenever the validation value first decreased. For the Clarke and Wright application, presented in Chapter 5, the validation value fluctuated considerably, making the early stopping approach less reliable. In this case, we opted to train for a fixed time, selecting the weights corresponding to the lowest validation value as the trained weights once the training is finalized.

Functions evaluate_training_quality(M, I) and evaluate_validation_quality(M, I) of Algorithm 1.3 have to be designed depending on the metaheuristic, the problem and the ML model at hand. A straightforward approach is to have them consist of an execution of the metaheuristic guided by the model M on problem instance I. Other methods for evaluating quality could be considered, as running the metaheuristic might be too computationally expensive as the evaluation function is called many times during the training process.

In the following chapters, we demonstrate the application of this framework to three algorithms applied to different combinatorial optimization problems. The first two are applied to two different string-related problems, and the last one is applied to two variants of the Electric Vehicle Routing Problem (EVRP). The next chapter introduces the two string-related problems tackled and presents the methods from the literature employed by our approaches.

Chapter 2

The LCSqS and RLCS Problems

2.1 Introduction

This chapter provides a detailed introduction to the Longest Common Square Subsequence (LCSqS) and Restricted Longest Common Subsequence (RLCS) problems, the two string-related combinatorial optimization problems used as examples for applying the offline learning framework. Moreover, we introduce the algorithmic components from the literature employed by the Biased Random Key Genetic Algorithm (BRKGA) for solving the LCSqS and the Beam Search (BS) for solving the RLCS.

Both problems are variants of the well-known Longest Common Subsequence (LCS) problem, which we introduce first.

Definition 2.1.2 (String). *A string is a finite sequence of characters drawn from a finite set Σ, referred to as its alphabet. If s is a string with characters $\mathtt{s}^1, \mathtt{s}^2, \ldots, \mathtt{s}^n$, we denote it as $s = \mathtt{s}^1\mathtt{s}^2 \cdots \mathtt{s}^n$. The empty string, represented by ε, is the one that contains no characters.*

Definition 2.1.3 (String length). *Given a string s over an alphabet Σ, its length, denoted by $|s|$, is the number of characters it contains.*

Definition 2.1.4 (String subsequence). *Given a string s, a subsequence of s is a string formed by deleting zero or more characters from s without altering the order of the remaining characters.*

Definition 2.1.5 (LCS problem). *Let $\{s_1, s_2, \ldots, s_m\}$ be a finite set of strings over an alphabet Σ. The LCS problem with input strings $\{s_1, s_2, \ldots, s_m\}$ seeks to find the longest possible string that is a subsequence of all given input strings.*

Definition 2.1.6 (String concatenation). *Let $s_1 = \mathtt{s}_1^1\mathtt{s}_1^2 \cdots \mathtt{s}_1^{n_1}$ and $s_2 = \mathtt{s}_2^1\mathtt{s}_2^2 \cdots \mathtt{s}_2^{n_2}$ be two strings over an alphabet Σ. The concatenation of s_1 and s_2, denoted $s_1 \cdot s_2$, is the string formed by appending the characters of s_2 to the end of s_1: $s_1 \cdot s_2 = \mathtt{s}_1^1\mathtt{s}_1^2 \cdots \mathtt{s}_1^{n_1}\mathtt{s}_2^1\mathtt{s}_2^2 \cdots \mathtt{s}_2^{n_2}$.*

Definition 2.1.7 (Square string)**.** *Let s be a string over an alphabet Σ. s is a square string if it can be expressed as $s = t \cdot t$, where t is some string over Σ.*

Definition 2.1.8 (LCSqS problem)**.** *Let $\{s_1, s_2, \dots, s_m\}$ be a finite set of strings over an alphabet Σ. The LCSqS problem with input strings $\{s_1, s_2, \dots, s_m\}$ involves finding the longest possible square string that is a subsequence of all given input strings.*

Definition 2.1.9 (RLCS problem)**.** *Let $S = \{s_1, s_2, \dots, s_m\}$ and $R = \{r_1, r_2, \dots, r_k\}$ be two finite sets of strings over an alphabet Σ. The RLCS problem with input strings S and restricted strings R consists of finding the longest possible string that is a subsequence of all strings in S while ensuring that none of the strings in R appear as a subsequence.*

The BRKGA for the LCSqS problem builds on the following proposition. It enables obtaining an approximate solution by selecting a cut point for each string and then solving the LCS problem using the resulting strings as input.

Proposition 2.1.1 (LCSqS to LCS reduction)**.** The LCSqS problem with input strings $S = \{s_1, s_2, \dots, s_m\}$ can be solved by computing the LCS of the $2m$ strings in the set:

$$S_p = \{s_1[1, p_1], s_1[p_1 + 1, |s_1|], \dots, s_m[1, p_m], s_m[p_m + 1, |s_m|]\} \tag{2.1}$$

for every possible cut-point selection

$$p = (p_1, \dots, p_m) \in \mathcal{P} = \prod_{i=1}^{m} \{1, \dots, |s_i| - 1\} \tag{2.2}$$

In particular, if t' is a maximal-length solution among the $|\mathcal{P}|$ LCS problems, then $t = t' \cdot t'$ is a solution to the original LCSqS problem.

Proof. Assuming that the LCS problem has been solved for all possible $p \in \mathcal{P}$ with input strings S_p, we define $k = |\mathcal{P}| = \prod_{i=1}^{m}(|s_i| - 1)$. Furthermore, let $\{t_1, \dots, t_k\}$ be set of solutions to these k LCS problems.

We obtain a maximal-length solution to the previous problems $t' = \arg\max\{|t_1|, \dots, |t_k|\}$, and define $t = t' \cdot t'$. t is an optimal solution to the LCSqS problem with input string $S = \{s_1, s_2, \cdots, s_m\}$ since it is a square subsequence of every string in S and has maximal length.

The first claim follows directly from the construction: each string $s_i \in \{s_1, \dots, s_m\}$ contains t' as a subsequence twice. Once in $s_i[1, p_i]$ and once in $s_i[p_i + 1, |s_i|]$. In order to prove that t is of maximal length, we argue by contradiction.

Suppose there exists a square subsequence $s = s' \cdot s'$ of every string in S with $|s| > |t|$. For every $i \in \{1, 2 \ldots, m\}$ we define $q_i \in \{1, |s_i| - 1\}$ as the smallest index such that s' is a subsequence of $s_i[1, q_i]$ and $s_i[q_i + 1, |s_i|]$. By construction, s' must be a solution to the LCS problem with the following input strings:

$$\{s_1[1, q_1], s_1[q_1 + 1, |s_1|], \ldots, s_m[1, q_m], s_m[q_m + 1, |s_m|]\} \tag{2.3}$$

By definition of t' we have $|t'| \geq |s'|$, which implies:

$$|t| = |t' \cdot t'| = |t'| + |t'| \geq |s'| + |s'| = |s' \cdot s'| = |s| \tag{2.4}$$

This contradicts our initial assumption that $|s| > |t|$.

Thus, by solving the k we have optimally solved the LCSqS problem, with t as an optimal solution. □

Thanks to the previous proposition, the LCSqS problem with input strings S can be solved by solving the LCS problem with input strings S_p by choosing p to be the set of cut points that leads to the longest solution of the LCS problem with input strings S_p. This forms the basis of the BRKGA decoder, where individuals represent cut points that are mapped to feasible solutions by approximately solving the corresponding LCS problem. To achieve this, a BS for the LCS problem is employed, which we introduce later in this section.

We now introduce a dynamic programming algorithm for the LCS problem, which is an exact method that can be applied to small problem instances and is used within the BRKGA for guiding the search process.

2.2 A Dynamic Programming Algorithm for the LCS Problem

When the number of input strings and their lengths are relatively small, an exact dynamic programming approach can be used to solve the LCS problem efficiently [58]. We present this algorithm in the case of two input strings, as this is the scenario relevant for the BRKGA.

Throughout this section, given a finite set of strings $\{s_1, s_2, \ldots, s_m\}$ we denote by $\mathrm{LCS}(s_1, s_2, \cdots, s_m)$ an optimal solution to the LCS problem with these strings as input.

This approach relies on the following two properties of the LCS.

Proposition 2.2.1. Let s and t be two strings over some alphabet Σ and let $\mathsf{a} \in \Sigma$. We have:

$$\mathrm{LCS}(s \cdot \mathsf{a}, t \cdot \mathsf{a}) = \mathrm{LCS}(s, t) \cdot \mathsf{a} \tag{2.5}$$

Proof. Since $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{a})$ ends in $\mathtt{a}$, the equality above is equivalent to saying that $\mathrm{LCS}(s, t)$ corresponds to the substring obtained by removing the last $\mathtt{a}$ from $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{a})$, which we denote by s'.

Now, suppose this is not the case and seek a contradiction. In this case, there must exist some other string s'' such that $|s''| > |s'|$ and s'' is a common subsequence of both s and t. By construction $s'' \cdot \mathtt{a}$ is a common subsequence of $s \cdot \mathtt{a}$ and $t \cdot \mathtt{a}$ with length $|s'' \cdot \mathtt{a}| = |s''| + 1 > |s'| + 1 = |s' \cdot \mathtt{a}| = |\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{a})|$, leading to contradiction. □

Proposition 2.2.2. Let s and t be two strings over an alphabet Σ and let $\mathtt{a}, \mathtt{b} \in \Sigma$ with $\mathtt{a} \neq \mathtt{b}$. The following holds:

$$\begin{gathered}\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b}) \text{ is one of the maximal-length strings from the set} \\ \{\mathrm{LCS}(s \cdot \mathtt{a}, t), \mathrm{LCS}(s, t \cdot \mathtt{b})\}\end{gathered} \tag{2.6}$$

Proof. We distinguish two cases:

- $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b})$ ends in $\mathtt{a}$.

  Since $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b})$ ends in $\mathtt{a}$, the final $\mathtt{b}$ from $t \cdot \mathtt{b}$ cannot be part of it, as $\mathtt{a} \neq \mathtt{b}$ and there is no $\mathtt{a}$ after the final $\mathtt{b}$. Thus, removing $\mathtt{b}$ from $t \cdot \mathtt{b}$ does not affect the LCS, leading to $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b}) = \mathrm{LCS}(s \cdot \mathtt{a}, t)$.

- $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b})$ does not end in $\mathtt{a}$.

  In this case the final $\mathtt{a}$ from $s \cdot \mathtt{a}$ cannot belong to $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b})$ by the same argument. Hence, $\mathrm{LCS}(s \cdot \mathtt{a}, t \cdot \mathtt{b}) = \mathrm{LCS}(s, t \cdot \mathtt{b})$.

□

These results allow us to derive the following recursive formulation, which forms the basis of the dynamic programming approach:

Observation 2.2.1. Let $s = \mathtt{s}_1\mathtt{s}_2 \cdots \mathtt{s}_n$ and $t = \mathtt{t}_1\mathtt{t}_2 \cdots \mathtt{t}_m$ be two strings over an alphabet Σ. Define the prefixes $s^i = \mathtt{s}_1 \cdots \mathtt{s}_i$ and $t^j = \mathtt{t}_1 \cdots \mathtt{t}_j$ for all $i = 0, \cdots n$ and $j = 0, \ldots, m$. We have that:

$$\mathrm{LCS}(s^i, t^j) = \begin{cases} \varepsilon & \text{if } i{=}0 \text{ or } j{=}0 \\ \mathrm{LCS}(s^{i-1}, t^{j-1}) \cdot \mathtt{s}_i & \text{if } i,j > 0 \text{ and } \mathtt{s}_i{=}\mathtt{t}_j \\ \arg\max\{|\mathrm{LCS}(s^i, t^{j-1})|, |\mathrm{LCS}(s^{i-1}, t^j)|\} & \text{if } i,j > 0 \text{ and } \mathtt{s}_i{\neq}\mathtt{t}_j \end{cases} \tag{2.7}$$

The first case is clear as $s^0 = t^0 = \varepsilon$ and the second and third directly result from Propositions 2.2.1 and 2.2.2 respectively.

This observation allows us to construct the LCS between s and t by iteratively constructing the LCS between each pair s^i and t^j, ultimately yielding the LCS of the full strings. This process is illustrated in the following example:

Example 2.2.1. *Consider $s =$ acgau and $t =$ gca. We compute the LCS between s and t using the dynamic programming algorithm. We consider the following table, which will keep LCS(s^{i-2}, t^{j-2}) in position i, j.*

	Ø	a	c	g	a	u
Ø						
g						
c						
a						

We begin by computing the second row and the second column, which correspond to the values of LCS(t^0, s^i) and LCS(s^0, t^j) for $i = 0, \ldots, 5$, $j = 0, \ldots, 3$. Note that all of these values are empty strings, as per the first case in Observation 2.2.1.

	Ø	a	c	g	a	u
Ø	ε	ε	ε	ε	ε	ε
g	ε					
c	ε					
a	ε					

Next, we compute the third row. Since s^1 and t^1 end in a different character, we apply the third case from Observation 2.2.1, yielding LCS$(s^1, t^1) = \varepsilon$. The same holds for LCS(s^2, t^1). However, s^3 and t^1 end in the same character, so we use the second case from the observation, resulting in LCS$(s^3, t^1) =$ g. Similarly, since s^4 and t^1 end in different characters, we again apply third case, which gives LCS$(s^4, t^1) =$ g. The same is true for s^5 and t^1, yielding LCS$(s^5, t^1) =$ g.

	Ø	a	c	g	a	u
Ø	ε	ε	ε	ε	ε	ε
g	ε	ε	ε	g	g	g
c	ε					
a	ε					

Now, we compute the last two rows in the same manner, using the expression from Observation 2.2.1. Since s^1 and t^2 end in different characters, we have LCS$(s^1, t^2) = \varepsilon$. *For s^2 and t^2, both end in* c, *so* LCS$(s^2, t^2) =$ c. *For s^3 and t^2, they end in different characters, so we apply the third case from the observation. This leads to* LCS(s^3, t^2) *being the set of maximal elements from* $\{$LCS(s^2, t^2), LCS$(s^3, t^1)\} = \{$c, g$\}$. *Since* $|$c$| = |$g$| = 1$, *we have* LCS$(s^3, t^2) = \{$c, g$\}$. *Similarly, the same applies for* LCS(s^4, t^2) *and* LCS(s^5, t^2), *yielding two longest common subsequences in these cases as well. After filling the fourth row, our table looks as follows:*

	Ø	**a**	**c**	**g**	**a**	**u**
Ø	ε	ε	ε	ε	ε	ε
g	ε	ε	ε	g	g	g
c	ε	ε	c	c,g	c,g	c,g
a	ε					

Doing the same for the last row, we obtain:

	Ø	**a**	**c**	**g**	**a**	**u**
Ø	ε	ε	ε	ε	ε	ε
g	ε	ε	ε	g	g	g
c	ε	ε	c	c,g	c,g	c,g
a	ε	a	a,c	a,c,g	ca,ga	ca,ga

Therefore, a solution for LCS(acgau, gca) *is any string from the set* $\{$ca, ga$\}$.

As we can see, given two strings s and t, this algorithm computes LCS(s, t) by evaluating LCS(s^i, t^j) for all i and j, applying observation 2.2.1 at each step. Initially, LCS(s^0, t^j) and LCS(s^i, t^0) are set to ε. Then, each LCS(s^i, t^j) is calculated by either using LCS(s^{i-1}, t^j) when s^i and t^j end with the same character, or LCS(s^{i-1}, t^j) and LCS(s^i, t^{j-1}) if otherwise.

In our case, we are interested in the length of the LCS rather than the LCS itself. To obtain the LCS length using the dynamic programming approach described earlier, we simply track the length of the LCS at each step of the algorithm. The pseudocode for this approach is provided in Algorithm 2.1.

As the reader may have noticed, this algorithm can be adapted to handle any number of input strings. This is achieved by generalizing Propositions 2.2.1 and 2.2.2 to accommodate multiple input strings. However, in practice, this

Algorithm 2.1 A dynamic programming approach for calculating the LCS length with two input strings

```
Input: Two strings $s$ and $t$
Output: Length of an LCS between $s$ and $t$
 1: $C \in M_{|s|,|t|}(\mathbb{Z})$
 2: for $i \in \{1, \ldots, |s|\}$ do
 3:     $C_{i,0} := 0$
 4: end for
 5: for $j \in \{1, \ldots, |t|\}$ do
 6:     $C_{0,j} := 0$
 7: end for
 8: for $i \in \{1, \ldots, |s|\}$ do
 9:     for $j \in \{1, \ldots, |t|\}$ do
10:         if $s_i = t_j$ then
11:             $C_{i,j} := C_{i-1,j-1} + 1$
12:         else
13:             $C_{i,j} := \max(C_{i-1,j}, C_{i,j-1})$
14:         end if
15:     end for
16: end for
17: return $C_{|s|,|t|}$
```

approach becomes too slow because the set that appears in the generalization of proposition 2.2.2 grows rapidly as the number of input strings increases, making the algorithm impractical when this value is large.

As mentioned earlier, we are primarily focused on the dynamic programming approach for two input strings. This algorithm is incorporated within the standard BRKGA to guide the search process as explained later in Chapter 3.

We now introduce the BS for the LCS problem, which is utilized within the BRKGA decoder to translate cut-points into valid LCSqS solutions. Additionally, its heuristic function, denoted as UB, serves as the heuristic function for the BS approach to solve the RLCS problem as well.

2.3 Beam Search for the LCS Problem

Beam Search (BS) is an incomplete tree search algorithm that expands nodes in a breadth-first manner while maintaining a set of candidate nodes, called the beam, at each iteration. Each node represents a partial solution to the problem at hand. The algorithm was first introduced in the context of scheduling [98]. The number of nodes (partial solutions) retained per iteration is determined by a parameter $\beta > 0$ known as the beam width. In each iteration, every node is expanded, and the best β child nodes are kept for the next step.

In the context of the LCS problem, BS was first applied by Blum et al. [20]. Here, nodes of the search tree represent different possible common subsequences between the input strings, and expanding a node involves appending exactly one character to the partial solution represented by the node. To determine which β nodes remain in the beam, a heuristic function is used to estimate the quality of each partial solution.

Several BS approaches have been proposed for the LCS problem. In our case, we adopt the method introduced by Djukanovic et al. [34]. To present this algorithm in detail, we first introduce some necessary notation.

Let $S = \{s_1, \ldots, s_m\}$ be the set of input strings. We denote by $s[j, j']$ the contiguous subsequence of string s starting at position j and ending at position j'. If $j > j'$, then $s[j, j'] = \varepsilon$. Given a vector $p^L = (p^L_1, \ldots, p^L_m) \in \mathbb{N}^m$ with $1 \leq p^L_i \leq |s_i|$ for all $i = 1, \ldots, m$, we define $S[p^L] = \{s_i[p^L_i, |s_i|] \mid i = 1, \ldots, m\}$. p^L is referred as a left position vector.

Every node v in the BS tree represents a partial solution, which is a common subsequence of all input strings. Each node is associated with its partial solution length l_v and a left position vector $p^{L,v}$. The left position vector $p^{L,v}$ is such that for each input string s_i, the index $p^{L,v}_i - 1$ is the smallest value for which $s_i[1, p^{L,v}_i - 1]$ contains the partial solution as a subsequence. For a given node v, the goal is to extend its partial solution by adding characters that are common to every string in $S[p^{L,v}]$. The objective is to iteratively add these characters in a way that leads to the construction of an LCS.

2.3.1 State Graph for the LCS Problem

The graph which BS partially constructs is known as the state graph. It is a directed acyclic graph $G = (V, A)$. An edge $a = (v_1, v_2) \in A$ with label $l(a)$ exists if (i) $l_{v_2} = l_{v_1} + 1$ and (ii) the partial solution obtained by appending letter $l(a)$ to the partial solution represented by v_1 has the left position vector p^{L,v_2} associated. Each node in the state graph represents a partial solution, which can be recovered by tracing back from the node to the root while following the appended characters. Because of this backtracking property, partial solutions are not explicitly stored in the nodes during the search process.

The root node r of G corresponds to the original problem. It is defined by the left position vector $p^{L,r} = (1, \ldots, 1)$, meaning that no characters have been selected yet, and its partial solution length is $l_r = 0$. When determining the successor nodes of a given node v, not every character needs to be considered. Some characters may be infeasible, meaning that appending them to v's partial solution leads to a sequence that is not a common subsequence of all input strings.

For a node v, the feasible characters are those that appear in every string from $S[p^{L,v}]$. The set of feasible characters for node v is denoted Σ_v. However, not all feasible characters need to be considered, as some would lead to suboptimal partial solutions. These characters are referred to as dominated characters. For every character $\mathsf{a} \in \Sigma_v$, we define $p^{L,v}_{i,\mathsf{a}}$ as the position of the first occurrence of a in the substring $s_i[p^{L,v}_i, |s_i|]$. A character $\mathsf{a} \in \Sigma_v$ is said to be dominated by another character $\mathsf{b} \in \Sigma_v$ if $p^{L,v}_{i,\mathsf{b}} < p^{L,v}_{i,\mathsf{a}}$ for every $i = 1, \ldots, m$. In this case, considering expanding v through character a is unnecessary, as it will never lead to a better solution than expanding v using character b. The set of feasible, non-dominated characters for a node v is denoted Σ^{nd}_v.

The state graph for the LCS problem is constructed by starting with the root node r, expanding it using the characters in Σ^{nd}_r and doing the same in a recursive manner, for every node obtained in this first expansion. Doing this until every node v that has not been expanded has $\Sigma^{nd}_v = \varnothing$ leads to the complete state graph. An example for the state graph will be provided in the next section, in Figure 2.1.

2.3.2 Beam Search

BS constructs a subgraph of G by considering only the best β nodes at each height level. If β is set to a sufficiently large value, the whole state graph is constructed.

In each iteration, the algorithm processes the current set of nodes (the beam) as follows. Firstly, for every node v in the beam, Σ^{nd}_v is calculated. After that, for each $\mathsf{a} \in \Sigma^{nd}_v$ a successor node v' of v is derived: $v' = (p^{L,v'}, l_v + 1)$ where $p^{L,v'}_i = p^{L,v'}_{i,\mathsf{a}} + 1$ for all $i = 1, \ldots, m$. The nodes that have $\Sigma^{nd}_v = \varnothing$ are called complete nodes, and represent common subsequences that cannot be extended further. After building the successors of every node in the beam, they are given a fitness value using a heuristic function h and the β best are retained, which form the next iteration's beam. Once every node in the beam is complete, the algorithm returns the subsequence corresponding to the node v with a largest value for l_v, which corresponds to the LCS found.

Algorithm 2.2 illustrates the BS procedure for the LCS problem.

Here, extend_and_evaluate() takes the current beam and expands each node by considering all feasible, non-dominated characters that can be added to the partial solution represented by that node. It then assigns them a heuristic value through the evaluation function h. Afterwards, reduce() returns the best β nodes from the newly expanded nodes, based on the heuristic values computed by h. These form the next iteration's beam.

In Figure 2.1, the state graph for the LCS problem with input strings $S = \{\mathtt{abcaabc}, \mathtt{cabacb}, \mathtt{aabacba}\}$ is displayed. Each node v shows the left position

Algorithm 2.2 Pseudo-code of the BS for the LCS problem

```
Input: Input strings $S = \{s_1, \dots, s_m\}$, beam width $\beta$ and heuristic function $h$
Output: An approximate LCS problem solution with $S$ as input strings
1: $B := \{r\}$
2: $s_{lcs} := \emptyset$
3: while $B \neq \emptyset$ do
4:     $V_{ext} = \mathsf{extend\_and\_evaluate}(B, h)$
5:     update $s_{lcs}$ if a complete node with a largest $l_v$ value is found
6:     $B := \mathsf{reduce}(V_{ext}, \beta)$
7: end while
8: return $s_{lcs}$
```

vector $p^{L,v}$ and the length of the corresponding partial solution l_v.

The root node r represents the empty string, so $l_r = 0$ and $p^{L,r} = (1,1,1)$, indicating that this partial solution has not used any part of the input strings. To extend r, we first compute Σ_r^{nd}. As observed, every letter from the alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ appears in every string from $S[p^{L,r}] = S$, making every character feasible. However, b is dominated by a, so it does not need to be considered in order to expand r. This is because a appears before b in every string from $S[p^{L,r}] = S$.

Since $\Sigma_r^{nd} = \{\mathsf{a}, \mathsf{c}\}$, the state graph has two nodes at the second depth level. One corresponds to the extension of the empty string with a, and the other corresponds to extending it with c. The left position vectors for these two nodes can now be computed. For the right node, the left position vector is $(4, 2, 6)$, as the first c appears in position 3 in abcaabc, at position 1 in cabacb, and at position 5 in aabacba. Similarly, the left position vector for the left node is $(2, 3, 2)$. Both nodes have $l_v = 1$, as their partial solutions are of length one.

The rest of the state graph is constructed in the same manner until the leaf nodes v are reached, where no further extension is possible because $\Sigma_v^{nd} = \varnothing$.

BS would generate a subgraph of this graph. If $\beta = 1$, only one node would be considered for further expansion at each level, with the node selected based on the better value for h. Since the maximum number of nodes per level is 3, a BS with $\beta \geq 3$ would construct the entire state graph for these input strings.

The final aspect to explain is the guiding function h, which is used to select the best β nodes at each level. Within $\mathsf{extend\_and\_evaluate}()$ from Algorithm 2.2, h provides a fitness measure for each node generated by expanding the nodes in the beam. Then, in $\mathsf{reduce}()$, the β nodes with the best values for h are retained, while the others are discarded. In the next subsection, we will explore two potential designs for h.

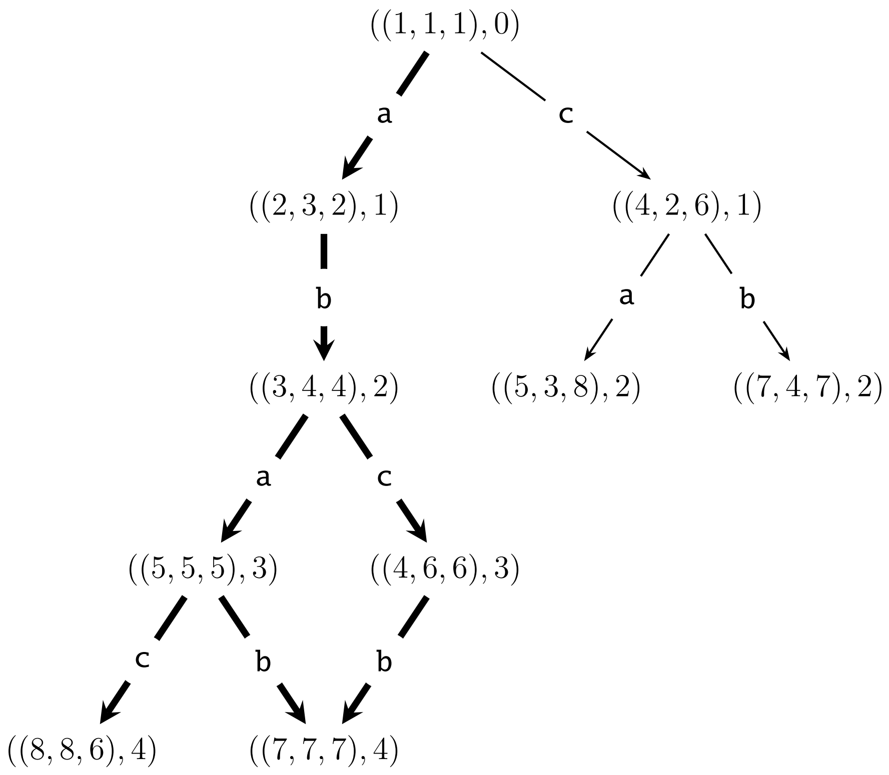


Figure 2.1 LCS problem state graph for $S = \{\mathtt{abcaabc}, \mathtt{cabacb}, \mathtt{aabacba}\}$. It contains five complete nodes. The three bold paths from $((8,8,6),4)$ and $((7,7,7),4)$ to the root node are the longest in the graph. Hence, they represent the three optimal solutions for this problem instance, abac, abab, and abcb respectively.

2.3.3 Guiding Function

We propose two designs for the guiding function h, with the choice between them being determined by a parameter of the BRKGA. The first design, introduced in [20], provides an upper bound on how much a given node can be extended. We refer to this design as UB. Although this upper bound is not tight, it has proven effective in guiding the BS.

The second design, introduced in [34], provides an expected value for how much a particular node can be extended under certain assumptions. This guiding function will be referred to as EX.

Upper bound UB

Given a string s over an alphabet Σ, we denote the number of occurrences of a character $\mathtt{a} \in \Sigma$ in s as $|s|_{\mathtt{a}}$. Using this notation, the guiding function UB applied to a node v is defined as follows:

$$\mathrm{UB}(v) = \sum_{\mathtt{a}\in\Sigma} \min_{1\le i\le m} \{|s_i[p_i^{L,v}, |s_i|]|_{\mathtt{a}}\} \tag{2.8}$$

As can be observed, $\mathrm{UB}(v)$ provides an upper bound on the number of characters that can be appended to the partial solution represented by v. This is because the appended characters must appear in every string from $S[p^{L,v}] = \{s_1[p_i^{L,v}, |s_i|], \dots, s_m[p_m^{L,v}, |s_m|]\}$, and therefore, for every character $\mathtt{a} \in \Sigma$, no more than $\min_{1\le i\le m}\{|s_i[p_i^{L,v}, |s_i|]|_{\mathtt{a}}\}$ a's can be appended. This reasoning applied to every character leads to the expression for UB.

Expected value EX

This guiding function involves assigning to a node v an approximation of the expected number of characters with which it can be extended. It requires knowledge of the probability $P(p,q)$ that a uniform random string of length p is a subsequence of a uniform random string of length q. These probabilities are computed in [93], where the following recurrence for $P(p,q)$ is derived:

$$P(p,q) = \begin{cases} 1 & \text{if } p = 0 \\ 0 & \text{if } p > q \\ \frac{1}{|\Sigma|}P(p-1,q-1) + \frac{|\Sigma|-1}{|\Sigma|}P(p,q-1) & \text{otherwise} \end{cases} \tag{2.9}$$

Now, the expected amount by which a certain node v can be extended can be approximated. This corresponds to approximating the expected length of a LCS of $S[p^{L,v}]$.

Let Y be the discrete random variable representing the length of an LCS of a set S' of m strings over an alphabet Σ . The approximation derived is based on the following two assumptions:

- All strings from S' are uniform
- The event that a given string over Σ is a common subsequence of all strings in S' is independent of the corresponding events for the other strings.

An LCS for input strings S' can have a length of at most $l_{\max} = \min\{|s| \mid s \in S'\}$, so $Y \in \{0, \dots, l_{\max}\}$. Let $Y_k \in \{0,1\}$ be the binary random variable which takes the value 1 if there exists a subsequence of length k common to all strings in S', and 0 otherwise. The following holds:

$$\begin{aligned}
P(Y=k) &= P(Y_k = 1 \cap Y_{k+1} = 0) \\
&= P(Y_k = 1) P(Y_{k+1} = 0 \mid Y_k = 1) \\
&= P(Y_k = 1)\big(1 - P(Y_{k+1} = 1 \mid Y_k = 1)\big) \\
&= P(Y_k = 1) - P(Y_k = 1) P(Y_{k+1} = 1 \mid Y_k = 1) \\
&= P(Y_k = 1) - P(Y_k = 1 \cap Y_{k+1} = 1) \\
&= P(Y_k = 1) - P(Y_{k+1} = 1) \\
&= \mathbb{E}(Y_k) - \mathbb{E}(Y_{k+1})
\end{aligned} \tag{2.10}$$

Where the last equality follows from Y_k being a binary random variable, which leads to:

$$\mathbb{E}(Y_k) = 1 \cdot P(Y_k = 1) + 0 \cdot P(Y_k = 0) = P(Y_k = 1) \tag{2.11}$$

Using this, the following expression for $\mathbb{E}(Y)$ is obtained:

$$\mathbb{E}(Y) = \sum_{1 \le k \le l_m} k \cdot P(Y = k) = \sum_{1 \le k \le l_m} k \cdot (\mathbb{E}(Y_k) - \mathbb{E}(Y_{k+1})) = \sum_{1 \le k \le l_m} \mathbb{E}(Y_k) \tag{2.12}$$

Finally, the probability that the strings in S' have no subsequence of length k is equal to:

$$P(Y_k = 0) = 1 - P(Y_k = 1) = 1 - \mathbb{E}(Y_k) \tag{2.13}$$

On the other hand, using the second assumption and the fact that there are $|\Sigma|^k$ strings of length k over the alphabet Σ, we have that the latter probability can also be written as:

$$\left(1 - \prod_{i=1}^{m} P(k, |s_i|)\right)^{|\Sigma|^k} \tag{2.14}$$

Therefore $\mathbb{E}(Y_k) = 1 - (1 - \prod_{i=1}^{m} P(k, |s_i|))^{|\Sigma|^k}$. Now, using the previously derived expression for $\mathbb{E}(Y)$, we obtain:

$$\mathbb{E}(Y) = \sum_{1 \le k \le l_m} \mathbb{E}(Y_k) = \sum_{1 \le k \le l_m} \left(1 - \left(1 - \prod_{i=1}^{m} P(k, |s_i|)\right)^{|\Sigma|^k}\right) \tag{2.15}$$

Using this for $S' = S[p^{L,v}]$, the guiding function EX evaluated at a node v is obtained:

$$\mathrm{EX}(v) = \sum_{1 \le k \le l_m} \mathbb{E}(Y_k) = \sum_{1 \le k \le l_m} \left(1 - \left(1 - \prod_{i=1}^{m} P(k, |s_i| - p_i^{L_v} + 1)\right)^{|\Sigma|^k}\right) \tag{2.16}$$

Here it has been used that every string $s_i[p_i^{L,v}, |s_i|]$ from $S[p^{L,v}]$ has length $|s_i| - p_i^{L,v} + 1$. In practice, the latter calculation is performed in a decomposed manner, and a Taylor approximation is applied to avoid issues with floating point arithmetic. More details can be found in [34, 36].

Chapter 3

BRKGA for the LCSqS Problem

3.1 Introduction

This chapter introduces the Biased Random Key Genetic Algorithm (BRKGA) for the Longest Common Square Subsequence (LCSqS) problem and explains how the offline learning mechanism is integrated into it. The content is based on publications [105, 107]. The first presents the BRKGA, while the second applies the offline learning framework to it.

A BRKGA [55] is a specialized variant of the previously introduced Random Key Genetic Algorithm (RKGA). The key distinction between a BRKGA and an RKGA lies in the mating process: in a BRKGA, one parent is always selected uniformly from the elite population, while the other is chosen from the non-elite population. Once two parents are selected for mating, the value at the i-th position of the offspring's vector is inherited from the elite parent with a probability $\rho_\mathrm{e} \in (0.5, 1]$, and from the non-elite parent otherwise. The parameter ρ_e, referred to as the elite inheritance probability, controls the bias toward the elite parent. Algorithm 3.1 shows a pseudo-code for a BRKGA.

As with RKGAs, the only problem-specific component in BRKGA is the decoder, which maps individuals to solutions for the given problem. This corresponds to the function $\mathsf{evaluate}()$ in Algorithm 3.1, which decodes each individual and assigns a corresponding fitness value.

A key aspect of the decoding process is that, beyond simply mapping a point from the hypercube to a feasible solution, the search can be guided using a greedy heuristic. Our BRKGA for the LCSqS problem leverages this feature, with different greedy information designs considered.

Moreover, the offline learning mechanism is incorporated at this stage to learn instance-specific greedy information, enhancing the search effectiveness. The next section provides details on this problem-specific evaluation, including the decoder, while the implementation of learning-based guidance is discussed later.

Algorithm 3.1 The general structure of a BRKGA

```
Input: Values for the parameters $p_{\text{size}}, p_{\text{e}}, p_{\text{m}},$ and $\rho_{\text{e}}$
 1: $P$ := generate_initial_population()
 2: evaluate($P$)
 3: while termination conditions not met do
 4:     $P'$ := new_empty_population()
 5:     $P$ := sort_depending_on_fitness($P$)
 6:     for $i \in \{1, \ldots, p_e\}$ do
 7:         $P'_i := P_i$
 8:     end for
 9:     for $i \in \{1, \ldots, p_m\}$ do
10:         $P'_{p_e+i}$ := generate_random_individual()
11:     end for
12:     for $i \in \{1, \ldots, p_{\text{size}} - p_e - p_m\}$ do
13:         select $\mathbf{v}^1 \in P_{\text{e}}$ randomly
14:         select $\mathbf{v}^2 \in P \setminus P_{\text{e}}$ randomly
15:         $\mathbf{v} := (v_i^1$ with probability $\rho_{\text{e}}$ and $v_i^2$ otherwise$)_{i=1}^{m}$
16:         $P'_{p_e+p_m+i} := \mathbf{v}$
17:     end for
18:     $P := P'$
19:     evaluate($P$)
20: end while
```

To explore different strategies for guiding the search, we have considered multiple approaches for defining both the greedy information and the fitness measure. The goal is to determine the most effective designs through experimental testing. To facilitate this, all the resulting options are encapsulated within a single decoder, which allows for flexibility in selecting and comparing different configurations. The following subsection details the structure and functioning of this decoder.

3.1.1 Decoder

The decoding process in the BRKGA for the LCSqS problem leverages Proposition 2.1.1, which enables solving the LCSqS problem by splitting each input string into two parts and then solving an Longest Common Subsequence (LCS) problem with the resulting substrings. To find an approximate solution efficiently, the Beam Search (BS) algorithm for the LCS problem introduced in the previous chapter is employed.

With this goal, each individual consists of m values in $[0, 1]$, with m being the number of input strings. The decoder follows a two-step process. First it translates an individual into a set of cut points that split each input string into two. Then, it applies the BS to the resulting strings and concatenates the obtained solution with itself, obtaining the approximate LCSqS solution associated with the individual.

We designed the mapping of an individual $\mathbf{v} = (v_1, v_2, \ldots, v_m) \in [0,1]^m$ to a cut-point vector as follows:

$$(v_1, v_2, \ldots, v_m) \in [0,1]^m \longmapsto (v_1|s_1|, v_2|s_2|, \ldots, v_m|s_m|) \in \prod_{i=1}^{m} \{0, 1, \ldots, |s_i|\} \quad (3.1)$$

Hereby, every product is rounded to the closest integer.

Each element $(c_1, c_2, \ldots, c_m) \in \prod_{i=1}^{m} \{0, 1, \ldots, |s_i|\}$ is interpreted as a cut-point vector, where for each $i = 1, 2, \ldots, m$, the string s_i is cut so that c_i characters remain in the first part of the cut.

Once this association is made, the BS described in Chapter 2 is used, introducing parameters β and h, which denote its beam width and heuristic function respectively. Algorithm 3.2 illustrates the decoder.

Algorithm 3.2 The decoder of the BRKGA for the LCSqS problem

```
Input: Input strings $S = \{s_1, \ldots, s_m\}$, an individual $\mathbf{v}$, beam width $\beta$, and
    heuristic function $h$
Output: The LCSqS solution associated with $\mathbf{v}$
1: $\mathbf{v}' := \text{greedy\_transformation}(\mathbf{v})$
2: $\mathbf{p}^{\mathbf{v}} := \text{map\_to\_cut\_points}(\mathbf{v}')$
3: $S_{\mathbf{p}^{\mathbf{v}}} :=$ LCS problem instance induced by $\mathbf{p}^{\mathbf{v}}$
4: $t^{\mathbf{v}} := \text{beam\_search\_for\_LCS\_problem}(S_{\mathbf{p}^{\mathbf{v}}}, \beta, h)$
5: return $t^{\mathbf{v}} \cdot t^{\mathbf{v}}$
```

As one can see, before mapping a hypercube element to a set of cut points using Equation (3.1), greedy information is employed to influence these hypercube elements. In the following section, we will explore our various proposals for the greedy information.

3.1.2 Greedy Information

Function greedy_transformation() applies a greedy bias to an individual $v = (v_1, \ldots, v_m)$ before mapping it to a cut point vector. We propose several options for this transformation, all of which follow the same basic principle: first, a promising hypercube point $u = (u_1, \ldots, u_m)$ is identified, and then individual v is adjusted toward this hypercube point by the following update:

$$v_i := v_i + \gamma \cdot (u_i - v_i). \quad (3.2)$$

Hereby, $\gamma \in [0,1]$ is the so-called *greedy rate*, controlling the degree to which v_i is adjusted toward u_i for each $i = 1, \ldots, m$. When $\gamma = 1$, v_i is directly replaced by u_i, while when $\gamma = 0$, no greedy information is used. The three different approaches for constructing the vector u are described next.

- The first approach for greedy information we designed involves setting $u = (0.5, \ldots, 0.5)$. This choice is motivated by the fact that, generally, the middle point of each string serves as an effective cut location, as these cuts maximize the minimum length of the resulting substrings. This is clearly desirable when we have no information about the distribution of characters throughout the strings and we assume them to be uniform.

- The second approach for greedy information involves assigning u_i the following value for all $i = 1, \ldots, m$:

  $$\underset{r \in [0,1]}{\arg\min} \sum_{\mathsf{a} \in \Sigma} \Big| \big|s_i[1, r \cdot |s_i|]\big|_{\mathsf{a}} - \big|s_i[r \cdot |s_i| + 1, |s_i|]\big|_{\mathsf{a}} \Big| \tag{3.3}$$

  The products $r \cdot |s_i|$ are rounded to the closest integer. $\big|s_i[1, r \cdot |s_i|]\big|_{\mathsf{a}}$ and $\big|s_i[r \cdot |s_i| + 1, |s_i|]\big|_{\mathsf{a}}$ denote the number of occurrences of character a in the two strings obtained after cutting s_i at position $r \cdot |s_i|$.

  This approach selects the cut that maximizes the overall balance of each character's quantity on both sides of the cut. The value that minimizes the above expression may not be unique; in such cases, a random value from those that minimize it is selected.

  With this design, the greedy value incorporates information about the character distribution across the strings and uses it to determine the best cuts. These chosen cuts are generally reasonable, as, in the absence of detailed information about the character distribution in a set of strings, the LCS is typically expected to be longer when the input strings have a similar number of occurrences of each character.

- The last considered greedy information approach consists in setting u_i to the following value for all $i = 1, \ldots, m$:

  $$\underset{r \in [0,1]}{\arg\max} \left| \mathrm{LCS}\Big(s_i[1, r \cdot |s_i|], s_i[r \cdot |s_i| + 1, |s_i|] \Big) \right| \tag{3.4}$$

  Hereby, the same notation as before is used, and $\big|\mathrm{LCS}(s_i[1, r \cdot |s_i|], s_i[r \cdot |s_i| + 1, |s_i|])\big|$ denotes the length of an LCS between the two strings obtained by cutting s_i at position $r \cdot |s_i|$. This length is computed using the dynamic programming approach (Algorithm 2.1) introduced in Chapter 2. This approach is suitable for obtaining greedy information due to the efficiency of the dynamic programming method for two input strings.

This greedy value design is also logical because the ultimate goal when determining the cuts is to maximize the LCS length between the resulting strings. Since it is computationally infeasible to select cuts that maximize this value for all strings simultaneously, we instead choose each cut to maximize the LCS length between the two resulting parts of the string.

Generally, we can expect the set of cut-points obtained using this approach to be close to those that maximize the LCS length across all the strings, which are computationally infeasible to calculate.

3.1.3 Measure of Fitness

To evaluate an individual v, we naturally consider the length of the LCSqS solution t^v obtained from the decoder. The value of this primary objective function is denoted by $f_1(v)$. During the study of our algorithm's behavior, we observed that many individuals shared the same primary objective function value, which indicates the presence of plateaus in the search space. To address this, we introduced an additional way to differentiate between such solutions, in the form of a secondary objective function value. The following options are considered for designing the secondary objective function value $f_2(v)$ of an individual v.

1. The first approach favors individuals whose cut points are more concentrated around the central positions of the input strings. We calculate the secondary objective function value $f_2(v)$ using the following expression:

$$\sum_{i=1}^{m} \left|2 \cdot p_i^v - |s_i|\right|. \tag{3.5}$$

   This sum results in a lower value when cuts are more centered and a higher value otherwise. This is because $|2 \cdot \text{cp}_{I_i} - |s_i||$ can be rewritten as $|\text{cp}_{I_i} - (|s_i| - \text{cp}_{I_i})|$, which represents the absolute difference between the number of characters on either side of the cut.

2. The second option favors individuals whose cut points create a better balance between the quantity of each character on both sides of the cut.

In this case, $f_2(v)$ is calculated by

$$\max_{1\leq i\leq n}\left\{\sum_{\mathsf{a}\in\Sigma}\left|\big|s_i[1,p_i^v]\big|_{\mathsf{a}}-\big|s_i[p_i^v+1,|s_i|]\big|_{\mathsf{a}}\right|\right\}. \tag{3.6}$$

Note that this approach is similar to the one used in the second greedy transformation design.

3. The third approach favors individuals whose cut points maximize the LCS distance between each side of the cut. It defines $f_2(v)$ by

$$-\min_{1\leq i\leq n}\left\{\left|\mathrm{LCS}(s_i[1,p_i^v],s_i[p_i^v+1,|s_i|])\right|\right\}. \tag{3.7}$$

   The motivation behind this approach is the same to that of the third greedy value approach. The negation of the sum ensures consistency with the other secondary objective function designs, where individuals with a lower value are preferred.

To compare two individuals, v and v', within the BRKGA, both the primary and selected secondary objective functions are used in a lexicographical manner: v is considered better than v' if and only if $f_1(v) > f_2(v')$ or $f_1(v) = f_2(v')$ and $f_2(v) < f_2(v')$.

The algorithm design is now complete, and the final step is to determine the preferred greedy value and secondary measure of fitness. We will select between the available designs through parameter tuning, which will be carried out during the experimental evaluation.

3.2 Implementing the Offline Learning Component

Following the general offline learning framework introduced in Chapter 1, the components that need to be described include the search component parameterized by the Machine Learning (ML) model, the ML model used with the features extracted, and the methods employed to compute the training and validation values.

As mentioned, the offline learning framework is applied to the original BRKGA in the greedy information vector u. Specifically, with this implementation, u_i is determined by a feed-forward neural network (NN) that receives features of the input string s_i along with some global features of the entire instance. From now on, we refer to this learning variant as Brkga-Learn, while denoting the standard one by Brkga.

3.2.1 Neural Network and Features

As explained, the neural network is applied separately to each input string, predicting a value for the corresponding greedy component u_i. Six features are used as input to the neural network: the first two are specific to each string, while the last four are global features. This approach aims to capture information both about individual strings and the overall problem instance.

Given a problem instance consisting of input strings $S = \{s_1, s_2, \ldots, s_m\}$, we denote by $\mathrm{gv2}_i$ and $\mathrm{gv3}_i$ the values for the second and third greedy value designs, as outlined in the previous subsection, for the i-th input string, respectively.

The following features are extracted for each string s_i, $i = 1, 2, \ldots, m$:

$$X = \left(\mathrm{gv2}_i,\ \mathrm{gv3}_i,\ \overline{\mathrm{gv2}},\ \overline{\mathrm{gv3}},\ \sigma(\mathrm{gv2}),\ \sigma(\mathrm{gv3})\right) \tag{3.8}$$

Here, $\overline{\mathrm{gv2}}$ and $\overline{\mathrm{gv3}}$ represent the averages of the second and third greedy values across all strings in S, while $\sigma(\mathrm{gv2})$ and $\sigma(\mathrm{gv3})$ denote the corresponding sample standard deviations, respectively:

$$\overline{\mathrm{gv2}} = \frac{\sum_{i=1}^{m} \mathrm{gv2}_i}{m},\ \sigma(\mathrm{gv2}) = \sqrt{\frac{\sum_{i=1}^{m} \left(\mathrm{gv2}_i - \overline{\mathrm{gv2}}\right)^2}{m-1}} \tag{3.9}$$

$$\overline{\mathrm{gv3}} = \frac{\sum_{i=1}^{m} \mathrm{gv3}_i}{m},\ \sigma(\mathrm{gv3}) = \sqrt{\frac{\sum_{i=1}^{m} \left(\mathrm{gv3}_i - \overline{\mathrm{gv3}}\right)^2}{m-1}} \tag{3.10}$$

Feature values are standardized before being fed into the neural network to ensure a mean of zero and a standard deviation of one. The neural network consists of a single node in the output layer with a sigmoid activation function, as it is applied to one string at a time, outputting a promising cut point in the form of a value in $(0, 1)$. Furthermore, tuning indicated that a single dense hidden layer with five nodes and a ReLU activation function is sufficient. This choice also aligns with the general guideline of selecting a hidden size between the input layer (6 nodes) and the output layer (1 node). More complex networks were tested as well, but they did not lead to noticeably better results. Figure 3.1 provides a graphical representation of the architecture used.

The expressions for the ReLU and sigmoid functions are given below.

$$\mathrm{ReLU}(x) = \max\{0, x\},\ x \in \mathbb{R} \tag{3.11}$$

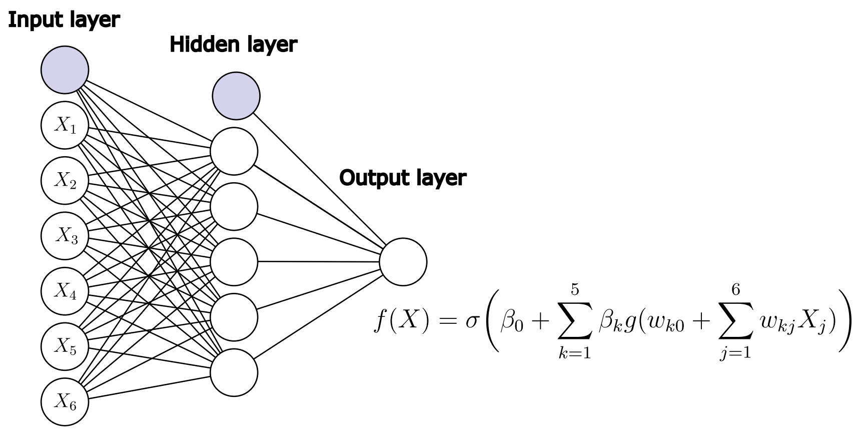

Input layer
Hidden layer
Output layer
X_1
X_2
X_3
X_4
X_5
X_6
$f(X) = \sigma\Big(\beta_0 + \sum_{k=1}^{5} \beta_k g(w_{k0} + \sum_{j=1}^{6} w_{kj} X_j)\Big)$

Figure 3.1 A graphical representation of the employed feed-forward neural network. $f(X)$ is the output of the neural network for the feature value vector $X = (X_1, X_2, \ldots, X_6)$. The sets $\{w_{k,j}\}$ and $\{\beta_k\}$ represent the 41 parameters of the neural network. Specifically, $\{w_{k,j}\}$ transform the features into the hidden layer, and $\{\beta_k\}$ transform the hidden layer into the output. Additionally, the lines from the top node in each layer represent the biases, which are parameters $\{w_{k0}\}$ and β_0.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}},\ x \in \mathbb{R} \tag{3.12}$$

3.2.2 Evaluation of an Individual

The computation of the training value proceeds as follows: First, a value u_i is obtained by supplying the neural network (equipped with the weights of the individual) for every string s_i of each problem instance in the training set. Then, Equation (3.1) is applied to generate the corresponding cut points. Afterward, BS is applied on the strings obtained after cutting to evaluate the cut points. The average of the obtained solution lengths for all training instances is then the training value, which is used as the measure of fitness of the corresponding individual. Running the BRKGA guided by the model with the individual weights is impractical, as it would lead to excessively long training times due to the large number of individual evaluations required during training. However, this can be done for validation values, as they have to be computed with much less frequency.

The BRKGA is executed to compute validation values by running the algorithm for the same time limit as used in the experimental evaluation on the validation instances, guided by the vector u obtained from the neural network with the corresponding individuals' weights. The average of the resulting LCSqS lengths is then used as the validation value. Again, it is important to note that, for validation, running the BRKGA instead of BS is computationally feasible, as new best individuals are found only sporadically.

3.3 Experimental Evaluation

In this section, we evaluate bot the standard Brkga and Brkga-Learn to compare their performance. Moreover, we also compare to the previously state-of-the-art algorithm from the literature for solving the LCSqS problem, which is a Reduced Variable Neighborhood Search (RVNS) introduced in [33] that also employed BS as one of its components. We denote the latter by Rvns from now on.

To do this experimental evaluation, we train Brkga-Learn on two benchmark sets, tune the three approaches and obtain numerical results. The first benchmark set, referred to as Random, consists of instances where strings are generated uniformly at random. In contrast, the second set, called Non-Random, consists of instances with non-uniform strings, which are created by implanting specific patterns.

3.3.1 Problem Instances

The first benchmark set, called Random, was previously used to evaluate Rvns in its original publication [33]. It was originally introduced in the context of the LCS problem in [15]. This set includes both tuning and evaluation instances, all of which are generated uniformly at random. There are ten evaluation instances for each combination of $n \in \{100, 500, 1000\}$, $m \in \{10, 50, 100, 150, 200\}$, and $|\Sigma| \in \{4, 12, 20\}$. Here, n represents the length of the input strings, m denotes the number of strings, and $|\Sigma|$ refers to the size of the alphabet. In total, there are 450 problem instances. Additionally, for every combination of n, m, and $|\Sigma|$, there are three tuning instances.

The second benchmark set, called Non-Random, is generated by ourselves. It consists of non-uniform instances, with the goal of testing the algorithms on instances with different characteristics. Similar to the Random set, we include both tuning and evaluation instances. The instances are generated by implanting a square pattern string into each input string of an instance. For each combination of $n \in \{100, 500, 1000\}$, $m \in \{10, 50, 100, 150, 200\}$, and type $\in \{1, 2\}$ (where "type" refers to the method of implanting the pattern, as described below), we generate ten evaluation instances. The alphabet size is varied based on string length, with values set to 12, 15, and 18 for $n = 100$, $n = 500$, and $n = 1000$, respectively. Therefore, the Non-Random benchmark set contains 90 problem instances in total. Additionally, three tuning instances are created for each combination of n,

m, and type.

The patterns used are determined solely by the length of the input strings. They consist of the following strings, each concatenated with itself.

- For input strings of length $n = 100$:
  `commonsubsequence`
- For input strings of length $n = 500$:
  `longestcommonsubsequenceproblem`
- For input strings of length $n = 1000$:
  `longestcommonsquaresubsequenceproblem`

In both type 1 and type 2 instances, the pattern is implanted into each string by selecting positions equal to the length of the pattern and filling them sequentially with the pattern's letters. Then, the remaining empty positions in each string are filled uniformly at random with characters from the alphabet.

The difference between type 1 and type 2 instances lies in how the positions for the pattern string are selected. In type 1 instances, the positions are chosen uniformly at random. In contrast, for type 2 instances, the probability of selecting a particular position is decreased from left to right, with the probability for position i given by:

$$p_i = \frac{p'_i}{\sum_{k=1}^{n} p'_k} \quad \text{with} \quad p'_i = 2 - \frac{i}{n-1}, \quad i = 1, \ldots, n. \tag{3.13}$$

This way, the probability of selecting the first position is twice as high as the probability of selecting the last position, with the probability decreasing linearly from left to right.

3.3.2 Training and Parameter Tuning

To ensure a fair comparison, we allocate the same computation time limit to the three algorithms. Specifically, a time limit of 600 CPU seconds was set for each algorithm execution. This time limit was also the one applied for each algorithm execution during parameter tuning and for the calculation of the validation value during the Brkga-Learn trainings. The benchmark sets Random and Non-Random consist of 150 and 100 instances, respectively, for each $n \in \{100, 500, 1000\}$, where n represents the string length in the instances. Additionally, Non-Random instances are further divided into two sets based on the parameter $type$, which specifies how the patterns were implanted.

Table 3.1 Parameter values obtained after tuning Brkga and Brkga-Learn on the LCSqS problem benchmark set Random. Three tunings are performed for each algorithm, one for each size: $n = 100, 500, 1000$.

	Allowed Range	Brkga			Brkga-Learn		
p_e	$\{0.01, 0.02, \ldots, 0.45\}$	0.21	0.36	0.21	0.15	0.21	0.31
p_m	$\{0.01, 0.02, \ldots, 0.45\}$	0.09	0.20	0.40	0.13	0.03	0.28
ρ_e	$\{0.30, 0.31, \ldots, 0.99\}$	0.64	0.69	0.53	0.35	0.45	0.68
p_{size}	$\{10, 11, \ldots, 1000\}$	954	282	535	706	329	737
$of2$	$\{1, 2, 3\}$	2	2	2	1	3	2
γ	$\{0.00, 0.01, \ldots, 0.99\}$	0.80	0.93	0.97	0.51	0.92	0.98
β	$\{1, 2, \ldots, 1000\}$	21	139	137	113	53	587
h	$\{\text{UB}, \text{EX}\}$	UB	UB	UB	UB	UB	UB
gv	$\{1, 2, 3\}$	1	1	1	-	-	-

Table 3.2 Parameter values obtained after tuning Brkga and Brkga-Learn on the LCSqS problem benchmark set Non-Random. Three tunings are performed for each algorithm, one for each size: $n = 100, 500, 1000$.

	Allowed Range	Brkga			Brkga-Learn		
p_e	$\{0.01, 0.02, \ldots, 0.45\}$	0.42	0.12	0.19	0.18	0.16	0.23
p_m	$\{0.01, 0.02, \ldots, 0.45\}$	0.01	0.18	0.04	0.04	0.33	0.02
ρ_e	$\{0.30, 0.31, \ldots, 0.99\}$	0.55	0.55	0.60	0.47	0.58	0.56
p_{size}	$\{10, 11, \ldots, 1000\}$	836	334	113	906	372	218
$of2$	$\{1, 2, 3\}$	2	1	3	2	2	1
γ	$\{0.00, 0.01, \ldots, 0.99\}$	0.76	0.84	0.90	0.79	0.84	0.87
β	$\{1, 2, \ldots, 1000\}$	949	19	70	388	6	6
h	$\{\text{UB}, \text{EX}\}$	EX	UB	UB	UB	UB	UB
gv	$\{1, 2, 3\}$	3	1	1	-	-	-

We train Brkga-Learn separately based on n for the Random instance set and on n and *type* for the Non-Random instance set. As a result, three separate training procedures are conducted for the Random benchmark set, and six for the Non-Random set. We tune the parameters based on n for both instance sets, following the same procedure for all algorithms. Additionally, different equivalently generated instances are used for training, parameter tuning, and evaluation. For each Random training, fifteen instances are used for calculating training values and another fifteen for validation values, one for each combination of m (number of strings) and $|\Sigma|$ (alphabet size). Likewise, Non-Random trainings use two sets of ten instances, with two instances for every possible value of m.

Regarding the training RKGA, we apply it using the following default parameters: a population size of 20, one elite individual, and seven mutants. In the BS used to compute training values, a beam width of $\beta = 250$ is applied alongside the UB guiding function presented in Equation (2.8) of Chapter 2. Meanwhile, for the Brkga executions performed during validation value computations, we adopt the parameter values produced by the parameter tuning for the original Brkga, found in Tables 3.1 and 3.2.

We carried out each training, tuning and evaluation run on a cluster of machines with 10-core Intel Xeon processors at 2.2 GHz and 8 GB of RAM. Parallelism in the training runs was applied in the calculation of validation and training values. Each training utilized 10 cores, distributing the training and validation instances across them. However, no parallelism was applied for parameter tuning or the final experimental evaluation, as was the case with the original Brkga and Rvns. Early stopping was employed for the training runs of Brkga-Learn, meaning they were terminated whenever the validation value decreased, indicating potential overfitting. On average, training runs lasted about one hour, with runs involving larger instances requiring up to four hours.

Tables 3.1 and 3.2 report the results of the parameter tuning for Brkga and Brkga-Learn. For each algorithm, three parameter sets are presented, corresponding to $n = 100$, $n = 500$, and $n = 1000$, shown from left to right. Specifically, p_e, p_m, ρ_e, p_{size}, and $of2$ refer to the proportion of elites, the proportion of mutants, the elite inheritance probability, the population size, and the secondary objective function design, respectively. Additionally, γ is the greedy rate, which controls the extent of greedy information used, and β and h are the parameters of the BS, representing the beam width and the guiding function design. Finally, gv is the greedy value design employed by Brkga, which is replaced by the neural network in the learning variant.

To perform parameter tuning, we utilized the automatic configuration tool *irace* [87]. For benchmark set Random, one tuning instance was selected for every combination of m and $|\Sigma|$. Similarly, for benchmark set Non-Random, one instance was chosen for every combination of m and $type$. This resulted in 15 tuning instances for each parameter tuning run concerning the Random benchmark set, and 10 instances for each Non-Random run. Each tuning process was allocated a budget of 5000 algorithm executions. Table 3.1 presents the allowed values for each parameter during the tuning process. These ranges were the same for both instance sets.

Regarding the parameter settings of Rvns, they were also set by *irace* using the same budget. For benchmark set Random the parameters produced were $\alpha = 0.9$, $\sigma = 5$ and $\beta = 100$ for $n = 100$; $\alpha = 0.9$, $\sigma = 10$ and $\beta = 200$ for $n = 500$; $\alpha = 0.9$, $\sigma = 20$ and $\beta = 200$ for $n = 1000$. With the guiding function EX being the preferred one for all sizes. For benchmark set Non-Random, the following configuration was yielded: $\alpha = 0.7$, $\sigma = 5$ and $\beta = 10$ for $n = 100$; $\alpha = 0.9$, $\sigma = 10$ and $\beta = 200$ for $n = 500$; $\alpha = 0.9$, $\sigma = 20$ and $\beta = 200$ for $n = 1000$. Moreover, EX was chosen as the BS heuristic function for $n = 100$ and UB for the rest.[1]

Table 3.3 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Random instances with string length $n = 100$.

		Brkga-Learn		Brkga		Rvns	
m	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
10	4	**28.44**	98.36	28.34	19.06	27.46	58.78
	12	**8.94**	11.73	8.00	0.06	8.32	19.11
	20	**4.20**	0.04	4.00	0.00	3.92	8.79
50	4	19.02	144.95	**19.94**	85.65	18.62	32.86
	12	**4.00**	0.03	**4.00**	1.25	3.98	2.75
	20	**1.40**	0.39	0.20	0.00	0.20	0.02
100	4	15.72	168.17	**17.16**	68.45	16.32	41.25
	12	**2.42**	30.20	2.20	5.38	1.60	0.02
	20	**0.12**	12.91	0.00	0.00	0.00	0.02
150	4	13.16	160.99	**15.94**	53.37	15.28	101.42
	12	**2.00**	0.02	**2.00**	0.20	0.40	0.02
	20	**0.00**	0.00	**0.00**	0.00	**0.00**	0.03
200	4	12.02	34.86	**14.78**	92.28	14.02	5.96
	12	**2.00**	0.29	1.60	1.08	0.00	0.03
	20	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00

3.3.3 Results for Benchmark Set Random

The results obtained for the benchmark set Random are presented in Tables 3.3–3.5. These tables contain results for instances with $n = 100$, $n = 500$, and $n = 1000$, respectively. For each combination of n, m, and $|\Sigma|$, as well as for each algorithm, we report the average length of the best solutions found ($\overline{|s|}$) and the average time required to find these best solutions ($\bar{t}_{best}[s]$). Since each group consists of ten instances and each algorithm was applied ten times to each instance, the results for each of the 45 table rows represent averages over 100 runs. In each row, the best result is highlighted in bold.

We can observe that the results for this benchmark set do not differ significantly between the three algorithms, although Brkga-Learn performs slightly better than the rest on average. Similarly, Brkga is often better than Rvns on average.

[1]For the meaning of these parameters we refer to the original publication of Rvns [33].

Table 3.4 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Random instances with string length $n = 500$.

		Brkga-Learn		Brkga		Rvns	
m	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
10	4	**159.56**	235.68	158.94	226.69	157.58	137.93
	12	**59.90**	47.88	59.60	51.22	58.94	80.61
	20	**36.12**	11.69	36.04	7.97	36.04	63.97
50	4	**126.40**	297.95	125.76	146.66	125.10	89.62
	12	**40.06**	65.44	40.00	52.29	39.30	65.65
	20	**22.00**	25.71	21.82	14.19	21.46	76.73
100	4	**117.26**	246.59	116.82	102.85	116.54	81.61
	12	**34.30**	55.78	34.12	21.53	34.00	15.96
	20	**18.00**	0.70	**18.00**	0.60	**18.00**	22.21
150	4	**112.50**	125.87	**112.50**	64.91	112.48	61.43
	12	**32.00**	2.71	**32.00**	2.14	**32.00**	31.68
	20	**16.02**	0.99	16.00	0.16	16.00	4.26
200	4	109.96	109.96	**110.00**	26.02	**110.00**	36.60
	12	**30.08**	17.80	30.04	6.96	30.00	6.34
	20	15.76	74.79	**15.90**	73.57	14.80	82.82

Table 3.5 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Random instances with string length $n = 1000$.

		Brkga-Learn		Brkga		Rvns	
m	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
10	4	**324.42**	155.60	323.98	164.38	322.96	181.82
	12	125.44	78.44	**125.78**	110.02	125.00	112.43
	20	77.68	46.47	**78.02**	107.96	77.50	100.46
50	4	**263.90**	88.78	263.74	96.49	262.82	77.69
	12	**87.62**	99.69	**87.62**	87.74	86.56	77.88
	20	50.16	17.68	**50.54**	43.04	50.00	29.99
100	4	**249.30**	70.03	248.84	78.81	248.16	43.58
	12	78.36	44.55	**78.48**	58.38	78.02	26.25
	20	**44.00**	3.68	**44.00**	1.76	**44.00**	48.63
150	4	**242.22**	62.90	242.06	73.12	241.30	52.63
	12	74.26	40.61	**74.42**	49.93	74.02	24.30
	20	41.26	100.94	**41.32**	87.43	40.22	40.29
200	4	**237.50**	64.25	237.32	69.46	236.88	129.61
	12	**72.02**	19.07	72.00	13.19	71.68	88.88
	20	39.88	124.41	**39.94**	77.83	38.64	86.65

It is important to note that, for this set of instances, the original Brkga used the first greedy information design, which simply biases cuts toward the middle of every string. As these instances consist of uniform strings, the first greedy information approach already produces a good prediction for the optimal cut point, leaving little room for improvement to the learned guidance of Brkga-Learn.

3.3.4 Results for Benchmark Set Non-Random

The results for benchmark set Non-Random are shown in Tables 3.6–3.8, which follow the same structure as outlined in the previous section.

Table 3.6 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Non-Random instances with string length $n = 100$.

		Brkga-Learn		Brkga		Rvns	
m	*type*	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	32.24	97.06	32.14	114.59	**32.56**	79.64
	2	30.56	66.05	30.84	95.25	**32.26**	118.42
50	1	**25.78**	151.84	24.98	227.49	22.70	102.23
	2	**25.28**	100.28	24.86	183.25	22.60	111.26
100	1	**22.16**	107.93	19.34	180.61	18.04	84.55
	2	**21.98**	120.53	20.08	149.02	18.62	105.24
150	1	**19.36**	127.27	16.76	154.21	16.50	81.51
	2	**19.76**	161.05	16.68	147.78	15.22	95.14
200	1	**18.10**	136.83	14.72	145.22	14.98	86.77
	2	**18.58**	120.13	14.70	160.61	13.46	87.22

In this case, Brkga-Learn consistently and significantly outperforms the original Brkga, except for instances with a very low number of input strings (m), where the results are inconclusive. For all other instances, Brkga-Learn yields better solutions than Brkga, demonstrating a clear benefit from the proposed ML guidance in the case of non-uniform strings. Regarding the Rvns, its solutions are of much worse quality than the ones of the BRKGA-based approaches, showing that it does not adapt well to solving non-uniform strings. The only exception is with the instances with $m = 10$ and $n = 100$, for which interestingly Rvns performs best.

In order to measure the statistical significance of the differences between the obtained solution lengths of Brkga-Learn and Brkga on the Non-Random benchmark set, we employed the signed-rank Wilcoxon test [128].

Table 3.7 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Non-Random instances with string length $n = 500$.

		Brkga-Learn		Brkga		Rvns	
m	*type*	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	66.18	39.68	**70.92**	129.82	29.48	32.67
	2	**70.14**	60.28	64.78	117.03	32.56	36.44
50	1	58.58	160.06	**59.16**	287.05	25.74	78.84
	2	**60.76**	218.02	55.32	187.35	23.94	67.24
100	1	**52.58**	205.24	49.30	321.75	22.44	63.15
	2	**53.60**	222.17	51.00	277.44	21.02	50.23
150	1	**51.08**	286.49	46.52	344.74	21.22	58.55
	2	**48.80**	285.49	48.78	315.31	22.00	54.17
200	1	**48.18**	327.75	43.62	354.31	21.46	60.57
	2	**45.60**	368.48	43.60	407.89	20.60	38.65

Table 3.8 Comparison of Brkga-Learn, Brkga and Rvns on the LCSqS problem Non-Random instances with string length $n = 1000$.

		Brkga-Learn		Brkga		Rvns	
m	*type*	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	90.50	96.65	**91.14**	113.29	51.24	62.16
	2	90.92	130.95	**91.38**	102.84	52.86	66.85
50	1	**66.16**	154.49	65.40	284.90	53.18	101.13
	2	**66.28**	157.41	63.94	255.81	51.68	74.58
100	1	**61.66**	228.93	59.60	346.65	50.52	104.43
	2	**60.32**	223.04	57.88	266.50	52.36	122.27
150	1	**57.70**	286.83	55.74	306.60	50.32	157.27
	2	**57.20**	258.09	54.38	195.92	50.90	129.77
200	1	**54.62**	282.27	52.46	72.89	50.92	152.97
	2	**54.48**	268.08	52.06	149.85	49.88	162.23

This test evaluates the one-sided alternative hypothesis that a solution value obtained by Brkga-Learn is, in the expected case, larger than the corresponding solution value obtained by Brkga. We obtained a p-value of less than 10^{-4}, which indicates that the observed differences are statistically highly significant.

Chapter 4

Beam Search for the RLCS Problem

4.1 Introduction

This chapter presents the Beam Search (BS) for the Restricted Longest Common Subsequence (RLCS) problem and its integration with the offline learning framework for learning its heuristic function. The content is based on publications [38, 39]. The first presents the BS while the second applies the offline learning framework to it.

The BS for the RLCS problem, a variant of the Longest Common Subsequence (LCS) problem introduced in Chapter 2, is closely related to the BS for the LCS problem, also presented in that chapter. The only difference with the latter are the nodes of the graph partially constructed during the BS execution, the state graph. In the case of the RLCS problem, nodes also represent partial solutions and hence, in addition to the partial solution length l_v and the left position vector $p^{L,v}$, they also contain information regarding the restricted strings.

Consider a set of input strings $S = \{s_1, s_2, \dots, s_m\}$ and restricted strings $R = \{r_1, r_2, \dots, r_k\}$. A node v represents a partial solution s^v and consists of a triple $(l_v, p^{L,v}, r^v)$. As with the LCS problem, l_v is the length of s^v and $p^{L,v}$ is the left position vector, the vector such that for each input string s_i the index $p_i^{L,v} - 1$ is the smallest value for which $s_i[1, p_i^{L,v} - 1]$ contains s^v as a subsequence. Finally, r^v is the vector that contains information about the restricted strings. For each restricted string r_j, the index r_j^v is such that the prefix $r_j[1, r_j^v]$ is not contained as a subsequence of s^v, but $r_j[1, r_j^v - 1]$ is.

As with the LCS problem, edges represent partial solution extensions. There is an edge a between two nodes $v_1 = (l_{v_1}, p^{L,v_1}, r^{v_1})$ and $v_2 = (l_{v_2}, p^{L,v_2}, r^{v_2})$ with label $l(a)$ if (i) $l_{v_2} = l_{v_1} + 1$ and (ii) the partial solution represented by node v_2 is obtained by appending character a to the partial solution represented by v_1.

For extending a node v, its successor nodes have to be determined. This consists of identifying the characters that can feasibly extend the partial solution s^v represented by v. The same set of available characters as in the case of the

LCS problem is considered, and is further restricted by taking into account the restricted strings. First, the characters occurring in each string of set $S[p^{L,v}]$ are considered. Then, the characters $\mathtt{a} \in \Sigma$ that cause one of the restricted patterns $r_i \in R$ to be a subsequence of $s^v \cdot \mathtt{a}$ are removed. Finally, the dominated letters as in the case of the LCS problem are also omitted. Remember, that a character $\mathtt{a} \in \Sigma$ is said to dominate a character $\mathtt{b} \in \Sigma$ if $p^{L,v}_{i,\mathtt{a}} < p^{L,v}_{i,\mathtt{b}}$ for all $i = 1, \ldots, m$, where $p^{L,v}_{i,\mathtt{a}}$ and $p^{L,v}_{i,\mathtt{b}}$ denote the first occurrence in $s[p^{L,v}_i, |s_i|]$ of character a and b respectively. As a appears before b in the remaining part of each input string, extending the partial solution through b will never lead to a better solution than doing it through a. We denote the set of non-dominated feasible characters to extend the partial solution of a node v by Σ^{nd}_v.

As with the LCS state graph, the state graph for the RLCS problem is obtained by starting with the root node $r = (0, (1, \ldots, 1), (1, \ldots, 1)$, which represents the empty string, expanding it with the characters in Σ^{nd}_r, and doing the same for every obtained node iteratively, until every node v not yet expanded has $\Sigma^{nd}_r = \emptyset$. Figure 4.1, shows an example state graph for the RLCS problem with a particular set of input and restricted strings. It is similar to the LCS state graph, the only difference lies in the information kept about the restricted strings.

The BS for the RLCS problem works in the same way as the BS for the LCS problem, described in Algorithm 2.2 of Chapter 2. Again, the only two differences lie in the structure of nodes, which now keep track of which suffixes of the restricted strings are subsequence of the node's partial solution, and in the further restriction of the characters considered for extension.

The final component to define is the heuristic function h. Given a set of input strings $S = \{s_1, s_2, \ldots, s_m\}$ and a set of restricted strings $R = \{r_1, r_2, \ldots, r_k\}$, every valid RLCS solution is also a solution to the LCS problem for the input strings S. As a result, any heuristic function designed for the LCS problem can also serve as a valid heuristic for the RLCS problem. We take advantage of this by employing the UB guiding function, which was originally introduced for the LCS. However, we also consider an alternative heuristic function, shown to deliver better performance in some cases. This is similar to the heuristic function EX for the LCS problem and we describe it next.

At a given node v and for an arbitrary string s of length k, let X_i be the discrete random variable for each $i = 1, 2, \ldots, m$, where X_i takes the value one if s is a subsequence of $s_i[p^{L,v}_i + 1, |s_i|]$, and zero otherwise. Assuming these events are independent, we can write:

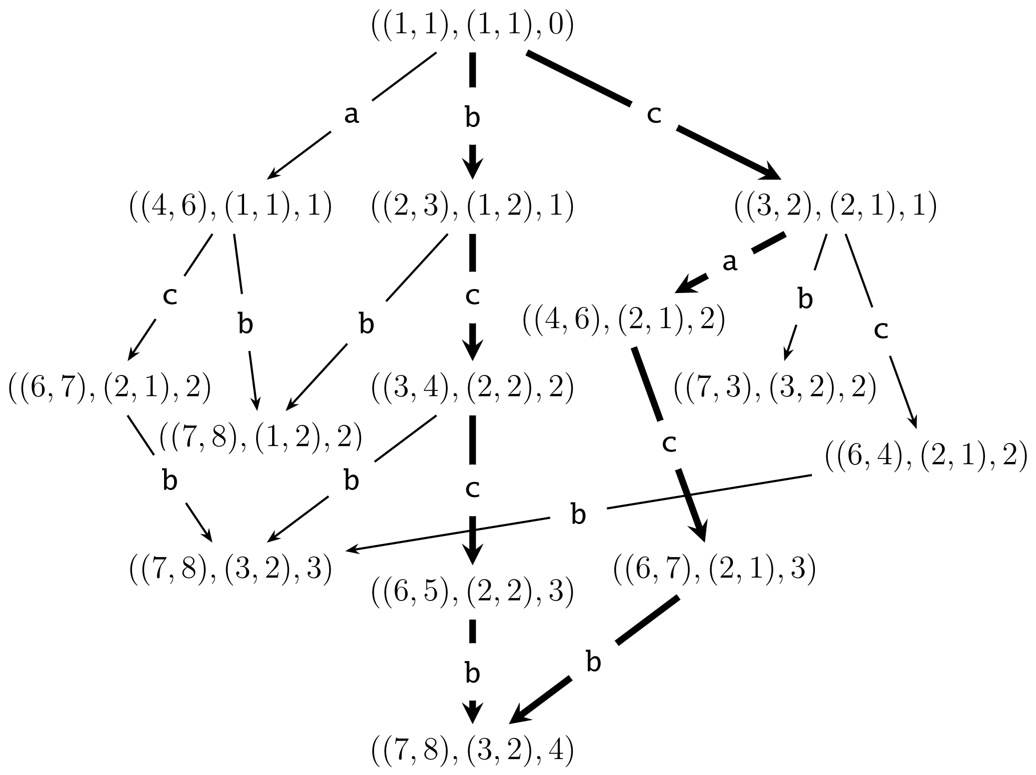


Figure 4.1 RLCS problem state graph for $S = \{\texttt{bcaacbb}, \texttt{cbccacb}, \}$, $R = \{\texttt{cbb}\}$. It contains four complete nodes. The two bold paths from the root node to $((7,8),(3,2),4)$ to are the longest in the graph. Hence, they represent the two optimal solutions for this problem instance, bccb and cacb respectively.

$$P\left(\bigcap_{i=1}^{m} E_i = 1\right) = \prod_{i=1}^{m} P(E_i = 1) = \prod_{i=1}^{m} P(k, |s_i| - p_i^{L,v} + 1) \tag{4.1}$$

where $P(k, |s_i|)$ represents the probability that an arbitrary string of length $|s_i| - p_i^{L,v} + 1$ contains an arbitrary string of length k as a subsequence. Assuming both strings are generated uniformly at random, the same recurrence relation as in the EX heuristic heuristic proposed by [93], described in Expression 2.9 of Chapter 2, can be employed to compute these probabilities. This approximation serves as the heuristic for node v.

$$H_{RLCS}(v) = \prod_{i=1}^{m} P(k, |s_i| - p_i^{L,v} + 1) \tag{4.2}$$

Here, k is considered a parameter that is heuristically determined at each level of the BS. In [93], this heuristic guiding function is applied to the LCS problem, where the value of k is computed using the following expression:

$$k = \max\left\{1, \left\lceil \min_{v \in V_{ext}} \frac{|s_i| - p_i^v + 1}{|\Sigma|} \right\rceil \right\} \tag{4.3}$$

As stated in the original paper [93], the underlying intuition is that the probability of extending a partial solution decreases as $|\Sigma|$ increases and increases as $\min_{v \in V_{ext}}(|s_i| - p_i^v + 1)$ grows. We propose an improved approach by avoiding the consideration of all nodes in V_{ext}, which can lead to an underestimation of k. Instead, we determine a more suitable value of k based on a subset $V'_{ext} \subseteq V_{ext}$ consisting of promising nodes.

To determine V'_{ext} at each level, we sort all nodes in decreasing order according to their UB value, introduced in Equation (2.8) on page 36. A tie-breaking mechanism is applied, utilizing information from the restricted strings through the so-called $R_{\min}$ score, which is defined as:

$$R_{\min}(v) = \min\{|r_i| - l^v + 1 \mid i = 1, \ldots, k\} \tag{4.4}$$

where larger values are preferred. The size of V'_{ext} is determined by the algorithm parameter *percent_extension* $\in [0, 100]$, which sets the size of V'_{ext} as a percentage of the size of V_{ext}.

Once V'_{ext} is selected, Equation (4.3) is applied using V'_{ext} instead of V_{ext} to determine the value of k. If two nodes have the same value according to H_{RLCS}, the node with the larger $R_{\min}$ value is prioritized.

4.2 Implementing the Offline Learning Component

The search component that is parameterized is the heuristic function h, and the model used is a feed-forward neural network. Given a node, the neural network processes a set of features and outputs a heuristic value for that node. Specifically, in the case of the BS applied to the RLCS problem, a Biased Random Key Genetic Algorithm (BRKGA) was employed for training instead of a standard Random Key Genetic Algorithm (RKGA), as this resulted in better optimization of the neural network parameters.

Notably, this modification introduces an additional parameter $\rho_e > 0$ and alters the mating mechanism by ensuring that one parent is always selected from the elite population and the other from the non-elite population. Each gene then inherits the elite parent's value with probability ρ_e. The general structures of RKGA and BRKGA are outlined in Pseudocodes 1.2 and 3.1, on pages 40 and 21, respectively.

4.2.1 Neural Network and Features

The same structure as in Brkga-Learn from the previous chapter is followed. Both node–specific features and global features from the given problem instance are incorporated, enabling the model to consider both the overall characteristics of the instance and the specific properties of each node.

The features specific to the node are as follows. Recall that a node v is stored as a tuple $(l_v, p^{L,v}, r^v)$ in the state graph. Here, l_v represents the length of the node's partial solution. The vector $p^{L,v}$ tracks the suffixes of the input strings available for further extension of the partial solution represented by v. Finally, the vector r^v keeps track of the prefixes of the restricted strings that are subsequences of the same partial solution. These three values are used to define the node-specific features.

The lengths of the vectors $p^{L,v}$ and r^v depend on the number of input and restricted strings, respectively. Additionally, the scale of their values is influenced by the lengths of these strings. To ensure that the number of features and their scale remain comparable across different instance sizes, the information contained in these vectors is summarized as follows. First, both vectors are normalized with respect to the string length in the following manner:

$$\tilde{p}_i^{L,v} = \frac{p_i^{L,v}}{|s_i|} \text{ for } i \in \{1,\dots,m\} \text{ and } \tilde{r}_j^v = \frac{r_j^v}{|r_j|} \text{ for } j \in \{1,\dots,k\} \tag{4.5}$$

The maximum, minimum, average, and sample standard deviation of the resulting standardized vectors are then used as the corresponding node features, ensuring that the number of features remains independent of the instance size:

$$\begin{aligned}\Big(\max\left(\tilde{p}^{L,v}\right), \min\left(\tilde{p}^{L,v}\right), \mathrm{avg}\left(\tilde{p}^{L,v}\right), \mathrm{sd}\left(\tilde{p}^{L,v}\right),& \\ \max\left(\tilde{r}^{v}\right), \min\left(\tilde{r}^{v}\right), \mathrm{avg}\left(\tilde{r}^{v}\right), \mathrm{sd}\left(\tilde{r}^{v}\right)\Big)&\end{aligned} \tag{4.6}$$

To incorporate global information about the problem instance, the following features are included: (i) alphabet size ($|\Sigma|$), (ii) number of input strings (m), and (iii) number of restricted strings (k). Two benchmark sets of problem instances will be employed for evaluating the BS: Random and Abstract. In the case of the first one, the length of the input strings (n) and the length of the restricted strings ($|r_0|$) are also included as additional features, since, within each instance, all input strings and all restricted strings have the same length, respectively.

Therefore, a total of 13 features are extracted when addressing a problem instance from benchmark set Random, while 11 features are extracted when dealing with an instance from the benchmark set Abstract. Finally, before being

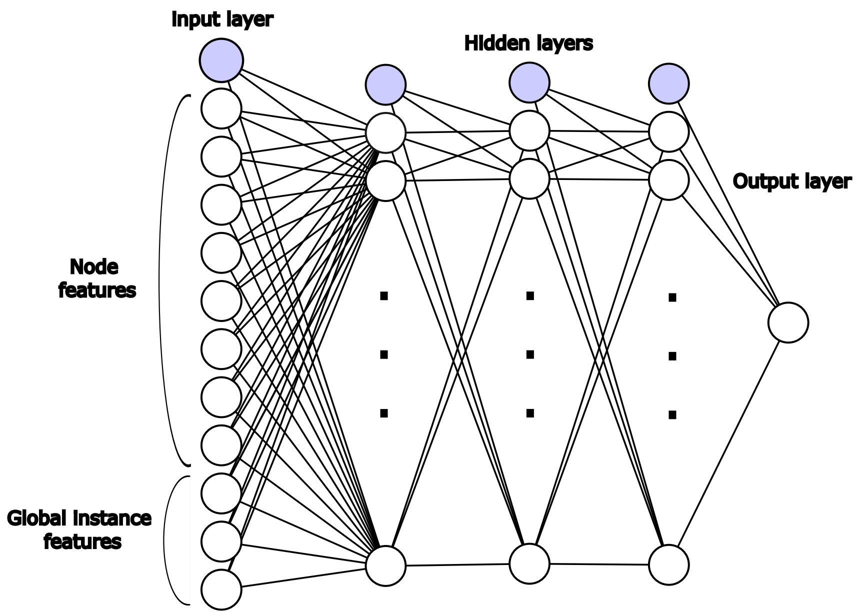


Figure 4.2 A graphical representation of the feed-forward neural network employed for benchmark set Abstract. The lines from the top node in each layer represent the biases. Remember that two extra instance features are considered in the context of benchmark set Random, as opposed to the benchmark set Abstract.

fed into the neural network, all features are standardized to have a mean of zero and a variance of one.

After conducting several preliminary experiments, we selected a neural network architecture consisting of three hidden layers. The first two hidden layers contain 10 nodes each, while the final hidden layer comprises 5 nodes. All hidden layers utilize the `sigmoid` activation function. The structure of the chosen neural network is depicted in Fig. 4.2.

4.2.2 Evaluation of an Individual

Since BS executions are generally fast, the entire algorithm is run to compute both training and validation values. The evaluation of an individual, i.e., a set of neural network weights, is performed as follows. First, the neural network is initialized with the individual's weights. Then, BS, guided by the neural network, is applied to each instance from the training set. The average length of the obtained solutions is recorded as the training value. Similarly, the validation value is computed whenever a new best individual is identified. This is done by executing BS guided by the neural network once again, but this time on the validation instances. The validation value is then set as the average length of the solutions obtained. Recall that validation values are used to determine when to end the training process as their decrease might indicate overfitting.

Regarding the beam width β used for computing training and validation values, its value was varied depending on the benchmark set. This will be described in the next section.

4.3 Experimental Evaluation

This section presents a comprehensive experimental comparison between the standard BS, guided by the probability-based heuristic introduced at the beginning of this chapter, denoted as BS-Prob and the BS guided by the learned heuristic function, referred to as BS-Learn. The evaluation is conducted using two benchmark sets, which we introduce below. We do not include a comparison to the previous state-of-the art heuristic algorithms for the RLCS problem due to their bad relative performance to the BS variants presented, as can be seen in [38].

4.3.1 Problem Instances

The first benchmark set, Random, follows a similar structure to the benchmark set of the same name used for evaluating the BRKGA for the Longest Common Square Subsequence (LCSqS) problem. It consists of instances where strings are generated uniformly at random. Specifically, five instances were generated for each combination of parameters: $m \in \{3, 5, 10\}$, $k \in \{3, 5, 10\}$, $n \in \{200, 500, 1000\}$, $|r| \in \{0.01 \cdot n, 0.02 \cdot n, 0.05 \cdot n\}$, and $|\Sigma| \in \{4, 20\}$. Here, m and k represent the number of input and restricted strings, respectively, while n and $|r|$ denote their corresponding lengths, and $|\Sigma|$ is the alphabet size. Notably, in this benchmark set, all input strings within an instance share the same length, and the same holds for the restricted strings. In total, this dataset comprises 810 RLCS instances.

The benchmark set Abstract consists of 298 instances, where the input strings are derived from the existing Abstract dataset in the literature, previously used for the LCS problem in [96]. The instances in this benchmark set comprise strings available at `https://cwi.ugent.be/respapersim`. These strings correspond to abstracts of real scientific papers written in English and are specifically designed for experiments assessing research paper similarity. Notably, for a subset of abstracts, each pair is accompanied by a label indicating whether the corresponding research papers were classified as similar or dissimilar by an expert.

The Abstract instances were generated by forming two groups of 12 instances each, one group with abstracts deemed as similar (POS) and one group with abstracts deemed as dissimilar (NEG). Then, for each of these groups and $m \in$

$\{10, 11, 12\}$, $\binom{12}{m}$ different instances were generated containing m input strings (one for each combination). This lead to one instance containing all 12 strings, 12 instances containing 11 strings and, 66 instances containing ten strings for each POS and NEG strings. Moreover, for $m \in \{3, 4, \ldots, 9\}$ ten of the corresponding $\binom{12}{m}$ instances were also added to the benchmark set.

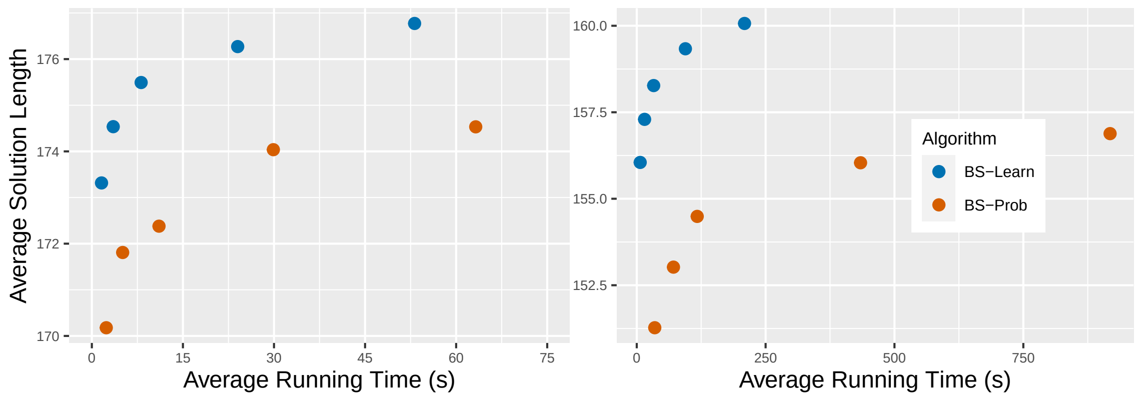
Restricted strings are added to these input strings to construct RLCS problem instances. These restricted strings consist of the sixty most frequently occurring words in research literature, as identified in [26]. The purpose of this approach is to facilitate abstract comparisons while excluding the most common words in academic writing, which are typically not relevant for assessing content similarity.

4.3.2 Training and Parameter Tuning

BS does not impose a time limit, as its execution time is inherently regulated by the beam width parameter, β. This parameter controls the trade-off between computational effort and solution quality: a larger β results in higher-quality solutions but increases runtime. To ensure a fair comparison between BS-Prob and BS-Learn, both methods must use the same beam width. Based on preliminary experiments, we selected $\beta = 5000$ to present the numerical results, as it achieved high-quality solutions while maintaining reasonable execution times. Apart from β, BS-Learn does not require any additional parameters. In contrast, BS-Prob includes the *percent_extensions* parameter, which was set to $\frac{100}{3}$ following preliminary experimentation.

Figure 4.3 illustrates how varying the beam width β affects the trade-off between solution quality and execution time. Each subplot corresponds to a different benchmark set and presents five data points per algorithm, reflecting the average solution length and average runtime for the following beam width values: $\beta = 500, 1000, 2000, 5000, 10000$. Within each algorithm's plot, points are ordered from left to right according to increasing β values. As expected, both BS variants yield better average solution quality as β increases, albeit at the cost of longer execution times.

Notably, for larger beam widths, further increasing β results in only marginal improvements in solution quality while significantly increasing running time. This effect is particularly evident when comparing $\beta = 5000$ and $\beta = 10000$. The plots also highlight the superior average performance of BS-Learn over BS-Prob: BS-Learn consistently produces longer average solutions and requires less execution time across all considered values of β. Interestingly, the performance gap is especially pronounced in the Abstract benchmark set, particularly at higher beam widths.



(a) Benchmark set RANDOM. **(b)** Benchmark set ABSTRACT.

Figure 4.3 Relation between beam width and the balance between solution quality and execution time for the RANDOM and ABSTRACT benchmark sets.

For the BS-LEARN training parameters, we used the same default RKGA training settings as in the case of the LCSqS problem, which consist of 20 individuals, one elite individual, and seven mutants. Additionally, since a BRKGA was used for training, an elite inheritance probability ρ_e of 0.5 was applied. Other values were tested without significant impact, so such default was chosen. For computing training and validation values, we set $\beta = 100$ for the RANDOM benchmark set and $\beta = 200$ for the ABSTRACT benchmark set. The latter setting likely performed better due to the greater difficulty of the instances in the ABSTRACT set.

We trained BS-LEARN separately for each benchmark set. For the RANDOM set, training involved one instance for every combination of m, k, n, $|r|$, and $|\Sigma|$ to compute the training and validation values, resulting in a total of 162 different instances for each. It is important to note that these instances were not used for the algorithm's evaluation. For the ABSTRACT set, three instances for each value of k were used to compute the training and validation values, except for $k = 11$, for which 5 instances were used, and $k = 12$, for which only one instance was available. In this case, the instances used for training were also used for evaluating the algorithm due to the limited number of available instances.

All experiments were conducted in single-threaded mode on an Intel Xeon E5-2640 processor with a clock speed of 2.40 GHz and 16 GB of RAM. The training durations were approximately 15 hours for the RANDOM benchmark set and 63 hours for the ABSTRACT benchmark set. It is important to note that, compared to the BRKGA trainings for the LCSqS problem, a significantly larger number of training instances were used here. This is because the computation of validation values is much faster than in the case of the LCSqS problem. The training process was executed in single-threaded mode. However, a similar multi-threaded training approach, as used for the BRKGA, could be implemented, which would likely reduce training times.

4.3.3 Results for Benchmark Set Random

The results obtained for the benchmark set Random are presented in Tables 4.1–4.9. Each table corresponds to a specific combination of $n \in \{200, 500, 1000\}$ and $m \in \{2, 5, 10\}$. For each combination of $|k|$, $|p|$, and $|\Sigma|$, as well as for each algorithm, we report the average length of the best solution found ($\overline{|s|}$) and the average runtime ($\bar{t}_{\text{best}}[s]$). Each group of instances consists of five individual instances, and the results are averaged over the five runs. For clarity, the best results in each table are highlighted in bold.

Table 4.1 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 3$ and input string length $n = 200$.

			BS-Prob		BS-Learn	
k	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	2	4	**61.20**	0.64	**61.20**	0.46
		20	**44.80**	28.72	**44.80**	4.04
	4	4	**93.80**	2.88	**93.80**	2.11
		20	**46.00**	16.35	**46.00**	5.93
	10	4	106.60	3.32	**106.80**	3.48
		20	**45.80**	6.44	**45.80**	6.59
5	2	4	**52.80**	0.02	**52.80**	0.02
		20	**43.80**	34.26	**43.80**	4.73
	4	4	**91.80**	2.65	**91.80**	2.06
		20	**45.20**	18.93	**45.20**	7.52
	10	4	105.60	3.80	**106.00**	3.82
		20	**44.60**	8.55	**44.60**	8.91
10	2	4	**37.00**	0.00	**37.00**	0.00
		20	**41.00**	35.73	**41.00**	6.18
	4	4	**87.80**	2.52	**87.80**	1.76
		20	**44.20**	30.50	**44.20**	11.33
	10	4	101.00	3.69	**101.20**	3.62
		20	**45.00**	11.64	44.80	12.50

Table 4.2 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 5$ and input string length $n = 200$.

			BS-Prob		BS-Learn	
k	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	2	4	**74.20**	1.37	**74.20**	1.02
		20	**29.40**	19.37	**29.40**	4.27
	4	4	88.80	3.15	**89.20**	3.01
		20	**30.00**	7.92	**30.00**	5.46
	10	4	94.20	3.59	**94.60**	4.26
		20	**29.40**	5.17	**29.40**	5.53
5	2	4	**55.00**	0.63	**55.00**	0.44
		20	**27.60**	23.75	**27.60**	4.77
	4	4	**89.80**	3.33	89.40	2.77
		20	**29.60**	10.41	**29.60**	6.39
	10	4	**93.20**	3.84	**93.20**	3.92
		20	**30.20**	6.21	**30.20**	6.50
10	2	4	**38.00**	0.00	**38.00**	0.00
		20	**29.20**	27.78	**29.20**	5.82
	4	4	**74.40**	2.11	**74.40**	1.69
		20	**29.80**	13.46	**29.80**	8.63
	10	4	89.60	3.84	**90.20**	3.69
		20	**29.60**	8.63	**29.60**	9.12

For instances with $n = 200$, BS-Prob and BS-Learn perform similarly, with the latter obtaining slightly better solutions in the cases with $m = 10$.

Table 4.3 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 10$ and input string length $n = 200$.

			BS-Prob		BS-Learn	
k	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	2	4	65.80	1.27	**66.00**	1.37
		20	**19.40**	15.05	**19.40**	4.58
	4	4	78.00	3.23	**78.20**	3.29
		20	**19.60**	6.59	**19.60**	4.89
	10	4	80.80	3.72	**81.60**	4.06
		20	**19.20**	5.46	**19.20**	4.65
5	2	4	**58.60**	0.01	**58.60**	0.01
		20	**18.60**	15.00	**18.60**	4.45
	4	4	75.00	2.85	**75.80**	2.67
		20	**19.20**	7.08	**19.20**	5.02
	10	4	**82.20**	4.11	82.00	4.36
		20	**19.20**	5.92	**19.20**	5.32
10	2	4	**36.00**	0.00	**36.00**	0.00
		20	**18.60**	16.77	**18.60**	4.27
	4	4	**70.60**	2.54	70.40	1.90
		20	**19.40**	7.56	**19.40**	5.40
	10	4	81.80	4.23	**82.00**	4.14
		20	**19.20**	6.66	**19.20**	5.75

Table 4.4 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 3$ and input string length $n = 500$.

			BS-Prob		BS-Learn	
k	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	5	4	**246.40**	8.65	246.00	6.97
		20	118.40	85.78	**118.60**	22.64
	10	4	**256.00**	8.77	**256.00**	8.55
		20	117.60	31.05	**118.20**	27.60
	25	4	269.20	9.05	**275.00**	11.35
		20	116.80	24.64	**117.20**	28.78
5	5	4	**237.40**	8.16	237.20	6.72
		20	115.00	110.09	**115.40**	28.76
	10	4	254.00	9.30	**256.20**	9.26
		20	117.60	40.08	**117.80**	35.97
	25	4	262.80	10.57	**267.80**	12.27
		20	118.00	30.76	**118.60**	37.06
10	5	4	**220.00**	7.78	**220.00**	5.30
		20	**116.40**	134.52	116.20	36.24
	10	4	247.00	9.76	**252.20**	9.11
		20	115.80	66.76	**117.00**	50.64
	25	4	261.20	10.64	**266.20**	12.53
		20	**118.40**	43.13	118.00	50.67

As n increases, the differences between the algorithms become more pronounced. Specifically, for $n = 500$, the results are mixed, with BS-Learn outperforming BS-Prob in instances with $m = 3$, where it achieves the best performance in 12 cases, while BS-Prob is the best in 4 cases. For $m = 5$, BS-Learn obtains the best results in 6 cases, while BS-Prob does so in 8 cases. Finally, for $m = 10$, both approaches achieve the best results in 6 cases, with a tie in the remaining instances.

For the largest instances, BS-Learn proves to be the superior approach, consistently finding better solutions on average compared to BS-Prob across all values of m. Specifically, BS-Learn outperforms BS-Prob in 15, 15, and 13 cases for instances with $m = 3$, $m = 5$, and $m = 10$, respectively.

As in the case of the LCSqS problem, we employ the signed-rank Wilcoxon test [128] to measure the statistical significance of the observed differences. We test the one-sided alternative hypothesis that a solution obtained by BS-Learn is

Table 4.5 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 5$ and input string length $n = 500$.

			BS-Prob		BS-Learn	
k	$\|r\|$	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
3	5	4	**225.60**	9.40	225.40	9.40
		20	**80.00**	52.52	79.60	22.50
	10	4	**230.00**	9.72	229.00	9.20
		20	**79.00**	22.79	78.80	25.19
	25	4	233.60	9.29	**243.20**	12.59
		20	79.80	22.61	**80.40**	26.35
5	5	4	**213.80**	8.36	212.80	8.00
		20	**79.00**	68.42	78.80	25.43
	10	4	**228.80**	10.11	228.40	9.83
		20	**80.00**	25.79	**80.00**	30.39
	25	4	235.80	9.96	**239.20**	12.86
		20	79.00	25.42	**79.40**	29.36
10	5	4	**213.00**	8.84	212.80	8.01
		20	**78.20**	89.87	**78.20**	30.74
	10	4	**230.60**	10.52	**230.60**	9.23
		20	79.20	31.53	**79.40**	37.59
	25	4	225.00	10.21	**237.60**	13.60
		20	**79.20**	30.26	**79.20**	36.05

Table 4.6 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instanceswith number of input strings $m = 10$ and input string length $n = 500$.

			BS-Prob		BS-Learn	
k	$\|r\|$	$\|\Sigma\|$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
3	5	4	202.40	9.65	**205.00**	12.77
		20	**53.80**	36.37	53.40	22.87
	10	4	**208.40**	11.10	203.80	9.54
		20	**54.00**	25.14	**54.00**	24.34
	25	4	212.00	10.67	**213.00**	12.95
		20	**53.20**	24.60	**53.20**	24.42
5	5	4	**194.60**	9.18	193.20	8.78
		20	53.20	41.49	**53.40**	24.49
	10	4	200.60	9.55	**205.40**	12.52
		20	53.20	25.11	**53.40**	25.75
	25	4	**210.60**	11.18	210.40	13.35
		20	**53.80**	25.07	**53.80**	25.51
10	5	4	**186.40**	8.46	183.80	7.74
		20	**53.40**	55.23	**53.40**	24.38
	10	4	**204.00**	10.74	**204.00**	12.68
		20	**54.00**	25.61	53.80	27.30
	25	4	209.60	12.01	**209.80**	14.59
		20	**53.60**	25.77	**53.60**	26.32

longer than that obtained by BS-Prob in the expected case. By testing separately based on n, we obtain p-values of 0.06, less than 10^{-3}, and less than 10^{-9} for $n = 200$, $n = 500$, and $n = 1000$, respectively. This indicates that, for $n = 500$ and $n = 1000$, the differences are statistically highly significant.

4.3.4 Results for Benchmark Set Abstract

Numerical results for the Abstract dataset are provided in Tables 4.10–4.11. The results are divided into two tables: one for the NEG instances and the other for the POS instances. Each row presents the average results obtained by BS-Prob and BS-Learn, similar to the results for the Random benchmark set. The first two values in each row define the instance group, as they correspond to the number of input strings and the group's number of instances.

For NEG instances, BS-Learn performs better for instances with $m \leq 7$. Interestingly, a similar pattern emerges for the POS instances, where BS-Learn

Table 4.7 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 3$ and input string length $n = 1000$.

			BS-Prob		BS-Learn	
k	$\|r\|$	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{t}_{best}[s]$	$\overline{\|s\|}$	$\overline{t}_{best}[s]$
3	10	4	505.00	17.69	**507.40**	16.55
		20	**238.40**	149.54	237.00	53.04
	20	4	512.60	18.47	**522.20**	17.26
		20	240.80	65.68	**241.00**	69.08
	50	4	539.20	19.76	**550.40**	24.44
		20	241.00	59.27	**242.20**	68.62
5	10	4	470.40	17.40	**505.60**	17.01
		20	**237.60**	186.22	236.80	68.37
	20	4	503.00	19.50	**513.20**	18.88
		20	238.00	75.00	**240.40**	81.20
	50	4	526.00	19.95	**541.80**	24.80
		20	239.60	72.77	**240.60**	86.11
10	10	4	464.40	17.65	**485.40**	17.49
		20	**232.80**	241.04	229.20	85.40
	20	4	505.20	20.98	**508.80**	19.80
		20	233.40	96.73	**238.80**	113.63
	50	4	517.80	22.29	**537.60**	29.33
		20	236.20	94.58	**238.20**	120.36

Table 4.8 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 5$ and input string length $n = 1000$.

			BS-Prob		BS-Learn	
k	$\|r\|$	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{t}_{best}[s]$	$\overline{\|s\|}$	$\overline{t}_{best}[s]$
3	10	4	**456.80**	20.40	455.40	18.64
		20	**162.40**	83.59	**162.40**	54.00
	20	4	462.40	20.08	**463.20**	20.80
		20	162.80	52.39	**163.00**	60.69
	50	4	477.20	21.09	**488.00**	24.83
		20	160.80	53.10	**162.20**	63.47
5	10	4	440.60	19.25	**451.80**	19.85
		20	**161.80**	115.87	161.60	63.09
	20	4	459.00	19.82	**463.80**	23.11
		20	162.40	57.74	**163.20**	70.56
	50	4	474.80	21.52	**487.80**	27.09
		20	164.80	58.68	**165.40**	69.01
10	10	4	399.60	16.77	**451.60**	21.54
		20	161.20	159.76	**161.40**	80.51
	20	4	456.60	22.86	**460.40**	23.03
		20	160.00	68.93	**161.40**	86.91
	50	4	454.60	20.82	**481.80**	30.35
		20	161.60	68.25	**163.60**	86.79

is the superior approach for instances with $m \leq 10$. The better performance of BS-Prob for the NEG instances compared to the POS ones may be attributed to the fact that NEG instances are less similar. This aligns with the probabilistic heuristic function employed by BS-Prob, which assumes that the events in which an arbitrary string is a subsequence of the input strings are independent.

The superiority of BS-Prob over BS-Learn for larger values of m can be attributed to BS-Prob benefiting from a larger number of input strings, which allows for more differentiated UB values. This, in turn, enables a more relevant subset of extensions V'_{ext} to be selected. On the other hand, BS-Learn may require more intensive training for larger instances. One possible solution could be to train the model separately depending on the number of input strings.

Generally, the RLCS solutions for instances in the POS set are larger than those for the corresponding-sized instances in the NEG set, except for the smallest instances with $m = 3$.

Table 4.9 Comparison of BS-Prob and BS-Learn on the RLCS problem Random instances with number of input strings $m = 10$ and input string length $n = 1000$.

			BS-Prob		BS-Learn	
k	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	10	4	412.20	19.39	**413.00**	24.09
		20	**111.00**	69.50	110.00	59.45
	20	4	402.20	21.46	**411.20**	22.71
		20	111.00	56.20	**111.40**	62.42
	50	4	**429.80**	23.14	427.40	28.76
		20	110.40	57.99	**111.00**	60.76
5	10	4	400.60	21.57	**409.20**	26.62
		20	**111.40**	85.40	110.60	60.84
	20	4	414.60	22.65	**416.40**	27.00
		20	111.40	58.55	**112.00**	64.55
	50	4	427.40	22.75	**427.80**	29.68
		20	110.40	58.35	**110.80**	64.80
10	10	4	395.00	21.12	**405.80**	22.05
		20	**110.80**	77.61	110.60	62.35
	20	4	**414.20**	24.91	413.20	26.23
		20	111.60	60.03	**111.80**	64.85
	50	4	409.20	22.52	**427.00**	31.77
		20	110.40	58.99	**110.80**	61.11

Table 4.10 Comparison of BS-Prob and BS-Learn on the RLCS problem Abstract instances of type POS.

		BS-Prob		BS-Learn	
m	#inst	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	10	225.20	649.78	**246.50**	150.53
4	10	211.60	604.41	**225.50**	124.65
5	10	188.60	540.93	**203.80**	110.23
6	10	171.40	507.56	**178.50**	97.64
7	10	162.20	492.51	**168.50**	95.84
8	10	151.50	467.62	**154.10**	95.97
9	10	146.70	435.61	**149.70**	90.67
10	66	136.23	377.25	**137.83**	88.49
11	12	**133.08**	331.96	132.25	89.61
12	1	**131.00**	294.85	129.00	80.55

Table 4.11 Comparison of BS-Prob and BS-Learn on the RLCS problem Abstract instances of type NEG.

		BS-Prob		BS-Learn	
m	#inst	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
3	10	231.90	644.95	**248.80**	141.06
4	10	210.10	635.06	**222.80**	123.69
5	10	190.50	589.71	**193.60**	98.51
6	10	172.60	460.74	**173.30**	94.92
7	10	167.50	456.79	**168.50**	89.19
8	10	**149.50**	433.96	148.00	85.06
9	10	**146.00**	422.70	144.50	83.53
10	66	**134.05**	343.09	132.50	79.44
11	12	**129.58**	321.23	128.25	83.44
12	1	**126.00**	296.43	124.00	72.82

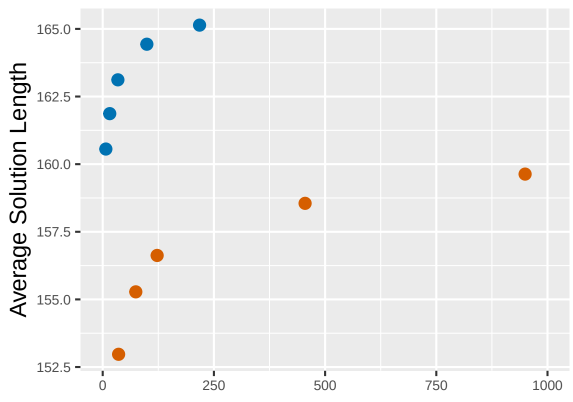
This observation aligns with the classification from the literature, which categorizes these instances into either the positive (POS) or negative (NEG) group based on the similarity of the research papers represented by the input strings. Furthermore, restricting the 60 most frequent academic words results in solutions for NEG and POS instances of the same size having a length difference approximately 2% larger compared to when no strings are restricted, as noted in [96]. This suggests that incorporating the most common academic words as restricted strings aids in determining the similarity of the abstracts in a given instance.

We again test the statistical significance of the results obtained using the signed-rank Wilcoxon test [128]. The POS and NEG instances are tested separately, with further separation into instances with $m \leq 7$ and $m > 7$ for the NEG instances, and instances with $m \leq 10$ and $m > 10$ for the POS instances.

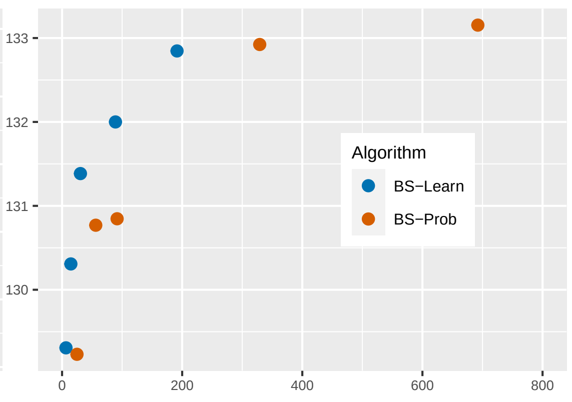
For the NEG instances with $m \leq 7$ and the POS instances with $m \leq 10$, we test the hypothesis that a solution obtained by BS-Learn is longer than one obtained by BS-Prob in the expected case. This produces p-values of less than 10^{-5} and 10^{-15}, respectively.

For the NEG instances with $m > 7$ and the POS instances with $m > 10$, we test the hypothesis that a solution obtained by BS-Prob is longer than one obtained by BS-Learn in the expected case. This produces p-values of less than 10^{-8} and of 0.08, respectively.

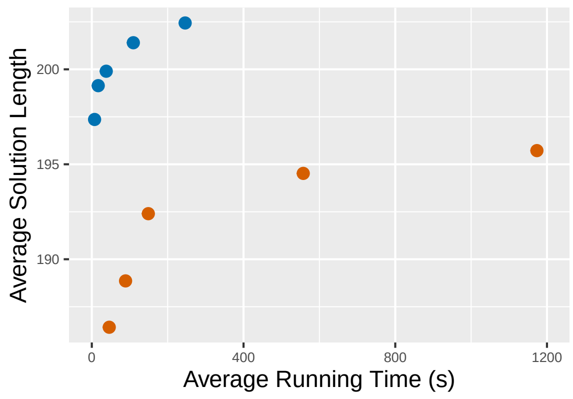
Figure 4.4 presents the average results and running times obtained by both BS variants across different beam widths for the four subsets of the Abstract benchmark set: NEG instances with $m \leq 7$, NEG instances with $m > 7$, POS instances with $m \leq 10$, and POS instances with $m > 10$. The same structure is followed as with Figure 4.3 for plotting. Interestingly, the overall performance patterns remain consistent across all beam widths, with one notable exception, POS instances with $m > 10$. In this subset, BS-Learn outperforms BS-Prob for $\beta = 500$ and $\beta = 2000$, while it underperforms in the remaining cases.



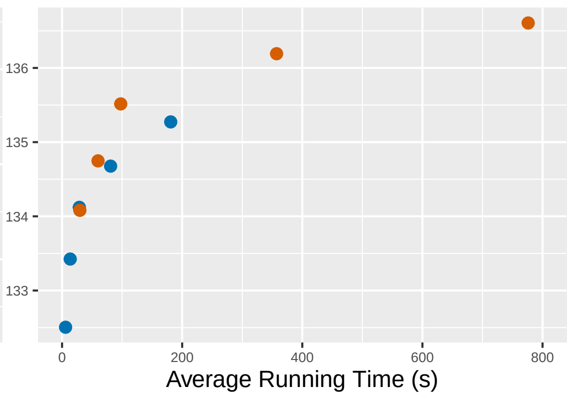
(a) POS instances with $m \leq 10$.



(b) POS instances with $m > 10$.



(c) NEG instances with $m \leq 7$.



(d) NEG instances with $m > 7$.

Figure 4.4 Relation between beam width and the balance between solution quality and execution time for different groups of the Abstract instances

Chapter 5

The Clarke and Wright Heuristic for VRPs

5.1 Introduction

This chapter presents a last application of our offline learning framework. So far, we have presented two applications to algorithms applied to solving-string related problems. In this case, the problem tackled is of a reasonably different kind. In particular, we present an application of our offline framework to the Clarke and Wright heuristic, which can be applied to obtain reasonably good solutions to most Vehicle Routing Problem (VRP) variants. The content is based on a journal article currently under review.

In its simplest form, the VRP, introduced in 1959 by Dantzig and Ramser [28], aims to find the optimal set of routes for a fleet of vehicles so that every customer in a given set is served. Many variants of the problem exist, each with its own particular characteristics designed to model realistic scenarios. Some notable variants include the Vehicle Routing Problem with Time Windows (VRPTW) [30], which considers time slots in which deliveries need to take place, the Vehicle Routing Problem with Pickup and Delivery (VRPPD) [91], which considers pickup requirements on top of the delivery ones, and the Electric Vehicle Routing Problem (EVRP) [79] and its variants, which consider electric delivery vehicles subject to battery-capacity constraints.

One of the simplest and most studied VRP variants is the Capacitated Vehicle Routing Problem (CVRP) [124], which considers delivery vehicles with a limited capacity. It can be defined as follows. Given a depot D, a fleet of m identical vehicles each with capacity $C > 0$ and n customers each with a demand $d_i \geq 0$, the goal of the CVRP is to find a set of routes with minimum routing cost such that:

- Each customer is served once by one vehicle.
- Each route starts and ends at the depot.
- The total demand on each route does not exceed the vehicle capacity C.

Algorithm 5.1 Pseudo-code of the Clarke and Wright heuristic

```
Input: Problem instance $I$
Output: Routing solution $S$
1: $S :=$ create_initial_solution($I$)
2: savings_list := compute_savings_list($I, S$)
3: for $s \in$ savings_list do
4:     $r :=$ merge_routes($s, S$)
5:     if is_valid($r$) then
6:         accept_merge($S, r$)
7:     end if
8: end for
9: return $S$
```

Routing cost is generally defined as follows:

$$\sum_{r \in \mathcal{R}} \sum_{(i,j) \in r} c_{ij} \tag{5.1}$$

$\mathcal{R}$ represents the solution, which is a set of routes. c_{ij} represents the travel cost between customers i and j. This value is usually defined as the distance, time, or energy consumption. Other objectives are also considered in the literature, such as minimizing the number of vehicles used, or the maximum route length.

The Clarke and Wright heuristic is a fast algorithm for obtaining reasonably good solutions to VRPs which can be applied to most problem variants [24]. It is based on iteratively merging the routes of a solution, which is initialized as the trivial one, consisting of one route per customer, each route going from the depot to the customer and back.

Algorithm 5.1 presents the standard Clarke and Wright heuristic. As mentioned, the algorithm begins by constructing an initial solution in method create_initial_solution(). This initial solution is formed by one route per customer, each consisting of a path from the depot to the customer and back. Once this is done, function compute_savings_list() calculates the potential savings in terms of objective function value for merging each pair of routes. For a pair of customers c_i and c_j, the potential saving from merging corresponds to the difference in routing cost between the corresponding individual routes and the merged one. Note that if we denote the depot by 0, this value can be expressed as:

$$(c_{0i} + c_{i0} + c_{0j} + c_{j0}) - (c_{0i} + c_{ij} + c_{j0}) = c_{i0} + c_{0j} - c_{ij}. \tag{5.2}$$

Here, c_{kl} is the cost of travel between nodes k and l, measured as distance, time or energy consumption depending on the objective value employed.

Algorithm 5.2 Pseudo-code of the dynamic Clarke and Wright heuristic

```
Input: Problem instance $I$
Output: Routing solution $S$
1: $S$ := create_initial_solution($I$)
2: routes_merged := true
3: while routes_merged do
4:     routes_merged := false
5:     savings_list := compute_savings_list($I, S$)
6:     for $s \in$ savings_list do
7:         $r$ := merge_routes($s, S$)
8:         if is_valid($r$) then
9:             accept_merge($s, r$)
10:            routes_merged := true
11:            break
12:        end if
13:    end for
14: end while
15: return $S$
```

These savings are then sorted in descending order so that the best merges are considered first. Afterwards, the algorithm iterates over the ordered savings list, and merges the pair of routes corresponding to each entry using merge_routes(). A merge corresponding to customers c_i and c_j is then accepted if deemed valid based on the following checks:

1. Customer c_i must be the last customer in its current route, i.e., it must be the last customer visited before returning to the depot.

2. Customer c_j must be the first customer in its route, i.e., it must be the first customer visited after leaving the depot.

3. The merged route must satisfy all problem-specific constraints, such as capacity limits in the CVRP.

This process continues until no further merge can be performed, when the resulting solution is returned.

An important observation is that for simple VRP variants in which cost does not depend on load, the savings values are valid throughout the execution of the algorithm. This is because cost is additive and terms corresponding to previous merges are canceled in the subtraction of Equation (5.2). This is not true if load affects cost, as then these terms cannot be canceled as previous merges affect load and therefore cost. In these cases it can be beneficial to recompute the savings list after every successful merge. This strategy leads to the variant of the Clarke and Wright heuristic presented in Algorithm 5.2, which we denote as dynamic Clarke and Wright.

This is the variant of the Clarke and Wright we will apply our offline framework to, since the VRP variants we consider have load-dependent costs. In particular we will apply the offline framework to learning the savings values of the dynamic Clarke and Wright heuristic applied to two variants of the EVRP, which consider load to determine energy consumption.

Although we focus on these two VRP variants it is worth noting that the offline learning framework can be applied for learning the savings values for any other VRP variant, and could also be applied to the standard Clarke and Wright which does not recompute the savings list after every successful merge.

5.1.1 The Electric Vehicle Routing Problem (EVRP)

EVRPs are an important family of VRPs [79]. In these problems, delivery vehicles are electric, which adds both constraints regarding battery capacity and optional visits to charging stations for recharging. These VRP variants have gained importance in recent years as delivery companies are slowly adapting electric vehicles with the main goal of reducing emissions.

There are many different EVRP variants, to take into account different characteristics. We will apply our learning Clarke and Wright heuristic to two particular EVRP variants: the Capacitated Electric Vehicle Routing Problem (CEVRP) [67], and the Electric Vehicle Routing Problem with Road Junctions and Road Types (EVRP-RJ-RT) [5]. The first one is a standard variant which considers load capacities on top of battery-related constraints. The second extends the CEVRP by modelling road junctions, which are nodes that join two or more road segments (edges). Moreover, this variant considers road types, which define minimum and maximum allowed speeds for each edge.

Similarly to the VRP, given a set of customers, in its simplest form the EVRP aims to find a set of routes with minimum routing cost so that:

1. Each customer is served once by one vehicle.
2. Each route starts and ends at the depot.
3. The battery level of each electric vehicle is always between 0 and its battery capacity.
4. A vehicle's battery is fully charged when visiting a charging station.

As it happens with the VRP, the EVRP is often considered with capacity constraints. The routing cost is often total energy consumption, number of vehicles or a combination of the two. In our case we adopt a lexicographic objective function for both considered EVRP variants, where the primary goal is to minimize the number of vehicles used, and the secondary goal is to minimize total energy consumption.

The first variant considered is the CEVRP, as introduced in [90], extended with additional constraints limiting route duration. These constraints model realistic considerations such as driver working hours, increasing the realism of the problem setting. From now on, we refer to this variant simply as CEVRP.

Note that we consider asymmetric travel costs between nodes. In real-world scenarios, the cost of traveling from one location to another may differ from the reverse trip due to factors such as one-way streets, traffic rules, or terrain variations. Consequently, assuming symmetric travel costs as is often done in simplified formulations can result in overly idealized models.

The mathematical formulation builds upon the model proposed in [90], with modifications to incorporate route duration constraints. To represent these constraints, the depot is divided into two nodes: 0_d for departures and 0_a for arrivals. Furthermore, the objective function employs a sufficiently large constant $K > 0$ to ensure that minimizing the number of vehicles used takes precedence over minimizing total energy consumption.

The problem is defined on a directed graph $G = (V, E)$, where V consists of the depot nodes $0_d, 0_a$, the set of customers $\mathcal{C}$ and the set of charging stations $\mathcal{S}$. To allow multiple visits to the same physical charging station, $\mathcal{S}$ includes multiple copies of each station. The edge set E contains all possible directed edges between distinct nodes, with the exceptions that: no edges connect a charging station to any of its copies, and the depot node 0_d is linked to all nodes except 0_a, while 0_a is reachable from all nodes except 0_d.

Each edge $(i, j) \in E$ is associated with an energy consumption E_{ij} and a travel time t_{ij}. The energy consumption depends on the load and a constant travel speed v, while travel time depends only on v. Each customer $i \in \mathcal{C}$ requires the delivery of a package with load q_i and a service time s_i. Routes in a solution have a maximum duration T_{max}, and vehicles are defined by their maximum carrying capacity C, maximum battery capacity Q, and charging speed g.

$$
\begin{aligned}
&\min \sum_{\substack{i,j\in V\\ i\neq j}} E_{ij}x_{ij} + K\sum_{j\in V} x_{0_d j} && && (1)\\
&\text{s.t} \sum_{\substack{j\in V\\ i\neq j}} x_{ij} = 1 && \forall\, i \in \mathcal{C} && (2)\\
&\sum_{\substack{j\in V\\ i\neq j}} x_{ij} \leq 1 && \forall\, i \in \mathcal{S} && (3)\\
&\sum_{\substack{j\in V\\ i\neq j}} x_{ij} - \sum_{\substack{j\in V\\ i\neq j}} x_{ji} = 0 && \forall\, i \in V \setminus \{0_d, 0_a\} && (4)\\
&f_{0_d} = 0 && && (5)\\
&f_i + (t_{ij} + s_i)x_{ij} - T_{\max}(1 - x_{ij}) \leq f_j && \forall\, i \in \mathcal{C},\, j \in V \setminus \{0_d\} && (6)\\
&f_i + (t_{ij} + g(Q - y_i))x_{ij} - T_{\max}(1 - x_{ij}) \leq f_j && \forall\, i \in \mathcal{S},\, j \in V \setminus \{0_d\} && (7)\\
&t_{0_d j} \leq f_j \leq T_{\max} - (t_{j0_a} + s_j) && \forall\, j \in V \setminus \{0_d, 0_a\} && (8)\\
&0 \leq m_{0_d} \leq C && && (9)\\
&0 \leq m_j \leq m_i - q_i x_{ij} + C(1 - x_{ij}) && \forall\, i, j \in V,\, i \neq j, 0_a,\, j \neq 0_d && (10)\\
&0 \leq y_j \leq y_i - E_{ij}x_{ij} + Q(1 - x_{ij}) && \forall\, i \in \mathcal{C},\, j \in V \setminus \{0_d\},\, i \neq j && (11)\\
&0 \leq y_j \leq Q - E_{ij}x_{ij} && \forall\, i \in \mathcal{S} \cup \{0_d\},\, j \in V \setminus \{0_d\},\, i \neq j && (12)\\
&x_{ij} \in \{0, 1\} && \forall\, i, j \in V,\, i \neq j, 0_a,\, j \neq 0_d && (13)
\end{aligned}
$$

A solution is represented by binary variables x_{ij}, which take the value 1 if a vehicle travels from node i to node j, and 0 otherwise. Equation (1) defines the objective function. The primary goal is to minimize the number of vehicles used, represented by the second summand, which counts the edges leaving the departure depot, each corresponding to a distinct route. Among solutions that use the same number of vehicles, preference is given to the one with the lowest total energy consumption, represented by the first summand.

Constraints (2)–(4) ensure connectivity. Constraint (2) guarantees that each customer is visited exactly once, constraint (3) allows visits to charging stations, and constraint (4) ensures that every visited node except the depot is also exited.

Constraints (5)–(8) impose the time limit for each route. Constraint (5) initializes the arrival time at the departure depot, while constraints (6), (7), and (8) define the timing relationships after visiting a customer, visiting a charging station, or leaving the depot, respectively. These constraints also implicitly eliminate subtours, as they enforce a monotonic time flow that can only start and end at the depots.

Constraints (9) and (10) govern the vehicle load, where m_i denotes the load at node i. Finally, constraints (11) and (12) manage the vehicle's battery charge, updating it after visiting customers or, alternatively, after stopping at a charging station or leaving the depot.

As mentioned, the EVRP-RJ-RT, introduced in [5], extends the CEVRP to a more realistic setting by modeling road junctions. Moreover, road segments have a type associated, defined by a minimum and maximum allowable speed, reflecting real-world constraints such as speed limits and road quality.

The problem is defined on a directed graph $G = (V, E)$, where two depot nodes are used: 0_d for departures and 0_a for arrivals. The graph is assumed to be strongly connected, meaning a path exists between any two nodes. The node set V includes the customers $\mathcal{C}$, the charging stations $\mathcal{S}$, and a set $\mathcal{J}$ of road junctions. To allow repeated visits, both $\mathcal{S}$ and $\mathcal{J}$ contain multiple copies of their respective elements.

Each edge $(i, j) \in E$ is associated with a specific road type, which imposes a speed interval defined by lower and upper bounds v_{ij}^{lb} and v_{ij}^{ub}. This captures variability in travel conditions more accurately than in the previous setting, where speed was treated as a fixed constant. Because the graph is not complete, standard neighborhood notation is used: for any node $u \in V$, $N(u) = \{v \in V \mid (u, v) \in E\}$ denotes its outgoing neighbors.

$$\min \sum_{\substack{i \in V \\ j \in N(i)}} E_{ij} x_{ij} + K \sum_{j \in N(0_d)} x_{0_d j} \tag{1}$$

$$\text{s.t} \sum_{j \in N(i)} x_{ij} = 1 \qquad \forall\, i \in \mathcal{C} \tag{2}$$

$$\sum_{j \in N(i)} x_{ij} \leq 1 \qquad \forall\, i \in \mathcal{S} \cup \mathcal{J} \tag{3}$$

$$\sum_{i \in N(j)} x_{ij} - \sum_{i \in N(j)} x_{ji} = 0 \qquad \forall\, j \in V \setminus \{0_d, 0_a\} \tag{4}$$

$$f_{0_d} = 0 \tag{5}$$

$$f_i + (t_{ij} + s_i)x_{ij} - T_{\max}(1 - x_{ij}) \leq f_j \qquad \forall\, i \in \mathcal{C} \cup \mathcal{J},\, j \in N(i) \tag{6}$$

$$f_i + (t_{ij} + g(Q - y_i))x_{ij} - T_{\max}(1 - x_{ij}) \leq f_j \qquad \forall\, i \in \mathcal{S},\, j \in N(i) \tag{7}$$

$$t_{0_d j} \leq f_j \leq T_{\max} - (t_{j0_a} + s_j) \qquad \forall\, j \in V \setminus \{0_d, 0_a\} \tag{8}$$

$$0 \leq m_{0_d} \leq C \tag{9}$$

$$0 \leq m_j \leq m_i - q_i x_{ij} + C(1 - x_{ij}) \qquad \forall\, i \in V,\, j \in N(i) \tag{10}$$

$$0 \leq y_j \leq y_i - E_{ij}x_{ij} + Q(1 - x_{ij}) \qquad \forall\, i \in \mathcal{C} \cup \mathcal{J},\, j \in N(i) \tag{11}$$

$$0 \leq y_j \leq Q - E_{ij}x_{ij} \qquad \forall\, i \in \mathcal{S} \cup \{0_d\},\, j \in N(i) \tag{12}$$

$$v_{ij}^{\text{lb}} \leq v_{ij} \leq v_{ij}^{\text{ub}} \qquad \forall\, i \in V,\, j \in N(i) \tag{13}$$

$$x_{ij} \in \{0, 1\} \qquad \forall\, i \in V,\, j \in N(i) \tag{14}$$

The model follows the same structure as the previous one, with neighborhoods being used to account for the graph's non-completeness. Additionally, constraint (13) is added, which regulates the speed allowed at each edge.

For both problems, the learning-based and standard Clarke and Wright heuristic attempt to repair routes that become infeasible after merging. This is handled by the function $is\_valid()$ in Algorithm 5.2. The repair process tries to repair battery-infeasible routes by inserting charging stations when necessary. It first identifies the earliest point in the route where the remaining battery is insufficient to traverse the next edge. From this point, it moves backward through the route, examining each preceding edge to determine whether a charging station can be inserted between its endpoints to restore feasibility. Among all candidate stations, it selects the one that minimizes the additional energy cost while ensuring the vehicle can complete the remainder of the route. If such a station is found, it is inserted, and the route is re-checked for infeasibilities. This process repeats until the route becomes feasible or no valid insertion exists, in which case the merge is deemed invalid and discarded.

Moreover, for the EVRP-RJ-RT both algorithms set the speed on each road segment to its maximum allowed value. Once a solution is built, a greedy post-processing step reduces speeds to lower energy consumption while keeping route durations within the time limit. This step begins by computing the slack time available for each route. The edges in each route are then sorted in descending order of speed, and their speeds are iteratively reduced in fixed steps. If lowering the speed of an edge still keeps the route within the allowed time, the change is accepted, and the corresponding energy savings are retained. This continues until either no further feasible reductions are possible or all edges reach their minimum allowed speeds. Energy consumption is then recalculated using the updated speeds.

Although the charging station insertion and speed adjustment methods are relatively simple, they are deliberately designed to remain straightforward. The aim of this work is not to optimize every component of the solution process, but rather to assess the effectiveness of the learned savings mechanism within the Clarke and Wright heuristic. These routines serve as supportive mechanisms to ensure feasibility and realistic energy modeling, enabling a focused evaluation of the learning-based approach.

5.2 Implementing the Offline Learning Component

In this case the search component parametrized is the savings values, which are computed after every successful merge and determine the sorting order of the savings list. As with the two previous applications we employ a feed-forward neural network to produce these values, replacing the standard computation that looks at the potential improvements in terms of cost. This neural network will process a set of features regarding a pair of routes in the savings list, and output a value indicating the quality of performing that merge. Following the framework, a BRKGA using full-sized instances will be used for training. Remember that the general structure of a BRKGA is presented in Pseudocode 3.1 of chapter 3.

5.2.1 Neural Network and Features

As before, the neural network takes both specific features of the savings list entry and global features of the problem instance. This way, the model considers both the global characteristics of the instance and the specific ones of the savings list entry to be evaluated.

The features specific to the savings list entry are extracted from the pair of routes r_i, r_j it represents and the corresponding customers u_i and u_j which will be bridged in case of merging. They consist of the following:

1. The distance between the depot and u_i via route r_i.
2. The distance between u_i and u_j.
3. The distance between u_j and the depot via route r_j.
4. The load of route r_i.

5. The load of route r_j.

To incorporate global information about the problem instance, the following features are added:

1. The total number of road junctions.
2. The total number of customers.
3. Seven statistical descriptors from each of two vectors:
   (a) The vector of depot-to-customer distances.
   (b) The vector of customer-to-customer distances.

   These statistical descriptors consist of the average, sample standard deviation, minimum, maximum, and values at the first, second, and third quartiles.

This results in 21 features associated with each savings list entry: 5 specific ones, and 16 global ones giving information about the whole problem instance. As with the other problems, before being fed into the neural network, all features are normalized to have a mean of zero and a variance of one.

Note that the number of road junctions is used as a feature even though road junctions are only present for the EVRP-RJ-RT. In practice, our experimental evaluation adapts EVRP-RJ-RT instances to the CEVRP; hence, using the number of road junctions makes sense for both problems, giving a measure of instance size.

For this problem, we selected a simple neural network architecture consisting of one hidden layer with 10 nodes using a `sigmoid` activation function. This simple model was selected because the road junctions in the EVRP-RJ-RT already produce a high overhead, as it will be shown in the experimental evaluation section.

5.2.2 Evaluation of an Individual

As running the Clarke and Wright heuristic is relatively fast, the entire algorithm is run to compute both training and validation values. An individual is evaluated as it was for the last problem. First, the neural network is set with the individual's weights. Then, the Clarke and Wright heuristic, using the neural network to produce its savings values, is applied to each instance from the training set. The average quality of the obtained solutions, in this case the average number of vehicles required and energy consumption, is set to be the individual's fitness. The validation value is computed similarly, this time running the Clarke and Wright heuristic using the neural network with the best-so-far individual's weights on the validation instances.

5.3 Experimental Evaluation

Here we present a comparison of the learning Clarke and Wright heuristic, which uses a neural network to compute savings values, against the standard Clarke and Wright which computes savings values in terms of consumption. From now on, we denote them as CW-Learn and CW respectively.

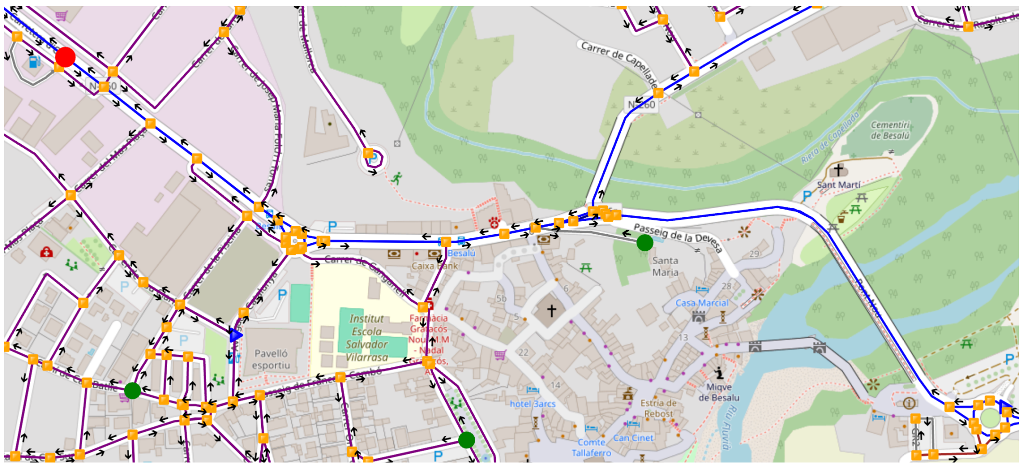
5.3.1 Problem Instances

The problem instances used in our experimental evaluation were generated with the `EVRPGen` instance generator [1], specifically designed to produce realistic EVRP-RJ-RT problem instances. This tool uses OpenStreetMap (OSM) road network data and allows generation based on user-defined parameters, such as the target area, the number of customers and charging stations, customer demand and service time distributions, and the maximum allowed route duration. Depot, customer, and charging station locations are selected from actual warehouses, shops, and charging or fuel stations in the chosen city.

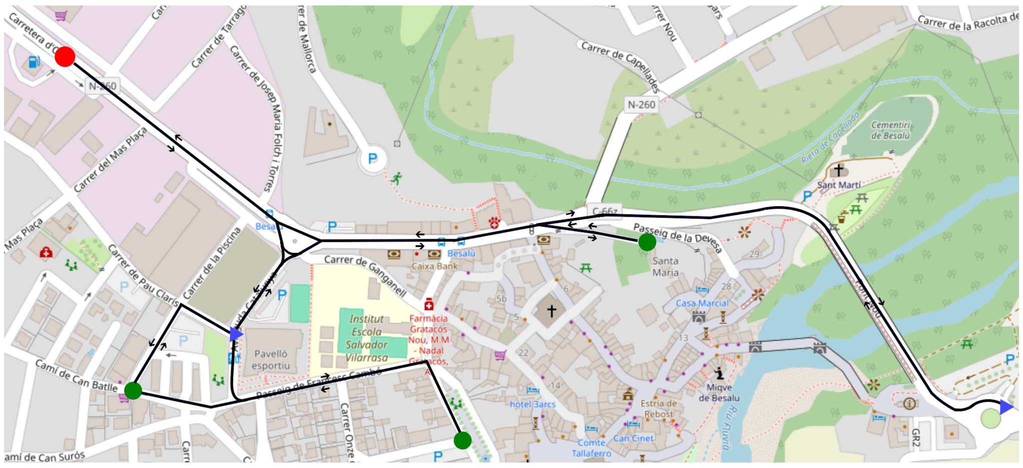
To adapt EVRP-RJ-RT instances for the CEVRP, which assumes a complete graph without road junctions, we replaced the generated graph with a complete one. The distance between any two nodes was defined as the length of the shortest path in the original graph, computed via Dijkstra's algorithm. Speeds were set to 90% of the average maximum allowable speeds in the corresponding EVRP-RJ-RT instance. This ensures each EVRP-RJ-RT instance can be seamlessly converted into a compatible CEVRP instance.

Figure 5.1 illustrates the process of adapting a problem instance. The red circle denotes the depot, yellow squares correspond to road junctions, green circles represent customers, and blue triangles indicate charging stations.

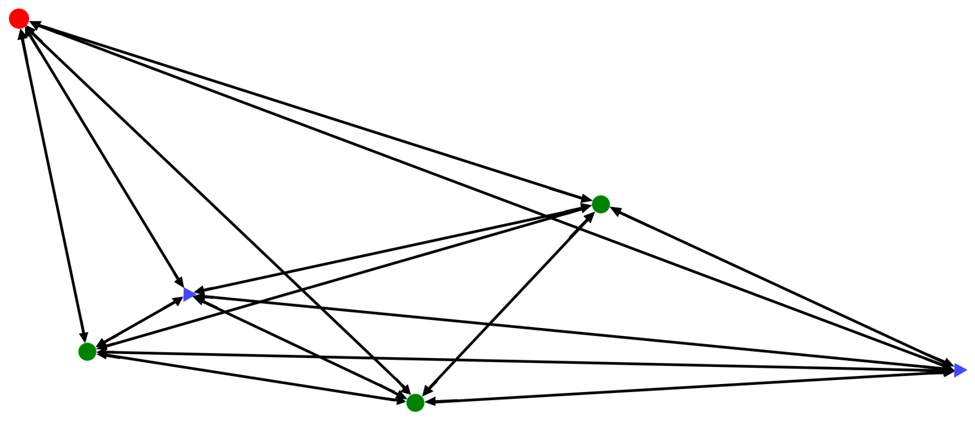
Figure 5.1a depicts the original EVRP-RJ-RT instance generated with `EVRPGen`. Its adapted version, shown in Figure 5.1b, corresponds to the CEVRP, where road junctions are removed and only the shortest paths are kept. Finally, Figure 5.1c presents the directed graph representation of the CEVRP instance. In this example, distances are symmetric, so a single bidirectional edge suffices for each pair of nodes. In the general case, however, distances may be direction-dependent, requiring two distinct edges per node pair, one for each direction.



(a) The original EVRP-RJ-RT problem instance viewed on the map.



(b) The corresponding CEVRP problem instance viewed on the map.



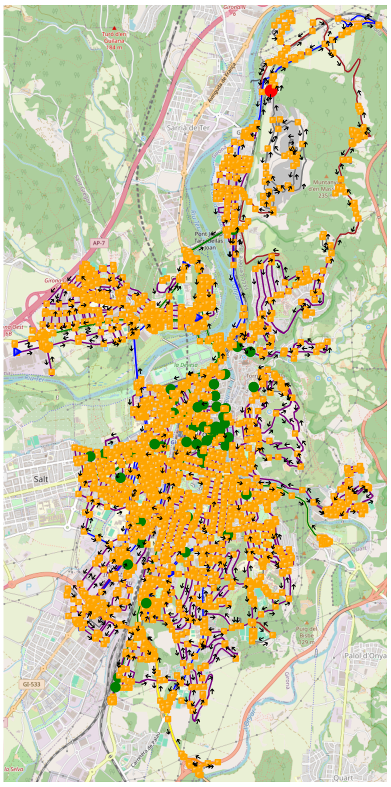
(c) The corresponding graph to the CEVRP problem instance.

Figure 5.1 Example of the transformation of an EVRP-RJ-RT problem instance to a CEVRP one.

We selected cities across Europe with populations of approximately 100,000, 500,000, and 2 million inhabitants for the benchmark set of problem instances. The cities in each group were:

- **100k**: Bolzano (Italy), Bruges (Belgium), Girona (Spain), Maastricht (Netherlands), Tarnów (Poland).
- **500k**: Bratislava (Slovakia), Lyon (France), Murcia (Spain), Nuremberg (Germany), Poznań (Poland).
- **2M**: Bucharest (Romania), Hamburg (Germany), Paris (France), Vienna (Austria), Warsaw (Poland).

For each city, instances with 50, 100, and 150 customers were generated, all with 10 charging stations. For every city–customer combination, 12 instances were created: two for training (one for training fitness computation and one for validation) and ten for testing. This resulted in 180 instances per size group. Training was performed separately for each group, with 15 training and 15 validation instances per category. The benchmark set is publicly available in [106].


(a) Complete instance view.

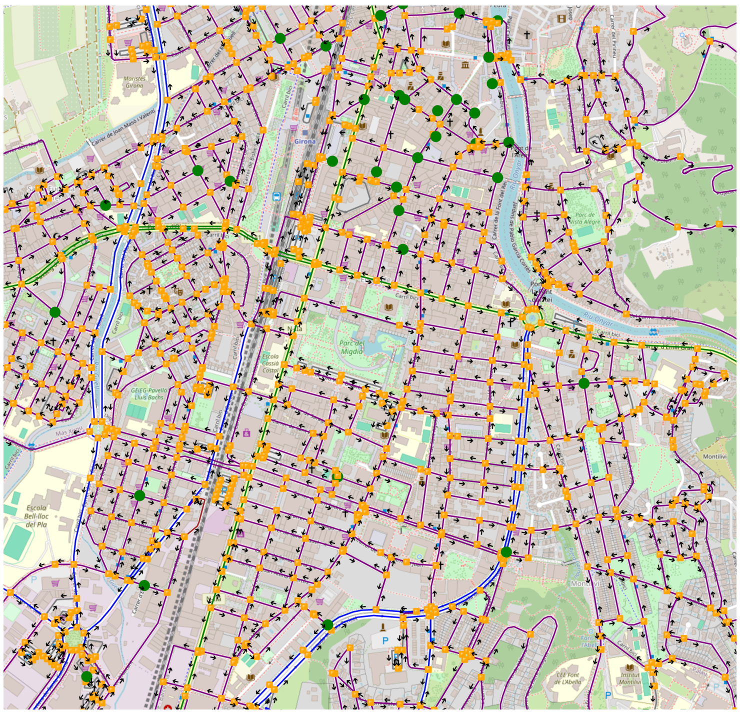
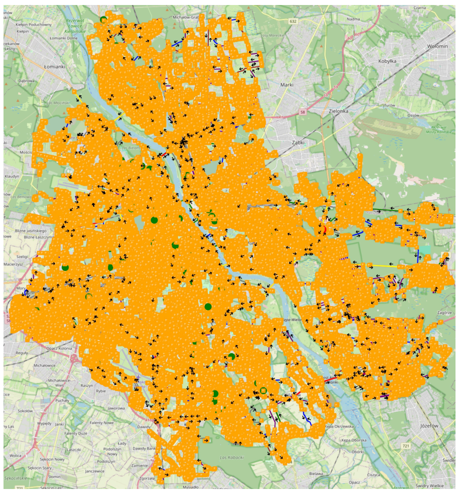

(b) Zoomed-in section.

Figure 5.2 Example of a problem instance based in Girona.


(a) Complete instance view.

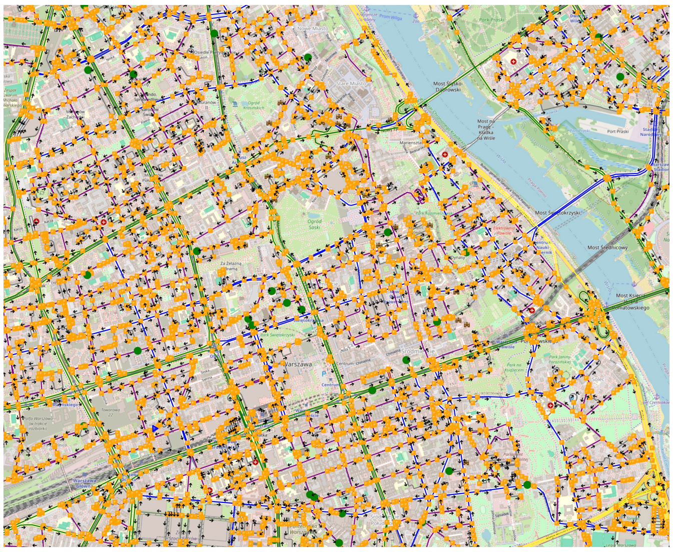

(b) Zoomed-in section.

Figure 5.3 Example of a problem instance based in Warsaw.

Figures 5.2 and 5.3 show two examples of generated problem instances: one based in Girona, one of the smallest cities considered, and another based in Warsaw, one of the largest. In these visualizations, road junctions are shown as yellow squares, the depot as a red circle, customers as green circles, and charging stations as blue triangles. Roads of different types are distinguished by different colors.

These visualizations help illustrate the high density of road junctions in the EVRP-RJ-RT, which contributes to its greater computational and memory demands. In contrast, the CEVRP stores only the shortest paths between customers, charging stations, and the depot, resulting in a far more compact graph.

Customer demands and service times were uniformly sampled from $(20, 100)$ kg and $(0.25, 0.5)$ h, respectively. Vehicle specifications were inspired by electric delivery vans used by companies like Amazon in the US: curb weight 3250 kg, load capacity 1000 kg, battery capacity 100 kWh, and charging rate 0.02 h/kWh. The maximum route duration was set to 8 hours, matching a standard workday.

As in [5], energy consumption was modeled following [9]. The power demand P (W) of a vehicle traveling a road segment is estimated by:

$$P = Mav + Mgv\sin(\theta) + 0.5C_dA_f\rho v^3 + MgC_r\cos(\theta)v \tag{5.3}$$

Here, M is the total vehicle weight (curb weight + load) in kg, v is average speed (m/s), a is acceleration (m/s^2), g is gravitational acceleration (approximated to 9.81 m/s^2), and θ is road slope in radians. The constants are frontal area A_f (m^2), air density ρ (kg/m^3), rolling resistance coefficient C_r, and drag coefficient C_d.

In our implementation, all variables are fixed except speed and load, which vary between edges in the EVRP-RJ-RT. In the CEVRP, speed is constant and only load varies. The energy consumption E_{ij} (J) from node i to j is approximated as:

$$E_{ij} \approx P\frac{d_{ij}}{v_{ij}} \approx \left(\alpha_{ij}(w + m_{ij}) + \beta v_{ij}^2\right)d_{ij} \tag{5.4}$$

where d_{ij} is distance, v_{ij} speed, m_{ij} load, w curb weight, $\alpha_{ij} = a + g\sin(\theta_{ij}) + gC_r\cos(\theta_{ij})$ is the road resistance factor, and $\beta = 0.5C_dA_f\rho$ captures aerodynamic and rolling effects. We set $\alpha_{ij} = 0.0981$ and $\beta = 2.11$ for all edges, based on realistic values for electric delivery vehicles [132].

5.3.2 Training

The Clarke and Wright heuristic does not require a time limit, since it terminates once no further merges are possible, either because all initial routes have been merged into a single route, or because no merge in the savings list can be applied. Moreover, the heuristic has no tunable parameters, so parameter tuning is unnecessary.

Regarding the training parameters, we used the same default BRKGA parameters as in the case of the BS for the RLCS problem, which consist of 20 individuals, one elite individual, seven mutants, and an elite inheritance probability ρ_e of 0.5.

As described earlier, CW-Learn was trained separately for each instance size, resulting in three independent training processes. Each training used one instance for every combination of city and customer count, yielding 15 training and 15 validation instances per size category.

All experiments were conducted in single-threaded mode on an Intel Xeon E5-2640 processor running at 2.40 GHz with 16 GB of RAM. The training process itself was parallelized using OpenMP. For the small and medium city trainings, we used 15 threads, while for the large cities only two threads were employed. This limitation for large cities was due to the high memory requirements, loading 15 large instances into RAM simultaneously was not feasible.

Training durations for the CEVRP were approximately 1, 2, and 5 hours for small, medium, and large cities, respectively. In contrast, EVRP-RJ-RT required significantly longer training: roughly 6, 2, and 5 days for the same categories. For applying the offline framework to the Clarke and Wright heuristic, training was performed with a fixed time limit, and the weights yielding the best validation value were the ones retained. Recall that this was done due to the high oscillation in validation values observed in this application, which made using early stopping unreliable. A total time limit of 7 days was used for both problems, so the reported training times correspond to when the best-validation weights were found. The substantial difference between the problems is primarily due to the additional computational overhead introduced by road junctions, as will be discussed later when presenting the results.

5.3.3 Results

Tables 5.1–5.6 present the numerical results obtained in the experimental evaluation. The first three correspond to the EVRP-RJ-RT, while the remaining three are for the CEVRP. Each line in these tables represents one combination of city and number of customers, and reports the average results over the ten instances associated with that configuration. The reported values include the number of electric vehicles required, the total energy consumption of the solution in Joules, and the computation time needed to obtain it in seconds. The best average value for each row is shown in bold. Remember that solution quality is primarily judged according to the number of vehicles, with total energy consumption serving as a tie-breaker when two solutions require the same fleet size.

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bolzano	50	**3.50**	**9.51**	0.26	3.60	8.06	0.48
	100	6.60	15.98	1.92	**6.50**	**13.04**	3.95
	150	**9.30**	**20.69**	5.50	9.60	17.15	11.39
Bruges	50	3.50	24.84	0.49	**3.50**	**19.04**	1.02
	100	**6.30**	**40.72**	2.64	6.50	33.90	6.80
	150	9.80	44.57	6.93	**9.70**	**36.90**	20.34
Girona	50	**3.50**	**6.84**	0.18	3.60	6.21	0.46
	100	**6.50**	**13.43**	1.30	6.60	11.49	3.63
	150	**9.60**	**18.16**	4.19	9.70	15.88	12.47
Maastricht	50	**3.60**	**11.47**	0.25	3.80	9.37	0.56
	100	6.70	19.20	1.38	**6.60**	**15.44**	3.30
	150	**9.80**	**23.46**	4.76	9.90	19.55	11.44
Tarnów	50	**3.50**	**10.64**	0.25	3.70	8.84	0.53
	100	**6.50**	**18.81**	1.57	6.60	15.02	4.27
	150	**9.50**	**24.24**	5.22	9.60	20.74	14.74

Table 5.1 Comparison of CW-Learn and CW on the EVRP-RJ-RT small instances.

When looking at the results as a whole, CW-Learn tends to produce better solutions than its standard counterpart. This advantage is visible in most instance groups, where CW-Learn achieves the best average number of vehicles in the

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bratislava	50	3.60	42.23	3.84	**3.60**	**33.31**	4.85
	100	6.70	72.02	9.60	**6.70**	**59.54**	28.39
	150	**9.80**	**84.70**	24.03	9.90	69.57	95.81
Lyon	50	**3.70**	**14.71**	0.61	3.90	10.48	1.88
	100	6.60	23.39	3.42	**6.50**	**18.99**	12.93
	150	**9.70**	**29.89**	9.94	9.80	24.10	37.78
Murcia	50	**3.80**	**27.26**	1.37	3.90	24.95	3.98
	100	6.80	61.95	5.67	**6.80**	**54.85**	23.05
	150	**9.60**	**77.80**	15.56	9.90	68.97	70.40
Nuremberg	50	3.80	30.22	2.51	**3.70**	**25.74**	3.38
	100	**6.50**	**54.26**	6.51	6.60	42.84	19.83
	150	**9.50**	**57.81**	17.24	9.70	48.50	60.61
Poznán	50	3.90	44.02	3.09	**3.80**	**30.44**	4.09
	100	**6.50**	**52.18**	8.04	6.70	38.83	22.06
	150	**9.50**	**65.05**	18.75	9.60	58.47	65.17

Table 5.2 Comparison of CW-Learn and CW on the EVRP-RJ-RT medium instances.

majority of cases. The main exception is the group of large EVRP-RJ-RT instances, where CW obtains better results in eight of the fifteen configurations.

We test the statistical significance of the results obtained using the signed-rank Wilcoxon test [128]. With this we only test regarding the main objective function value, which is the number of vehicles. This means that solutions requiring the same amount of vehicles are deemed as equivalent, independent of the energy consumption. This produces p-values of less than 0.01 in favour of CW-Learn for all tested groups, which consist of one group per problem and instance size. These results confirm that the learning-based approach significantly reduces the number of vehicles compared to the baseline. Importantly, note that CW-Learn was explicitly trained to minimize fleet size, and we observe that it very rarely produces solutions requiring more vehicles than CW. This occurred in fewer than 5 out of the 150 instances in each instance group.

An interesting trend emerges when examining total energy consumption: CW achieves lower average consumption in every single case. This is unsurprising when CW-Learn uses fewer vehicles, as these solutions may require longer or more complex routes that naturally lead to higher energy use. However, the fact that CW still consumes less energy even when the number of vehicles in the CW-Learn solutions is the same or bigger than in the CW ones is more intriguing. This can be explained by the design of the algorithms. CW is explicitly constructed to minimize energy consumption: at each step, it merges the two routes that yield the greatest energy saving. CW-Learn, in contrast, is trained

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bucharest	50	**3.30**	**39.78**	3.50	3.70	29.26	8.02
	100	6.80	59.81	9.84	**6.80**	**47.42**	33.24
	150	9.70	72.79	23.47	**9.60**	**57.96**	100.57
Hamburg	50	3.80	58.91	4.81	**3.80**	**46.93**	23.52
	100	**6.70**	**97.48**	12.85	6.90	79.73	55.88
	150	9.90	113.55	33.40	**9.90**	**95.38**	149.47
Paris	50	**3.50**	**20.13**	0.74	3.70	15.82	2.48
	100	6.20	33.40	3.73	**6.20**	**26.66**	14.48
	150	9.60	42.24	11.02	**9.60**	**34.13**	46.43
Vienna	50	3.90	33.02	1.58	**3.70**	**27.49**	6.72
	100	6.60	53.90	5.84	**6.60**	**44.70**	25.28
	150	**9.50**	**71.98**	15.97	9.70	61.83	75.90
Warsaw	50	**3.50**	**74.12**	4.59	3.80	50.72	18.44
	100	**6.50**	**98.76**	13.15	6.60	76.10	44.39
	150	**9.70**	**129.09**	24.69	9.90	104.65	103.32

Table 5.3 Comparison of CW-Learn and CW on the EVRP-RJ-RT large instances.

to minimize the number of vehicles first and only considers energy consumption when breaking ties. This alignment between the CW-Learn's training and the objective function explains its superior performance in reducing the fleet size, but also why it consistently performs worse in terms of energy. It would be an interesting extension to retrain CW-Learn with energy consumption as the sole optimization objective, so that the two algorithms could be compared in the exact context for which CW was originally designed.

To complement the statistical significance analysis we apply a binomial signed test [40] based on the lexicographic objective, where the number of vehicles is the primary criterion and energy consumption is used as a tie-breaker. In this evaluation the baseline achieves significantly more wins, because although the learning method sometimes reduces fleet size, more frequently both approaches require the same number of vehicles in which cases the baseline consistently consumes less energy as noted above. The binomial test produces p-values of less than 0.01, this time in favour of the standard Clarke and Wright. This shows that the learning variant is effective and important in achieving the primary goal of reducing vehicles, but that additional modelling of secondary objectives may be needed to also capture energy efficiency.

Regarding computation time, results are also interesting. Firstly, much more computation time is required for obtaining solutions to the EVRP-RJ-RT than to the CEVRP. In some of the largest cases, such as CW on Hamburg with 150 customers (row 6 of Table 5.3), average runtimes exceed 150 seconds, while for the CEVRP the time is always below 0.5 seconds. As hinted before, this

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bolzano	50	3.60	11.92	0.05	**3.60**	**10.07**	0.01
	100	**6.50**	**19.61**	0.44	6.60	16.39	0.10
	150	**9.30**	**24.96**	1.51	9.40	21.41	0.37
Bruges	50	3.50	39.39	0.05	**3.50**	**34.75**	0.01
	100	**6.40**	**62.93**	0.43	6.50	56.41	0.11
	150	**9.60**	**78.25**	1.55	9.80	70.23	0.39
Girona	50	3.50	10.09	0.05	**3.50**	**8.57**	0.01
	100	6.60	18.69	0.40	**6.50**	**15.99**	0.10
	150	9.60	25.13	1.50	**9.60**	**21.69**	0.37
Maastricht	50	3.80	22.53	0.05	**3.70**	**18.33**	0.01
	100	**6.60**	**35.02**	0.42	6.70	29.52	0.11
	150	**9.70**	**46.19**	1.53	10.00	39.67	0.39
Tarnów	50	**3.60**	**13.51**	0.05	3.70	11.54	0.01
	100	**6.40**	**23.31**	0.42	6.60	19.76	0.11
	150	**9.50**	**31.08**	1.47	9.60	26.92	0.38

Table 5.4 Comparison of CW-Learn and CW on the CEVRP small instances.

disparity stems from the large number of road junctions in the EVRP-RJ-RT, which significantly increase graph size and route complexity.

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bratislava	50	**3.50**	**84.88**	0.05	3.70	73.19	0.01
	100	6.60	151.59	0.45	**6.60**	**130.53**	0.08
	150	9.80	196.27	1.58	**9.80**	**168.19**	0.30
Lyon	50	**3.70**	**16.15**	0.05	3.80	13.67	0.01
	100	6.50	28.55	0.45	**6.50**	**24.44**	0.09
	150	**9.70**	**36.02**	1.58	9.90	30.38	0.29
Murcia	50	**3.80**	**32.75**	0.05	3.90	28.23	0.01
	100	**6.70**	**64.84**	0.45	6.80	56.84	0.08
	150	**9.70**	**85.79**	1.50	9.80	77.72	0.30
Nuremberg	50	3.80	55.37	0.05	**3.70**	**43.12**	0.01
	100	**6.50**	**81.06**	0.43	6.60	69.44	0.08
	150	**9.50**	**106.41**	1.55	9.70	92.49	0.31
Poznán	50	**3.70**	**60.18**	0.05	3.80	48.07	0.01
	100	**6.50**	**86.41**	0.43	6.80	73.73	0.08
	150	**9.60**	**115.29**	1.54	9.70	98.88	0.31

Table 5.5 Comparison of CW-Learn and CW on the CEVRP medium instances.

Other than the difference in required computation time between problems, another notable observation regards the difference in computation time between algorithms. For the EVRP-RJ-RT, CW-Learn is consistently faster than CW, often requiring only one-half or one-third of the computation time. For the CEVRP, however, the situation is reversed, and CW-Learn can be up to five times slower. The reason behind this lies in how road junctions affect both algorithms.

		CW-Learn			CW		
City	Customers	Evs	Consumption	Time	Evs	Consumption	Time
Bucharest	50	**3.30**	**44.34**	0.05	3.50	37.66	0.01
	100	**6.80**	**74.40**	0.46	6.90	63.73	0.08
	150	**9.60**	**95.73**	1.62	9.70	83.82	0.31
Hamburg	50	**3.50**	**69.96**	0.05	3.80	61.37	0.01
	100	**6.40**	**112.67**	0.45	6.70	100.71	0.09
	150	**9.90**	**149.15**	1.60	10.00	130.15	0.33
Paris	50	**3.50**	**23.01**	0.05	3.70	19.10	0.01
	100	**6.20**	**36.07**	0.42	6.30	31.38	0.08
	150	9.60	49.17	1.49	**9.60**	**42.07**	0.30
Vienna	50	**3.80**	**44.07**	0.05	3.90	38.91	0.01
	100	**6.40**	**75.02**	0.42	6.80	63.67	0.09
	150	**9.50**	**101.07**	1.56	9.70	87.11	0.31
Warsaw	50	**3.70**	**129.79**	0.05	3.90	113.66	0.01
	100	**6.40**	**220.60**	0.42	6.60	185.28	0.09
	150	**9.60**	**291.16**	1.45	10.00	253.51	0.30

Table 5.6 Comparison of CW-Learn and CW on the CEVRP large instances.

In CW, savings are computed as the reduction in total energy consumption, which requires traversing the routes corresponding to each savings list entry. In EVRP-RJ-RT, these routes contain many road junctions, meaning that each computation involves summing over a large number of edges, which creates significant overhead. CW-Learn instead calculates savings by extracting features and performing a forward pass through its neural network. Only the distance features are affected by road junctions, and since these distances are precomputed, the cost of computing savings scales much better. In CEVRP, where there are no road junctions, CW's energy-saving calculations are much faster, while CW-Learn still incurs the cost of the neural network's forward passes, leading to its relative slowdown.

Figures 5.4 and 5.5 provide a visual summary of these patterns. The first shows, for each city and customer count, the average percentage reduction in the number of vehicles achieved by one algorithm compared to the other. Positive values indicate that CW-Learn requires fewer vehicles, while negative values indicate the opposite. The second figure presents the same type of comparison but for computation time. Together, these figures confirm what is seen in the tables: CW-Learn generally reduces the number of vehicles, particularly in the CEVRP instances, while computation time improvements are strongly problem-dependent, with CW-Learn being much faster in the EVRP-RJ-RT but slower in the CEVRP.

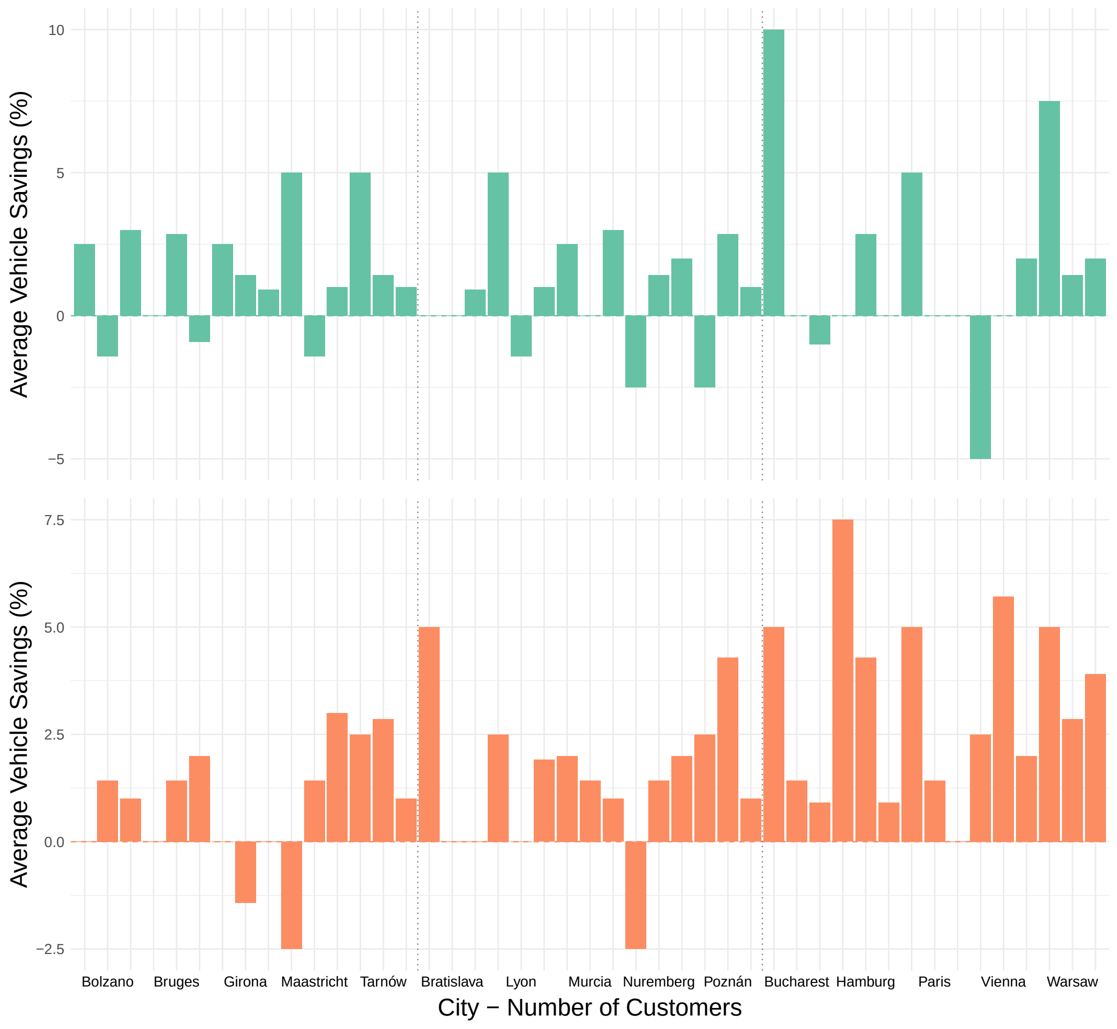


Figure 5.4 Average percentage reduction in the number of vehicles required by one algorithm compared to the other for the EVRP-RJ-RT (top) and the CEVRP (bottom). Each cluster of three bars represents a city, with bars corresponding to customer counts of 50, 100, and 150. Positive values indicate that the learning variant outperforms the standard, while negative values indicate the standard variant performs better than the learning approach.

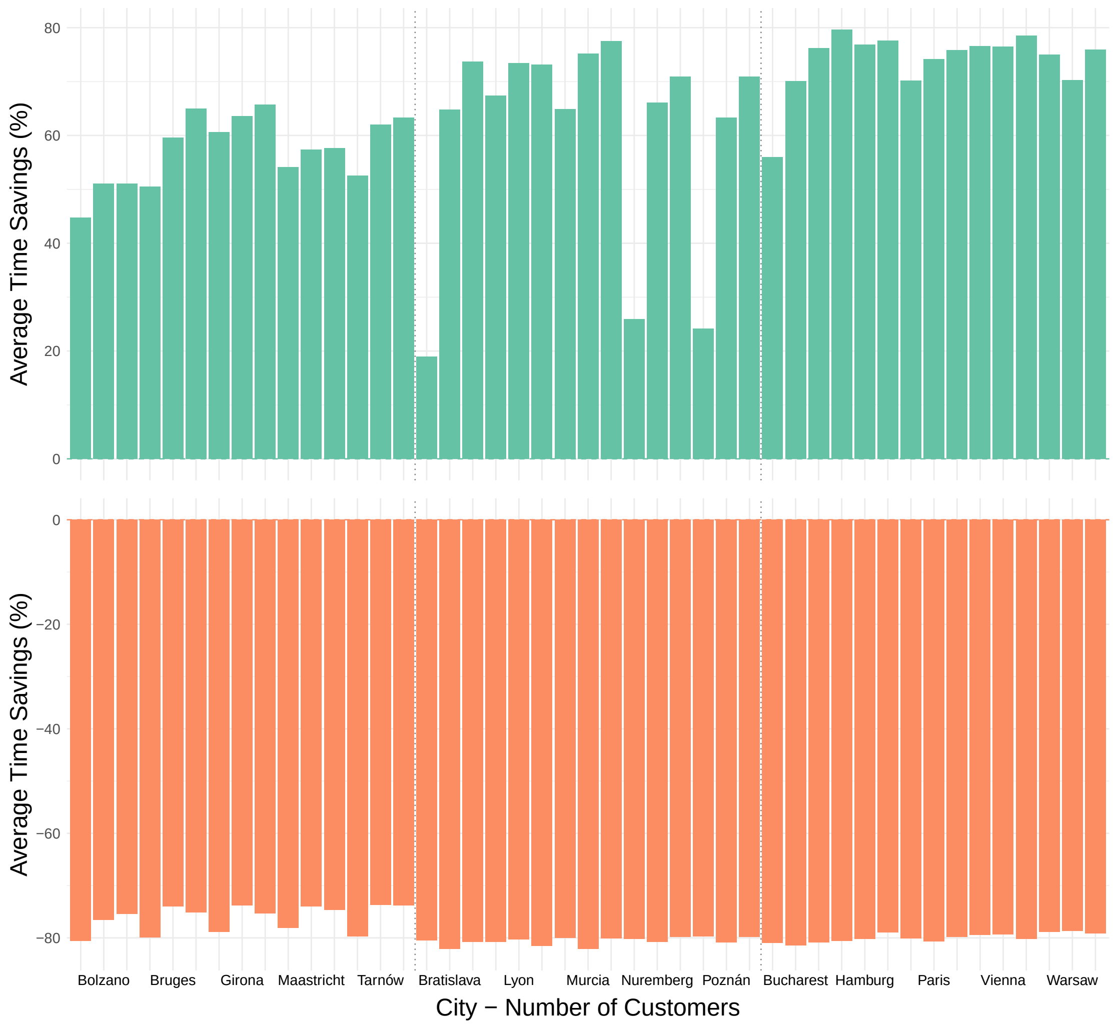


Figure 5.5 Average percentage reduction in execution time by one algorithm compared to the other for the EVRP-RJ-RT (top) and the CEVRP (bottom). Each cluster of three bars represents a city, with bars corresponding to customer counts of 50, 100, and 150. Positive values indicate that the learning variant requires less time than the standard, while negative values indicate the standard variant requires less time than the learning approach.

Part II

Online Learning

Chapter 6

CMSA

6.1 Introduction

Construct, Merge, Solve, and Adapt (CMSA) is a hybrid metaheuristic introduced rather recently [13, 17, 21]. The main idea of the algorithm is to iteratively apply an exact solver restricted to an evolving sub-instance C' of the original optimization problem, with the goal that the exact solver eventually finds a high-quality solution in it. Each iteration begins with the *construct* step, where multiple feasible solutions are probabilistically generated for the full problem instance. Next, in the *merge* step, solution components present in these generated solutions are incorporated into C'. An exact solver is then applied in the *solve* step restricted to the current sub-instance. Finally, in the *adapt* step, outdated solution components are systematically removed from C' based on an aging mechanism. CMSA is broadly applicable to optimization problems that allow for (1) the probabilistic generation of feasible solutions and (2) the use of an exact solver that can be applied restricted to sub-instances.

Since its introduction in 2016, CMSA has been effectively applied to various combinatorial optimization problems. Some of its most recent applications include the variable-sized bin packing problem [4], the electric vehicle routing problem with simultaneous pickup and deliveries [3], as well as a bus driver scheduling problem with complex break constraints [110]. Additionally, CMSA has been utilized for test data generation in software product lines [48] and for scheduling the maintenance of nuclear power plants [43]. Furthermore, a variant designed to mitigate the parameter sensitivity observed in certain applications denoted Adapt-CMSA was recently introduced [2].

The research presented in this second part of the thesis introduces and explores two approaches that integrate Machine Learning (ML) to enhance CMSA. Both methods incorporate learning into the *construct* step of CMSA, utilizing the solutions generated by the exact solver as feedback to refine the construction of new solutions.

The first approach, called RL-CMSA, draws inspiration from the multi-armed bandit problem in Reinforcement Learning (RL). It maintains a quality score for each solution component, guiding solution construction based on these values. At the end of each iteration, these scores are updated by identifying which components are part of the solution generated by the exact solver.

The second approach, called DL-CMSA, employs a more advanced learning mechanism by integrating Deep Learning (DL). It considers the partial solution under construction when selecting the next solution component during the *construct* step. At the end of each iteration the parameters of the neural network are updated to make the selections performed by the exact solver more likely in future iterations.

The next section offers a comprehensive explanation of the standard CMSA, while the next two chapters introduce RL-CMSA and DL-CMSA. In addition, various designs for the learning mechanisms of RL-CMSA are explored. Following this, in Chapter 9, the three CMSA variants are experimentally assessed on three well-known combinatorial optimization problems: the Minimum Dominating Set (MDS) problem, the Far From Most String (FFMS) problem, and the Maximum Independent Set (MIS) problem. Finally, the concluding chapter presents a summary and suggestions for future work related to this research.

6.2 The Standard CMSA

To apply CMSA to a combinatorial optimization problem, the first step is to define a set C of solution components such that every valid solution to the problem can be represented as a subset of the complete set of solution components C. For instance, in the case of the classic Traveling Salesman Problem (TSP), the edge set of the complete input graph is a possible definition of C. This is because any valid tour can be expressed as a subset of edges.

In the following explanation of CMSA, we assume a general set of solution components $C = \{c_1, c_2, \ldots, c_n\}$, the existence of a method for probabilistically generating solutions to the given problem, and the availability of a solver that can produce solutions restricted to a subset of solution components. While this solver does not necessarily have to be exact and could, for instance, be another metaheuristic, we assume it to be an exact solver, as is typically the case in CMSA applications.

Algorithm 6.1 outlines the structure of the standard CMSA. Initially, the sub-instance C' is set to empty, and the *best-so-far* solution S_{bsf} is initialized as NULL. Subsequently, the main loop of the algorithm begins, during which the

Algorithm 6.1 Pseudo-code of the standard CMSA

```
Input: Set $C$ of solution components for the problem instance to be solved
Input: Values for parameters $n_a$, $age_{\max}$ and $t_{\text{ILP}}$
Output: A solution to the problem at hand
1:  $S_{\text{bsf}} = \text{null}, C' = \emptyset$
2:  while termination conditions not met do
3:      for $j = 1, \ldots, n_a$ do
4:          $S :=$ probabilistic_solution_construction()
5:          for all $c_i \in S$ and $c_i \notin C'$ do
6:              $\text{age}_{c_i} = 0$
7:              $C' := C' \cup \{c_i\}$
8:          end for
9:      end for
10:     $S_{\text{opt}} :=$ apply_exact_solver$(C', t_{\text{ILP}})$
11:     if $S_{\text{opt}}$ is better than $S_{\text{bsf}}$ then $S_{\text{bsf}} := S_{\text{opt}}$ end if
12:     adapt$(C', S_{\text{opt}}, age_{\max})$
13: end while
14: return $S_{\text{bsf}}$
```

construct, *merge*, *solve*, and *adapt* steps are executed in sequence until a specified time limit is reached. The steps can be described as follows:

1. In the *construct* step, n_a solutions to the problem are generated probabilistically.

2. The *merge* step involves adding to C' the solution components c_i that appear in at least one of the n_a constructed solutions and are not already present in C'. Additionally, the ages of these components are set to 0.

3. The *solve* step employs an exact solver with a time limit t_{ILP} restricted to the problem instance C', obtaining a solution S_{opt} for the given problem.

4. Finally, the *adapt* step involves increasing the age of solution components in $C' \setminus S_{\text{opt}}$ by one, resetting the age of the components in S_{opt} to 0, and removing from C' those solution components whose age reaches the maximum allowed age, $age_{\max}$.

Here, n_a, t_{ILP}, and $age_{\max}$ are algorithm parameters that define the number of solutions generated in the *construct* step, the time allocated to the exact solver in the *solve* step, and the maximum permissible age for solution components in the sub-instance, respectively.

The *construct* and *solve* steps are the only problem-specific components of the algorithm. Typically, the *construct* step employs a probabilistic solution generation mechanism combined with a greedy function adapted to the specific problem, while the *solve* step relies on an exact method designed for solving the given problem.

Chapter 7

RL-CMSA

This chapter introduces the RL-CMSA variant and is based on publications [102, 103]. The first presents an initial implementation of the algorithm; the second, which is a journal extension, explores additional algorithmic component designs and tackles a broader set of problems.

RL-CMSA maintains a quality measure q_i for each solution component $c_i \in C$, referred to as the q-values. The complete vector of these values is denoted as $\mathbf{q}$. The probabilistic construction of solutions in RL-CMSA is guided by these q-values, where components with higher values have a greater probability of being selected. Furthermore, after each iteration of CMSA, once the exact solver has been applied, the q-values are updated. Specifically, the values associated with solution components in C' that appear in the solution S_{opt} returned by the exact solver are increased, whereas the values corresponding to components in C' that do not belong to S_{opt} are decreased.

Algorithm 7.1 outlines the general framework of RL-CMSA. Initially, the stored sub-instance C' is set to empty, the *best-so-far* solution S_{bsf} is initialized to NULL, and all q-values are set to zero. Within the main loop, the algorithm follows the standard four CMSA steps along with an additional fifth step, referred to as the *learn* step. The four standard CMSA steps remain unchanged, except for the *construct* step.

Similar to traditional CMSA, the *construct* step in RL-CMSA involves the probabilistic generation of n_a solutions. However, in RL-CMSA, this process is influenced by the q-values: solution components with higher q-values are more likely to be selected. The newly introduced *learn* step updates the q-values and computes a convergence measure, which may trigger a restart of the learning process if necessary. Such a restart involves (1) resetting all q-values to zero and (2) emptying the sub-instance C' based on a predefined parameter, as explained below.

Various designs were explored for the new solution construction mechanism and the update of the q-values in our work [103]. These designs were partially

Algorithm 7.1 Pseudo-code of RL-CMSA

```
Input: Set $C$ of solution components for the problem instance to be solved
Input: Values for parameters $n_a$, $\text{age}_{\max}$, $t_{\text{ILP}}$, $cf_{\text{limit}}$ and $b_{\text{reset}}$
Output: A solution to the problem at hand
 1: $S_{\text{bsf}} = \text{null}, C' = \emptyset$
 2: $q_i = 0$ for $i = 1, \ldots, n$
 3: while termination conditions not met do
 4:    for $j = 1, \ldots, n_a$ do
 5:       $S :=$ probabilistic_solution_construction($\mathbf{q}$)
 6:       for all $c_i \in S$ and $c_i \notin C'$ do
 7:          $\text{age}_{c_i} = 0$
 8:          $C' := C' \cup \{c_i\}$
 9:       end for
10:    end for
11:    $S_{\text{opt}} :=$ apply_exact_solver($C', t_{\text{ILP}}$)
12:    if $S_{\text{opt}}$ is better than $S_{\text{bsf}}$ then $S_{\text{bsf}} := S_{\text{opt}}$ end if
13:    adapt($C', S_{\text{opt}}, age_{\max}$)
14:    update_q_values($\mathbf{q}, C', S_{\text{opt}}$)
15:    $cf =$ compute_convergence_factor($\mathbf{q}$)
16:    if $cf > cf_{\text{limit}}$ then
17:       $q_i = 0$ for $i = 1, \ldots, n$
18:       if $b_{\text{reset}} = \textit{true}$ then $C' = \emptyset$ end if
19:    end if
20: end while
21: return $S_{\text{bsf}}$
```

inspired by prior research on the multi-armed bandit problem, a classic problem in Reinforcement Learning (RL) [80]. The concept of multi-armed bandits was first introduced by Robbins in 1952 [109].

In its simplest form, a multi-armed bandit problem consists of a set of k probability distributions $\{D_1, \ldots, D_k\}$. The objective is to devise a sampling strategy for an agent that does not have prior knowledge of these distributions. The agent iteratively samples from these distributions, receiving a reward at each iteration. From a technical perspective, the goal is to design a sampling strategy that maximizes the total accumulated reward. The problem is often framed metaphorically, where the distributions correspond to k arms of a slot machine, and the agent represents a gambler aiming to maximize their earnings.

As observed, the RL strategy incorporated into CMSA in this work leads to a scenario that closely resembles a multi-armed bandit problem. In RL-CMSA, the sampling process corresponds to selecting one of the available solution components. However, a crucial distinction exists: instead of assigning rewards after each individual sample or after every solution construction, RL-CMSA assigns rewards only after the *solve* step. These rewards are determined based on the outcome obtained by the exact solver when solving the current sub-instance.

7.1 Update of the q-values

In the *learn* step, consisting of lines 14 to 19 of Algorithm 7.1, the q-values associated with solution components in C' are updated based on whether they are part of the solution S_{opt} obtained by the exact solver during the *solve* step. Notably, this approach differs from standard CMSA, where the quality of a solution component is typically evaluated using a myopic measure that considers only its immediate benefit when added to a partial solution under construction. Moreover, the quality is not directly linked to the objective function value of the final solution in which the component was included. Instead, the quality of a component is assessed in comparison to all other solution components in the sub-instance C'. Specifically, the q-value q_i of a solution component c_i is increased if it is part of S_{opt} and decreased otherwise. This adjustment is made by assigning a reward $R > 0$ in the former case and $-R$ in the latter. We consider the following three strategies for updating the q-values:

1. The initial approach involves accumulating the rewards over time. In each iteration, once the reward $r_i \in \{R, -R\}$ for a solution component $c_i \in C'$ is determined, its q-value is updated in the following manner:

$$q_i := q_i + r_i \tag{7.1}$$

2. The second approach relies on averaging the rewards received over time. To facilitate this, a variable n_i keeps track of how many times the q-value q_i of a solution component c_i has been updated since the algorithm began. In each iteration, after determining the reward $r_i \in \{R, -R\}$ for a solution component $c_i \in C'$, its q-value is updated as follows:

$$q_i := q_i + \frac{1}{n_i}(r_i - q_i). \tag{7.2}$$

3. The final design extends the previous one by substituting $1/n_i$ with a constant step-size parameter $\alpha > 0$. Consequently, the update of the q-value q_i for a solution component $c_i \in C'$ is given by the following expression:

$$q_i := q_i + \alpha(r_i - q_i). \tag{7.3}$$

The first design option may lead to a bias toward solution components that are selected more frequently. For instance, with this approach, a solution component that receives a reward R would accumulate the same q-value as one that was awarded rewards R, $-R$, and R. In contrast, the second design avoids this issue by setting the q-values to the average of the rewards, ensuring that the number of times a solution component is selected does not introduce bias. The second and third designs are widely used strategies for updating q-values in multi-armed bandit problems [122]. It is worth noting that, due to the constant step-size α, the third design places more emphasis on recent rewards than on older ones. This can be advantageous in the RL-CMSA context, where rewards might shift over time.

7.2 Solution Construction in RL-CMSA

The updated *construct* step utilizes the q-values for probabilistic solution generation. This updated construction process, corresponding to the function probabilistic_solution_construction($\mathbf{q}$) of Algorithm 7.1, begins with an empty solution $S = \emptyset$. Then, at each step, one of the solution components that can feasibly extend the current partial solution is added to it until the solution is complete. We denote by $C_{\text{feas}} \subseteq C \setminus S$ the set of feasible solution components with respect to the partial solution S. Additionally, recall that q_i represents the q-value associated with solution component $c_i \in C$. We propose two distinct approaches for selecting a solution component from C_{feas}.

7.2.1 Softmax Selection

The first proposed design introduces a real parameter $dr \in [0, 1]$, referred to as the determinism rate. At each step of the construction process, a solution component is selected as follows:

1. With probability dr, a solution component is randomly selected from those in C_{feas} that have the highest q-value.
2. Otherwise, with probability $1 - dr$, the selection is made using a roulette wheel approach, where the probability p_i of selecting solution component $c_i \in C_{\text{feas}}$ is given by

$$p_i = \frac{e^{\beta q_i}}{\sum_{c_k \in C_{\text{feas}}} e^{\beta q_k}} \quad . \tag{7.4}$$

Here, $\beta \geq 0$ is a parameter that, in conjunction with dr, controls the balance between exploration and exploitation.

It is important to note that this first selection design may lead to the algorithm's convergence. Specifically, the q-values of certain solution components may become significantly larger than those of others, resulting in the same solution being repeatedly constructed and further increasing their q-values. To address this potential issue, we propose measuring the level of convergence as described below. This measurement is carried out once per iteration in the *learn* step, after the q-values have been updated. If high convergence is detected, the algorithm is reset by re-initializing the q-values to zero and emptying C' according to a specified parameter. See lines 15 to 19 of Algorithm 7.1. This mechanism relies on a convergence factor and a convergence factor limit. When the convergence factor exceeds the convergence factor limit, the algorithm is re-initialized.

The following describes how the convergence factor is calculated, which is done in method compute_convergence_factor($\mathbf{q}$) of Algorithm 7.1. For each solution component c_i in the most recently constructed solution S, the probability z_i of selecting c_i over all other solution components that are not part of S is computed. The convergence factor is then defined as the minimum of these probabilities for all $c_i \in S$. It is important to note that, according to this definition, the closer the convergence factor is to 1, the closer the algorithm is to converging. In this context, it is worth mentioning that the probability of selecting a particular solution component depends on the values of the parameters dr and β. Specifically, the probability z_i of selecting solution component c_i is given by:

$$z_i := dr \cdot \chi_i + (1 - dr) \cdot \frac{e^{\beta q_i}}{\sum_{c_k \notin S} e^{\beta q_k} + e^{\beta q_i}} \quad , \tag{7.5}$$

where χ_i is defined as:

$$\chi_i := \begin{cases} \frac{1}{|\{c_k \in C \setminus S \cup \{c_i\} \mid q_k = q_i\}|} & \text{if } q_i = \max\{q_k \mid c_k \in C \setminus S \cup \{c_i\}\} \\ \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

The expression for z_i is derived from the selection process in the *construct* step. With probability dr, a solution component is selected uniformly at random from the solution components that have the highest q-value. With the complementary probability $1 - dr$, the selection is made in a roulette-wheel-based manner, where the probabilities are given by the softmax function. After calculating the probabilities z_i for each solution component $c_i \in S$, the convergence factor is computed as $cf := \min\{z_i \mid c_i \in S\}$.

Once the convergence factor cf is calculated, the algorithm checks whether it exceeds the convergence factor limit, which is defined by the parameter $cf_{\text{limit}} \in [0, 1]$. If this condition is met, the algorithm is re-initialized as follows:

1. The q-values are reset to zero.
2. Sub-instance C' is cleared based on the Boolean parameter b_{reset}. If $b_{\text{reset}} =$ *true*, C' is set to $\emptyset$. Otherwise, if $b_{\text{reset}} =$ *false*, C' remains unchanged.

Emptying C' during the re-initialization of the algorithm effectively removes all the information accumulated by the learning process. On the other hand, if C' is not cleared, some of the previously gathered information is retained.

7.2.2 UCB Selection

As an alternative to the Softmax selection, we also explore the Upper Confidence Bound (UCB) selection method [122]. This approach, originally developed for the multi-armed bandit problem, focuses on sampling from the distribution set based on their potential to be optimal. We view this as a different approach to addressing convergence, as this selection strategy ensures sufficient exploration over time. Similarly to Softmax selection, this method was implemented as follows:

1. With probability dr, a random solution component is selected from those with the highest q-value.
2. Otherwise, with probability $1 - dr$, UCB selection is used, which involves randomly selecting among the solution components whose q-values maximize the following expression:

$$q_i + \rho \cdot \sqrt{\frac{\log(n)}{n_i}} \tag{7.7}$$

   Here, $\rho > 0$ is a parameter, n denotes the current iteration number, $\log(n)$ represents the natural logarithm of n, and n_i is the number of times solution component c_i has been selected up to the present iteration.

The square-root term in the UCB expression represents the uncertainty in the estimate of q_i. Every time a solution component is selected, its corresponding square-root term decreases, thereby reducing the estimated uncertainty in its

q-value. On the other hand, if a solution component is not chosen in an iteration, the square-root term increases.

Note that if the q-values are unbounded, this method becomes unsuitable, as the square-root term loses its effectiveness once the q-values become sufficiently large. This issue may arise with the first and third designs we proposed for updating the q-values. For this reason, we will apply this method in conjunction with the second design mentioned above for updating the q-values, which sets the q-values to the average of the rewards observed so far.

7.3 Variants Considered

To experimentally evaluate RL-CMSA in [103], we considered the following four algorithm variants, each utilizing different designs for updating the q-values and selecting solution components during the solution construction. The four RL-CMSA variants are described as follows:

1. **RL-CMSA-1**: This variant is characterized by the first design for updating the q-values (summation of the rewards) and the use of Softmax selection. The reward (R) is set to one.

2. **RL-CMSA-2**: This variant uses the second design for updating the q-values (average rewards) and Softmax selection. The value of the reward (R) is treated as a parameter of the algorithm.

3. **RL-CMSA-3**: This variant is identical to RL-CMSA-2, except that the third design (average rewards + step-size) is used for the q-value update.

4. **RL-CMSA-4**: This variant uses the second design for the q-value update, combined with the UCB selection for solution construction as an alternative method to avoid convergence. The reward (R) is set to one.

These four variants were experimentally evaluated in [103] on the Minimum Dominating Set (MDS) and the Far From Most String (FFMS) problems. The results obtained showed that the first variant, RL-CMSA-1, was superior to the rest with statistical significance. Hence, in this thesis we consider only the first variant for evaluation, which we simply denote RL-CMSA from now on. In Chapter 8 the standard CMSA, RL-CMSA, and DL-CMSA are experimentally compared on three combinatorial optimization problems: the MDS problem, the FFMS problem, and the Maximum Independent Set (MIS) problem.

Note that by design, the quality values in RL-CMSA do not differentiate between the partial solution being constructed. Clearly, the solution under construction has a significant impact on which solution components should be added next. DL-CMSA, introduced in the next chapter, addresses this by utilizing a neural network to take into account partial solutions.

Chapter 8

DL-CMSA

This chapter presents DL-CMSA and is based on the paper "Improving the CMSA Algorithm with Online Deep Learning", presented at the European Conference on Artificial Intelligence (ECAI) of 2025 [104].

As mentioned earlier, RL-CMSA does not take into account the partial solution under construction when deciding which solution component to select next. In other words, the q-values are independent of the solution being constructed. However, the current partial solution at each construction step clearly impacts the quality of the available solution components at that moment. Therefore, it is a natural idea to extend RL-CMSA in this direction. One possible approach is to mirror the process used in RL-CMSA by maintaining separate q-values for each possible partial solution. However, this is clearly not feasible in practice due to the enormous number of possible partial solutions.

DL-CMSA incorporates information from the partial solution currently under construction by employing a neural network Q to approximate this corresponding separate q-value vector. The role of Q is to estimate, for each possible solution component, how beneficial it would be to extend the current partial solution with that component. To achieve this, the neural network receives as input a binary encoding of the partial solution and produces as output a quality score for each solution component.

Specifically, DL-CMSA assumes a fixed order of the solution components in order to ensure consistent encodings. Let $C = (c_1, c_2, \ldots, c_n)$ denote the ordered set of all solution components associated with the problem instance being tackled. The input and output layers of Q are then both of size n. The input is a one-hot encoding of the partial solution S, which is a binary vector representing S, with its i-th entry set to one if $c_i \in S$, and zero otherwise. Given this representation, the j-th entry of the output vector is intended to reflect the estimated quality of extending the partial solution S with component c_j.

Just like the q-values in RL-CMSA, the parameters of the network Q are updated at the end of each iteration, based on the solution provided by the

Algorithm 8.1 Pseudo-code of DL-CMSA

```
Input: Ordered set $C = (c_1, \ldots, c_n)$ of solution components for the problem
    instance to be solved
Input: Values for parameters $n_a$, $t_{\text{ILP}}$, $\text{age}_{\text{max}}$, $\varepsilon_{\text{dec}}$ and $\varepsilon_{\text{min}}$
Output: A solution to the problem at hand
 1: $S_{\text{bsf}} := \text{null},\ C' := \emptyset,\ \varepsilon := 1$
 2: Initialize the weights of neural network $Q$ randomly
 3: while termination conditions not met do
 4:     for $j = 1, \ldots, n_a$ do
 5:         $S :=$ probabilistic_solution_construction$(Q, \varepsilon)$
 6:         for all $c_i \in S$ and $c_i \notin C'$ do
 7:             $\text{age}_{c_i} := 0$
 8:             $C' := C' \cup \{c_i\}$
 9:         end for
10:     end for
11:     $S_{\text{opt}} :=$ apply_exact_solver$(C', t_{\text{ILP}})$
12:     if $S_{\text{opt}}$ is better than $S_{\text{bsf}}$ then $S_{\text{bsf}} := S_{\text{opt}}$ end if
13:     adapt$(C', S_{\text{opt}}, age_{\text{max}})$
14:     Order the solution components in $S_{\text{opt}} := (c_{k_i})_{i=1}^{m}$
15:     for $(\hat{S}, c_i) \in \{(\{c_{k_i}\}_{i=1}^{j}, c_{k_{j+1}})\}_{j=0}^{m-1}$ do
16:         Set target $y^i := Q_\theta(\phi(\hat{S})), \quad y^i_{c_i} := y^i_{c_i} + 1$
17:         Compute loss $\mathcal{L}(\theta) := \|Q_\theta(\phi(\hat{S})) - y\|^2$
18:     end for
19:     Apply gradient descent step on average loss
20:     if $\varepsilon > \varepsilon_{\text{min}}$ then $\varepsilon := \varepsilon \cdot \varepsilon_{\text{dec}}$ end if
21: end while
22: return $S_{\text{bsf}}$
```

exact solver. A gradient descent update is used to adjust the parameters of Q to increase the likelihood of performing selections similar to those done by the exact solver.

A challenge arises from the fact that the order in which the solver constructs its solution is typically unknown. To address this, we assume the solver constructs solutions in the same order as the one defined for the one-hot encoding. Given the ordered solution $S_{\text{opt}} = (c_{k_1}, c_{k_2}, \ldots, c_{k_m})$ produced by the solver, the parameters of the neural network Q are updated with the objective of increasing the output of Q for the corresponding (partial solution, solution component) pairs:

$$\{(\emptyset, c_{k_1}), (\{c_{k_1}\}, c_{k_2}), (\{c_{k_1}, c_{k_2}\}, c_{k_3}), \ \ldots, (\{c_{k_1}, \ldots, c_{k_{m-1}}\}, c_{k_m})\}. \tag{8.1}$$

This is done by setting a target for each partial solution as the output of Q for that solution, where the component corresponding to the associated solution component is incremented by one. The loss is then calculated, and a gradient descent step is executed using the Adam optimizer [75] on the average loss.

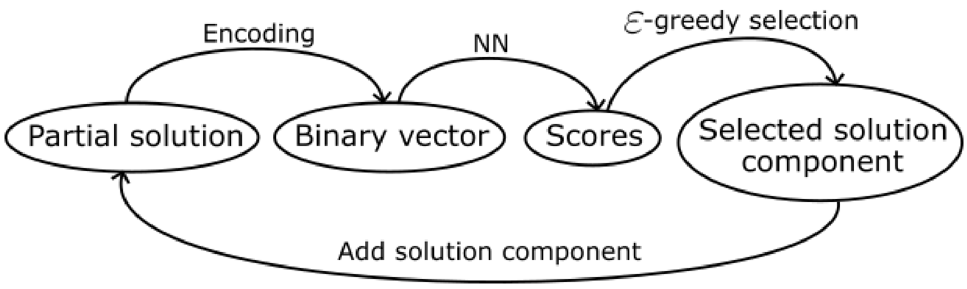


Figure 8.1 Diagram depicting the construction process of DL-CMSA. The algorithm iteratively builds a partial solution S using the neural network Q and epsilon-greedy selection.

The neural network Q is employed in the *construct* step of DL-CMSA to generate solutions. Just like in RL-CMSA, the solution construction begins with an empty solution, and solution components are added iteratively. In this case, the neural network Q replaces the q-values during the construction process. In particular, the next solution component is selected using epsilon-greedy selection, as this method has shown better performance for this algorithm. Given the set $C_{\text{feas}} \subseteq C$ of available solution components for extending a partial solution S, the next solution component added to S is selected as follows:

1. With probability ε, a random solution component from C_{feas} is selected.
2. Otherwise, a random solution component is selected from those in C_{feas} that have the maximum output for Q.

ε is initialized at one and is gradually reduced based on two parameters: $\varepsilon_{\text{dec}} \in [0, 1)$, which controls the rate of decrease for ε at each iteration, and $\varepsilon_{\text{min}} \in [0, 1]$, which defines the minimum value that ε can reach.

Algorithm 8.1 outlines the overall structure of DL-CMSA and Figures 8.1 and 8.2 provide diagrams representing the construct and learning steps of DL-CMSA, respectively. As mentioned earlier, the neural network Q takes the place of the q-value vector in RL-CMSA for solution construction, while the gradient descent step replaces the straightforward q-value update mechanism of RL-CMSA. The construction process, which now employs the neural network Q, corresponds to line 6. Meanwhile, the new update process corresponds to lines 16 to 21. In this context, we use the notation $Q_\theta(\phi(\hat{S}))$, where $\phi(\hat{S})$ is the one-hot encoding of the partial solution $\hat{S}$, and θ denotes the parameters of the neural network being updated.

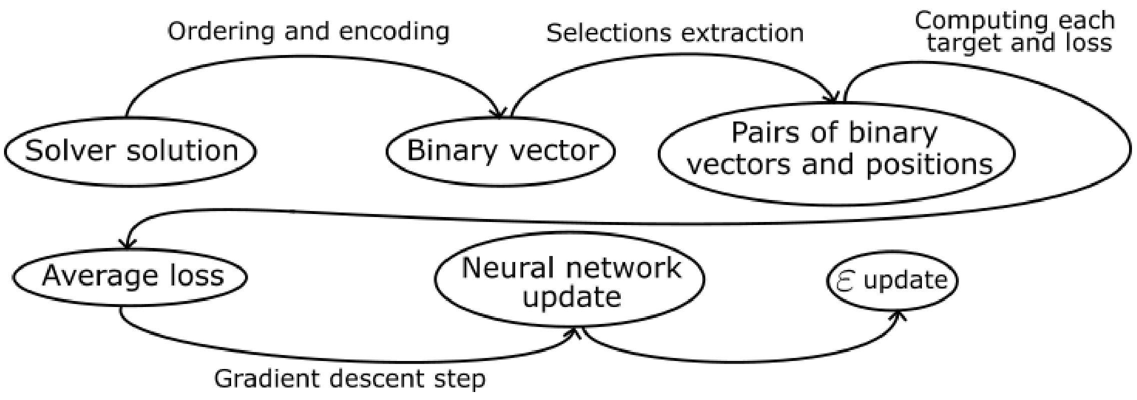


Figure 8.2 Diagram depicting the learning process of DL-CMSA. The algorithm computes an average loss over the selections forming the solution produced by the exact solver and applies a gradient descent step. Lastly, it decreases the parameter ε.

It is worth noting that DL-CMSA does not incorporate a restart mechanism, as no convergence issues have been observed, which can be effectively managed by ensuring a sufficiently large value for ε_{min}.

To better illustrate the construction and learning steps in DL-CMSA, let us consider a small example using a Set Cover problem instance. Given a set of elements U and a collection of subsets of elements S, this problem aims to identify a smallest sub-collection of S whose union equals U. For this problem, a straightforward definition for the set of solution components is $C := S$, which we use for the following example. Consider $U = \{1, 2, 3, 4\}$ and $S = \{c_1 = \{1, 2\}, c_2 = \{2, 3\}, c_3 = \{3, 4\}, c_4 = \{4\}\}$. For the solution component order, we choose (c_1, c_2, c_3, c_4).

The construction process starts with an empty solution $S = \emptyset$. Its one-hot encoding is $(0, 0, 0, 0)$, as none of the four solution components belong to it. The neural network Q_θ takes this encoding as input and produces output scores, for example: $Q_\theta((0, 0, 0, 0)) = (0.3, 0.6, 0.4, -0.1)$. Using ε-greedy selection, suppose component c_2 is chosen. The partial solution becomes $S = \{c_2\}$, and hence its encoding $(0, 1, 0, 0)$. At the next step, the network outputs $Q_\theta((0, 1, 0, 0)) = (0.7,\ 0.2,\ 0.5, 0.9)$. Now the ε-greedy selection can choose any of the available solution components given S, which are all except c_2. If it selects c_4, the solution is still not complete, and again Q is fed with the encoding of the current partial solution which is now $(0, 1, 0, 1)$. The next selection will be the last, as a complete solution will be produced by any of the two possible selections.

Regarding the learning step, assume the exact solver outputs the following solution: $S_{\text{opt}} = \{c_1, c_3\}$. To train the network, this solution is binary encoded and decomposed into pairs of partial solutions and the next selected component following the order assumed. In this case these pairs are: $((0,0,0,0), c_1), ((1,0,0,0), c_3)$. For each pair, a loss is computed with respect to a target that is the current output of the neural network with the position of the solution component selection corresponding to the pair incremented by one. For example, as $Q((0,0,0,0)) = (0.3, 0.6, 0.4, -0.1)$, the target would be $(0.3, 0.6, 0.4, -0.1) + (0,1,0,0) = (0.3, 1.6, 0.4, -0.1)$. And therefore, the loss for this selection $\mathcal{L}(\varphi) = ||Q_\varphi((0,0,0,0)) - (0.3, 1.6, 0.4, -0.1)||^2$. In the same way, the loss would be computed for the second pair of partial solution and solution component selected. Afterwards, the average loss would be considered and the Adam optimizer [75] used to perform a gradient descent step with respect to it.

Chapter 9

Experimental Evaluation

This chapter provides an experimental evaluation of RL-CMSA and DL-CMSA. Specifically, we compare the standard CMSA, RL-CMSA, and DL-CMSA across three combinatorial optimization problems: the Far From Most String (FFMS) problem, the Minimum Dominating Set (MDS) problem, and the Maximum Independent Set (MIS) problem. Notably, these are three NP-Hard problems, meaning that they cannot be solved in polynomial time unless P = NP [51, 82].

The chapter is organized into separate sections, each dedicated to one of the three problems. Each section follows a consistent structure: first, the problem is formally defined; next, the set of solution components C used is described. Then, the probabilistic solution generation method employed by the standard CMSA is explained, along with the construction mechanism utilized by the learning-based variants. Following this, the exact solver applied is discussed, and finally, the parameter tuning process and experimental results are presented and thoroughly analyzed. Before diving into the problem-specific evaluations, we first list the parameters of each algorithm, which will be fine-tuned for each problem.

It is important to note that all algorithms were executed in single-threaded mode on a computing cluster equipped with 10-core Intel Xeon processors running at 2.2 GHz, with 8 GB of RAM per machine. Additionally, each algorithm utilized an Integer Linear Programming (ILP) model in the *solve* step, which was solved using the commercial solver CPLEX version 22.1.1 [65].

9.1 Algorithm Parameters

Each algorithm utilizes the standard CMSA parameters: t_{ILP}, n_a, and $age_{\max}$. These parameters respectively define the time limit allocated to the exact solver in the *solve* step, the number of solutions generated in the *construct* step, and the age threshold used in the *adapt* step. Additionally, every algorithm incorporates three parameters related to the *solve* step, which influence the behavior of CPLEX. These parameters are $\text{cplex}_{\text{warmstart}}$, $\text{cplex}_{\text{emphasis}}$, and $\text{cplex}_{\text{abort}}$.

The first parameter, $\text{cplex}_{\text{warmstart}}$, determines whether an initial solution is provided to CPLEX. If set to *true*, the best-so-far solution is used to warm-start the solving process. The second parameter, $\text{cplex}_{\text{emphasis}}$, governs the trade-off between proving optimality quickly and enhancing the best-found solution efficiently. When set to *true*, CPLEX employs its highest heuristic emphasis; otherwise, the default setting is used. Lastly, $\text{cplex}_{\text{abort}}$ dictates whether the CPLEX execution terminates upon discovering a solution that improves the best-so-far solution. This can be advantageous since CPLEX may otherwise allocate significant computational resources to bound computations necessary for proving optimality.

For the standard CMSA, in addition to the parameters mentioned above, each problem-specific solution generation method includes additional parameters that regulate the diversity of the generated solutions. These parameters will be introduced in their respective sections.

For RL-CMSA, the additional parameters consist of dr, β, cf_{limit}, and b_{reset}. Here, dr is the determinism rate used when selecting solution components, β is a parameter in the Softmax function, cf_{limit} sets the threshold for the convergence factor, and b_{reset} is a Boolean parameter indicating whether C' should be emptied upon re-initialization of the q-values.

Finally, DL-CMSA introduces the parameters ε_{dec} and ε_{min} for solution construction, which represent the decrease rate of ε and its minimum allowable value, respectively.

Additionally, we consider the number of hidden layers n_{layers} in the neural network Q, the number of hidden nodes per layer n_{nodes}, and the learning rate lr used by the Adam optimizer during the gradient descent step as parameters of the algorithm. The activation function was not tuned, a Leaky ReLU [88] was employed as the default choice.

9.2 Application to the FFMS Problem

9.2.1 Problem Definition

The Far From Most String (FFMS) problem [47] is a combinatorial optimization problem that arises in bioinformatics and belongs to the family of sequence consensus problems. These problems have applications in various fields, including molecular biology [58]. Given a set of input strings of equal length over an alphabet Σ and a threshold $t > 0$, the objective is to find a string of the same length that maximizes the number of input strings whose *Hamming* distance from it is at least t.

$$S = \{s_1 = \texttt{baba}, s_2 = \texttt{bbba}, s_3 = \texttt{abaa}\},\ \Sigma = \{\texttt{a}, \texttt{b}\},\ t = 3$$

$$s = \texttt{aaab},\ \begin{matrix} d_H(s, s_1) = 3 \\ d_H(s, s_2) = 4 \\ d_H(s, s_3) = 2 \end{matrix} \Rightarrow f_1(s) = 2$$

Figure 9.1 Example of an FFMS problem instance and a solution. The instance is defined by the top line, which specifies the set of input strings S, the alphabet Σ, and the threshold *th*.

For two strings s and s' of length m, their *Hamming* distance $d_H(s, s')$ is defined as the number of positions where their corresponding characters differ, i.e.,

$$d_H(s, s') := \left|\{k \in \{1, \ldots, m\} \mid s[k] \neq s'[k]\}\right| \tag{9.1}$$

An instance of the FFMS problem is represented as $(\mathcal{S}, \Sigma, t)$, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is a set of n input strings of length m over the alphabet Σ, and t is a threshold satisfying $0 < t \leq m$. Any string of length m over Σ is considered a feasible solution. The objective is to determine a feasible string s that maximizes the following function:

$$f_1(s) := \left|\{s' \in \mathcal{S} \mid d_H(s, s') \geq t\}\right| \tag{9.2}$$

In practice, our algorithm implementations incorporate a secondary objective function to distinguish between solutions that yield the same value for the primary objective function (Equation (9.2)). This secondary objective function is defined as follows:

$$f_2(s) := \sum_{s' \in \{t \in \mathcal{S} \mid d_H(t,s) \geq t\}} d_H(s, s') + \max_{s' \in \{t \in \mathcal{S} \mid d_H(t,s) < t\}} \{d_H(s, s')\} \tag{9.3}$$

A higher value of $f_2(s)$ reduces the likelihood that a minor modification in s results in a decrease in the primary objective function $f(s)$. Consequently, a solution s is considered superior to another solution s' (i.e., $f(s) > f(s')$) if and only if: (1) $f_1(s) > f_1(s')$, or (2) $f_1(s) = f_1(s')$ and $f_2(s) > f_2(s')$.

This lexicographic objective function was introduced in [16] to mitigate the adverse impact of large plateaus in the search space of the FFMS problem.

Figure 9.1 shows an FFMS instance and a solution. The solution $s =$ aaab is valid since it has the same length as the strings in S and uses only characters from Σ. As the threshold is 3 and s achieves a Hamming distance of at least 3 in two strings of S, its objective value is 2.

9.2.2 Solution Components

A natural definition for the set C of solution components in the FFMS problem is to include one solution component for each position-character pair. Specifically, for each position $k = 1, \ldots, m$ in a solution string and each character $a \in \Sigma$, the set C contains the corresponding solution component $c_{k,\mathsf{a}}$.

$$C := \{c_{k,\mathsf{a}} \mid k = 1, \ldots, m, \text{ and } \mathsf{a} \in \Sigma\} \quad . \tag{9.4}$$

Thus, at each step $j = 1, \ldots, m$ of the solution construction process, the set of feasible solution components is $C_{\text{feas}} := \{c_{j,\mathsf{a}} \mid \mathsf{a} \in \Sigma\}$, where selecting $c_{j,\mathsf{a}}$ represents the addition of character a at the current position j.

9.2.3 Probabilistic Solution Construction

In the following, we explain how solutions are generated in both standard CMSA and the learning variants. All algorithms employ the same solution construction mechanism, where a letter for each position $j \in \{1, \ldots, m\}$ is determined sequentially from position 1 to m. In other words, at each j-th construction step, exactly one solution component from $C_{\text{feas}} = \{c_{j,\mathsf{a}} \mid \mathsf{a} \in \Sigma\}$ is selected. However, the method of selection differs between CMSA and the learning variants. In standard CMSA, a probabilistic approach is used based on the following greedy function. For a given position $1 \leq j \leq m$ and character $\mathsf{a} \in \Sigma$, the corresponding frequency $f_{j,\mathsf{a}}$ is defined as:

$$f_{j,\mathsf{a}} := \frac{\left|\{s \in \mathcal{S} \mid s[j] = \mathsf{a}\}\right|}{|\mathcal{S}|} \tag{9.5}$$

For selecting a letter for position j, the following procedure is employed:

1. With probability $0 \leq dr_{\text{CMSA}} \leq 1$, the solution component (letter-position assignment) with the lowest frequency value is selected, with ties being broken randomly.

2. Otherwise, with probability $1 - dr_{\text{CMSA}}$, a solution component is chosen from C_{feas}, with the letter probabilities being proportional to the inverse of their frequencies. Specifically, the probability of selecting solution component $c_{j,\mathsf{a}}$ is given by:

$$\frac{1/f_{j,\mathsf{a}}}{\sum_{\alpha \in \Sigma} 1/f_{j,\alpha}} \tag{9.6}$$

Here, $dr_{\text{CMSA}} \in [0, 1]$ is a parameter known as the determinism rate of CMSA.

In contrast, RL-CMSA selects a letter for each position j of a solution s based on the q-values. A q-value is maintained for each solution component, meaning a q-value $q_{j,\mathsf{a}}$ is kept for every position-character pair. At the j-th solution construction step, one of the solution components is chosen from $C_{\text{feas}} = \{c_{j,\mathsf{a}} \mid \mathsf{a} \in \Sigma\}$ using Softmax selection.

DL-CMSA uses the output of the neural network Q to determine the next character to add. Recall that partial solutions are encoded by a vector, where each solution component is represented by a binary value, resulting in $m \cdot |\Sigma|$ values in total. Specifically, for this problem, the solution components are ordered by position and character. For each position $i = 1, \ldots, m$ and character $j = 1, \ldots, |\Sigma|$, the encoding position $i \cdot j$ takes a value of 1 if the partial solution contains the j-th character of the alphabet in its i-th position. For instance, if $m = 4$ and $\Sigma = \{a, b, c\}$, the partial solution $S = aca$ would be encoded as $(1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)$. At each construction step, the output from Q, given the current partial solution, is combined with ε-selection to decide which character is added next.

9.2.4 ILP Model and Sub-Instance Solving

In CMSA algorithms, sub-instances are typically modeled using Integer Linear Programming (ILP) models, which are solved at each iteration by an ILP solver. As noted earlier, this work employs the commercial solver CPLEX for this purpose. The standard ILP model for the FFMS problem consists of two sets of binary variables. The first set contains a variable $x_{j,\mathsf{a}}$ for each position $j = 1, \ldots, m$ and character $\mathsf{a} \in \Sigma$, while the second set contains a variable y_j for every position $j = 1, \ldots, m$.

$$\max \sum_{i=1}^{n} y_i \tag{9.7}$$

$$\text{subject to} \sum_{\mathsf{a} \in \Sigma} x_{j,\mathsf{a}} = 1, \qquad \text{for } j = 1, \ldots, m \tag{9.8}$$

$$\sum_{j=1}^{m} x_{j,s_i[j]} \leq m - t \cdot y_i, \qquad \text{for } i = 1, \ldots, n \tag{9.9}$$

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

The variable $x_{j,\mathsf{a}}$ is assigned a value of one if character a is selected for position j in the solution string, and zero otherwise. Constraint (9.8) ensures that only one character is selected for each position. Furthermore, constraint (9.9), in conjunction with the maximization objective, guarantees that y_i equals 1 if and only if the Hamming distance between the solution string and the input string s_i is at least *th*.

To solve a sub-instance $C' \subseteq C$, for every $c_{j,\mathtt{a}} \in C \setminus C'$, the constraint $x_{j,\mathtt{a}} = 0$ is added to the ILP model. In other words, the variables corresponding to solution components that are not included in sub-instance C' are fixed to zero.

9.2.5 Experimental Evaluation

We generated a set of benchmark instances to evaluate the performance of the different CMSA variants in the context of the FFMS problem. Each instance consists of n strings of length m, with characters drawn from an alphabet Σ of size $|\Sigma|$ uniformly at random. Additionally, each instance has its associated threshold, denoted as *th*, specified as a proportion of m. The benchmark set includes 720 instances for each value of $|\Sigma| \in \{4, 12, 20\}$. These instances are further divided into 30 instances for every combination of $n \in \{100, 200, 300, 400\}$ and $m \in \{100, 500, 1000\}$. Furthermore, two threshold values, depending on $|\Sigma|$, are considered for all instances: $(0.8m, 0.85m)$ for instances with $|\Sigma| = 4$, $(0.97m, 1.0m)$ for $|\Sigma| = 12$, and $(0.99m, 1.0m)$ for $|\Sigma| = 20$.

In addition to the previously mentioned instances, the benchmark set also comprises 72 tuning instances, one for each combination of n, m, $|\Sigma|$, and *th*. Each algorithm underwent two separate tuning procedures: the first for all instances with the lower threshold *th* from each threshold pair, and the second for those with the higher threshold. This was based on prior research indicating that changes in the threshold value have a more significant effect on the problem's characteristics than variations in n or m. To minimize the number of tuning instances, instances with $m = 500$ were excluded. As a result, each tuning run utilized 24 instances. Moreover, a total of 3000 algorithm runs were allocated to each tuning process, with each algorithm run being given a time limit of 600 CPU seconds for both tuning and evaluation phases.

Table 9.1 presents the results obtained from the parameter tuning process. The first column lists the names of the parameters that were tuned, while the second column specifies the allowed numerical range for each. For each algorithm, two columns report the selected parameter values by *irace* [87], one corresponding to the lower-threshold instances and the other to the higher-threshold instances.

Some consistent patterns emerge from the parameter tuning results. Across all cases, a high value for t_{ILP} is selected, indicating the importance of giving the exact solver sufficient time. (Interestingly, significantly lower values are selected for the MDS and MIS problems, as shown in Tables 9.5 and 9.8.) Another trend concerns

Table 9.1 Parameter values obtained after tuning the three CMSA variants for the FFMS problem. Two tuning runs are performed for every algorithm. One for the lower thresholds $(0.8m, 0.97m, 0.99m)$ and one for the higher thresholds $(0.85m, 1.00m, 1.00m)$.

	Allowed Range	CMSA		RL-CMSA		DL-CMSA	
t_{ILP}	$\{1, 2, \ldots, 50\}$	48	32	36	28	27	38
n_a	$\{1, 2, \ldots, 50\}$	17	35	14	46	19	36
age_{max}	$\{1, 2, \ldots, 10\}$	7	10	9	7	8	9
$\text{cplex}_{\text{warmstart}}$	$\{0, 1\}$	1	1	1	1	1	1
$\text{cplex}_{\text{emphasis}}$	$\{0, 1\}$	1	1	1	1	1	1
$\text{cplex}_{\text{abort}}$	$\{0, 1\}$	0	1	0	0	0	0
dr_{CMSA}	$\{0, 0.01, \ldots, 0.99\}$	0.35	0.48	-	-	-	-
dr	$\{0, 0.01, \ldots, 0.99\}$	-	-	0.36	0.95	-	-
β	$\{0.0, 0.01, \ldots, 2.0\}$	-	-	0.70	1.12	-	-
b_{reset}	$\{0, 1\}$	-	-	1	0	-	-
cf_{limit}	$\{0.90, 0.91, \ldots, 1.0\}$	-	-	0.95	0.95	-	-
n_{layers}	$\{1, 2, 3\}$	-	-	-	-	1	1
n_{nodes}	$\{10, 11, \ldots, 1000\}$	-	-	-	-	37	225
lr	$\{0.001, 0.002, \ldots, 0.2\}$	-	-	-	-	0.132	0.159
ε_{dec}	$\{0.950.0.951, \ldots, 1\}$	-	-	-	-	0.962	0.958
ε_{min}	$\{0, 0.01, \ldots, 0.1\}$	-	-	-	-	0.04	0.06

the number of constructed solutions n_a, which is always set to a higher value for the higher-threshold instances. Additionally, the CPLEX-related parameters show clear and consistent preferences: $\text{cplex}_{\text{warmstart}}$ and $\text{cplex}_{\text{emphasis}}$ are always set to 1, while $\text{cplex}_{\text{abort}}$ is set to 0 in all but one case. For DL-CMSA, the tuning process favors a single hidden-layer neural network with similar values for lr, ε_{dec}, and ε_{min} across instance types, except for the number of hidden nodes, which is notably larger for the higher-threshold instances.

The results obtained by the three CMSA variants are reported in Tables 9.2–9.4. For each combination of *th*, n, m, and each algorithm, we present the average length of the best solutions found, as well as the average execution time required to obtain these best solutions. These two values are shown in columns $\overline{|s|}$ and $\bar{t}_{best}[s]$, respectively. The three tables correspond to instances with $|\Sigma| = 4$, $|\Sigma| = 12$, and $|\Sigma| = 20$, respectively. As each combination of *th*, n, and m includes 30 distinct benchmark instances the reported values represent averages over the corresponding 30 values.

The results indicate that RL-CMSA achieves the best average performance. Standard CMSA ranks second overall, while DL-CMSA shows the weakest average performance. Interestingly, for the high-threshold instances with $|\Sigma| = 4$, RL-CMSA performs poorly, with standard CMSA yielding the best results.

Table 9.2 Comparison of the three CMSA variants for the FFMS problem instances with alphabet size $|\Sigma| = 4$.

			CMSA		RL-CMSA		DL-CMSA	
th	n	m	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
$0.8m$	100	100	**76.07**	129.23	75.97	134.00	75.40	116.96
		500	79.37	193.15	**81.07**	258.66	78.80	150.81
		1000	78.97	249.11	**83.27**	380.96	77.10	217.75
	200	100	98.57	106.89	**101.43**	216.27	96.93	148.98
		500	**94.20**	234.71	94.07	277.99	91.70	188.68
		1000	86.70	417.10	**92.97**	532.67	81.90	455.36
	300	100	118.50	114.87	**119.63**	362.91	114.80	162.35
		500	96.37	295.47	**99.03**	368.70	93.47	362.06
		1000	88.33	472.47	**94.63**	565.84	84.60	492.72
	400	100	141.80	170.77	**144.97**	328.59	138.97	140.59
		500	95.27	384.24	**101.10**	453.05	92.57	462.19
		1000	86.40	544.58	**95.23**	577.30	76.57	560.73
$0.85m$	100	100	**38.40**	106.95	38.37	40.91	38.33	62.78
		500	**27.97**	225.42	27.93	55.87	27.93	72.28
		1000	**25.20**	229.84	24.50	62.49	24.57	107.17
	200	100	**46.53**	120.81	46.07	48.35	46.30	75.68
		500	**27.93**	217.18	27.17	115.07	27.03	106.26
		1000	**24.93**	283.50	24.77	92.43	24.83	214.71
	300	100	49.80	95.61	**50.40**	40.95	50.37	53.81
		500	**27.60**	220.7	27.17	104.47	27.23	145.69
		1000	**25.13**	310.81	23.90	183.55	24.53	326.27
	400	100	**53.57**	106.36	51.00	76.36	51.53	75.85
		500	27.50	289.57	27.07	165.47	**27.73**	130.13
		1000	**24.03**	336.02	23.10	228.00	23.43	411.30

Regarding computation time, all three algorithms require a similar amount of time to find their best solutions. The only notable exception occurs in the instances with $|\Sigma| = 12$, where RL-CMSA utilizes a significantly larger portion of the 600-second time limit compared to the other two variants. This extended search effort may indicate that RL-CMSA is less prone to getting trapped in local optima, which could help explain its superior performance.

Figure 9.2 presents Critical Difference (CD) plots for the FFMS problem results, generated using the *R* package `scmamp` [22]. Each plot displays the average rank of the algorithms along the x-axis, with horizontal bars connecting those whose differences are not statistically significant. Statistical significance is assessed using the Friedman rank-sum test, with Finner's procedure [49] as the post-hoc method for pairwise comparisons, applying a significance level of 0.05.

The CD plot in Figure 9.2a, which aggregates all instances, shows RL-CMSA

Table 9.3 Comparison of the three CMSA variants for the FFMS problem instances with alphabet size $|\Sigma| = 12$.

			CMSA		RL-CMSA		DL-CMSA	
th	n	m	$\overline{\lvert s \rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s \rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s \rvert}$	$\bar{t}_{best}[s]$
$0.97m$	100	100	74.10	212.25	**74.13**	262.15	73.63	242.33
		500	66.63	277.79	**67.43**	390.97	67.17	180.70
		1000	64.00	293.85	64.40	292.47	**64.73**	226.14
	200	100	**98.00**	274.53	97.87	280.40	97.20	191.88
		500	74.07	316.19	**75.43**	419.16	74.83	250.32
		1000	67.17	316.43	68.23	297.67	**68.67**	270.92
	300	100	111.50	276.12	**113.23**	249.19	112.00	141.37
		500	76.10	327.43	**78.07**	438.66	76.83	286.99
		1000	67.90	305.39	**69.73**	385.95	66.90	233.66
	400	100	123.20	284.09	**124.00**	282.25	122.50	159.29
		500	77.57	334.70	**80.53**	467.82	78.47	246.64
		1000	68.23	343.07	**69.97**	368.69	69.23	265.24
$1.0m$	100	100	31.43	120.36	**31.53**	198.29	31.33	127.89
		500	**19.33**	98.86	19.30	332.61	19.17	130.15
		1000	17.00	46.18	**17.07**	297.36	17.00	186.82
	200	100	34.60	153.45	**34.73**	204.35	34.40	157.16
		500	19.23	76.03	**19.37**	256.75	19.10	116.91
		1000	17.00	51.30	**17.03**	267.31	17.00	217.02
	300	100	36.23	129.72	**36.33**	238.39	36.23	129.02
		500	19.20	97.21	**19.53**	211.86	19.10	128.80
		1000	16.97	154.56	**17.00**	257.58	16.93	209.56
	400	100	**37.23**	127.64	**37.23**	215.16	36.93	122.34
		500	19.07	81.34	**19.40**	295.79	19.33	139.66
		1000	**16.97**	106.38	**16.97**	300.63	16.93	222.01

achieving the best average rank, followed by standard CMSA, and then DL-CMSA. All pairwise differences are statistically significant. Figures 9.2b and 9.2c report CD plots for the lower and higher threshold instances, respectively. For the lower threshold set, performance differences are more pronounced and mirror the overall ranking. In contrast, for the higher threshold instances, differences in average rank are smaller: CMSA ranks best, followed closely by RL-CMSA, with no statistically significant difference between them. DL-CMSA ranks lowest, though the difference compared to CMSA is not statistically significant.

For this problem, the standard CMSA just presented was the state-of-the-art algorithm until 2024. Then, a new variant of CMSA, which hybridizes the standard algorithm with a particular population-based metaheuristic known as a bacterial algorithm, showed better performance [99].

Table 9.4 Comparison of the three CMSA variants for the FFMS problem instances with alphabet size $|\Sigma| = 20$.

			CMSA		RL-CMSA		DL-CMSA	
th	n	m	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
$0.99m$	100	100	**86.93**	264.18	86.80	340.17	86.80	137.30
		500	**84.33**	289.07	84.23	263.97	84.13	224.17
		1000	81.13	238.01	**82.20**	363.23	80.63	265.10
	200	100	**117.93**	288.90	117.53	290.23	117.83	217.09
		500	93.60	383.37	93.13	222.34	**94.00**	267.38
		1000	86.53	403.65	**87.97**	392.85	86.57	274.71
	300	100	136.23	287.32	**136.43**	293.30	136.03	253.69
		500	**97.87**	423.38	97.47	264.78	97.37	286.13
		1000	88.40	397.48	**89.57**	366.82	88.10	300.55
	400	100	**151.33**	317.64	151.10	297.16	150.33	212.57
		500	**99.53**	381.12	98.93	242.14	98.93	315.44
		1000	89.03	387.21	**90.93**	383.92	88.87	334.50
$1.0m$	100	100	**62.23**	65.82	62.07	279.75	62.20	123.72
		500	43.63	273.42	**43.80**	234.20	43.67	227.71
		1000	37.83	187.77	**38.03**	326.79	37.70	405.85
	200	100	77.10	212.56	**77.20**	245.91	76.90	214.12
		500	43.77	237.56	44.13	286.73	**44.40**	262.02
		1000	37.47	172.53	**38.20**	256.08	37.70	385.84
	300	100	**84.63**	170.86	84.27	243.04	83.93	251.11
		500	44.57	278.63	44.53	257.92	**44.67**	282.16
		1000	38.03	244.75	**38.20**	281.44	37.87	404.62
	400	100	**90.00**	194.44	89.60	256.06	89.10	168.68
		500	44.37	282.37	44.87	275.58	**45.13**	284.81
		1000	37.87	193.87	**38.07**	260.2	38.00	405.26

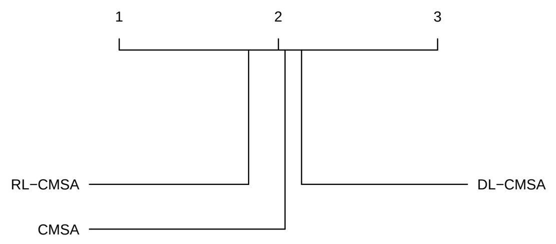
9.3 Application to the MDS Problem

9.3.1 Problem Definition

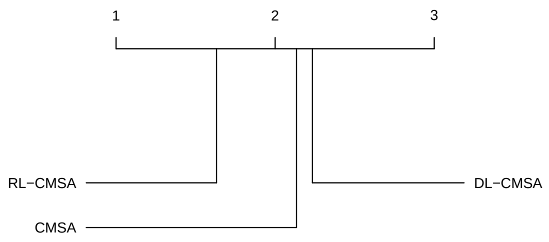
The Minimum Dominating Set (MDS) problem is another well-known combinatorial optimization problem. Given an undirected graph, the goal of the MDS problem is to identify the smallest subset of nodes such that every node in the graph is either included in this subset or is adjacent to at least one node in it. Formally, let $G = (V, E)$ be an undirected graph. The MDS problem seeks the smallest subset $\tilde{V} \subseteq V$ such that for each node $v \in V$, at least one of the following two conditions holds:

1. $v \in \tilde{V}$

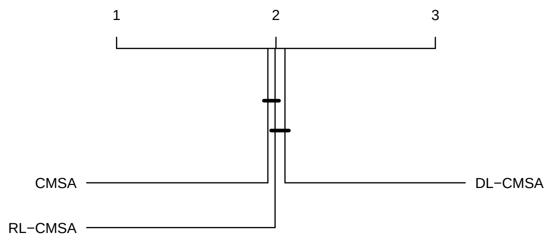
2. $v' \in \tilde{V}$ for some $v' \in N(v)$



(a) All instances.



(b) Instances with thresholds $(0.80m, 0.97m, 0.99m)$.



(c) Instances with thresholds $(0.85m, 1.00m, 1.00m)$.

Figure 9.2 CD plots concerning the FFMS problem results.

Here, $N(v)$ denotes the set of neighbors of a node v in the graph G. Specifically, $N(v) := \{v' \in V \mid (v', v) \in E\}$. A subset of nodes that satisfies the two conditions above is referred to as a dominating set of G. Therefore, the MDS problem focuses on finding dominating set of minimal size, as indicated by its name. The MDS problem has various applications in fields such as wireless sensor networks [100] and natural language processing [116].

Figure 9.3 illustrates two solutions to the MDS problem on the given graph. They are both valid solutions, as every node is orange or neighbor of an orange one. The solution on the right is optimal, since the graph admits no smaller dominating set.

9.3.2 Solution Components

A natural way to define the solution components for the MDS problem is to introduce a solution component for each node in the input graph. The set of solution components is then $C = V$. For convenience, we will abuse notation and refer to solution components directly as the nodes, i.e., $C := \{v_1, \ldots, v_n\}$, where each solution component v_i corresponds to a node in the input graph G. At each step of the solution construction process for a partial solution $S \subseteq C$, the set of available solution components $C_{\text{feas}} \subseteq C = V$ consists of all nodes except those that are already covered by a node in S and have no uncovered neighbors.

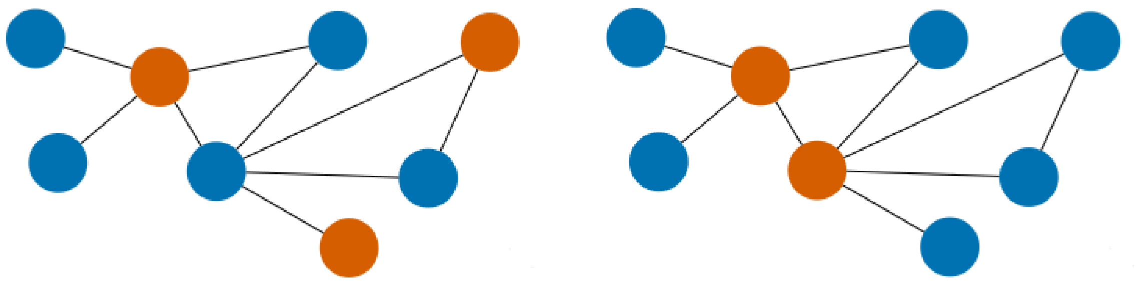


Figure 9.3 Example of an MDS problem instance and two solutions. The problem instance corresponds to the graph, and the nodes forming the two solutions are highlighted in orange.

9.3.3 Probabilistic Solution Construction

All CMSA variants considered follow the same solution construction mechanism. The process begins with an empty solution, $S := \emptyset$, and at each step, a single node (solution component) is added until a valid solution, which is a dominating set, is achieved. In this context, $C_{\text{feas}} \subseteq C$ represent the set of feasible solution components at the current step as discussed previously.

The standard CMSA employs the following greedy function to select, at each construction step, a node from C_{feas}. To define this greedy function, let $N[v] := N(v) \cup \{v\}$ represent the closed neighborhood of node v, and let $N[v \mid S] \subseteq N[v]$ denote the set of uncovered neighbors of v relative to the current partial solution S. Based on this, the process for selecting a node to add to S in CMSA is as follows:

1. With a probability $0 \leq dr_{\text{CMSA}} \leq 1$, a node $v \in C_{\text{feas}}$ is selected as follows:

$$v := \underset{v' \in C_{\text{feas}}}{\arg\max} \left\{ \left| N[v' \mid S] \right| \right\} \tag{9.10}$$

2. Otherwise, with probability $1 - dr_{\text{CMSA}}$, a subset of $\min\left\{ l^{\text{size}}_{\text{CMSA}}, |C_{\text{feas}}| \right\}$ nodes from C_{feas} is selected and stored in $L \subseteq C_{\text{feas}}$, such that:

$$\left| N[v \mid S] \right| \leq \left| N[v' \mid S] \right| \quad \text{for all } v \in L, v' \in C_{\text{feas}} \setminus L \tag{9.11}$$

   A node $v \in L$ is then selected uniformly at random and added to S.

Here, dr_{CMSA} and $l^{\text{size}}_{\text{CMSA}}$ are parameters of the CMSA algorithm.

Unlike standard CMSA, the learning-based variants do not rely on this greedy function. Instead, RL-CMSA utilizes the set of q-values, where each node (solution component) $v_i \in C$ is assigned a corresponding value q_i. At each construction step, a node is selected using softmax selection (see Section 7.2 of Chapter 7).

In DL-CMSA, the construction process is driven by the neural network Q, which evaluates the current partial solution and assigns a quality score to each solution component. In order to get fed into the neural network, recall that partial solutions are one-hot encoded based on the solution components. Since, in the MDS problem, these components correspond to the nodes of the input graph, each partial solution is represented as a binary vector, where each entry indicates the presence or absence of a node. The ordering of nodes is predefined when the problem instance is first processed by the algorithm and remains consistent throughout its execution.

9.3.4 ILP Model and Sub-Instance Solving

The following simple ILP model for the MDS problem is solved by CPLEX at each iteration. It keeps a binary variable x_i for every node $v_i \in V$, which takes value one if v_i forms part of the solution and zero otherwise.

$$\min \sum_{v_i \in V} x_i \tag{9.12}$$

$$\text{subject to} \sum_{v_j \in N(v_i)} x_j + x_i \geq 1, \qquad \text{for } v_i \in V \tag{9.13}$$

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

Constraints (9.13) ensure that the solutions form valid dominating sets by requiring that each node is either included in the solution or covered by at least one of its neighbors. Additionally, the objective function minimizes the total number of selected nodes, ensuring the smallest possible dominating set.

To solve a sub-instance $C' \subseteq C$, we introduce the constraint $x_j = 0$ for all $v_j \in C \setminus C'$. This ensures that the variables corresponding to solution components (nodes) that are not part of sub-instance C' are fixed to zero in the ILP model.

9.3.5 Experimental Evaluation

To conduct the experimental evaluation on the MDS problem, we utilize a benchmark set composed of graphs of varying sizes and densities, generated using three well-known graph models: Erdös-Rényi [45], Watts-Strogatz [127], and Barabási-Albert [7].

The Erdös-Rényi model, one of the most widely used random graph models, generates graphs based on two parameters: the number of nodes and the probability that an edge exists between any given pair of nodes. The Watts-Strogatz model is designed for constructing small-world networks, which exhibit both a short average path length between nodes and a high degree of local clustering. Finally, the Barabási-Albert model generates graphs characterized by a scale-free degree distribution, where most nodes have relatively few connections, while a few nodes possess significantly higher degrees.

A total of thirty graphs were generated for each combination of graph model, number of nodes $|V| \in \{500, 1000, 1500, 2000\}$, and four distinct density levels. The density of the graphs is controlled by parameters p, k, and m corresponding to the three graph models, respectively. The four density levels considered are $p \in \{0.00416381, 0.0062414, 0.0103881, 0.020705\}$ for Erdös-Rényi and $k, m \in \{2, 3, 5, 10\}$ for Watts-Strogatz and Barabási-Albert.

For clarity, these density levels will be referred to as the 1st, 2nd, 3rd, and 4th density levels, respectively. Consequently, the benchmark set comprises 480 graphs per model, resulting in a total of 1440 instances. Additionally, the set includes one tuning instance for each combination of graph model, density level, and graph size.

The three CMSA variants were tuned using the tuning instances corresponding to the lowest and highest density levels, meaning the instances related to the 2nd and 3rd density levels were excluded in order to speed up the process. This results in a total of 24 tuning instances. Similar to the FFMS problem, the tuning was performed using the *R* package *irace* [87], with a budget of 3000 experiments per tuning run. For both tuning and evaluation phases, each algorithm execution was allocated a time limit of 150, 300, 450, and 600 CPU seconds for instances of size 500, 1000, 1500, and 2000, respectively.

Table 9.5 presents the parameter configurations selected by *irace* for each algorithm on the MDS problem. The table follows the same structure as used for the FFMS problem. However, in this case, only a single value per parameter and algorithm is reported, as each algorithm was tuned just once.

An interesting observation concerns the preferred value for the parameter t_{ILP}, which sets the time limit given to CPLEX at each iteration. Notably, for the two learning-based variants, the selected value is significantly smaller than the one chosen for the standard CMSA. A lower value for this parameter results in a greater number of iterations, which is advantageous for the learning processes in RL-CMSA and DL-CMSA. This preference likely reflects the effectiveness of the learning mechanisms, which benefit more from frequent updates than from allocating additional time to the exact solver. Another noteworthy pattern

Table 9.5 Parameter values obtained after tuning the three CMSA variants for the MDS problem. Every algorithm is tuned exactly once.

	Allowed Range	CMSA	RL-CMSA	DL-CMSA
t_{ILP}	$\{1, 2, \ldots, 20\}$	13	6	6
n_a	$\{1, 2, \ldots, 50\}$	4	10	3
age_{max}	$\{1, 2, \ldots, 10\}$	3	1	4
$\text{cplex}_{\text{warmstart}}$	$\{0, 1\}$	0	0	0
$\text{cplex}_{\text{emphasis}}$	$\{0, 1\}$	1	1	1
$\text{cplex}_{\text{abort}}$	$\{0, 1\}$	0	0	0
dr_{CMSA}	$\{0.0, 0.01, \ldots, 0.99\}$	0.29	-	-
$l^{\text{size}}_{\text{CMSA}}$	$\{3, 4, \ldots, 50\}$	35	-	-
dr	$\{0.0, 0.01, \ldots, 0.99\}$	-	0.44	-
β	$\{0.0, 0.01, \ldots, 2.0\}$	-	0.28	-
b_{reset}	$\{0, 1\}$	-	1	-
cf_{limit}	$\{0.90, 0.91, \ldots, 1.0\}$	-	0.98	-
n_{layers}	$\{1, 2, 3\}$	-	-	2
n_{nodes}	$\{10, 11, \ldots, 1000\}$	-	-	30
lr	$\{0.001, 0.002, \ldots, 0.2\}$	-	-	0.168
ε_{dec}	$\{0.950.0.951, \ldots, 1\}$	-	-	0.963
ε_{min}	$\{0, 0.01, \ldots, 0.1\}$	-	-	0.05

emerges in the CPLEX-related parameters: all three algorithm set $\text{cplex}_{\text{warmstart}}$ and $\text{cplex}_{\text{abort}}$ to 0, while assigning a value of 1 to $\text{cplex}_{\text{emphasis}}$.

The results are summarized in Tables 9.6 and 9.7. For each combination of graph size, graph type, and density level, the tables report the average size of the solutions found and the average time taken by the algorithms to obtain them. Table 9.6 includes results for problem instances with 500 and 1000 nodes, while Table 9.7 covers instances with 1500 and 2000 nodes. Each value represents the average over the 30 instances of every configuration.

The main observation is that, for this problem, both RL-CMSA and DL-CMSA achieve better results on average compared to the standard CMSA. The performance of the two learning-based variants is largely similar, with no consistent pattern favoring one over the other.

Regarding the time required to find the best solutions, DL-CMSA is the most time-consuming approach. This is likely due to the computational overhead introduced by its Deep Learning (DL) mechanism, which is significantly greater than the simpler update procedure used in RL-CMSA. In contrast, the standard CMSA is the fastest, likely because it tends to get stuck in local optima more quickly, as reflected in its overall inferior performance.

Figure 9.4 presents the CD plots for the MDS problem results. As with the FFMS problem, each plot displays the average rank of the algorithms along the

Table 9.6 Comparison of the three CMSA variants for the MDS problem instances of size $|V| = 500$ and $|V| = 1000$

			CMSA		RL-CMSA		DL-CMSA	
$\|V\|$	Graph Type	Density	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
500	Barabási-Albert	1st	**101.23**	0.04	**101.23**	0.13	**101.23**	0.54
		2nd	**71.97**	0.96	**71.97**	3.11	**71.97**	5.77
		3rd	47.90	18.01	**47.87**	20.58	**47.87**	17.92
		4th	27.13	18.94	**27.10**	28.65	27.17	23.29
	Erdös Rényi	1st	**209.97**	0.20	**209.97**	0.08	**209.97**	0.69
		2nd	**153.37**	0.66	**153.37**	0.17	**153.37**	0.84
		3rd	101.90	42.66	101.67	28.90	**101.63**	26.81
		4th	60.37	55.80	60.60	41.67	**60.17**	59.14
	Watts-Strogatz	1st	110.33	42.43	**110.13**	39.12	110.17	36.94
		2nd	82.40	70.20	82.23	57.67	**82.20**	46.38
		3rd	**57.20**	72.67	57.67	38.12	**57.20**	80.23
		4th	**34.87**	59.06	35.13	51.79	34.93	80.61
1000	Barabási-Albert	1st	**202.07**	0.37	**202.07**	0.43	**202.07**	0.58
		2nd	145.07	15.69	**145.00**	20.90	**145.00**	26.26
		3rd	92.27	64.60	**92.20**	41.20	92.27	35.24
		4th	**50.40**	72.50	50.67	56.36	50.47	52.83
	Erdös Rényi	1st	241.43	90.70	**241.17**	20.15	**241.17**	30.36
		2nd	175.30	145.99	**174.37**	114.85	174.40	167.53
		3rd	122.07	126.97	**120.67**	176.69	121.00	227.16
		4th	75.73	114.87	**75.23**	220.15	75.80	197.33
	Watts-Strogatz	1st	220.90	163.33	220.10	83.28	**220.03**	85.63
		2nd	166.33	151.48	165.03	132.51	**164.80**	200.25
		3rd	117.37	156.41	**115.33**	167.53	115.70	244.17
		4th	72.10	126.95	**70.63**	245.70	71.27	258.31

x-axis, with horizontal bars indicating non-statistically significant differences in performance. Statistical significance is assessed using the Friedman rank-sum test, with Finner's procedure [49] applied as the post-hoc method, using a significance level of 0.05.

Figure 9.4a, which considers all instances together, confirms the trends observed in the results tables: RL-CMSA and DL-CMSA perform similarly, while the standard CMSA performs worse. The difference in performance between RL-CMSA and DL-CMSA is not statistically significant, whereas both learning variants significantly outperform the standard CMSA. This same pattern is evident in Figures 9.4c and 9.4d, corresponding to the Erdös-Rényi and Watts-Strogatz instances, respectively. In contrast, Figure 9.4b, which concerns the Barabási-Albert instances, shows that all three algorithms achieve very similar performance, with no statistically significant differences. This is likely due to the

Table 9.7 Comparison of the three CMSA variants for the MDS problem instances of size $|V| = 1500$ and $|V| = 2000$

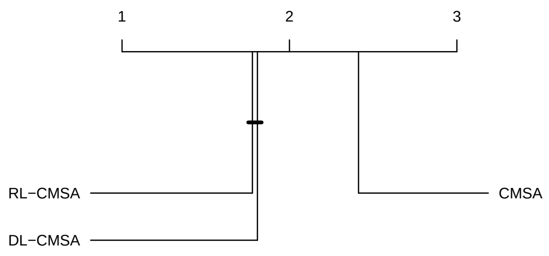
			CMSA		RL-CMSA		DL-CMSA	
$\lvert V\rvert$	Graph Type	Density	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
1500	Barabási-Albert	1st	303.97	18.43	**303.83**	1.52	**303.83**	1.74
		2nd	213.73	48.63	**213.67**	61.82	213.70	52.58
		3rd	136.83	146.81	**136.77**	98.99	**136.77**	84.69
		4th	**73.30**	107.81	73.60	128.50	73.53	77.18
	Erdös Rényi	1st	263.17	261.17	261.30	172.09	**261.07**	241.29
		2nd	198.47	221.23	**194.80**	296.68	194.97	356.25
		3rd	140.50	194.25	138.17	271.25	**137.83**	386.43
		4th	87.40	135.35	**86.13**	335.87	86.23	375.31
	Watts-Strogatz	1st	331.10	284.44	329.60	160.07	**329.53**	153.96
		2nd	251.77	239.13	248.30	225.37	**248.17**	261.99
		3rd	178.60	216.38	174.57	233.85	**173.73**	363.34
		4th	110.97	143.81	**106.73**	328.33	107.07	395.86
2000	Barabási-Albert	1st	403.70	23.70	**403.57**	1.73	**403.57**	2.35
		2nd	285.67	81.66	**285.40**	121.12	285.50	74.81
		3rd	180.93	177.78	**180.87**	149.31	181.00	109.86
		4th	**95.57**	152.56	95.70	215.08	95.73	115.67
	Erdös Rényi	1st	289.10	255.75	284.17	290.83	**283.27**	391.59
		2nd	218.77	283.50	214.07	401.48	**213.47**	478.74
		3rd	156.30	208.89	**153.50**	369.36	153.63	535.36
		4th	96.17	181.82	**95.07**	436.07	96.70	510.05
	Watts-Strogatz	1st	442.63	367.59	440.70	200.91	**440.60**	193.46
		2nd	337.77	328.00	331.67	292.25	**331.30**	314.12
		3rd	241.97	243.74	233.13	378.62	**231.97**	438.04
		4th	152.57	58.85	**145.80**	421.32	147.93	577.52

relative simplicity of these instances, as Barabási-Albert graphs tend to include high-degree nodes that are obvious candidates for inclusion in the solution.

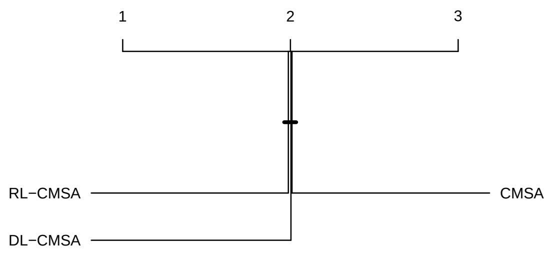
9.4 Application to the MIS Problem

9.4.1 Problem Definition

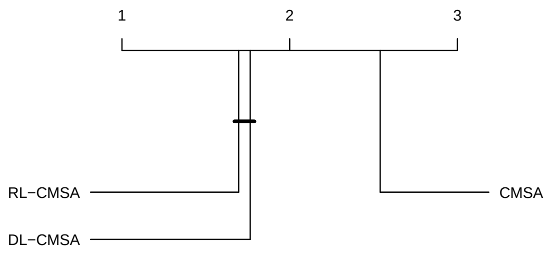
The Maximum Independent Set (MIS) problem is another well-known combinatorial optimization problem in graph theory. As with the MDS problem, given an undirected graph $G = (V, E)$ and a node $v \in V$, we define the closed neighborhood of v as $N[v] := N(v) \cup \{v\}$, where $N(v)$ is the set of neighbors of v. A set $S \subset V$ is considered a feasible solution to the MIS problem if, for any pair of distinct nodes $v, u \in S$, there is no edge between them, i.e., $(v, u) \notin E$. In other words, a feasible solution S ensures that no two nodes in S are adjacent.



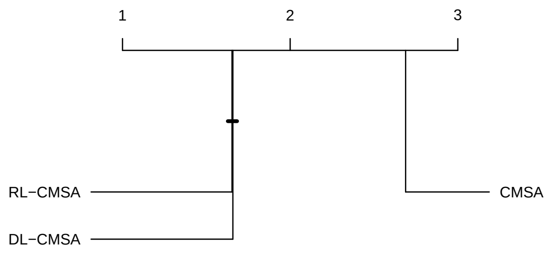
(a) All instances.



(b) Barabási-Albert instances.



(c) Erdös-Rényi instances.



(d) Watts-Strogatz instances.

Figure 9.4 CD plots concerning the MDS problem results.

The objective of the problem is to find the largest feasible solution.

Figure 9.5 illustrates two solutions to the MIS problem on the given graph. The solutions are valid as no pair of orange nodes are neighbors. Moreover, the solution on the right is optimal, since the graph admits no independent set of larger size.

9.4.2 Solution Components

For the set of solution components, we follow the same straightforward approach as in the MDS problem, defining C as the set of nodes in the input graph, i.e., $C = V$. As in the previous case, we will directly represent the solution components by the nodes themselves. At each construction step, the set of available solution components, denoted as C_{feas}, consists of the nodes that do not have any neighbors included in the current partial solution.

9.4.3 Probabilistic Solution Construction

As with the previously discussed problems, all CMSA variants utilize the same solution construction mechanism. This process begins with an empty solution, and at each step, a solution component is added by selecting it from the set of available solution components C_{feas}, defined previously.

The standard CMSA uses the following greedy function to choose a node from C_{feas} at each step of the construction process.

1. With a probability dr_{CMSA}, a node from C_{feas} with the least amount of neighbors is randomly selected.

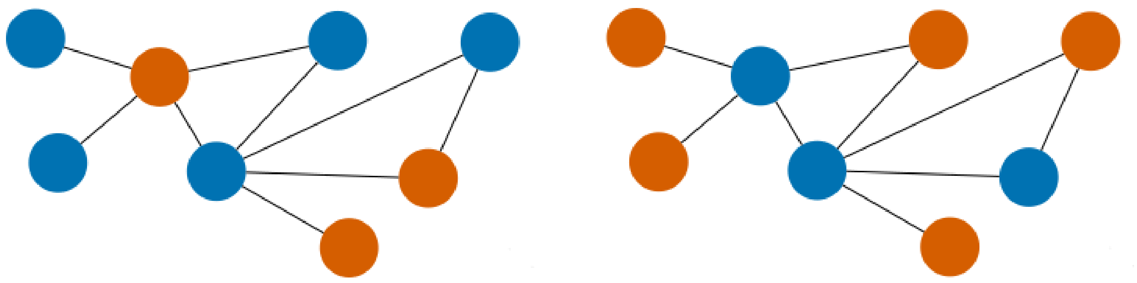


Figure 9.5 Example of an MIS problem instance and two solutions. The problem instance corresponds to the graph, and the nodes forming the two solutions are highlighted in orange.

2. Otherwise, a set L is created, containing the l_{size} nodes from C_{feas} with the fewest neighbors. A node is then chosen uniformly at random from L.

As with the previous problems, RL-CMSA and DL-CMSA do not utilize this approach, as they generate solutions using the quality value vector q and the neural network Q, respectively. In DL-CMSA, a one-hot encoding of the current partial solution is provided at each construction step. This encoding is based on the solution components, which, in this case, correspond to the nodes of the graph. Similarly to the MDS problem, an arbitrary ordering of the nodes is determined when the algorithm first processes the problem instance. This order is then consistently used for encoding the partial solutions.

9.4.4 ILP Model and Sub-Instance Solving

The following ILP model for the MIS problem is employed at each iteration, along with CPLEX, restricted to the sub-instance. It uses a binary variable x_i for each node $v_i \in V$, where x_i takes the value of one if v_i is included in the solution and zero otherwise.

$$\max \sum_{v_i \in V} x_i \tag{9.14}$$

$$\text{subject to} \quad x_i + x_j \leq 1, \qquad \text{for } e = (v_i, v_j) \in E \tag{9.15}$$

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

Constraint (9.15) ensures that no two neighbors are part of the solution. To apply this ILP model restricted to the sub-instance C', the following additional constraints are imposed: $x_i = 0$ for all $v_i \in C \setminus C'$.

9.4.5 Experimental Evaluation

The same graph models used for generating problem instances in the MDS problem were employed here as well: Barabási-Albert [7], Watts-Strogatz [127], and Erdös-Rényi [45]. In this case, the graph densities of the Barabási-Albert and Watts-Strogatz were varied across different graph sizes. As with the MDS problem, 30 graphs were generated for each graph type and for every combination of $|V| \in \{500, 1000, 1500, 2000\}$ and four distinct graph densities, resulting in a total of 480 graphs for each model. Additionally, one extra instance was also generated for each combination of graph type, density, and size for the purpose of parameter tuning.

In this case, the four corresponding density levels are achieved with $p \in \{0.01, 0.04, 0.07, 0.1\}$ for all Erdös-Rényi graph sizes, while for the Watts-Strogatz and Barabási-Albert graphs, the following values are used: $k, m \in \{3, 10, 17, 25\}$ for $|V| = 500$, $k, m \in \{5, 20, 35, 50\}$ for $|V| = 1000$, $k, m \in \{7, 30, 52, 75\}$ for $|V| = 1500$, and $k, m \in \{10, 40, 70, 100\}$ for $|V| = 2000$.

As with the previous problems, the algorithm variants are tuned using the *R* package *irace* [87]. Again, half of the tuning instances are used, with the instances corresponding to the 2nd and 3rd density levels excluded to speed up the procedure. This amounts to 24 tuning instances. A budget of 3000 algorithm runs was allocated to the tuning process of the three algorithms. The same time limits as in the case of the MDS problem were applied to each tuning and evaluation run, consisting of 150, 300, 450, and 600 seconds for instances of 500, 1000, 1500, and 2000 nodes, respectively.

Table 9.8 presents the parameters obtained through the tuning process. A similar trend to that observed for the MDS problem can be noted regarding the setting of t_{ILP}. Specifically, significantly smaller values are selected for the learning variants compared to the standard CMSA. This suggests a favorable performance of the learning-based approaches, as allocating less time to the exact solver emphasizes the role and effectiveness of the learning mechanisms.

Among the CPLEX parameters, the only one exhibiting a consistent pattern is $\text{cplex}_{\text{emphasis}}$, which is set to 1 in all cases. An additional noteworthy observation concerns the determinism rate of the standard CMSA, dr_{CMSA}, which is set to the minimum allowed value. This indicates that increased diversity during the solution construction phase benefits the standard approach.

The results are summarized in Tables 9.9 and 9.10, which follow the same structure as those presented for the MDS problem. As in the case of the latter, the learning variants consistently outperform the standard CMSA on average. Among the two learning-based approaches, their performance is comparable on

Table 9.8 Parameter values obtained after tuning the three CMSA variants for the MIS problem. Every algorithm is tuned exactly once.

	Allowed Range	CMSA	RL-CMSA	DL-CMSA
t_{ILP}	$\{1, 2, \ldots, 20\}$	20	9	9
n_a	$\{1, 2, \ldots, 50\}$	6	23	7
age_{max}	$\{1, 2, \ldots, 10\}$	3	1	3
$\text{cplex}_{\text{warmstart}}$	$\{0, 1\}$	0	1	0
$\text{cplex}_{\text{emphasis}}$	$\{0, 1\}$	1	1	1
$\text{cplex}_{\text{abort}}$	$\{0, 1\}$	1	1	0
dr_{CMSA}	$\{0.0, 0.01, \ldots, 0.99\}$	0.01	-	-
$l^{\text{size}}_{\text{CMSA}}$	$\{3, 4, \ldots, 100\}$	72	-	-
dr	$\{0.0, 0.01, \ldots, 0.99\}$	-	0.36	-
β	$\{0.0, 0.01, \ldots, 2.0\}$	-	0.73	-
b_{reset}	$\{0, 1\}$	-	1	-
cf_{limit}	$\{0.90, 0.91, \ldots, 1.0\}$	-	0.97	-
n_{layers}	$\{1, 2, 3\}$	-	-	1
n_{nodes}	$\{10, 11, \ldots, 1000\}$	-	-	254
lr	$\{0.001, 0.002, \ldots, 0.2\}$	-	-	0.199
ε_{dec}	$\{0.950.0.951, \ldots, 1\}$	-	-	0.959
ε_{min}	$\{0, 0.01, \ldots, 0.1\}$	-	-	0.05

the instances with 500 and 1000 nodes. However, for the larger instances with 1500 and 2000 nodes, DL-CMSA achieves superior solution quality.

Regarding computation time, DL-CMSA requires the most time to find its best solutions, which is consistent with the overhead introduced by its DL mechanism. Interestingly, the standard CMSA is the second slowest approach, despite performing significantly worse than RL-CMSA. This suggests that, although CMSA spends more time than RL-CMSA, it is less effective in utilizing this time to improve solution quality.

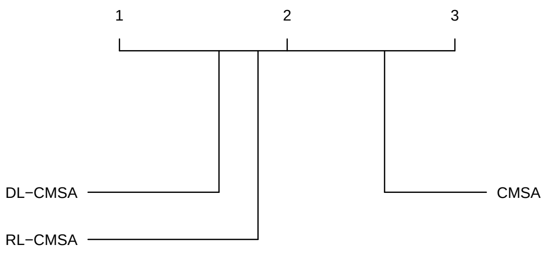
The CD plots for the MIS problem are presented in Figure 9.6. In this case, we organize the plots by the number of nodes, as this grouping reveals more insightful patterns. Figure 9.6a shows the CD plot for all instances combined, indicating that DL-CMSA is the best-performing approach, followed by RL-CMSA, with all pairwise differences between the three algorithms being statistically significant. Similar results are observed when grouping by graph type. Figures 9.6b–9.6e display the CD plots for graphs with 500, 1000, 1500, and 2000 nodes, respectively. As reflected in the numerical tables, the performance differences among algorithms are statistically significant for the larger instances (1500 and 2000 nodes), whereas RL-CMSA and DL-CMSA are statistically equivalent for the smaller instances (500 and 1000 nodes). In all cases, the standard CMSA is the worst-performing algorithm with statistical significance.

Table 9.9 Comparison of the three Construct, Merge, Solve, and Adapt variants for the Maximum Independent Set problem instances of size $|V| = 500$ and $|V| = 1000$

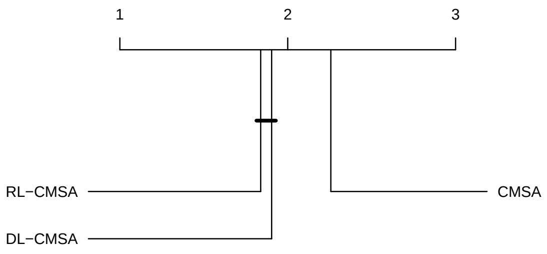
			CMSA		RL-CMSA		DL-CMSA	
$\|V\|$	Graph Type	Density	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
500	Barabási-Albert	1st	242.47	5.15	**242.53**	0.46	**242.53**	1.19
		2nd	148.30	34.98	**148.73**	24.95	147.80	92.81
		3rd	111.67	39.73	112.50	39.29	**112.53**	52.83
		4th	88.80	51.08	**89.43**	35.50	89.27	46.55
	Erdös Rényi	1st	217.90	21.77	**217.93**	21.74	217.73	62.96
		2nd	104.30	59.38	104.60	46.29	**104.80**	61.13
		3rd	71.67	58.07	71.97	68.78	**72.30**	76.84
		4th	54.97	67.00	55.03	57.80	**55.23**	81.87
	Watts-Strogatz	1st	174.90	18.67	175.03	27.46	**175.20**	38.90
		2nd	89.80	50.93	**90.20**	49.78	89.90	68.66
		3rd	62.70	71.57	**63.10**	55.73	62.97	86.87
		4th	46.93	59.72	**47.57**	60.92	47.30	74.32
1000	Barabási-Albert	1st	403.77	82.40	406.17	53.88	**406.23**	86.80
		2nd	204.90	139.43	208.63	99.14	**208.97**	147.18
		3rd	147.27	138.37	**149.17**	142.64	148.20	125.04
		4th	116.20	120.38	**116.93**	139.75	113.40	134.49
	Erdös Rényi	1st	311.83	172.03	314.20	106.12	**315.47**	161.51
		2nd	131.07	197.69	131.00	139.05	**131.40**	236.06
		3rd	83.83	198.28	85.30	156.04	**85.43**	235.78
		4th	63.17	176.71	**64.00**	156.86	63.70	217.28
	Watts-Strogatz	1st	269.20	153.01	270.20	102.55	**272.03**	230.96
		2nd	109.73	144.60	**111.87**	157.73	110.17	233.01
		3rd	72.33	204.80	73.50	141.08	**73.87**	223.63
		4th	54.70	186.87	**55.30**	128.91	55.03	193.99

Table 9.10 Comparison of the three Construct, Merge, Solve, and Adapt variants for the Maximum Independent Set problem instances of size $|V| = 1500$ and $|V| = 2000$

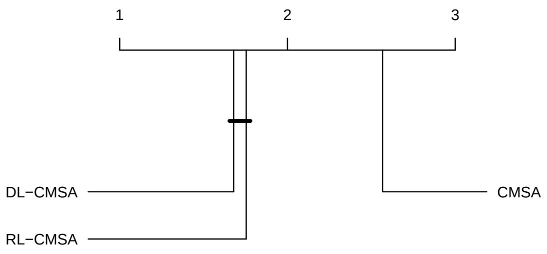
			CMSA		RL-CMSA		DL-CMSA	
$\lvert V\rvert$	Graph Type	Density	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
1500	Barabási-Albert	1st	522.93	219.24	529.23	138.89	**530.43**	250.63
		2nd	243.40	269.95	247.50	100.17	**250.17**	256.45
		3rd	172.37	213.55	173.17	148.83	**174.70**	279.92
		4th	**133.00**	214.62	132.63	155.81	132.80	238.52
	Erdös Rényi	1st	371.17	253.86	376.03	223.51	**377.87**	295.45
		2nd	142.67	293.73	142.53	176.77	**144.80**	345.35
		3rd	87.33	225.45	91.33	170.54	**92.47**	322.01
		4th	64.23	227.44	**68.40**	203.67	68.20	308.46
	Watts-Strogatz	1st	331.40	312.74	332.93	248.16	**334.33**	394.90
		2nd	119.57	259.25	123.10	233.47	**123.67**	351.93
		3rd	76.10	279.75	80.23	220.24	**80.30**	319.77
		4th	56.83	305.76	59.50	165.15	**59.60**	285.76
2000	Barabási-Albert	1st	591.23	399.37	604.13	220.43	**606.43**	444.02
		2nd	269.63	373.66	275.10	246.50	**278.50**	424.62
		3rd	188.20	325.21	**189.03**	245.82	181.43	318.55
		4th	145.47	331.82	143.50	208.51	**145.73**	387.46
	Erdös Rényi	1st	413.97	409.75	419.07	279.78	**422.33**	449.55
		2nd	147.17	291.38	151.30	274.14	**151.80**	452.41
		3rd	89.17	305.93	**95.63**	274.19	94.57	405.05
		4th	64.50	250.79	70.10	251.55	**70.50**	380.90
	Watts-Strogatz	1st	351.67	415.24	353.23	368.81	**359.17**	502.23
		2nd	123.67	297.47	129.60	271.44	**131.90**	487.21
		3rd	77.03	279.67	**84.63**	314.65	84.10	469.82
		4th	56.90	285.13	**62.97**	352.36	62.33	414.77



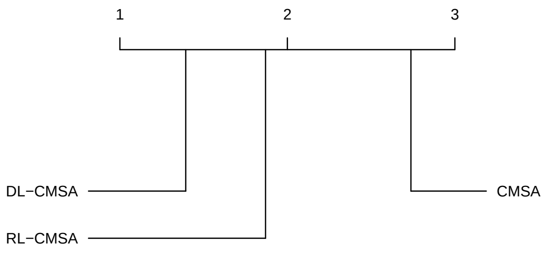
(a) All instances.



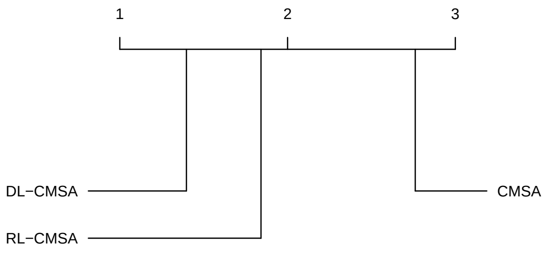
(b) Instances with 500 nodes.



(c) Instances with 1000 nodes.



(d) Instances with 1500 nodes.



(e) Instances with 2000 nodes.

Figure 9.6 CD plots concerning the MIS problem results.

9.5 In-Depth Algorithmic Analysis

This section analyzes the behavior of RL-CMSA and DL-CMSA, the two new CMSA variants introduced. Experimental results show that RL-CMSA consistently outperforms the standard CMSA across all three problems. DL-CMSA, in contrast, achieves the best performance on the MIS problem, performs comparably to RL-CMSA on the MDS problem, and significantly underperforms in comparison to the other variants on the FFMS problem. This section aims to explain the reasons behind these differences in performance.

For the MDS and MIS problems, four figures are included, based on data gathered from repeated experimental runs. These runs were conducted separately from the performance evaluations to ensure that the data collection process did not impact algorithm performance. The first two figures present data related to the solution construction process for certain selected instances. The first plot illustrates how the average quality of the solutions constructed in each iteration evolves over time, while the second tracks the evolution of the subinstance size prior to the *solve* step. Together, these plots offer insight into the learning dynamics of the algorithms, with the second plot additionally reflecting the diversity of the constructed solutions.

The remaining two figures analyze the computational overhead introduced by the learning mechanisms of RL-CMSA and DL-CMSA. These are shown as boxplots based on data collected from all problem instances, grouped by relevant instance characteristics. The first plot displays the average time overhead per iteration caused by the new *learning* steps and the modified *construct* steps, highlighting how this overhead varies across instance types. The second plot reports the average number of solution component selections made during the *learning* step, as well as the average number of solution components forming the CPLEX solution produced in the *solve* step. This plot helps to illustrate how the number of selected components influences overhead and how this behavior varies depending on instance characteristics. For the FFMS problem, the first three figures are also included; however, the fourth is replaced by a table, since the number of solution component selections in this case depends solely on the number of input strings in the problem instance.

9.5.1 FFMS Problem

The plots corresponding to the FFMS problem can be found in Figures 9.7 - 9.9. Remember that for this problem RL-CMSA performed best, followed by the standard CMSA and DL-CMSA and that the differences between the three

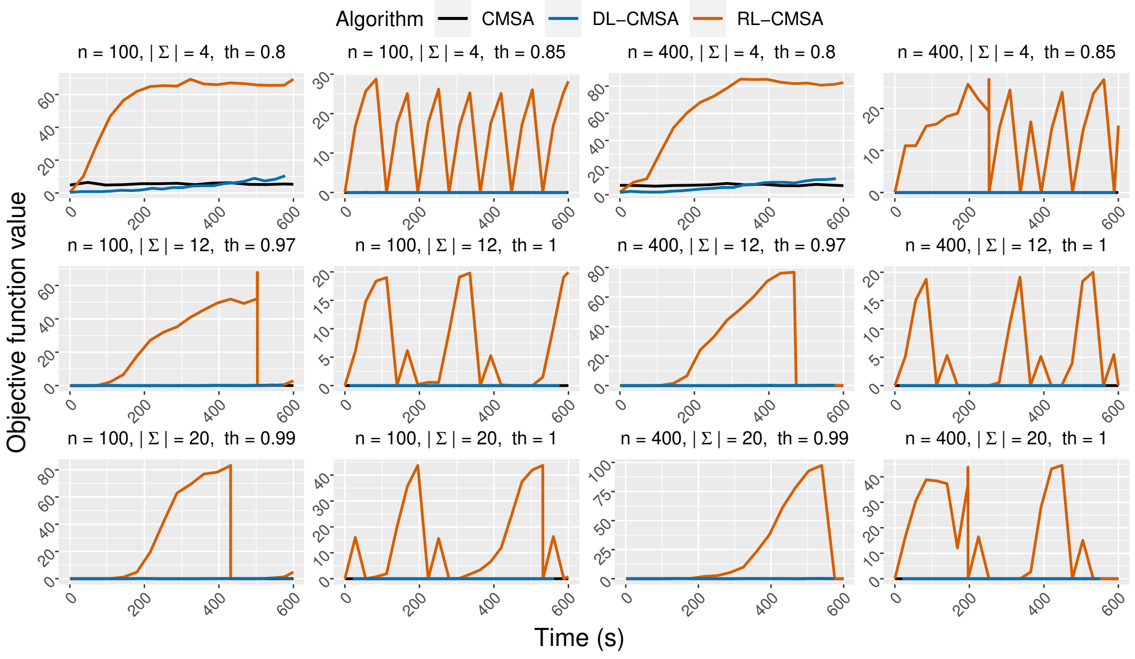


Figure 9.7 Evolution of the average quality of the constructed solutions per iteration over time for a set of selected FFMS problem instances.

algorithms were found to be statistically significant.

Figure 9.7 shows the evolution of the average quality of the solutions constructed in one iteration over time for a set of selected instances. For this problem, one instance for every combination of alphabet size $|\Sigma|$ and threshold *th* for a quantity of $n = 100$ and $n = 400$ strings of a length of $m = 500$ was selected.

The figure highlights a major limitation of the standard CMSA and DL-CMSA approaches: both consistently generate low-quality solutions across all instances. In most cases, the average objective function value of the solutions they construct remains at zero. Recall that for the FFMS problem, this value represents the number of input strings for which the solution string has a Hamming distance greater than *th*. Only two of the considered instances, those with $n = 100$, $|\Sigma| = 4$, $t = 0.85m$ and $n = 400$, $|\Sigma| = 4$, $t = 0.8m$, show DL-CMSA eventually producing solutions with an average objective value above zero. In these cases, a slow upward trend suggests some degree of learning.

In contrast, RL-CMSA demonstrates consistent learning behavior across all instances, as evidenced by steadily increasing objective values in its constructed solutions. Occasional drops in the curves reflect algorithm restarts. Furthermore, a distinct pattern emerges between low and high threshold instances, corresponding to the first and third columns versus the second and fourth columns of plots, respectively. This is probably due to the separate parameter tuning. For low-threshold instances, lower values for dr and β were

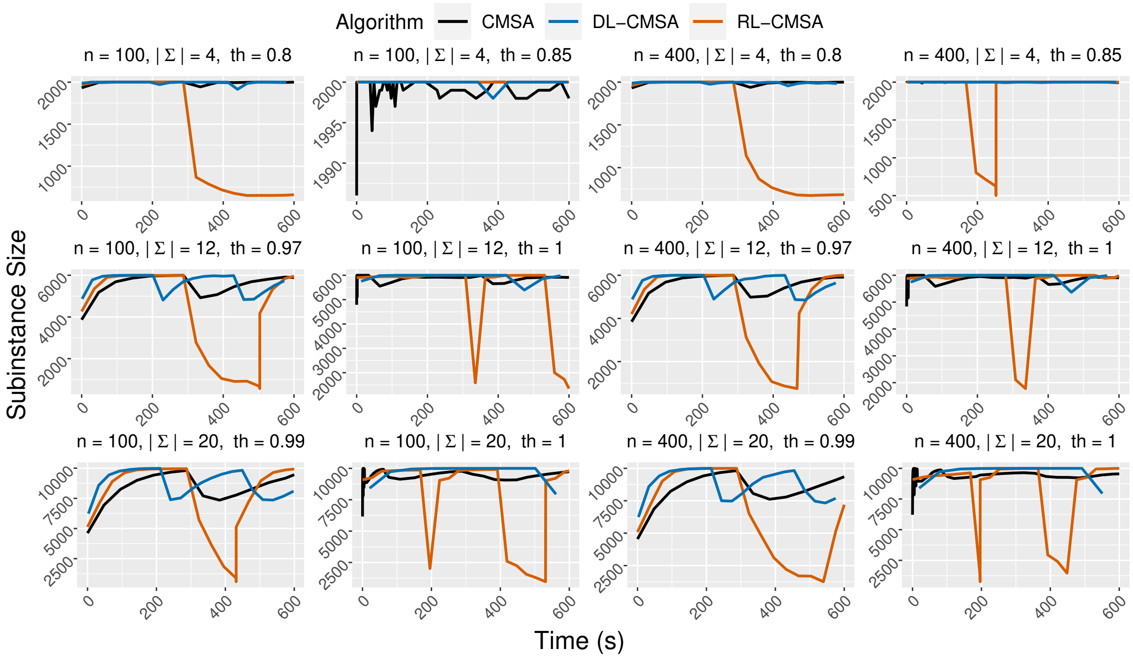


Figure 9.8 Evolution of the subinstance size over time for a set of selected FFMS problem instances.

selected, resulting in flatter learning curves compared to those for high-threshold settings.

Figure 9.8 presents the evolution of subinstance size over time for the three algorithms, using the same selected FFMS instances as in the previous figure. For both the standard CMSA and DL-CMSA, the subinstance size remains large and relatively stable across iterations, typically close to the full instance size of $500 \cdot |\Sigma|$. This behavior is consistent with their weaker performance, as applying the exact solver to such large subinstances limits its effectiveness.

In contrast, RL-CMSA in most cases exhibits a decreasing subinstance size over time, which facilitates the exact solver's ability to identify better-quality solutions. These reductions are periodically interrupted by sharp increases, indicating algorithm restarts. An exception is observed for the instance with $n = 100$, $|\Sigma| = 4$, and $t = 0.85$, where RL-CMSA's subinstance size remains high and constant. This anomaly results from frequent restarts preventing the subinstance from shrinking effectively. Specifically, the algorithm's maximum solution component age, $\text{age}_{\text{max}} = 7$, selected during tuning with *irace*, is relatively high compared to the total number of iterations (23) for this instance, limiting the opportunity for subinstance size reduction.

Figure 9.9 illustrates the average computational overhead introduced by RL-CMSA and DL-CMSA. The top part of the figure shows the overhead from modifications in the *construct* step, while the bottom part displays the overhead

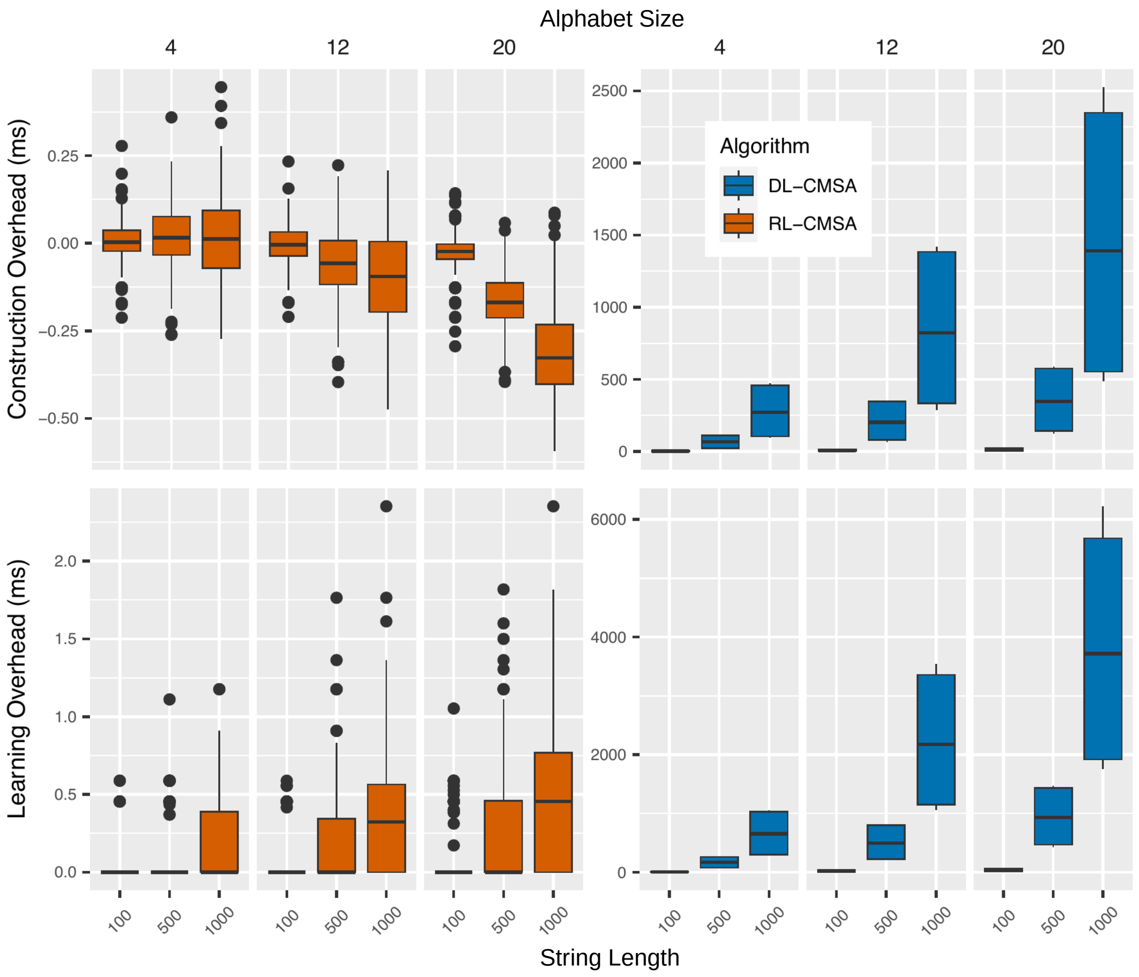


Figure 9.9 Average time overhead for (top) the construction of a solution, and (bottom) the update of the learning procedure for the FFMS problem.

from the *learning* step, both relative to the standard CMSA. Boxplots are provided for each, with instances grouped by alphabet size $|\Sigma|$ (indicated along the top axis) and string length m (shown on the bottom axis). Remember that, for RL-CMSA the *learning* step includes updating the q-values, computing the convergence factor, and executing a restart if convergence is detected. For DL-CMSA, it involves updating the neural network parameters, which requires a gradient descent step using Adam over the loss computed from the CPLEX selections, as well as reducing the exploration rate ε.

This plot highlights the main drawback of DL-CMSA: the significant overhead introduced in both its *construct* and *learning* steps. In both cases, the overhead is considerably higher than that of RL-CMSA and increases substantially with the alphabet size $|\Sigma|$ and the string length m. For example, for instances with $|\Sigma| = 20$ and $m = 1000$, the median overhead reaches nearly 1.5 seconds for the *construct* step and 4 seconds for the *learning* step. These high overheads, combined with the large CPLEX time limits selected by *irace*, hinder DL-CMSA's performance, as too few iterations are completed for effective learning. In contrast, RL-CMSA exhibits much lower overheads, and in some cases, even achieves negative overhead relative to the standard CMSA during the *construct* step. Its *learning* step introduces minimal overhead, with median values below 0.5 milliseconds even for the largest instances. As with DL-CMSA, RL-CMSA's overhead increases with $|\Sigma|$ and m, but remains far more manageable.

Table 9.11 Number of solution component selections for constructing a solution in both the *construct* and *solve* steps, and neural network input and output sizes for the FFMS problem.

m	$\|\Sigma\|$	Selections	Input and Output sizes
100	4	100	400
	12	100	1200
	20	100	2000
500	4	500	2000
	12	500	6000
	20	500	10000
1000	4	1000	4000
	12	1000	12000
	20	1000	20000

Table 9.11 reports the number of solution component selections made during the *construct* step and the number of components included in the CPLEX solution, along with the input and output sizes of the neural network. These values help explain the overheads discussed previously. Recall that, for the FFMS problem,

the solution components are position-character pairs, and constructing a solution requires one selection per string position. Consequently, the number of selections during construction is always equal to the string length m. The input and output sizes of the neural network correspond to the total number of solution components, given by $m \cdot |\Sigma|$. Both the number of selections and the neural network dimensions increase with m and $|\Sigma|$, which directly impacts runtime overhead. The number of selections determines how many forward passes the neural network must perform during construction, while the size of the network affects the duration of each pass. Additionally, one gradient descent step is performed in the *learning* step for every selection made by CPLEX, and the size of the network again heavily influences the cost of each step. These factors explain the substantial overheads observed in Figure 9.9, which grow rapidly as m and $|\Sigma|$ increase.

As we will see, in the MDS and MIS problems these overheads are much smaller, due to both fewer selections and significantly smaller network sizes. This difference accounts for DL-CMSA's poor performance on the FFMS problem relative to the other two.

9.5.2 MDS Problem

Figures 9.10–9.13 present the plots corresponding to the MDS problem. In this case, both RL-CMSA and DL-CMSA performed best, with no statistically significant difference between them.

As before, Figure 9.10 shows the evolution of the average quality of the solutions constructed by the three algorithms over time, for a selected set of instances. Since the MDS problem is a minimization problem, lower objective values (i.e., smaller dominating sets) indicate better solutions. The plots were generated with one instance of the third density level for every combination of graph type and graph size $|V|$.

The behavior of RL-CMSA closely resembles the one it displayed for the FFMS problem, with clear signs of learning and periodic restarts. DL-CMSA also demonstrates learning, as the average size of its constructed solutions steadily decreases over time, though at a slower rate than RL-CMSA. This difference may be attributed to the values of their respective parameters, dr and β for RL-CMSA, and ε_{dec} and lr for DL-CMSA.

The standard CMSA, in contrast, shows no improvement over time, as expected from a variant that lacks a learning mechanism. However, unlike in the FFMS problem, it constructs solutions of relatively good quality compared to the learning variants.

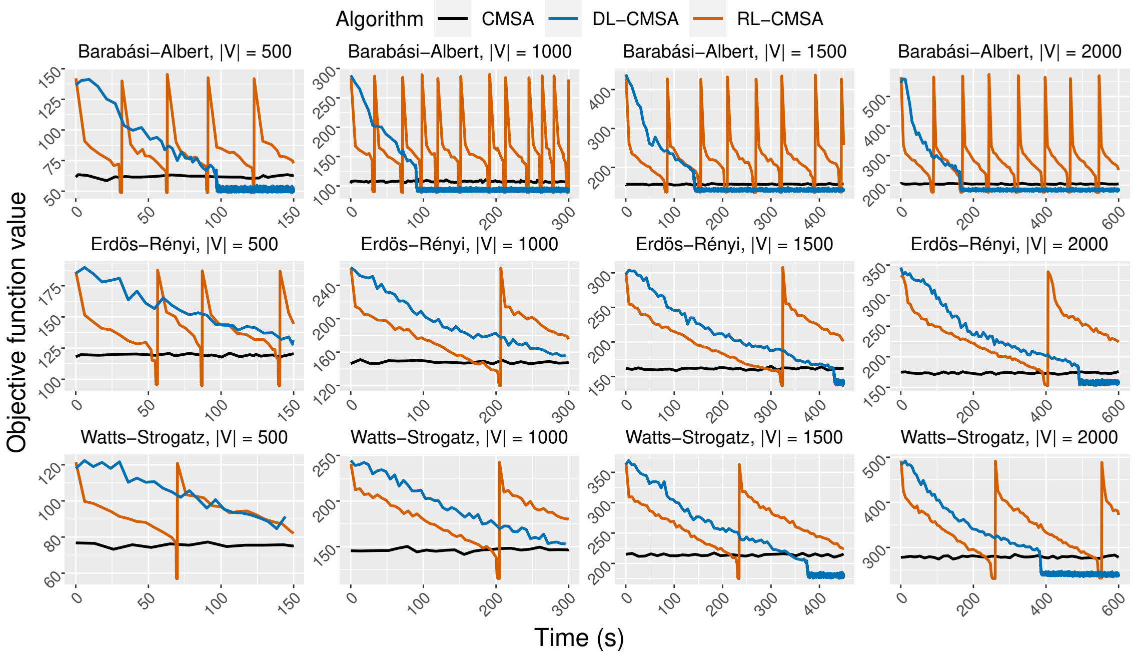


Figure 9.10 Evolution of the average quality of the constructed solutions per iteration over time for a set of selected MDS problem instances.

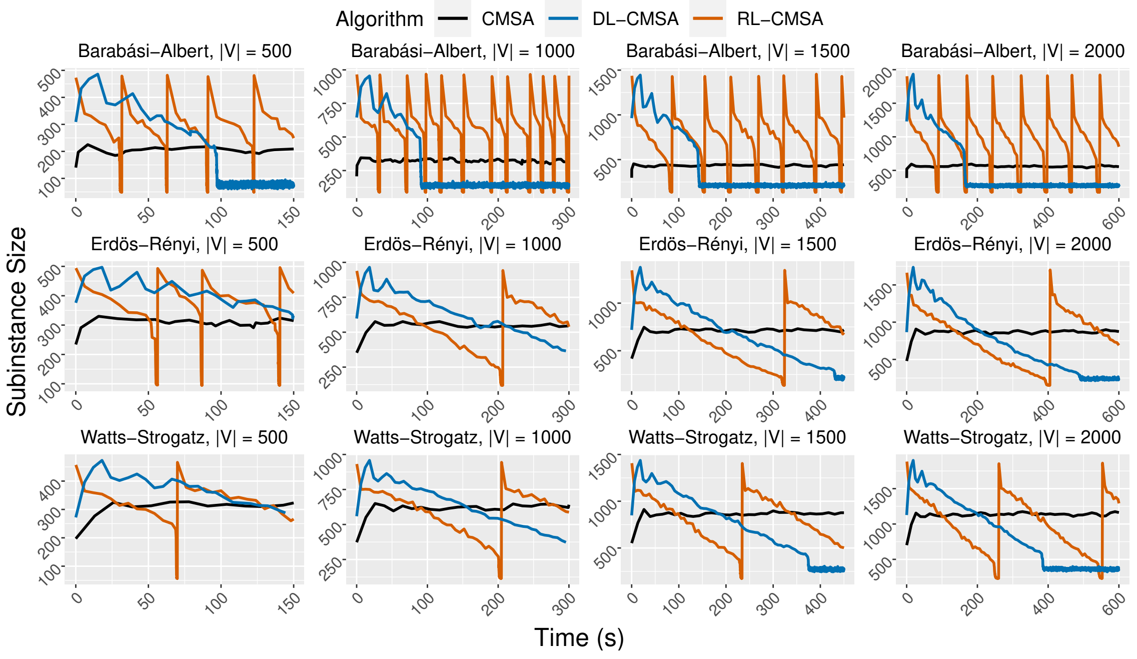


Figure 9.11 Evolution of the subinstance size over time for a set of selected MDS problem instances.

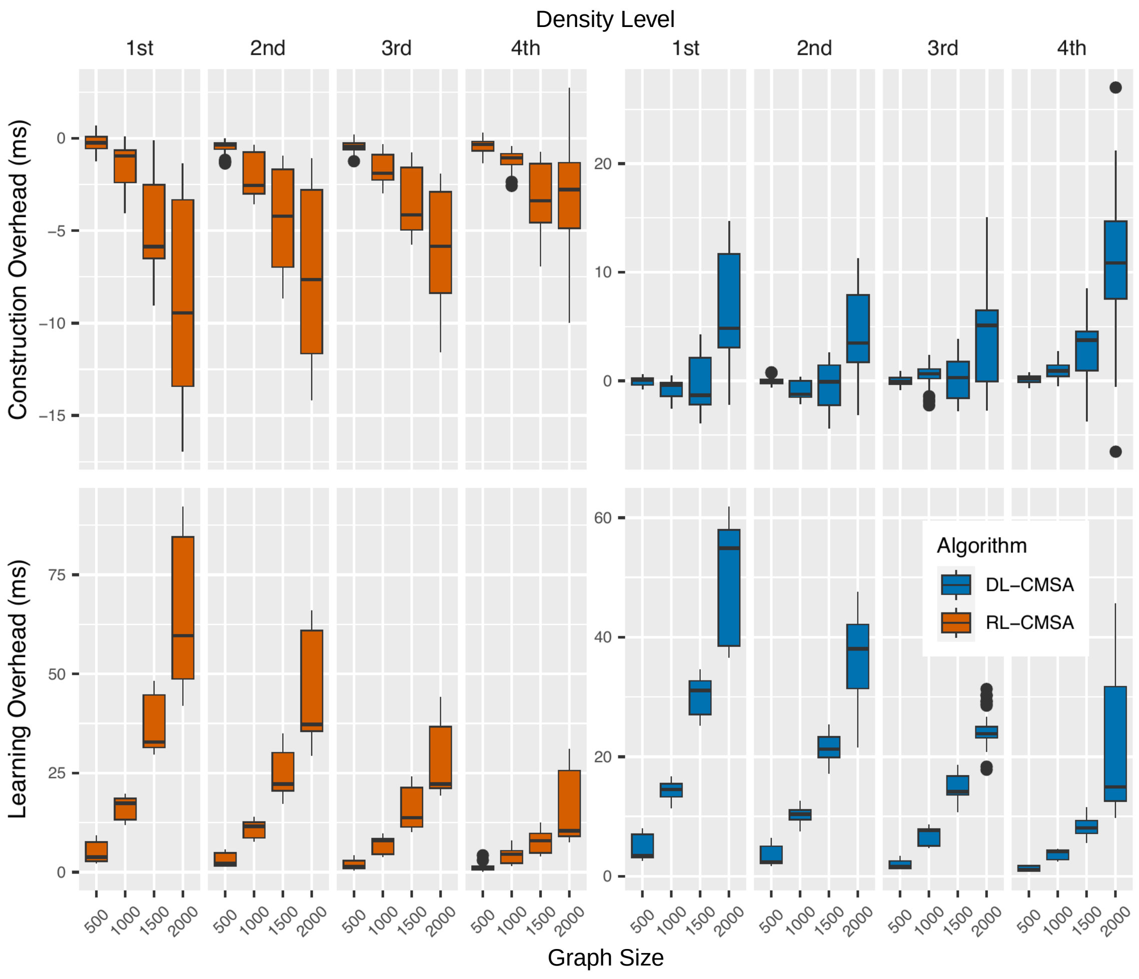


Figure 9.12 Average time overhead for (top) the construction of a solution, and (bottom) the update of the learning procedure for the MDS problem.

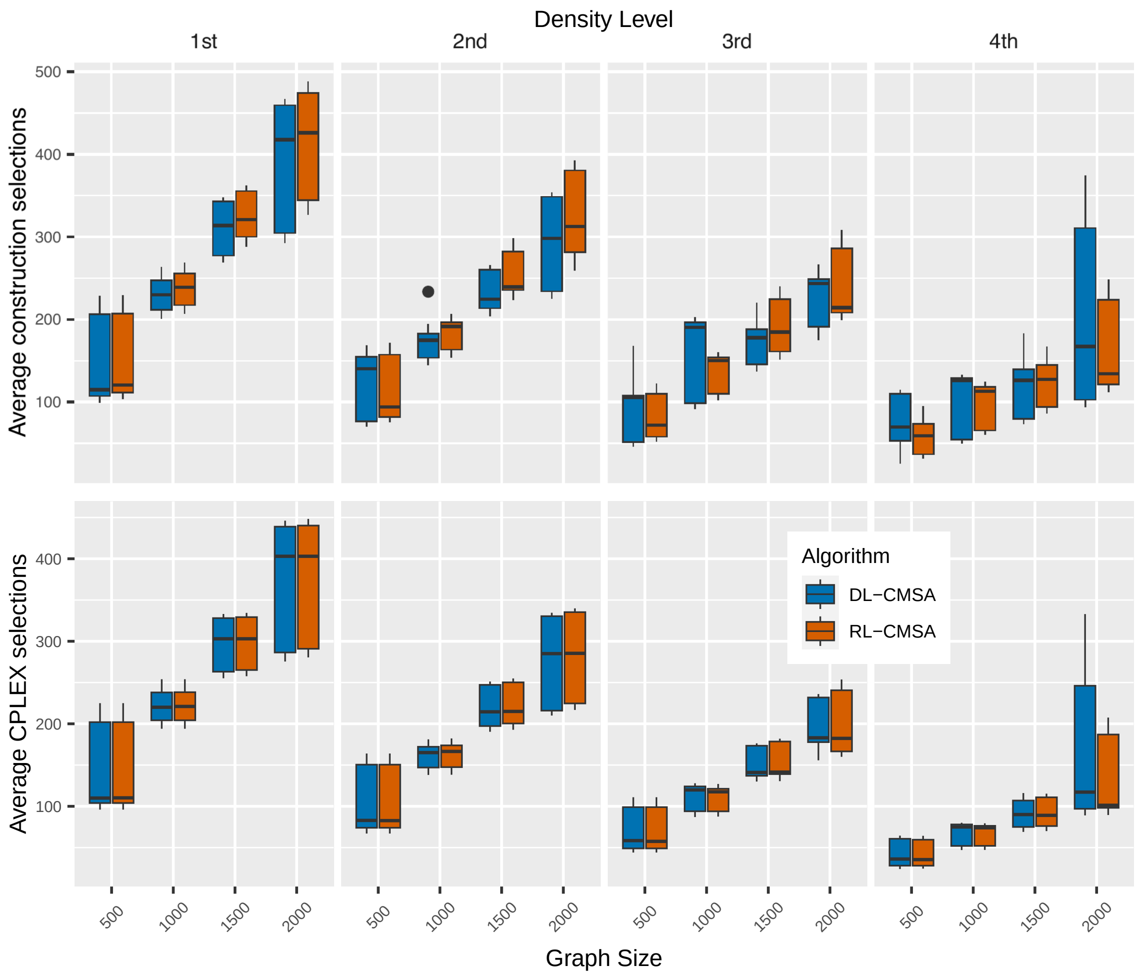


Figure 9.13 Average solution component selections for (top) the construction of a solution, and (bottom) done by CPLEX for the MDS problem.

A pattern emerges for Barabási–Albert instances: RL-CMSA converges quickly and restarts multiple times during execution, while DL-CMSA also converges faster than on other graph types. This behavior can be attributed to the structure of Barabási–Albert graphs, which typically include a few high-degree nodes and many low-degree ones. The learning variants rapidly identify the importance of high-degree nodes, as CPLEX consistently selects them in the *solve* step.

One further observation concerns DL-CMSA's performance on instances with 500 and 1000 nodes. For these, the quality of its constructed solutions generally falls short of RL-CMSA's, except in the easier Barabási–Albert cases. This suggests that the time limits used in the experiments may have been too restrictive for DL-CMSA on smaller instances, and that its performance could potentially improve with longer time budgets.

Figure 9.11 illustrates the evolution of subinstance size over time for the selected MDS instances. These plots largely mirror the trends observed in Figure 9.10, as reductions in subinstance size tend to coincide with improvements in the quality of constructed solutions.

This correlation arises because, as learning progresses, the algorithm increasingly favors a smaller subset of frequently selected solution components, leading to more compact subinstances.

A notable observation is that the standard CMSA tends to maintain larger subinstances than the learning variants, even when achieving similar or better solution quality. The rate at which subinstance sizes decrease also differs between RL-CMSA and DL-CMSA. For RL-CMSA, the decrease is more pronounced, which can be attributed to *irace* selecting a low maximum age parameter ($\text{age}_{\text{max}} = 1$). In contrast, DL-CMSA exhibits a more gradual reduction in subinstance size, consistent with a higher age_{max} value of 4. These settings influence how quickly older, less frequently selected components are removed, directly affecting the evolution of subinstance size.

Figure 9.12 shows the average overhead introduced by RL-CMSA and DL-CMSA compared to the standard CMSA on the MDS problem. The instances are grouped by density level (top labels) and graph size (bottom labels). The most prominent observation is the significantly lower overheads for DL-CMSA compared to what was observed for the FFMS problem, an important factor contributing to its improved relative performance here. Specifically, for DL-CMSA, the overhead from the modified solution construction process typically remains under 20ms, while the overhead from the new learning step is mostly below 60ms.

For RL-CMSA, the average overhead during solution construction is often negative, meaning it tends to construct solutions faster than the standard CMSA. This is due to the slow greedy function employed by the standard CMSA for this problem, which requires sorting the available solution components depending on their number of uncovered neighbors at every construction step. Interestingly, the learning overhead for RL-CMSA is comparable to, and in some cases slightly higher than, that of DL-CMSA across several instance groups.

Figure 9.13 presents the average number of solution components selected during construction and the average number included in the CPLEX-generated solutions. For the MDS problem, these values correspond to the average sizes of the constructed and CPLEX solutions, respectively. Compared to the FFMS problem, the number of selections is lower, and more importantly, the neural network used by DL-CMSA has considerably smaller input and output sizes, as they correspond to the total number of solution components which is the graph size for this problem. This reduced model size, along with a more modest architecture selected by *irace*, contributes directly to DL-CMSA's lower time overheads in this setting.

Lastly, the number of selections decreases with increasing density (since denser graphs allow smaller dominating sets) and increases with graph size, reflecting the natural scaling behavior of the problem.

9.5.3 MIS Problem

The plots for the MIS problem correspond to Figures 9.14 – 9.17. For this problem, DL-CMSA was the superior algorithm, followed by RL-CMSA, with the differences between the three algorithms being statistically significant.

Figure 9.14 illustrates the average solution construction quality per iteration for a selection of MIS instances, while Figure 9.15 shows the corresponding evolution of subinstance size over time. The chosen instances are some of those for which DL-CMSA outperformed RL-CMSA, aiming to better understand the factors behind DL-CMSA's superior performance on this problem. In the presented runs, DL-CMSA achieved solutions with sizes approximately 2.12% to 5.88% larger than those found by RL-CMSA.

A key observation is that, for some instances, DL-CMSA attains a higher maximum average solution quality within an iteration compared to RL-CMSA, something not observed for the MDS problem, where DL-CMSA's peak iteration quality was typically similar to or below that of RL-CMSA. However, this alone does not fully explain DL-CMSA's stronger overall performance. For most 1500-node instances, DL-CMSA does not outperform RL-CMSA in terms of maximum average solution construction quality, yet still delivers better overall

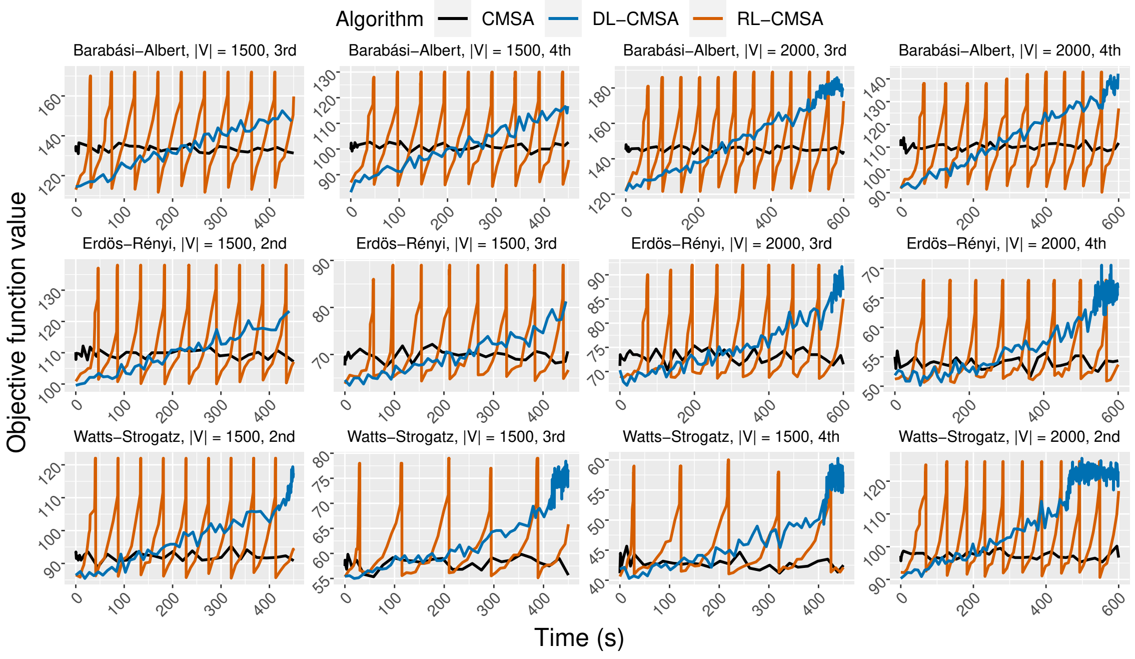


Figure 9.14 Evolution of the average quality of the constructed solutions per iteration over time for a set of selected MIS problem instances.

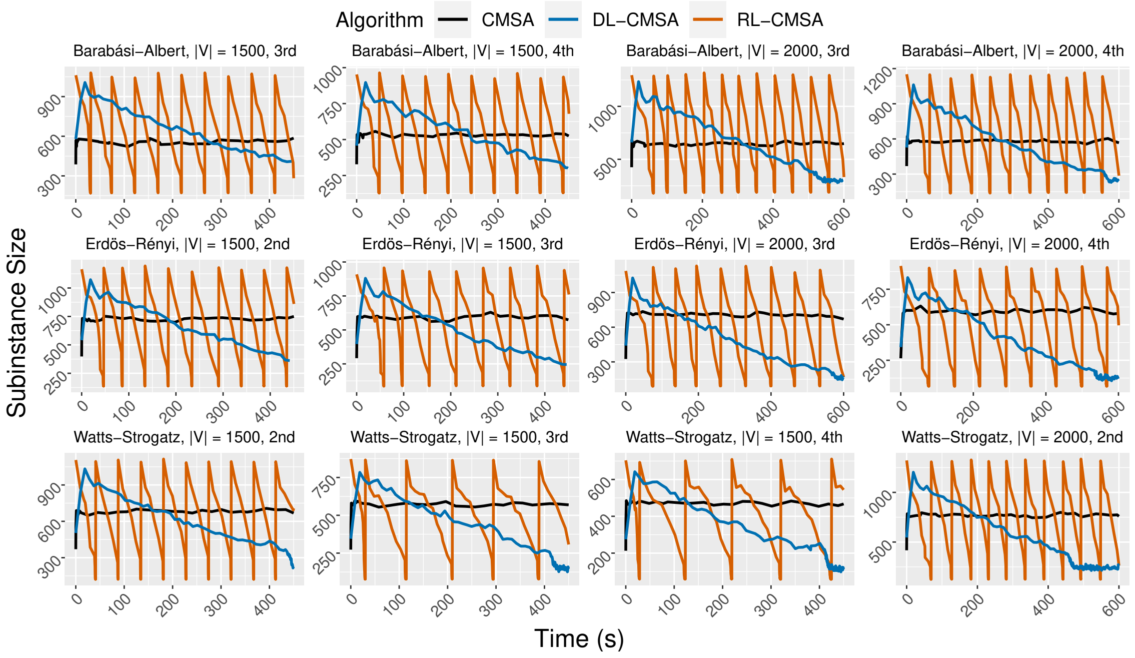


Figure 9.15 Evolution of the subinstance size over time for a set of selected MIS problem instances.

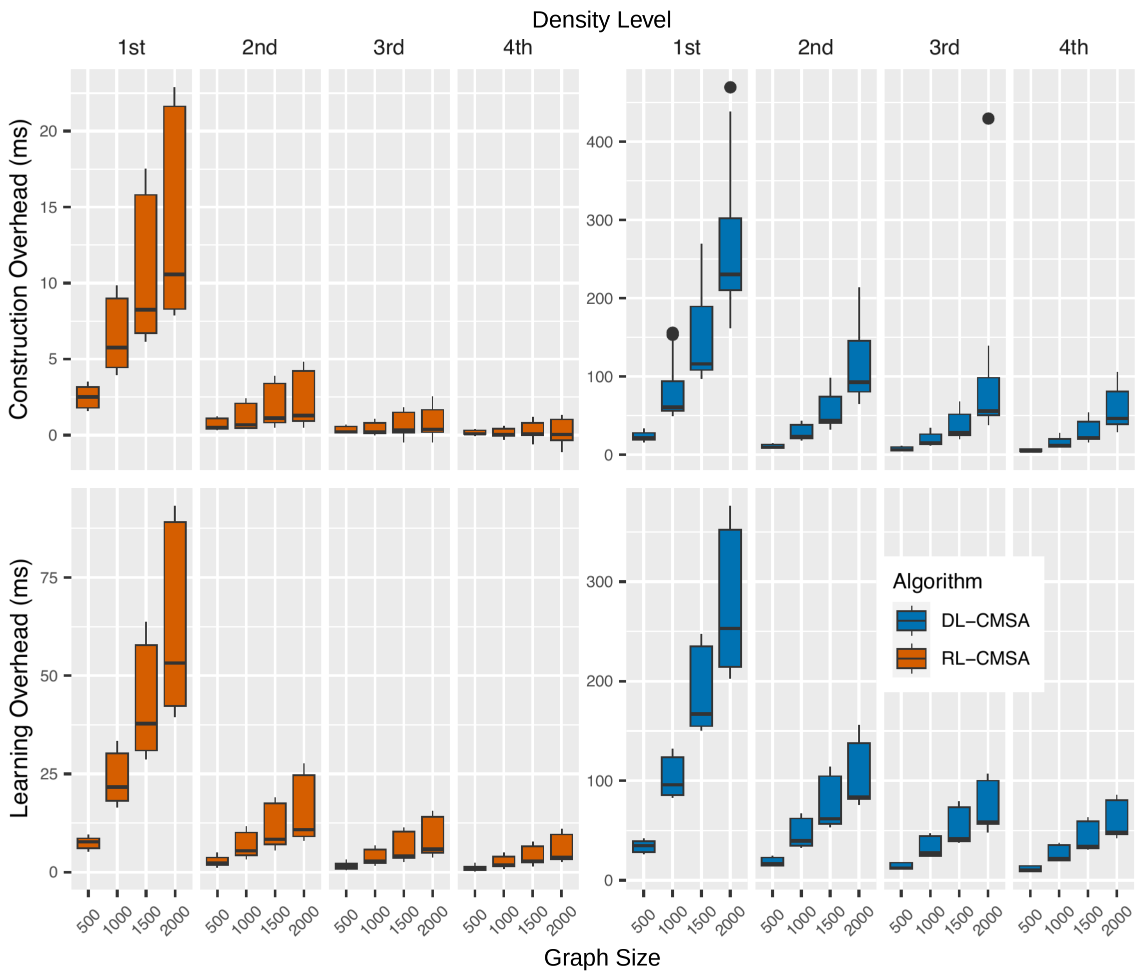


Figure 9.16 Average time overhead for (top) the construction of a solution, and (bottom) the update of the learning procedure for the MIS problem.

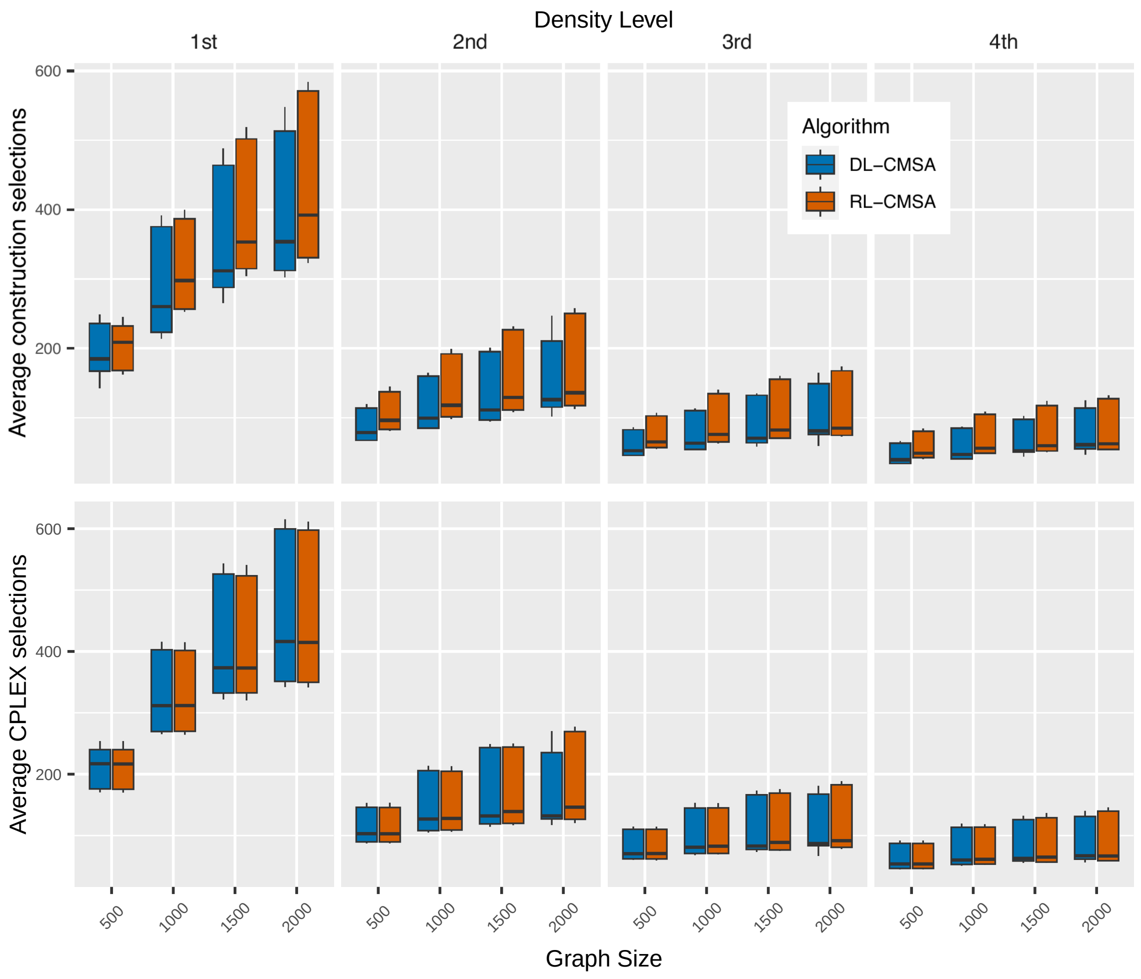


Figure 9.17 Average solution component selections for (top) the construction of a solution, and (bottom) done by CPLEX for the MIS problem.

results.

A notable difference appears in the learning patterns of the two algorithms. RL-CMSA exhibits steep increases in solution quality and sharp reductions in subinstance size, indicating that it tends to concentrate its best constructions within just a few iterations. In contrast, DL-CMSA tends to maintain a broader range of iterations where it constructs diverse high-quality solutions. This diversity is highly advantageous for the *solve* step, as it provides CPLEX with richer subinstances. This might be one of the main reasons behind the better performance of DL-CMSA for this problem.

Importantly, the improved performance of DL-CMSA does not stem from differences in the selection mechanisms. RL-CMSA was re-tuned and evaluated using a greedy epsilon-selection strategy, and the performance gap persisted.

As with the MDS problem, DL-CMSA appears to suffer from tight time constraints on smaller instances (500, 1000, and 1500 nodes), where it fails to match the solution construction quality of RL-CMSA. However, for larger instances (2000 nodes), DL-CMSA not only matches but sometimes surpasses RL-CMSA in terms of average construction quality, suggesting that its performance could further improve with extended time limits.

Figure 9.16 presents the computational overhead of the learning variants relative to the standard CMSA for the MIS problem. Compared to the MDS problem, DL-CMSA exhibits noticeably higher overheads, despite the average number of selections and the neural network input/output sizes being similar. This discrepancy is likely due to *irace* selecting a larger neural network architecture for the MIS problem, which increases the cost of both forward passes during the *construct* step and the gradient descent updates in the *learning* step.

RL-CMSA also incurs greater overheads during solution construction for the MIS problem than for the MDS, while its learning step overhead remains comparable. This difference arises from the distinct solution construction mechanisms used by the standard CMSA for the two problems. Specifically, the greedy functions differ: as already mentioned, in MDS constructing a solution requires sorting available components by their number of uncovered neighbors at each iteration, a computation that depends on the current partial solution. In contrast, MIS uses a simpler greedy function, the number of neighbors, which remains static throughout construction. This makes MIS constructions inherently faster for the standard CMSA, thereby amplifying the relative overhead of RL-CMSA's construction phase for this problem.

As expected, overheads increase with graph size and decrease with density, which can be explained by the average number of selections performed, shown in

Figure 9.17. Notably, the reduction in selections with increasing density is more pronounced for MIS than for MDS, while the increase in selections with graph size is less steep.

Chapter 10

CONCLUSIONS AND FUTURE WORK

10.1 Conclusions

This thesis contributed to the growing and promising field of integrating Machine Learning (ML) techniques into combinatorial optimization. Specifically, it explored enhancements to metaheuristics, versatile search frameworks that can be adapted to a wide range of optimization problems. The work was structured in two main parts. The first part focused on offline learning approaches, where ML models are trained prior to the execution of the optimization algorithm and then applied during its runtime. The second part addressed online learning methods, in which the models are continuously updated and improved as the optimization algorithm progresses.

The offline learning part of this thesis focused on implementing a general framework for learning search components within metaheuristics. This framework involves parametrizing a metaheuristic's search component using an ML model, which is trained via a Genetic Algorithm (GA). During this training process, the GA maintains a population of parameter configurations, which are evolved over generations guided by two sets of full-sized problem instances used for evaluation. This approach was applied to two distinct metaheuristics targeting two variants of the well-known Longest Common Subsequence (LCS) problem: a Biased Random Key Genetic Algorithm (BRKGA) for the Longest Common Square Subsequence (LCSqS) problem, and a Beam Search (BS) for the Restricted Longest Common Subsequence (RLCS) problem. Additionally, it was applied to the Clarke and Wright heuristic for two variants of the Electric Vehicle Routing Problem (EVRP).

In all cases, applying the framework resulted in improved performance. The first application was evaluated on two sets of instances: one with strings generated uniformly at random, and another with strings containing implanted patterns, which were more challenging. For the simpler instances, the offline learning approach did not yield improvements, as a basic heuristic outperformed the learned model in both speed and quality. However, on the more difficult instances, the learning-based approach achieved significant gains, as the ML model was able to capture patterns beyond the reach of the simpler heuristic.

In the second application, also two distinct sets of problem instances were used. Across both, the learning-enhanced BS outperformed its standard counterpart on average, not only in solution quality but also in execution time. This improvement was due to the fact that the original BS relied on a complex heuristic, which scaled poorly compared to the lightweight neural network used in the learned version. One of the key advantages of offline learning was highlighted here: although training incurs a cost upfront, it can ultimately lead to faster algorithms at runtime, depending on the complexity of the baseline heuristics. For the first benchmark set, performance improvements grew with the length of the input strings. For the second set, a trend emerged showing the standard algorithm outperforming the learned variant when the number of input strings became large.

In the last application, two variants of the EVRP were used for evaluation: the Capacitated Electric Vehicle Routing Problem (CEVRP) and the Electric Vehicle Routing Problem with Road Junctions and Road Types (EVRP-RJ-RT). A realistic benchmark set of problem instances based on European cities was used in both cases. The learning Clarke and Wright heuristic achieved a better average performance in minimizing the number of vehicles required compared to the standard Clarke and Wright for both problems. This is thanks to the flexibility of the learning approach, which can be trained to minimize the number of vehicles, while the standard algorithm is specifically designed to minimize energy consumption and cannot be adapted for minimizing the number of vehicles in a straightforward way. This application also presented interesting differences in execution time: for the CEVRP, the standard Clarke and Wright presented much faster executions, while it was the other way around for the EVRP-RJ-RT. This is because the road junctions present in the EVRP-RJ-RT affect the standard heuristic much more than the learning variant.

Overall, this part of the thesis demonstrated the effectiveness of offline learning in enhancing metaheuristics. Despite the need for training, the resulting models were able to improve both performance and efficiency in many scenarios.

The second part of this thesis introduced two new variants of the Construct, Merge, Solve, and Adapt (CMSA) metaheuristic from the literature. Applying CMSA to a combinatorial optimization problem requires defining a set of solution components, C, such that any feasible solution can be expressed as a subset of C (for example, the set of graph edges in the case of the Traveling Salesman

Problem). CMSA maintains a subinstance of the problem, defined as a subset of C, and applies an exact solver to it at each iteration. This subinstance is dynamically modified throughout the search process with the goal that the exact solver will eventually discover a high-quality solution in it.

The subinstance is expanded during the *construct* and *merge* steps, where a number of solutions to the full problem are probabilistically generated and their corresponding components are added to the subinstance. Conversely, it is reduced during the *adapt* step by removing components that have not appeared in the exact solver's solution over a given number of iterations. In standard CMSA implementations, solution construction in the *construct* step is typically performed using a problem-specific greedy probabilistic heuristic, designed to ensure diverse and reasonably good solutions.

The two proposed variants, named RL-CMSA and DL-CMSA, replace this heuristic with online learning mechanisms that adapt solution construction based on feedback from the exact solver. RL-CMSA maintains a quality score for each solution component, which is updated depending on its presence in the exact solver's solution. At each iteration, components used in the solver's solution have their quality scores increased, while those present in the subinstance but not in the solution see their scores decreased. New solutions are constructed by sampling components with probabilities biased by these quality scores, encouraging the use of high-performing components.

DL-CMSA extends this idea by incorporating the partial solution under construction at each step. It replaces the static quality vector with a neural network, which takes the current partial solution as input and outputs a value for each component, representing its estimated usefulness in extending the current solution. This allows DL-CMSA to dynamically adapt its decisions based on solution context, rather than relying solely on static quality estimates.

The performance of RL-CMSA and DL-CMSA was evaluated on three combinatorial optimization problems: the Far From Most String (FFMS) problem, the Minimum Dominating Set (MDS) problem, and the Maximum Independent Set (MIS) problem. RL-CMSA consistently outperformed the standard CMSA, achieving statistically significant improvements across all three problems. DL-CMSA performed worse than the standard CMSA on the FFMS problem but outperformed it on the MDS and MIS problems. Compared to RL-CMSA, DL-CMSA achieved comparable results on the MDS problem and a statistically significantly better performance on the MIS problem.

The detailed analysis of algorithm behavior highlighted the reasons behind these performance differences. For the FFMS problem, DL-CMSA incurred substantial computational overhead from its neural network–based learning, which scaled poorly with string length and alphabet size. As a result, too few iterations were completed for effective learning, and the algorithm constructed mostly low-quality solutions. In contrast, RL-CMSA maintained low overhead and exhibited clear learning dynamics, with shrinking subinstances that enabled the solver to identify better solutions.

On the MDS problem, both learning variants benefited from smaller model sizes and more favorable scaling, keeping overheads modest. Here, RL-CMSA reduced subinstances more aggressively, while DL-CMSA achieved slower but steady improvements, leading to comparable overall performance.

For the MIS problem, DL-CMSA proved most effective. Although its overhead was higher than on the MDS, it maintained greater diversity in the high-quality solutions it constructed, providing richer subinstances for the solver. This broader exploration enabled it to outperform both RL-CMSA and the standard CMSA, especially on larger instances where time budgets allowed its learning mechanism to take full effect.

Overall, this part of the thesis demonstrated the effectiveness and possible risks of integrating online learning mechanisms into metaheuristics. While these approaches eliminate the need for prior training, they often incur considerable computational overhead, which may become a limiting factor in practice, depending on the problem and implementation.

10.2 Future Work

As discussed in the previous section, this thesis primarily investigated the integration of ML techniques into metaheuristics, both from offline and online perspectives. The emphasis was on the integration process rather than on optimizing the ML components themselves. For this reason, all models employed were simple feed-forward neural networks. This design choice leaves ample room for future research in which more sophisticated ML models are considered.

Regarding the offline learning part, one promising direction is to explore models that do not rely on handcrafted features. Feature engineering can be time-consuming, requires domain expertise, and risks discarding useful information. For the two string-related problems addressed here, the LCSqS and the RLCS, future work could leverage models designed to process text directly. Potential candidates include recurrent neural networks (RNNs) [57], convolutional neural networks (CNNs) [74] for sequences, or more recent transformer-based architectures such as BERT [31] or GPT [101] variants. These models could learn representations of strings end-to-end, potentially capturing structural similarities and repetitions more effectively than handcrafted features. Transfer learning from pre-trained language models might also reduce data requirements in this context.

For the Clarke and Wright heuristic, Graph Neural Networks (GNNs) present another natural extension. A GNN applied once to the graph representing the Vehicle Routing Problem (VRP) instance at hand could generate node embeddings encoding global spatial and cost structure. These embeddings could then be combined, along with pooled route embeddings, by a neural network to predict savings values for candidate merges. An alternative is a two GNN setup: one GNN encoding the global instance structure and another encoding the local subgraph corresponding to the routes under evaluation. Such models could capture both high-level problem context and detailed route interactions, reducing reliance on handcrafted savings features.

In the online setting, additional model classes could be explored within DL-CMSA. In particular, the choice of model could be tailored to the problem domain. For example, for graph-structured problems, a GNN could be used to represent the partial solution under construction, enabling the model to exploit structural regularities during the construction phase. Similarly, sequence models might be better suited for string-based problems.

Finally, both the offline learning framework and the online CMSA variants should be applied to a broader range of combinatorial optimization problems. Thus far, RL-CMSA and DL-CMSA were benchmarked on three rather academic problems, with the goal of comparison against the standard CMSA. A natural next step is to evaluate them on more realistic problems where CMSA has already demonstrated strong performance, such as in VRP variants and scheduling problems [13]. Such applications would allow testing the scalability and robustness of the learning-augmented approaches and would provide insights into their practical value.

Bibliography

[1] Mehmet Anil Akbay and Christian Blum. EVRPGen: A web-based instance generator for the electric vehicle routing problem with road junctions and road types. *Software Impacts*, 2025. doi:10.1016/j.simpa.2025.100778.

[2] Mehmet Anıl Akbay, Albert López Serrano, and Christian Blum. 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. doi:10.1007/s44196-022-00098-1.

[3] Mehmet Anıl Akbay, Christian Blum, and Can Berk Kalayci. Application of Adapt-CMSA to the Electric Vehicle Routing Problem with Simultaneous Pickup and Deliveries. In *International Conference on Computer Aided Systems Theory*, pages 90–106. Springer, 2024. doi:10.1007/978-3-031-82949-9_9.

[4] Mehmet Anıl Akbay, Christian Blum, and Can Berk Kalayci. CMSA based on set covering models for packing and routing problems. *Annals of Operations Research*, 343(1):1–38, 2024. doi:10.1007/s10479-024-06295-9.

[5] Mehmet Anil Akbay, Christian Blum, and Michella Saliba. The Electric Vehicle Problem with Road Junctions and Road Types: An Ant Colony Optimization Approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1454–1462, 2024. doi:10.1145/3638529.3653997.

[6] Claus Aranha, Christian L. Camacho Villalón, Felipe Campelo, Marco Dorigo, Rubén Ruiz, Marc Sevaux, Kenneth Sörensen, and Thomas Stützle. Metaphor-based metaheuristics, a call for action: the elephant in the room. *Swarm Intelligence*, 16(1):1–6, 2022. doi:10.1007/s11721-021-00202-9.

[7] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

[8] James C. Bean. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA Journal on Computing*, 6(2):154–160, 1994.

[9] Tolga Bektaş and Gilbert Laporte. The Pollution-Routing Problem. *Transportation Research Part B: Methodological*, 45(8):1232–1250, 2011. doi:10.1016/j.trb.2011.02.004.

[10] Richard Bellman. Dynamic Programming. *Science*, 153(3731):34–37, 1966. doi:10.1126/science.153.3731.34.

[11] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. doi:10.1016/j.ejor.2020.07.063.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006. ISBN 978-0-387-31073-2.

[13] Christian Blum. *Construct, Merge, Solve & Adapt*. Springer Cham, 2024. doi:10.1007/978-3-031-60103-3.

[14] Christian Blum and Marco Dorigo. The Hyper-Cube Framework for Ant Colony Optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2):1161–1172, 2004. doi:10.1109/TSMCB.2003.821450.

[15] Christian Blum and Paola Festa. *Metaheuristics for String Problems in Bio-informatics*. John Wiley & Sons, 2016. doi:10.1002/9781119136798.

[16] Christian Blum and Pedro Pinacho-Davidson. Application of Negative Learning Ant Colony Optimization to the Far From Most String Problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar)*, pages 82–97. Springer, 2023. doi:10.1007/978-3-031-30035-6_6.

[17] Christian Blum and Jaume Reixach. The Hybrid Metaheuristic CMSA. In Rafael Martí, Panos M. Pardalos, and Mauricio G.C. Resende, editors, *Handbook of Heuristics*. Springer Nature Switzerland, Cham, 2025. ISBN 978-3-319-07153-4. doi:10.1007/978-3-319-07153-4_79-1.

[18] Christian Blum and Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35:268–308, 2001. doi:10.1145/937503.937505.

[19] Christian Blum, Maria J. Blesa Aguilera, Andrea Roli, and Michael Sampels, editors. *Hybrid Metaheuristics*. Springer, 1st edition, 2008. doi:10.1007/978-3-642-30671-6.

[20] Christian Blum, Maria J. Blesa, and Manuel López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009. doi:10.1016/j.cor.2009.02.005.

[21] Christian Blum, Pedro Pinacho, Manuel López-Ibáñez, and José A Lozano. Construct, Merge, Solve & Adapt A new general algorithm for combinatorial optimization. *Computers & Operations Research*, 68:75–88, 2016. doi:10.1016/j.cor.2015.10.014.

[22] Borja Calvo and Guzmán Santafé Rodrigo. scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems. *The R Journal, Vol. 8/1, Aug. 2016*, 2016. doi:10.32614/RJ-2016-017.

[23] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[24] George Clarke and John W. Wright. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12(4):568–581, 1964.

[25] Peter Cowling, Graham Kendall, and Eric Soubeiga. A Hyperheuristic Approach to Scheduling a Sales Summit. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 176–190. Springer, 2000. doi:10.1007/3-540-44629-X_11.

[26] Averil Coxhead. A New Academic Word List. *TESOL Quarterly*, 34(2): 213–238, 2000. doi:10.2307/3587951.

[27] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th Conference on Genetic and Evolutionary Computation*, pages 913–920. Association for Computing Machinery, 2008. doi:10.1145/1389095.1389272.

[28] George B Dantzig and John H Ramser. The Truck Dispatching Problem. *Management science*, 6(1):80–91, 1959.

[29] Janez Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[30] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. *Operations Research*, 40(2):342–354, 1992. doi:10.1287/opre.40.2.342.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 4171–4186, 2019. doi:10.18653/v1/N19-1423.

[32] Marko Djukanović. *Exact and Heuristic Approaches for Solving String Problems from Bioinformatics*. PhD thesis, Technische Universität Wien, Vienna, Austria, 2020.

[33] Marko Djukanović, Günther R. Raidl, and Christian Blum. A Heuristic Approach for Solving the Longest Common Square Subsequence Problem. In *International Conference on Computer Aided Systems Theory*, pages 429–437. Springer, 2019. doi:10.1007/978-3-030-45093-9_52.

[34] Marko Djukanović, Günther R. Raidl, and Christian Blum. A Beam Search for the Longest Common Subsequence Problem Guided by a Novel Approximate Expected Length Calculation. In *Machine Learning, Optimization, and Data Science*, pages 154–167. Springer International Publishing, 2019. ISBN 978-3-030-37598-0. doi:10.1007/978-3-030-37599-7_14.

[35] Marko Djukanović, Christoph Berger, Günther R. Raidl, and Christian Blum. An A* search algorithm for the constrained longest common subsequence problem. *Information Processing Letters*, 166:106041, 2020. doi:10.1016/j.ipl.2020.106041.

[36] Marko Djukanović, Günther R. Raidl, and Christian Blum. Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, 114:104827, 2020. doi:10.1016/j.cor.2019.104827.

[37] Marko Djukanović, Günther R. Raidl, and Christian Blum. A Heuristic Approach for Solving the Longest Common Square Subsequence Problem.

In *International Conference on Computer Aided Systems Theory*, pages 429–437. Springer International Publishing, 2020. doi:10.1007/978-3-030-45093-9_52.

[38] Marko Djukanović, Aleksandar Kartelj, Tome Eftimov, Jaume Reixach, and Christian Blum. Efficient Search Algorithms for the Restricted Longest Common Subsequence Problem. In *International Conference on Computational Science*, pages 58–73. Springer, 2024. doi:10.1007/978-3-031-63775-9_5.

[39] Marko Djukanović, Jaume Reixach, Ana Nikolikj, Tome Eftimov, Aleksandar Kartelj, and Christian Blum. A learning search algorithm for the Restricted Longest Common Subsequence problem. *Expert Systems with Applications*, page 127731, 2025. doi:10.1016/j.eswa.2025.127731.

[40] Yadolah Dodge. *The Concise Encyclopedia of Statistics*. Springer Science & Business Media, 2008. doi:10.1007/978-0-387-32833-1.

[41] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.

[42] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999. doi:10.1109/CEC.1999.782657.

[43] Nicolas Dupin and El-Ghazali Talbi. Matheuristics to optimize refueling and maintenance planning of nuclear power plants. *Journal of Heuristics*, 27 (1):63–105, 2021. doi:10.1007/s10732-020-09450-0.

[44] Arkadiy Dushatskiy, Tanja Alderliesten, and Peter A.N. Bosman. A novel surrogate-assisted evolutionary algorithm applied to partition-based ensemble learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 583–591, 2021. doi:10.1145/3449639.3459306.

[45] Paul Erdös and Alfred Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[46] Rupert Ettrich, Marc Huber, and Günther R. Raidl. A Policy-Based Learning Beam Search for Combinatorial Optimization. In *European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar)*, pages 130–145. Springer, 2023. doi:10.1007/978-3-031-30035-6_9.

[47] Daniele Ferone, Paola Festa, and Mauricio G.C. Resende. On the Far from Most String Problem, One of the Hardest String Selection Problems. In

Dynamics of Information Systems: Computational and Mathematical Challenges, pages 129–148. Springer, 2014. doi:10.1007/978-3-319-10046-3_7.

[48] Javier Ferrer, Francisco Chicano, and José Antonio Ortega-Toro. CMSA algorithm for solving the prioritized pairwise test data generation problem in software product lines. *Journal of Heuristics*, 27(1):229–249, 2021. doi:10.1007/s10732-020-09462-w.

[49] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. 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. doi:10.1016/j.ins.2009.12.010.

[50] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979. ISBN 978-0-7167-1044-8.

[51] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979. ISBN 0716710447.

[52] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, volume 32, 2019.

[53] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 1998. doi:/10.1007/978-1-4615-6089-0.

[54] D. Goldberg. Genetic Algorithm in Search, Optimization, and Machine Learning. *Addison-Wesley, Reading, Massachusetts*, xiii, 1989.

[55] José Gonçalves and Mauricio Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17:487–525, 2011. doi:10.1007/s10732-010-9143-1.

[56] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*. MIT Press, 2016. ISBN 9780262035613.

[57] Alex Graves. Long Short-Term Memory. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 37–45. Springer, 2012. doi:10.1007/978-3-642-24797-2_4.

[58] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

[59] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[60] Alain Hertz and Daniel Kobler. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126(1): 1–12, 2000. doi:10.1016/S0377-2217(99)00435-X.

[61] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[62] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient Active Search for Combinatorial Optimization Problems. *arXiv preprint*, 2021. doi:10.48550/arXiv.2106.05126.

[63] Marc Huber and Günther R Raidl. Learning Beam Search: Utilizing Machine Learning to Guide Beam Search for Solving Combinatorial Optimization Problems. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 283–298. Springer, 2021. doi:10.1007/978-3-030-95470-3_22.

[64] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization: 5th International Conference*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

[65] IBM. *IBM ILOG CPLEX Optimization Studio*, 2023. URL `https://www.ibm.com/products/ilog-cplex-optimization-studio`. Version 22.1.1.0.

[66] Takafumi Inoue, Shunsuke Inenaga, Heikki Hyyrö, Hideo Bannai, and Masayuki Takeda. Computing longest common square subsequences. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-074-3. doi:10.4230/LIPIcs.CPM.2018.15.

[67] Ya-Hui Jia, Yi Mei, and Mengjie Zhang. A Bilevel Ant Colony Optimization Algorithm for Capacitated Electric Vehicle Routing

Problem. *IEEE Transactions on Cybernetics*, 52(10):10855–10868, 2021. doi:10.1109/TCYB.2021.3069942.

[68] Tao Jiang, Guohui Lin, Bin Ma, and Kaizhong Zhang. A General Edit Distance between RNA Structures. *Journal of Computational Biology*, 9(2): 371–388, 2002. doi:10.1089/10665270252935511.

[69] Yaochu Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing*, 9(1):3–12, 2005. doi:10.1007/s00500-003-0328-5.

[70] Syu-Ning Johnn, Victor-Alexandru Darvariu, Julia Handl, and Jörg Kalcsics. A Graph Reinforcement Learning Framework for Neural Adaptive Large Neighbourhood Search. *Computers & Operations Research*, 172:106791, 2024. doi:10.1016/j.cor.2024.106791.

[71] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. The traveling salesman problem. In *Handbooks in Operations Research and Management Science*, volume 7, pages 225–330. Elsevier, 1995. doi:10.1016/S0927-0507(05)80121-5.

[72] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995. doi:10.1109/ICNN.1995.488968.

[73] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, 2019. doi:10.1162/evco_a_00242.

[74] Yoon Kim. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, 2014. Association for Computational Linguistics. doi:10.3115/v1/D14-1181.

[75] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint*, 2014. doi:10.48550/arXiv.1412.6980.

[76] Scott Kirkpatrick, C.D. Daniel Gelatt Jr, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[77] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, Learn to Solve Routing Problems! *arXiv preprint*, 2018. doi:10.48550/arXiv.1803.08475.

[78] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNET Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, volume 25, 2012.

[79] Ilker Kucukoglu, Reginald Dewil, and Dirk Cattrysse. The electric vehicle routing problem and its variations: A literature review. *Computers & Industrial Engineering*, 161:107650, 2021. doi:10.1016/j.cie.2021.107650.

[80] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint*, 2014. doi:10.48550/arXiv.1402.6028.

[81] Felipe Lagos and Jordi Pereira. Multi-armed bandit-based hyper-heuristics for combinatorial optimization problems. *European Journal of Operational Research*, 312(1):70–91, 2024. doi:10.1016/j.ejor.2023.06.016.

[82] J Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Information and Computation*, 185 (1):41–55, 2003. doi:10.1016/S0890-5401(03)00057-9.

[83] Eugene L. Lawler and David E. Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[84] Stefan Lessmann, Marco Caserta, and Idel Montalvo Arango. Tuning metaheuristics: A data mining based approach for particle swarm optimization. *Expert Systems with Applications*, 38(10):12826–12838, 2011. doi:10.1016/j.eswa.2011.04.075.

[85] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003. doi:10.1007/0-306-48056-5_11.

[86] Shin-Yee Lu and King Sun Fu. A Sentence-to-Sentence Clustering Procedure for Pattern Analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 8 (5):381–389, 1978.

[87] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3, 2016. doi:10.1016/j.orp.2016.09.002.

[88] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng, et al. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, 2013.

[89] Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende, editors. *Handbook of Heuristics*. Springer, Cham, 2 edition, 2025. Forthcoming.

[90] Michalis Mavrovouniotis, Charalambos Menelaou, Stelios Timotheou, Georgios Ellinas, Christos Panayiotou, and Marios Polycarpou. A Benchmark Test Suite for the Electric Capacitated Vehicle Routing Problem. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020. doi:10.1109/CEC48606.2020.9185753.

[91] Hokey Min. The multiple vehicle routing problem with simultaneous delivery and pick-up points. *Transportation Research Part A: General*, 23 (5):377–386, 1989. doi:10.1016/0191-2607(89)90085-X.

[92] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997. doi:10.1016/S0305-0548(97)00031-2.

[93] Sayyed Rasoul Mousavi and Farzaneh Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012. doi:10.1016/j.cor.2011.02.026.

[94] Mohammadreza Nazari, Afshin Oroojlooy, Martin Takáč, and Lawrence V. Snyder. Reinforcement Learning for Solving the Vehicle Routing Problem. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9861–9871, 2018. doi:10.5555/3327546.3327651.

[95] Ferrante Neri, Carlos Cotta, and Pablo Moscato. *Handbook of Memetic Algorithms*, volume 379. Springer, 2011. doi:10.1007/978-3-642-23247-3.

[96] Bojan Nikolic, Aleksandar Kartelj, Marko Djukanović, Milana Grbic, Christian Blum, and Günther R. Raidl. Solving the Longest Common Subsequence Problem Concerning Non-Uniform Distributions of Letters in Input Strings. *Mathematics*, 9(13), 1515, 2021. doi:10.3390/math9131515.

[97] Teddy Nurcahyadi and Christian Blum. Adding Negative Learning to Ant Colony Optimization:A Comprehensive Study. *Mathematics*, 9:361, 2021. doi:10.3390/math9040361.

[98] Peng Si Ow and Thomas E. Morton. Filtered beam search in scheduling. *The International Journal of Production Research*, 26(1):35–62, 1988.

[99] Pedro Pinacho-Davidson, Christian Blum, María Angélica Pinninghoff, and Ricardo Contreras. Extension of CMSA with a Learning Mechanism: Application to the Far from Most String Problem. *International Journal of Computational Intelligence Systems*, 17(1):109, 2024. doi:10.1007/s44196-024-00488-7.

[100] Tayler Pino, Salimur Choudhury, and Fadi Al-Turjman. Dominating Set Algorithms for Wireless Sensor Networks Survivability. *IEEE Access*, 6: 17527–17532, 2018. doi:10.1109/ACCESS.2018.2819083.

[101] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI techinal report*, 2019.

[102] Jaume Reixach and Christian Blum. Extending CMSA with Reinforcement Learning: Application to Minimum Dominating Set. In *Metaheuristics International Conference*, pages 354–359. Springer, 2024. doi:10.1007/978-3-031-62922-8_27.

[103] Jaume Reixach and Christian Blum. How to improve “construct, merge, solve and adapt"? Use reinforcement learning! *Annals of Operations Research*, In press, 2024. doi:10.1007/s10479-024-06243-7.

[104] Jaume Reixach and Christian Blum. Improving the CMSA Algorithm with Online Deep Learning. In *ECAI 2025*, volume 413 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, October 2025. doi:10.3233/faia251352.

[105] Jaume Reixach, Christian Blum, Marko Djukanović, and Günther R Raidl. A Neural Network Based Guidance for a BRKGA: An Application to the Longest Common Square Subsequence Problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar)*, pages 1–15. Springer, 2024. doi:10.1007/978-3-031-57712-3_1.

[106] Jaume Reixach, Mehmet Anıl Akbay, and Christian Blum. Rjaume/EVRP-RJ-RT_Problem_Instances: EVRP-RJ-RT European Cities Benchmark Set , 2025. URL `https://doi.org/10.5281/zenodo.17055543`.

[107] Jaume Reixach, Christian Blum, Marko Djukanović, and Günther R. Raidl. A Biased Random Key Genetic Algorithm for Solving the Longest

Common Square Subsequence Problem. *IEEE Transactions on Evolutionary Computation*, 29(2):390–403, 2025. doi:10.1109/TEVC.2024.3413150.

[108] Mauricio G.C. Resendel and Celso C. Ribeiro. GRASP with path-relinking: Recent advances and applications. In *Metaheuristics: Progress as Real Problem Solvers*, pages 29–63. Springer, 2005. doi:10.1007/0-387-25383-1_2.

[109] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[110] Roberto Maria Rosati, Lucas Kletzander, Christian Blum, Nysret Musliu, and Andrea Schaerf. 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*, pages 254–267. Springer, 2022. doi:10.1007/978-3-031-27181-6_18.

[111] David Sankoff and Joseph B. Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication*, 1983.

[112] Camilo Chacón Sartori and Christian Blum. Boosting a Genetic Algorithm with Graph Neural Networks for Multi-Hop Influence Maximization in Social Networks. In *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 363–371. IEEE, 2022. doi:10.15439/2022F78.

[113] Camilo Chacón Sartori and Christian Blum. Improving Existing Optimization Algorithms with LLMs. *arXiv preprint arXiv:2502.08298*, 2025. doi:10.48550/arXiv.2502.08298.

[114] Camilo Chacón Sartori, Christian Blum, Filippo Bistaffa, and Guillem Rodríguez Corominas. Metaheuristics and Large Language Models Join Forces: Toward an Integrated Optimization Approach. *IEEE Access*, 2024. doi:10.1109/ACCESS.2024.3524176.

[115] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008. doi:10.1109/TNN.2008.2005605.

[116] Chao Shen and Tao Li. Multi-document summarization via the minimum dominating set. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 984–992, 2010. doi:10.5555/1873781.1873892.

[117] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529 (7587):484–489, 2016. doi:10.1038/nature16961.

[118] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. doi:10.1016/0022-2836(81)90087-5.

[119] Kenneth Sörensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. In *Handbook of heuristics*, pages 1–18. Springer, 2018. doi:10.1007/978-3-319-07153-4_4-2.

[120] James A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., 1987.

[121] Thomas Stützle and Holger H. Hoos. MAX–MIN ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000. doi:10.1016/S0167-739X(00)00043-1.

[122] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, Cambridge, MA, USA, 2018. ISBN 9780262039246.

[123] El-Ghazali Talbi. Machine Learning into Metaheuristics: A Survey and Taxonomy. *ACM Computing Surveys*, 54(6), 2021. doi:10.1145/3459664.

[124] Paolo Toth and Daniele Vigo. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*, 123 (1-3):487–512, 2002. doi:10.1016/S0166-218X(01)00351-1.

[125] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems*, volume 2, 2015.

[126] Qingguo Wang, Mian Pan, Yi Shang, and Dmitry Korkin. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 1287–1292, 2010. doi:10.1609/aaai.v24i1.7493.

[127] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998. doi:10.1038/30918.

[128] Robert F. Woolson. Wilcoxon signed-rank test. *Wiley Encyclopedia of Clinical Trials*, pages 1–3, 2007. doi:10.1002/0470011815.b2a15177.

[129] Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. NeuroLKH: combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, volume 34, pages 7472–7483, 2021.

[130] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi:10.1613/jair.2490.

[131] Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. DeepACO: Neural-enhanced ant systems for combinatorial optimization. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, volume 36, pages 43706–43728, 2023.

[132] Shuai Zhang, Yuvraj Gajpal, S.S. Appadoo, and M.M.S. Abdulkader. Electric vehicle routing problem with recharging stations for minimizing energy consumption. *International Journal of Production Economics*, 203: 404–413, 2018. doi:10.1016/j.ijpe.2018.07.016.