TU WIEN Informatics

Solving the Paintshop Scheduling Problem with Memetic Algorithms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Wolfgang Weintritt, BSc
Matrikelnummer 01327191

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu
Mitwirkung: Dipl.-Ing. Felix Winter, BSc

Wien, 15. Oktober 2020

Wolfgang Weintritt	Nysret Musliu

TU WIEN Informatics

Solving the Paintshop Scheduling Problem with Memetic Algorithms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Wolfgang Weintritt, BSc
Registration Number 01327191

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu
Assistance: Dipl.-Ing. Felix Winter, BSc

Vienna, 15th October, 2020

Wolfgang Weintritt	Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Wolfgang Weintritt, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Oktober 2020

Wolfgang Weintritt

Acknowledgements

I hereby express my sincere gratitude to my advisor Priv.-Doz. Dr. Nysret Musliu, and my co-advisor DI Felix Winter for their expertise, valuable feedback, patience, and encouragement throughout the whole thesis. Without them, this thesis would not have been possible.

I want to thank the DBAI group of the Institute of Information Systems for providing me with access to the computer cluster "cobra", which was invaluable for the evaluation.

Further, I would like to thank my family for supporting me under all circumstances and my cat Balu for putting a smile on my face in stressful times.

Special thanks to my friend Max for providing me with valuable tips regarding benchmarking and evaluation. Last but not least, I want to thank my girlfriend Ines for always supporting me, motivating me, and proofreading my work.

Kurzfassung

Das Paint-Shop-Schedling-Problem, abgekürzt PSSP, ist ein kürzlich vorgestelltes praxisnahes Planungsproblem. Das Ziel dieses Problems ist es, einen optimierten Zeitplan für das Lackieren von Autoteilen zu finden. Autoteile, welche lackiert werden sollen, werden auf Trägervorrichtungen platziert. Die Träger werden wiederum mittels Förderband in die Lackiererei transportiert, wo mehrere Lackierroboter die Autoteile lackieren. Das PSSP ist ein anspruchsvolles NP-hartes Problem. Beim Erstellen des Zeitplans müssen weitere Bedingungen beachtet werden, beispielsweise die vorhandenen Materialien und Träger, verbotene Träger- und Farbfolgen, Fälligkeitstermine, usw. Für viele der Probleminstanzen wurden noch keine optimalen Ergebnisse gefunden.

Die bestehende wissenschaftliche Literatur für das PSSP befasst sich mit exakten Lösungsverfahren und lokaler Suche. Allerdings wurden noch keine evolutionären Algorithmen zur Lösung des PSSP herangezogen. In dieser Diplomarbeit stellen wir einen memetischen Algorithmus zur Lösung des PSSP vor. Unser Algorithmus folgt der typischen Vorlage eines memetischen Algorithmus. Es wird zuerst eine initiale Bevölkerung generiert, gefolgt von stetiger Evolution. Selektions-, Rekombinations-, Mutations- und lokale Verbesserungsoperatoren werden auf jede Generation der Bevölkerung angewandt. Wir stellen drei neuartige Rekombinationsoperatoren vor, welche mit problemspezifischem Wissen arbeiten. Abschließend konfigurieren wir unseren Algorithmus mittels automatischem und manuellem Parametertuning.

Für die experimentelle Analyse unseres Algorithmus verwenden wir öffentlich verfügbare Probleminstanzen. Unser memetischer Algorithmus generiert konkurrenzfähige Lösungen für kleine und mittelgroße Probleminstanzen. Wir schaffen es für 8 der 24 Instanzen neue obere Schranken zu finden.

Abstract

The Paint Shop Scheduling Problem is a recently introduced real-life scheduling problem from the automotive industry. Car parts, which need to be painted, are placed upon carrier devices. These carriers are placed on a conveyor belt and moved into painting cabins, where painting robots paint the parts. The aim is to find an optimized production schedule for the painting of car parts. The Paint Shop Scheduling Problem is a challenging NP-hard problem that imposes constraints like due dates, forbidden carrier and color sequences, maximum and minimum block lengths, and material availability. While the problem has been tackled in literature, there are many problem instances for which the optimal solutions are still unknown.

Existing literature is focused on solving the Paint Shop Scheduling Problem with exact and local search approaches. However, population-based algorithms have not yet been applied to solve this problem. In this thesis, we propose a memetic algorithm to solve the Paint Shop Scheduling Problem. Our algorithm follows the typical template of a memetic algorithm. An initial population is generated, followed by the constant evolution of generations. Selection, crossover, mutation, and local improvement operators are applied in each generation. We design three novel crossover operators that consider problem-specific knowledge. Finally, we carefully configure our algorithm, including automated and manual parameter tuning.

Using a set of available real-life benchmark instances from the literature, we perform an extensive experimental evaluation of our algorithm. The experimental results show that our memetic algorithm yields competitive results for small- and medium-sized instances and is able to set new upper bounds for some of the problem instances.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Aims of this Thesis . . . 2
1.2 Main Results . . . 2
1.3 Organization . . . 3

2 The Paint Shop Scheduling Problem **5**
2.1 Problem Description . . . 5
2.2 Related Work . . . 8

3 Memetics **11**
3.1 Concepts and Terminology . . . 11
3.2 From Memetics to Memetic Algorithms . . . 13

4 Solving the PSSP with Memetic Algorithms **19**
4.1 Definitions . . . 19
4.2 Crossover Operators and Memetic Representations . . . 20
4.3 Algorithm . . . 24
4.4 Crossover Delta Evaluation . . . 28

5 Empirical Evaluation **31**
5.1 Methodology . . . 31
5.2 Setup . . . 32
5.3 Automated Parameter Tuning . . . 34
5.4 Manual parameter tuning . . . 40
5.5 Comparison to Literature Results . . . 56

6 Conclusions and Future Work **63**

List of Figures **65**

List of Tables **67**

List of Algorithms **69**

Bibliography **71**

CHAPTER 1

Introduction

Each day, large amounts of synthetic material pieces need to be painted in the automotive supply industry's paint shops. As the process of painting is costly and time-consuming, the industry's paint shops are highly automated. Material pieces are placed on carrier devices, which in turn are placed on a conveyor belt. The carrier devices are moved to painting cabins, where several painting robots paint the material pieces.

Usually, for such problems, human planners are constructing the production schedules. Due to the usually long planning horizons and the tight due dates, as well as a lot of other constraints, it takes a lot of time for human planners to create the schedules, and they are normally not able to find optimal solutions. Therefore, there is a strong need for automated approaches to generate effective schedules for this challenging problem.

In literature, this scheduling problem has recently been introduced by Winter et al. [2019], and is called the *Paint Shop Scheduling Problem*, abbreviated as PSSP. It is a complex combinatorial problem, and the decision variant of the problem was shown to be NP-complete by Winter and Musliu [2019a].

The PSSP is a challenging problem since it imposes a number of constraints in addition to due dates. Those include a limited amount of materials and carrier devices, forbidden color and carrier sequences, limited capacities per scheduling round, and minimum and maximum block sizes for carriers of the same type. The combined complexity of these constraints separates the PSSP from other scheduling problems in literature. The PSSP has a multi-objective cost function with two main objectives - to minimize the number of color changes and the number of carrier device changes per round. Peaks of these changes should not appear in the schedule. Therefore changes per rounds are squared, which penalizes peaks and balances changes over the scheduling horizon.

Different approaches have been considered in literature to solve the PSSP (Winter and Musliu [2019b], Winter et al. [2019]). Exact approaches have shown to work well for small instances. For some of the smaller instances, optimal solutions have been found

by Winter and Musliu [2019b]. However, for large instances, optimal solutions are not known yet.

Heuristics can generate feasible solutions in a reasonable time - Winter et al. [2019] achieved good results with simulated annealing. As of yet, it is not clear how metaheuristics other than simulated annealing - for example evolutionary algorithms - perform for this specific problem. To the best of our knowledge, the problem has not yet been tackled with memetic algorithms. As memetic algorithms have been successfully applied to other scheduling problems (e.g. Burke et al. [1995], Liu et al. [2007], Liu et al. [2013]), the investigation of a memetic approach for the PSSP is an interesting research question and could possibly lead to improved results.

1.1 Aims of this Thesis

The main objectives of this thesis are:

- Investigation of solution techniques for the PSSP with regards to memetic algorithms. Design of a memetic problem representation, several memetic operators, and various population construction strategies.
- Implementation of a parameterizable memetic algorithm for the PSSP, including different memetic operators and population construction strategies.
- Statistical evaluation of our proposed algorithm. This includes the use of automated parameter tuners and subsequent manual parameter tuning.
- Comparison to state-of-the-art results.

1.2 Main Results

The main contributions of this thesis are:

- We design a memetic representation of the PSSP. Further, we design three novel crossover operators and different population construction strategies.
- We implement an algorithm based on the concepts of Moscato [1989]. At first, a population of individuals is created. We propose several strategies for this process. Then, memetic evolution takes place - new individuals are created by applying memetic operators and local improvement methods. The algorithm is highly parameterizable, which helps in evaluating the performance of the different construction strategies and memetic operators.
- We experimentally evaluate the algorithm's performance on problem instances from the literature. Via automated-parameter-tuning, we try to optimize our algorithm's performance. We analyze the performance impact of different algorithm parameters.

- We compare our algorithm's results with the best literature results. We achieve competitive results for many of the problem instances and can improve upper bounds for 8 of the 24 instances.

1.3 Organization

The thesis is structured as follows. In Chapter 2, we give a formal definition of the PSSP take a look at related work in literature. The concepts of genetics and memetics are examined in Chapter 3. Memetic algorithms are explained, and memetic algorithms for related problems from the literature are investigated. Our algorithm is extensively described in Chapter 4. This chapter includes subsections for each step of the algorithm, like the memetic operators, the construction strategies, and the local improvement phase. In Chapter 5, we compare different parameter settings for our algorithm. We evaluate our algorithm's performance and compare it to literature results. Finally, we present our conclusions as well as remarks on possible future research in Chapter 6.

CHAPTER 2

The Paint Shop Scheduling Problem

The Paint Shop Scheduling Problem has recently been introduced by Winter et al. [2019]. It is a challenging problem - in fact, Winter and Musliu [2019a] have proven that the problem's decision variant is NP-complete. A detailed textual description and a formal problem description can be found in the work of Winter et al. [2019]. In this chapter, we will describe the PSSP, its hard constraints, and its objective function. Further, we take a look at different approaches that have been used for solving the PSSP in literature, as well as related problems.

2.1 Problem Description

In this section, a problem description based on the work of Winter et al. [2019] is given. In the automotive supply industry's paint shops, a large number of car parts need to be painted each day. A paint shop usually supplies different car manufacturers. Thus many different car parts have to be painted, such as engine covers, bumpers, and wheel rims. Just-in-time manufacturing (Sugimori et al. [1977]) is a commonly used concept in the car industry, which forces suppliers to adhere to tight due-dates. Therefore, the main goal of the PSSP is the construction of feasible production sequences.

Figure 2.1 outlines a paint shop's layout. Carrier devices are placed on a conveyor belt. Each carrier may transport well-defined combinations of raw material parts, as can be seen in Figure 2.2. Paint shop employees load the carrier devices with raw material pieces. Then, carriers are moved to the painting cabins, where the raw material parts are painted by several painting robots. After the carriers return from the painting cabins, the painted parts are unloaded by paint shop employees. The empty carrier may be removed from the conveyor belt or reused and loaded with raw material parts.

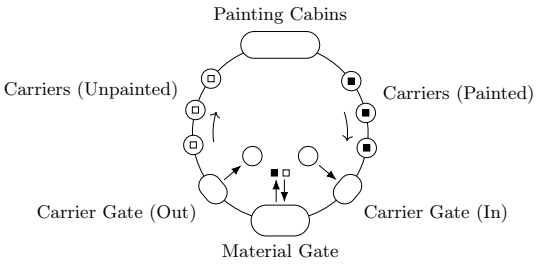

Figure 2.1: A typical layout of an automotive paint shop. A conveyor belt moves the carriers through the paint shop. After completing a round, carriers can be removed, or new carriers can be added. The figure is taken from Winter et al. [2019].

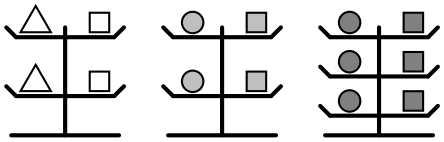

Figure 2.2: Schematic representation of three carriers. The two carriers on the left are from the same type, but loaded with different material configurations. The carrier on the right is a different carrier type. The figure is taken from Winter et al. [2019].

The PSSP has a multi-objective cost function with two main objectives - to minimize the number of color changes and the number of carrier device changes per round. The practical benefit of these objectives is to reduce waste and save costs. A good schedule should group requests with similar colors to reduce costs. Besides, it is preferential to keep the number of carrier changes as low as possible, since they may lead to delays in the schedule. Therefore, as many carrier devices as possible should be reused in the schedule's following round - this will be explained in the following paragraphs.

	R1	*R2*	*R3*	*...*
1	A1	A2	C1	...
2	A1	A2	C2	...
3	A2	C1	C3	...
4	B1	B2	B1	...
5	B2	B3	B2	...

Figure 2.3: A PSSP schedule in tabular form for three rounds. Columns depict rows of the schedule, while cells depict carrier, carrier configuration, and color. The figure is taken from Winter et al. [2019].

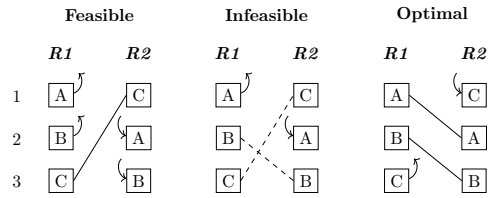

Figure 2.4: Three ways to reuse carrier devices in consecutive rounds. The left option, while viable, is not optimal. The middle option is infeasible since carrier C cannot be reused and placed before carrier B. The right option is the optimal solution - which means that the LCS of the two rounds' sequences is 'AB'. The figure is taken from Winter et al. [2019].

Because of the paint shop's circular layout, the schedule is organized in rounds. A schedule can be represented in tabular form, where columns are rounds, and table cells represent carrier type, carrier configuration, and the proposed color. Figure 2.3 depicts such a schedule. Carrier devices can be inserted and removed between rounds. However, if carrier devices of the same type are scheduled in consecutive rounds, it may be possible to reuse some of them, depending on the two rounds' order. The minimum amount of carrier changes between two consecutive rounds can be calculated by determining the LCS (longest common subsequence, Hirschberg [1977]) between the two sequences. Figure 2.4 sketches the schedules of two consecutive rounds and different ways in which carrier devices could be reused.

2.1.1 Hard Constraints

As mentioned above, a schedule for the PSSP has to fulfill a number of hard constraints to be considered feasible:

- All demands must be satisfied within time (overproduction is allowed).
- Carrier availability must be respected in each round.
- The minimum round capacity must be fulfilled in each round. If the maximum round capacity is not used, unplanned carrier positions must be scheduled last.
- Minimum and maximum carrier block length restrictions must be fulfilled.
- Forbidden carrier type sequences must not appear in the schedule.
- Forbidden color sequences must not appear in the schedule.

2.1.2 Objective function

As stated above, the PSSP's objective function has two minimization objectives. It aims to minimize the number of color change costs (cc) and carrier changes (sc). The objective function is shown in Equation 2.1. Each round's color change costs (cc_r) and carrier changes (sc_r) are squared, to balance the required changes over the scheduling horizon. This is done to avoid peaks of such changes within a single round, which could lead to delays in the schedule.

$$minimize \sum_{r \in \{0,\dots,n-1\}} sc_r^2 + \sum_{r \in R} cc_r^2 \tag{2.1}$$

where R is the set of rounds, and n is the schedule's number of rounds.

2.2 Related Work

Not too much work has been done on the PSSP since it is a relatively new problem. Winter and Musliu [2019b] tried to solve the problem with an exact method - constraint programming. Their approach works well for small instances, and they have been able to find optimal solutions for most of them. But they were not able to solve any large problem instances within a runtime limit of 6 hours. Winter et al. [2019] proposed a metaheuristic approach based on local search. They use a simulated annealing-based move acceptance function, which takes the search progress into account. Further, they use a tabu list to prevent the selection of recently performed moves. They propose a greedy construction heuristic to generate an initial solution for local search. This allows them to solve the largest instances within time constraints. The best results were achieved with the following combination of techniques: The initial solution was created with a greedy heuristic and passed to simulated annealing, where the neighborhood was built via min-conflicts heuristic. With this combination of techniques, they were able to find feasible solutions, even for the largest problem instances, within a time limit of one hour. In their work, they also introduce 24 real-life problem instances. We describe those instances in detail in Section 5.2.1, and use them to evaluate our algorithm.

While the PSSP has just recently been introduced in the literature, there is quite some work on the production scheduling problems of the automotive industry. The *Car Sequencing Problem (CSP)* is a related problem, where cars have to be sequenced along the production line, fulfilling a number of constraints. This problem also tackles the minimization of color changes. The CSP is a popular problem in literature, and has been tackled with different approaches, like constraint programming (Dincbas et al. [1988]), integer linear programming (Prandtstetter and Raidl [2008]), ant colony (Boysen and Fliedner [2007]) and a follow-up sequencing algorithm (Bysko and Krystek [2019]). Solnon et al. [2008] give an overview of state-of-the-art methods for this problem.

Spieckermann et al. [2004] also deal with the sequential ordering in automotive paint shops. The objective of the presented problem is to maximize the average color batch size. They present a branch&bound algorithm to solve the problem.

The work of Wang et al. [2011] focuses on reducing energy consumption in the automotive production process by optimizing schedules. The proposed problem is to reduce energy consumption over peak demand periods on a group of production lines of an automotive factory, while also obeying time and resource constraints. They use an evolutionary algorithm to solve the problem. Starting from a population of random schedules, they apply local transformations to create offspring. The best individuals of the union set of parents and children are selected for survival for the next generation.

Dörmer et al. [2015] consider the *Master Production Scheduling Problem (MPS)* in the context of the automotive industry. The customization of final products in the automotive industry involves large numbers of optional parts, which leads to a huge variety of operation times at the various stations of the assembly line. They develop

a mathematical model formulation for the MPS and propose several heuristic solution procedures to solve this problem, which focus on minimizing the workload variability.

The PSSP is different from the automotive scheduling problems discussed above since it does not deal with cars as a whole, but with car parts placed on carrier devices. It also has the uniqueness of a multi-objective quadratic cost function, taking the whole scheduling horizon into consideration, and punishing peaks in color and carrier device changes.

CHAPTER 3

Memetics

In this chapter, the terminology and scientific background of genes and memes is discussed. We look at the differences between genetic and memetic evolution and give a real-life example of memetic evolution. Furthermore, we highlight the differences between genetic and memetic algorithms and discuss their applications in the field of meta-heuristics. Finally, we examine state-of-the-art memetic algorithms for scheduling problems.

3.1 Concepts and Terminology

The basics of genes and memes are essential to understand *memetic algorithms*.

A *gene* is a replicable biological unit hosted by an individual. In Darwin's breakthrough work On the Origin of Species (Darwin [1859]) a new evolutionary concept - natural selection - is introduced.

In nature, individuals are pressured to extinct by selection. The fittest individuals with the best genes have the highest chance of reproduction. This is called "survival of the fittest". Genes are replicated through heredity from one generation to another. As a consequence, better genes are more likely to survive the evolutionary process and multiply. But this is a slow process, and it takes hundreds of thousands of years for species to evolve since the speed of evolution is (amongst others) limited by the generation time - the time between two consecutive generations (Sarich and Wilson [1973]). Mutation of genes also occurs randomly during the process, though mostly at the point of fertilization.

The term *meme* was first introduced by Dawkins [1976]. A meme is a cultural unit. Thus it can be a catchy tune, a specific greeting, a belief in a certain god, a way to dress, the human rights, etc. Multiple memes together form cultural constructs like music genres, cultures, religions, fashions, political views.

In various science disciplines like anthropology (Atran [2002]), psychology (Blackmore and Blackmore [2000]), and architecture/urban theory (Salingaros and Mehaffy [2006])

research has been done based on the concepts of memetic evolution. Metaheuristic techniques like memetic algorithms are just one of the use cases of memetics.

Genes have been the replicators on our planet for billions of years. They are passed from body to body via sperms and eggs. Memes, just like genes, are replicable units hosted by individuals. Memes are new kinds of replicators, passed from brain to brain. Genetic evolution provided the brains, which are the foundation for the new replicators: the memes. The evolution of memes is a process with many similarities to genetic evolution, but orders of magnitude faster.

Memes are replicated by communication (spoken, written) and constantly mutated by misunderstandings or deliberate adaptions. Only the best memes "survive" the selection and are passed on to other individuals.

The process of replication of memes - how they are copied from brain to brain - is called imitation. Dawkins [1976] proposed several qualities on which the survival of memes depends: longevity, fecundity, and copying-fidelity.

- *Longevity* of a single copy of a meme is limited by the lifespan of the person, which is the same as it is for genes.
- *Fecundity* of memes is more important than longevity. Some memes have great short time success (e.g. pop songs), while other memes have long-term success (e.g. religious belief). This is also similar to gene pools, where some genes have short and some long term success.
- The biggest difference between memes and genes is in their *copying-fidelity*. Genes have an all-or-nothing transmission. Memes, on the other hand, are exposed to a process of constant blending and mutation.

In genetics, genes are competing with their allele, which is the genetic counterpart. Memes, on the other hand, have no allele. The competition between memes is for brain time. Successful memes acquire more brain time, and thus are able to supersede weaker memes.

3.1.1 Real-Life Example

Let us discuss a real-life example of memetic evolution. We have the cultural entity "Formula 1 car engineering" consisting of the memes "type of drive", "engine type", and "aerodynamics". There are several memes competing for each of those properties. For example, engines can be electric, hybrid, naturally aspirated, turbocharged, have different amounts of cylinders, engine placement, fuel type, etc. There are infinite options for each of those memes. Each day, engineers have new ideas - and those memes are competing with all other memes. The consequence is that the whole cultural entity is evolving rapidly.

In the early years of Formula 1, the cars' shape was as streamlined as possible. In 1968, the Lotus engineers were the first to fit wings on their car in the Monaco Grand Prix. They were the fastest and won the race, alerting the other teams. The next race, Ferrari engineers equipped their car with wings and were faster than the other teams by a few seconds. The very next race, almost all cars were fitted with wings. In this example *replication* took place. The idea of putting wings on a race car replicated from the engineers of team Lotus to the other teams' engineers. This highlights how fast memetic evolution takes place compared to genetic evolution. The replication of the successful meme took only a few weeks.

When the Renault team pioneered turbocharged engines in Formula 1 in 1977, those engines were extremely powerful but also very unreliable and difficult to drive due to the turbo lag. For some years, only Renault used turbocharged engines, and with improved reliability and driveability, they became increasingly successful. More and more competitors switched to a turbocharged engine each year, and from 1983 to their ban in F1 in 1988, all of the top teams used turbocharged engines. The turbocharged engine meme had *mutated* several times until it successfully *superseded* the naturally aspirated engine meme.

Several times through Formula 1 history, teams and their engineers tried to use four-wheel drive (4WD) cars. The first and most successful was the Ferguson P99 raced in 1961, which even won an F1 event. Only seven other cars were built in the following years with 4WD, but none of them achieved much success. The 4WD meme thus *extincted*. It was not fit enough to survive the requirements of F1 engineering.

In the examples above, we can observe some of the differences between genetic and memetic evolution. Individuals (engineers) are constantly trying to improve the cultural entity as a whole. They are able to independently modify any of their memes. Due to high replication rates, the speed of evolution is magnitudes faster than the speed of genetic evolution.

3.2 From Memetics to Memetic Algorithms

3.2.1 Genetic algorithms

Genetic algorithms that try to mimic the biological mechanisms of evolution have first been proposed by (Holland [1992]). They are used to tackle optimization problems of all kinds. At the start, a population of candidate solutions is created. Each of these individuals has some properties (genes), which are altered and mutated during the process. Genes can be binary encoded, i.e. they can be represented by 0 or 1. During runtime, the population evolves towards better solutions by mimicking biological evolution.

After the initialization, there are three different operators, which are repetitively applied to the population: selection, crossover, and mutation. The operators are inspired by biological evolution. Fitter individuals have a higher chance of selection. Once two individuals are selected for reproduction, the crossover operator is executed. A crossover

function takes parts of the information of both parent individuals to create a new individual. The other genetic operator, which alters one individual, is the mutation operator. It is used to improve genetic diversity.

3.2.2 Memetic Algorithms

Memetic algorithms - like genetic algorithms - are population-based, evolutionary metaheuristics (Moscato [1989]). The survey of Moscato and Cotta [2019] provides an excellent overview of different techniques applied by memetic algorithms. The difference to genetic algorithms is that they mimic cultural instead of biological evolution. Again everything starts with a population of solution candidates. But the candidates' properties are memes, not genes. Memes are larger units than genes.

The basic idea of the memetic operators is the same as for genetic operators. Crossover operators do not have the limitations of biological reproduction, thus a child can have an arbitrary number of parents. A subset of each generations' individuals is selected to be improved via local search techniques. Selection, duration, and frequency of local improvement are all subject to parametrization.

The paragraph above captures the ideas of basic memetic algorithms, as proposed by Moscato [1989]. Some more recent approaches employ (Chen and Ong [2012], Smith [2005]) what is called *memetic computing*. As opposed to the basic memetic algorithms, all memetic operators act on meme basis. In fact, there is an explicit representation of memes alongside solutions, and those memes evolve as well. Selection is done on meme level, as memes compete with each other. Different features of the memes can be assessed. For example, memes with better fitness values have a higher chance to be selected. The local refinement step is also performed on meme level. Meta-Lamarckian learning and adaptive hyperheuristics are related approaches since they also work that way - i.e. a collection of memes is available, and a mechanism decides which one to apply.

The most recent developments in the field of memetic algorithms yielded co-evolving (Smith [2007]) and multimemetic algorithms (Nogueras and Cotta [2014]). Those memetic algorithms co-evolve memes that encode additional information, for example, definitions of local search operators. This may lead to more optimal choices of local search operators, depending on the state of the search process.

3.2.3 Designing a Memetic Algorithm

When designing a memetic algorithm for a specific problem, lots of aspects have to be considered. First of all, representations of genes, memes, individuals, as well as a fitness function to evaluate individuals or memes must be defined. These representations are the foundations of the algorithm.

For each phase of the memetic algorithm, a lot of decisions have to be made. The first phase is the construction of the initial population. Regarding the construction strategy,

we have to make some decisions. For example, if a random construction of solutions is sufficient or a specific heuristic should be used.

For the genetic part of the algorithm, suitable memetic operators have to be chosen. For this step, the representation of memes and individuals has to be kept in mind. When deciding upon the concrete implementation of the memetic operators, we have to consider some questions. Which memetic operators should be used? How often should these operators be applied? How are the parent individuals selected, and by which criteria? How does the interaction between the selected parents work? Do we need to repair the child solutions after creating them? How often shall individuals undergo mutation?

Memetic algorithms utilize local refinement its population. Again, there are a lot of design choices to be made for this part of the algorithm. Should exact methods or heuristics be used? If we use a local search heuristic, which neighborhood should be used? Which solution acceptance should be applied? Hill climbing? Simulated annealing? Should a tabu list be used?

Some algorithms also have another selection step at the end of each generation. Anew, this raises some questions. Should all generated children be part of the next generation, or is there another selection process? Should some individuals of the current generation survive? What should we do with duplicate solutions?

Some of the options presented above may be controlled by algorithm parameters. In Chapter 4 we present the implementation of our algorithm, including the memetic representation. Configurable parameters of our algorithm are listed in Section 5.2.3.

In the next section, we take a look at some state-of-the-art memetic algorithms from literature and summarize the authors' answers to the questions we posed above.

3.2.4 State-of-the-art of Memetic Algorithms

Memetic algorithms are flexible metaheuristic approaches (Cotta et al. [2018]). This means that they are higher-level procedures, and can be applied to solve a variety of problems. They are especially suited for NP-hard problems, which exact methods often cannot solve in reasonable time. Memetic algorithms have been applied to many NP-hard problems, including timetabling (Burke et al. [1995]), traveling salesperson problem (Gong et al. [2019]), break scheduling (Widl and Musliu [2010] and Widl and Musliu [2014]), graph coloring (Lü and Hao [2010]), graph partitioning (Benlic and Hao [2011]), job shop scheduling (Liu et al. [2013]), bin packing (Spencer et al. [2019]), and multidimensional knapsack (Puchinger et al. [2005]).

To the best of our knowledge, no research has been done on solving the Paint Shop Scheduling Problem (Winter et al. [2019]) with memetic algorithms. However, memetic algorithms have been used to successfully solve other scheduling problems. In the following, we will give two examples to illustrate how a popular scheduling problem - the Job Shop Scheduling Problem - is solved in literature with memetic algorithms.

In the Job Shop Scheduling Problem, n jobs have to be scheduled on m different machines. Each job consists of a set of operations, and each operation requires a different machine. Each operation has a processing time, which may vary from machine to machine. The jobs' operations must be processed in order. The makespan, which is the time between the start and the end of the schedule, has to be minimized.

Memetic algorithm for the Job Shop Scheduling Problem

In their work, Hasan et al. [2009] examine the performance of a genetic algorithm and three different memetic algorithms on the Job Shop Scheduling Problem. In their version, each job is performed on a machine exactly once, so each job has one operation per machine.

In their representation, a chromosome is mapped to a job pair-relationship-based binary string. For each job pair (j_u, j_v), there are m (number of machines) bits. The bit for machine m_x is 1, if j_u precedes job j_v on this machine, and 0 otherwise.

The phenotype (the observable characteristics, so the actual order of jobs for each machine) is derived from the chromosome with an algorithm called local harmonization. The resulting solution is then repaired.

The applied genetic operators are a simple two-point crossover and bit-flip mutation. Selection of the parent individuals is done by randomly choosing one parent from the elite class (top 15%) of individuals. The other parent is selected by performing a tournament between two individuals of the bottom 85% of the population. After the crossover operator is applied, the solution is repaired, s.t. it is feasible again.

The three memetic algorithms use different local search techniques. They introduce three priority rules: partial reordering (PR), gap reduction(GR), restricted swapping (RS). The best combination they found is GR and RS. GR is applied to every individual, while RS is only applied to 5% of randomly selected individuals each generation. This is because the role of RS is mostly to increase population diversity.

A solution found by local search is accepted if it improves the individual's fitness. It may still be accepted if its fitness is above a certain threshold.

To improve population diversity they vary mutation rates. If more than 50% of the elite class are the same solution, a higher mutation rate is employed.

The memetic algorithm outperforms the genetic algorithm considerably while also reducing computational time. In the experiments conducted by the authors, the memetic algorithm outperformed all other algorithms from the literature.

Memetic algorithm for the Multiobjective Flexible Job Shop Scheduling Problem

Yuan and Xu [2015] investigate the use of a memetic algorithm for the Multiobjective Flexible Job Shop Scheduling Problem (FJSP). The FJSP is a generalization of the

classical JSP. Each operation may be processed by any machine from the given set, rather than one specified machine. For this problem, three minimization objectives have to be achieved: minimizing the makespan, total workload, and critical workload. The problem is addressed in a Pareto manner, and the aim is to find Pareto-optimal solutions.

A chromosome consists of two vectors: a machine assignment vector and an operation sequence vector. The machine assignment vector u is an integer vector. Its length is equal to the number of operations O - it assigns operations to machines. If o_1 is executed on the machine with the ID 7, then the first element of the vector is 7. The operation sequence vector v is also an integer vector with the length O. This vector orders the operations by their priority. If o_3 is the operation with the highest priority, then the first element of the vector is 3.

There are two genetic operators - one for each chromosome vector. The crossover operator for vector u chooses a random set of vector positions first. Two children are generated by swapping values of the selected positions between parents. For vector v a modified order crossover is used. Two positions of the vector are randomly picked. The operations between the two positions are copied from one parent to the corresponding positions of the child. The empty positions in the child's vector are filled with the missing operations in the same order they appear in the second parent's vector. The second child is created by executing the same procedure and swapping parents. Finally, the child solutions are repaired, since infeasible operation sequences may occur.

The selection process is done by holding a tournament. For each tournament, a weight vector is randomly chosen from a set of 300 weight vectors. This vector is applied to the fitness function of the individuals.

They employ a hierarchical local search strategy to handle the three objectives.

It is interesting to note that they use NSGA-II (Deb et al. [2002]) as a framework for their memetic algorithm. They actively eliminate duplicates in the population by mutating them during each iteration. Not all offspring may be part of the next population, and some parents may survive the generation. The new population is created by choosing the best N individuals from the merged sets of old population, individuals created by memetic operations, and individuals created by local improvement.

CHAPTER 4

Solving the PSSP with Memetic Algorithms

In this chapter, we solve the Paint Shop Scheduling Problem with memetic algorithms. We propose a memetic algorithm based on the concepts discussed in Chapter 3. Different memetic representations and novel crossover operators are outlined. All phases of our memetic algorithm - including initialization, selection, crossover, mutation, and local search - will be explained in detail. After outlining the basic structure of our memetic algorithm, its steps are described in detail. In Chapter 5, the fitness of the solutions obtained by the various crossover operators is evaluated and compared.

4.1 Definitions

In the following chapters, we think of a solution as a sequence of rounds. Figure 4.1 is an example of the representation we use in the following chapters. Of course, each of the rounds still consists of a sequence of carrier devices.

Figure 4.1: Visual representation of a solution. Rounds are labeled R1 - R8. Round contents (carrier devices, their configuration, and the selected color) are not shown in detail since they are not relevant for our memetic representations.

The base algorithm is shown in Algorithm 4.1. It is basically the same skeleton used by most memetic algorithms. The only notable difference is that we have another selection step at the end of the algorithm, which includes shrinking of the population to the base population size.

The different parts of the algorithm are described in detail in the following chapters.

Algorithm 4.1: Memetic base algorithm

Input: generation strategy gs, population size ps, crossover population size cps,
Output: Fittest solution found

```
1  P ← generateInitialPopulation(gs)
2  while time is left do
       // 1. selection (elitism and k-tournament)
3      E ← fittest P ∈ P
4      M ← selectParentsForMemeticOperations(cps)
       // 2. crossover and mutation
5      N ← performMemeticOperations(M)
       // 3. local search
6      N ← improvePopulation(N)
       // 4. selection (shrinking)
7      P ← shrinkPopulation(N, P, ps)
8      P ← P ∪ E
9  end
10 return fittest P ∈ P
```

4.2 Crossover Operators and Memetic Representations

Crossover operators are an essential part of population-based algorithms.

Crossover operators may have a significant influence on the solution's quality (hereinafter called fitness). Finding effective crossover operators for the Paint Shop Scheduling Problem is not a trivial task, since partial solutions (memes) cannot be evaluated separately. This is because of multiple reasons:

- Costs caused by a single round or a group of rounds cannot be calculated without the neighboring rounds. This is because the carrier device sequencing is calculated by looking at the different carrier devices in between two consecutive rounds. However, carrier device sequencing is critical for a solution's costs - costs can grow fast because of its quadratic calculation.
- Demands must be fulfilled - those are hard constraints. But again, we cannot evaluate them for partial solutions.

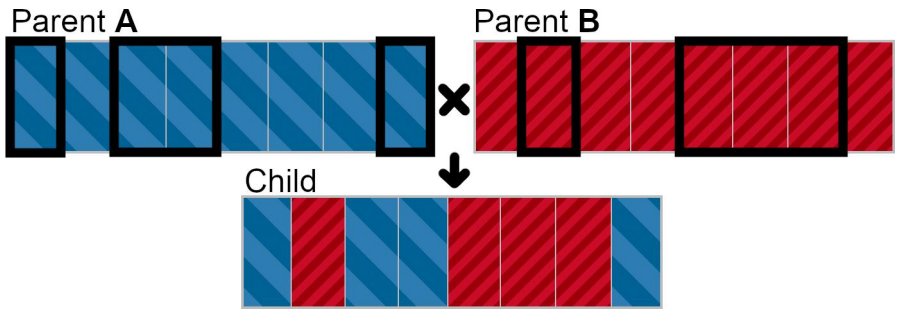

Figure 4.2: Vertical crossover. Parent solutions A and B are cut *vertically*. A child solution is created by merging those parts together.

- Between the last carrier device of a round, and the first carrier device of the next round, all constraints must still be met. A round could, for example, have an insufficient amount of carrier devices of type A at the end of the round, but if the next round has enough carrier devices of type A at its start, the constraint is fulfilled. Basically, the rounds are just a way of dividing the whole schedule, but the rounds cannot be viewed independently.

- Rounds do not have a fixed size; their size only has to be within a certain range. When evaluating partly solutions, smaller rounds usually have fewer constraint violations and lower costs than larger rounds.

These points make it difficult to evaluate and improve memes. Thus, for the first two crossover operators (vertical and horizontal crossover), full solutions are evaluated and selected as parents. We also do not improve the memes during the local search phase, but instead solutions as a whole.

4.2.1 Vertical Crossover

For this crossover operator rounds are taken from two parent solutions to create a new child solution. The solutions are cut in *vertical* direction - that's where the name comes from. A meme, therefore, is one round of the solution. Whether a round is taken from parent A or parent B is randomly chosen. Figure 4.2 depicts an example of such a crossover.

This crossover operator's main idea is to introduce a crossover operator that - while as simple as possible - still performs well. The operator uses one of the problem's natural units - the round - as its memes. A round is not too big and not too small to be a meme. Carrier device and color sequences are preserved. We do not have to worry about the round capacity constraints (too few/too many carrier devices in the round) or the carrier device availability constraint (too many carrier devices of one type used in the same round). Since we are merging rounds of different solutions together, some of the problems

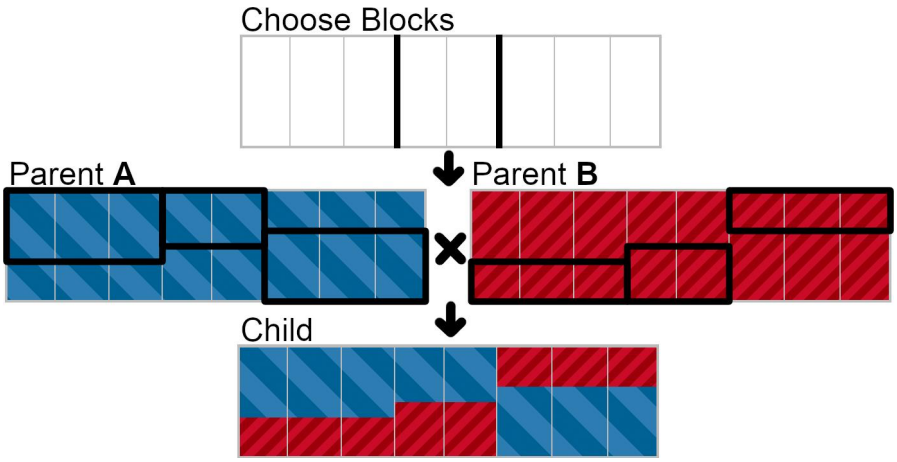

Figure 4.3: Horizontal crossover. In the first step, the parents are divided into blocks of rounds. Then, for each of those blocks, a *horizontal* cutting point is chosen. The upper and lower parts of the block are taken from different parents at random. Eventually, all of those selected block parts are merged together to create the offspring solution.

mentioned above can occur - such as higher costs due to carrier sequences of successive rounds or hard constraints being violated (demands, carrier device sequences, etc.).

4.2.2 Horizontal Crossover

This crossover operator cuts the solutions in *horizontal* direction. Again, two parent solutions are taken, cut, and a child solution is created by merging the pieces together.

First, the solution is split into round blocks of length l: $l \in [\log_3 R, \log_2 R]$, where R is the number of rounds in the instance. We choose a logarithmic block size since the round number can vary a lot between the instances. If we have an instance with just a few rounds, we still want to have a few cuts. However, for a problem instance with ten times as many rounds, we do not want ten times more cuts. Instead, we want the number of blocks - and thus the number of memes - to be more consistent for different instances.

For each of these round blocks, a horizontal cutting point is chosen. The horizontal cutting point h is randomly chosen: $h \in [r * 0.25, r * 0.75]$, where r is the minimum round length, i.e. the minimum number of carrier devices allowed per round. A meme is equivalent to one of the halves of such a block. The cutting point h is chosen from this interval to obtain blocks of various sizes, while at the same time avoiding blocks consisting of just a few carrier devices. Figure 4.3 illustrates a horizontal crossover.

This crossover operator aims to keep carrier device sequence costs low, since those are vital for a solution's fitness. The blocks ensure that we have a longer sequence of consecutive "half" rounds, thus helping us to achieve this goal. Many constraints, like the min/max block constraint or the forbidden color/block sequence constraint, depend on the sequence of blocks. The color costs also depend on the block sequence. Those constraints/costs

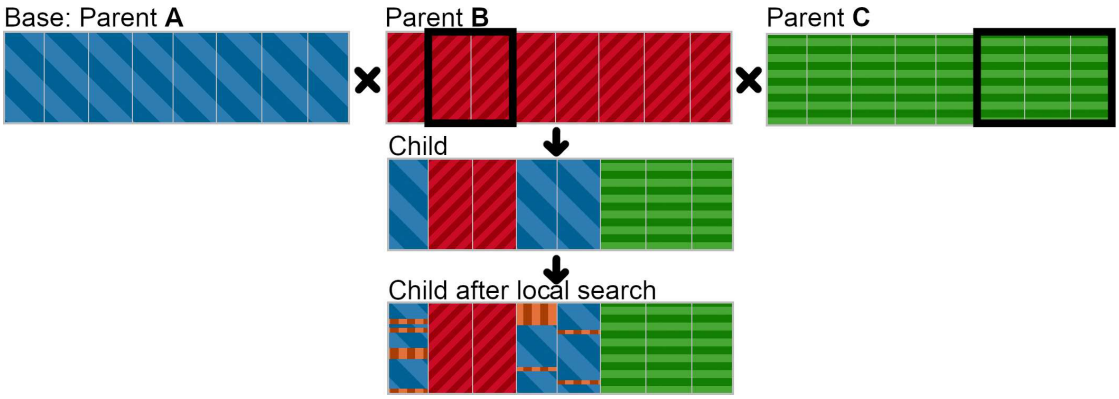

Figure 4.4: Cost and demand crossover. Blocks are selected for all parents except the base parent (A). The different blocks must be non-intersecting. Between these blocks must be a buffer zone with a minimum length of one round. Offspring is created by merging the blocks and the base solution together. Then, local search is conducted, but is only allowed to modify (i.e. repair) the buffer zones.

suffer from switching between solutions, which occurs now twice as often compared to the vertical crossover. Again, since we are merging different solutions together, some of the problems mentioned at the start of the chapter may occur.

4.2.3 Costs and Demand Crossover

This crossover operator is far more complex than the two crossover operators described above. The first difference is that the number of parents is not fixed at two, but is arbitrary (> 2). The second difference is that we conduct local search to repair the solution at the merging point.

There is one base parent solution. Blocks of rounds are chosen from different other parent solutions replacing those rounds in the base solution - those are the memes. Between the selected blocks must be a buffer zone. This buffer zone consists of rounds of the base solution and has a minimum size of one round. The size of the blocks is randomly chosen from the interval $[\log_3 l, \log_2 l]$, where l is the instance length. It's the same block size as the horizontal crossover, chosen for the same reasons stated there. The crossover's name comes from the fact that those blocks are chosen by their costs (color + carrier device sequence) and by a score of demand fulfillment.

The calculation of a block's costs is done in two steps. First, for each material, a scarcity score is calculated, as shown in Algorithm 4.2. The scarcity is calculated by contrasting the demanded material amount with the estimated amount, which can be held by the available carrier devices. Eventually, we have a score for each material, which is then normalized in the interval $[0.3, 1]$. This score is independent of the solution and depends only on the instance. Therefore it is only calculated once.

Algorithm 4.2: Material scarcity calculation

Input: Instance $\mathcal{I}$
Output: An array mapping material to its normalized scarcity

```
1  maximumMaterial ← initialize array with 0 for each material id
2  foreach carrier C in I.carriers do
3      foreach configuration CO in C.configurations do
4          foreach material capacity MC in CO.materialCapacities do
5              maximumMaterial[MC.materialId] += ∑(C.availabilities) *
                 MC.capacity
6          end
7      end
8  end
9  demandsByMaterial ← summed demand by material
10 materialScarcity ← []
11 foreach demand D in demandsByMaterial do
12     materialScarcity[D.materialId] ← sqrt( D / maximumMaterial[D.materialId] )
13 end
14 normalizedMaterialScarcity ← normalize materialScarcity in interval [0.3, 1],
     where higher score means higher scarcity
15 return normalizedMaterialScarcity
```

The second step is the calculation of the demand fulfillment and the final score for the block. Algorithm 4.3 depicts the implementation. The materials' scarcity calculated in step 1 is used here. For each of the carrier devices in the block's rounds, we look at the demands that this carrier device can fulfill. A carrier device gets a higher score if it fulfills urgent demands. This score gets divided by the material scarcity calculated in step 1. The score for the whole block is the average of all the carrier devices' scores.

The block's *final costs* are the block's color and carrier device change costs multiplied with the score from step 2.

After creating the child solution by merging the base solution and the other parent solutions' blocks together, a special local search operator is conducted. The operator is only allowed to manipulate the buffer zones. There is a higher chance in these zones than in the blocks for possible improvement since the blocks have been chosen for their low cost. Also, when merging parts of different solutions, constraints can easily be violated. The local search operator is stopped if no improvement is found for 6 cycles, or if the time limit of 1 second is passed.

4.3 Algorithm

All steps of the base algorithm depicted in 4.1 are illustrated in this chapter.

Algorithm 4.3: Cost and demand crossover cost calculation

Input: A block of rounds $\mathcal{B}$, the materials' scarcity $\mathcal{MS}$, the block's costs c
Output: The block's costs multiplied with the demand and scarcity score.

```
1  demandImportance = []
2  foreach round R in B do
3      foreach carrier C in R do
4          foreach material M on C do
5              ND ← most urgent demand affected by M
6              if ND is empty then
7                  importance ← 0
8              else
9                  importance ← (1−rounds until ND) / (rounds until end∗MS[M.id])
10             end
11             demandImportance[] ← importance
12         end
13     end
14 end
   // score is in [0,1], lower score means more important block
15 score = ∑(demandImportance) / demandImportance.length
16 return c ∗ (1 − score)
```

4.3.1 Selection

Selection of solutions is done at different stages of the algorithm.

- Selection of the *elitist*.

  The elitist is selected at the start of each generation, by choosing the individual with the best fitness (line 3 in Algorithm 4.1). It is selected to survive the current round. The elitist can still be selected as a parent for crossover operators or for mutation. Those operators are immutable, i.e. they do not modify the parents. Thus, the elitist will never be changed during this phase.

- Selection of individuals for crossover operators via *k-tournament* (see Miller and Goldberg [1996]).

  Crossover operators use memes of parents to create child individuals. The parents are chosen by running a k-tournament (line 4 in Algorithm 4.1). The k-tournament operator takes $k < |\mathcal{P}|$ individuals from the current generation and performs a k-tournament. The fittest individual out of the k competitors wins, and thus is selected. But there are different fitness functions for the various crossover operators - since for the cost and demand crossover blocks are competing to be selected instead of full individuals. The number of times a k-tournament is run likewise

depends on the crossover operator. For binary crossover operators, two parents are selected, while more parents are selected for the cost and demand. The k-tournament operator is also used for the mutation operator - where it only needs to be executed once.

- Selection of the *current generation's survivors (shrinking)*.

  This selection is only performed if the algorithm is configured to execute more crossover and mutation operators and thus create more offspring than there are members in the initial population ($\mathcal{N} > |\mathcal{P}| - 1$).

  In our implementation, the population size always stays the same. So only a certain number of created children can survive the current generation. The best $p = |\mathcal{P}| - 1$, which are non-duplicates, are selected for survival (line 7 in Algorithm 4.1). If there are too many duplicates in the new generation ($> 10\%$ of the population), we include individuals of the old generation for survival (i.e. the parents). This is done to prevent the population from converging towards the elitist, which would frequently happen before introducing this change. Eventually, the elitist is added to the new population.

4.3.2 Crossover and Mutation

Offspring is created by applying memetic operators to the population, like crossover and mutation operators. Mutation is done with probability m, while crossover operators are executed with probability $1 - m$. The parents for the crossover operators, as well as the individuals to be mutated, are selected via a k-tournament operator. m and the probabilities for the different crossover operators are passed as parameters to the algorithm (see Chapter 5 for a list of all parameters).

Our mutation operator chooses and executes a random local search move (insert/delete/swap block). The move is selected randomly and does not have to be at a conflict's position. The randomness of the move is on purpose - it is a move the local search algorithm would probably not make, thus promotes diversity in the population.

4.3.3 Local Search

The local search algorithm we use for improving the selected individuals is taken from Winter et al. [2019].

Local search is conducted on a few selected individuals $l \leq |\mathcal{P}|$ each generation (line 6 in Algorithm 4.1). The number of individuals selected for local search depends on the parameter ilf (individual learning frequency): $l = |\mathcal{P}| * ilf$. Child solutions are selected randomly for local search - fitness is not taken into account. The time reserved for local search is controlled by the parameter ili (individual learning intensity), denoted in seconds.

Local search specific parameters like the temperature are saved and passed to the next local search call. We save those local search parameters since the population converges

together. For a population consisting of rather fit individuals, the temperature thus will already be lower. If no improvements are found for a certain amount of time, reheating is used to escape local optima.

The local search algorithm eliminates violated hard constraints first, due to the cost function penalizing them heavily. For one hard constraint violation, a cost penalty equaling the maximum objective value is added.

There are three neighborhood moves: insertion, deletion, and swapping of carriers. All these moves can be done as block moves (modifying successive carriers).

The neighborhood is generated via min-conflicts heuristic. Positions involved in constraint violations and positions missing carriers are tracked. Some of those positions are randomly selected, and fitting moves are generated. A simulated annealing move acceptance function decides whether a move should be accepted (see Kirkpatrick et al. [1983]). The temperature function also supports reheating - a technique to escape local maxima in case no improvements can be found for a certain amount of iterations.

4.3.4 Initialization

The fitness and diversity of the initial population have a big influence on genetic and memetic algorithms' performance. There is a trade-off between execution time, quality, and population diversity when creating the initial population. We implemented three construction strategies. They are vastly different in terms of the qualities mentioned above.

Random Construction

The random construction is the fastest one of the three construction strategies. This strategy yields completely random solutions. Due to the fact that carriers and colors are randomly chosen, population diversity is very high.

Random Greedy Construction

This is another very fast construction strategy. Its implementation is outlined in Algorithm 4.4. Carriers and colors are again chosen randomly, but they cannot violate some of the constraints. Carrier block size constraints are always met, and only permitted color/carrier sequences are used. The population diversity is again very high since we choose carriers and colors randomly. The difference is that we try to avoid conflicts of minimum/maximum block size and of forbidden color/carrier sequences.

Greedy Construction

The third construction strategy is taken from Winter et al. [2019]. This construction strategy is a lot slower than the first two - and a lot more sophisticated. It is needed to achieve feasible results for the largest instances - see Chapter 5 for details.

Algorithm 4.4: Random greedy construction

Input: The history round $\mathcal{H}$, the number of rounds $\mathcal{R}$
Output: A solution without block size, block transition, color transition violations.

```
1  solution = H
2  blockSize = 0
3  color = last color in history round
4  foreach round r in R do
5      roundlength ← random in interval [min., max. round length]
6      for pos = 1 to roundlength do
7          if blocksize == 0 then
8              currentCarrier ← random carrier with allowed transition from last
                carrier
9              blockSize ← random blocksize in [min block size, max block size]
10         end
11         if random ∈ [0, 1] > 0.5 then
12             color = random color with allowed transition from last color
13         end
14         solution[r][pos] = newCarrier(currentCarrier, color)
15     end
16 end
17 return solution
```

The greedy construction heuristic is a two-phase algorithm. In the first phase, the round layout is constructed. In this step, carrier configurations and colors are assigned to the rounds, but not yet put in sequence, while considering the problem's hard constraints.

In the second phase, the carrier sequence for each round is determined. Hard constraints concerning sequencing are taken into consideration while trying to keep the amount of carrier and color changes low.

Once the greedy construction terminates and yields a single solution, we have to generate a full population from this one solution because of time constraints - for the largest instances, the construction can take up to 20 minutes. Therefore, for each additional population member, we modify the yielded solution. 3% of the carriers are modified by random local search moves. Afterwards, local search is conducted for 5 seconds to repair some of the conflicts introduced by the modifications.

4.4 Crossover Delta Evaluation

Evaluating solutions is a task which takes a lot of computation power. The costliest part of a local search algorithm is often the evaluation of the new solutions. A technique to speed up solution evaluation is *delta evaluation*. We can calculate the fitness of a

child solution s by using the fitness of its parent solutions $p_1, ..., p_n$, some cached data structures, and information about the genetic differences caused by the crossover or mutation operator.

Suppose we have a solution and its fitness. When applying a move, we would normally evaluate the modified solution from scratch. With delta evaluation, we can just evaluate the differences in the part of the solution that has changed. If our move adds additional carriers, we just need to consider how those carriers affect the various constraints - which the Paint Shop Scheduling Problem has many of (see Chapter 2). For example, for the delta evaluation of the demand constraint, we cache (among others) the number of scheduled pieces until a round. The move's impact on the demand constraint can easily be calculated by using this cache and information about the performed move, thus avoiding iterating through all the solution's rounds.

4.4.1 Full delta evaluation for crossover operators

For local search moves (insert/delete/swap blocks), Winter et al. [2019] already implemented delta evaluation. Based on their implementation, we add delta evaluation logic for each of our crossover operators.

One problem with this approach is that a lot of large data structures have to be cached. We also have to create a deep copy of those data structures for new population members. This is due to the fact that we cannot just modify one of the parent solutions and create the child by applying the crossover operator. Parents need to be immutable since they could be

1. selected as a parent for another crossover operator.
2. selected for mutation.
3. taken over into the next round (if there are too many duplicates).

For each new child solution, all the data structures need to be copied. Creating offspring is an essential task of a memetic algorithm, making this copy process very costly. Ultimately, copying those data structures makes the delta evaluation slower than just evaluating each solution from scratch.

4.4.2 Caching the carrier change costs

To improve the performance of the delta evaluation, we completely changed the approach. Via profiling, we identified the most time-consuming constraint evaluators. The constraint which takes (depending on the instance) 60% to 95% of the evaluation's runtime is the carrier change constraint. This is because, for this constraint, the *longest common sub-sequence* between each pair of consecutive rounds has to be calculated. Instead of deep copying the large data structures for each new child solution, we just cache the costs

of carrier changes per round directly in the solution. This simplifies the cost calculation a lot.

Let's assume we have a vertical crossover operator and take rounds 1 to 5 from solution A, and rounds 6 to 10 from solution B to create child solution C. We already know the carrier change costs from 9 of the 10 rounds for C and can just copy them from A and B's cache. We only need to calculate the carrier change costs for the "cutting" point, i.e. round 5.

Just by adding this simple improvement, the costs of evaluating child solutions have been reduced by 50% to 80%. For the largest instance available (*instance-200R*, runtime 1 hour, population of 10, no local search), the performance gained is especially impressive. Before, the algorithm only managed to create 6 generations while after the improvement, 43 generations could be created within the same runtime.

CHAPTER 5

Empirical Evaluation

In this chapter, we evaluate our memetic algorithm presented in Chapter 4. As memetic algorithms are often highly parameterizable, we configure many parameters for our algorithm, which may have an influence on the solutions' fitness. We use a state-of-the-art automated parameter tuning algorithm to find efficient parameter configurations. We then compare the automatically tuned configurations to a set of manually tuned configurations and evaluate their results statistically. Finally, we compare the solutions obtained by our algorithms to the best literature results.

5.1 Methodology

Since our algorithm is non-deterministic, 10 stochastic runs are executed for each parameter configuration. The result of those runs is used to calculate the mean, best, and worst fitness, and the standard deviation for the parameter configuration. All cost values listed in this chapter are averaged values if not specified otherwise.

Assessing performance differences between parameter settings may be hard when just considering the absolute solution costs for the different instances. To compare results between different benchmark instances, we use relative performance measures like *Relative Percentage Deviation (RPD)* and *Relative Deviation Index (RDI)*.

Equation 5.1 shows the calculation of the *Relative Percentage Deviation (RPD)*. For each instance I and solution S a relative cost is calculated. A lower score means that the solution's fitness is better. The best solution has a score of 0. A score of 0.5 for example means that the solution's fitness is 50% worse than the best solution's fitness.

$$RPD_{I,S} = \frac{cost_{I,S} - best_I}{best_I} \tag{5.1}$$

The second measure we use to calculate relative performance of configurations is the *Relative Deviation Index (RDI)* (see Equation 5.2). The difference to RPD is that the scores are scaled in the interval $[0, 1]$. This time, a score of 0.5 means that the solution's fitness lies exactly in the middle between the best and the worst solution's fitness.

$$RDI_{I,S} = \frac{cost_{I,S} - best_I}{worst_I - best_I} \tag{5.2}$$

We use the *Wilcoxon signed-rank test* as statistical method to assess whether the means of solution costs of two parameter settings differ significantly. It tests the null hypothesis that two related paired samples x and y come from the same distribution. In particular, it tests whether the differences $x - y$ follow a symmetric distribution around zero. It's a non-parametric version of the paired T-test.

5.2 Setup

5.2.1 Instances

The problem instances we use for benchmarking our algorithm are taken from Winter et al. [2019]. They provide 24 problem instances based on real life planning scenarios of the automotive industry. The instances have recently been used as benchmark instances for this problem. All instances are publicly available for download.[1]

The instances have six different planning horizons of 7, 20, 50, 70, 100, 200 rounds. For each of the planning horizons, there are two instances - one of them imposes forbidden color and carrier sequences. There are 12 big instances and 12 small instances. Small instances were created by scaling down big instances, via random selection of roughly 5 percent of the carrier devices, demands, materials, colors.

The instance set features a wide variety in instance size and complexity, which is ideal for spotting strengths and weaknesses of our algorithm and of certain parameter settings.

5.2.2 Testing Environment

We impose a time limit of one hour for every run. Each parameter setting needs to be tested for all instances of a type (small/big). To get a representative result, we repeat each run 10 times with different random seeds. This means that 120 hours of CPU time are needed to test a parameter configuration on one of the instance sets.

Since we have many different parameter settings to test, a lot of computation time is needed. We used a Intel Xeon E5-2650 v4 2.20GHz CPU for our benchmarks. The CPU has 12 cores and 24 threads. A run is executed on a single thread of the CPU. The whole system has 256 GB of RAM, so one thread has about 10 GB of RAM available. We had a cluster of 12 of those machines at disposal thanks to the DBAI institute of the TU Wien.

[1] https://dbai.tuwien.ac.at/staff/winter/ps_instances.zip

Since we got the original code for the algorithm from the authors of Winter et al. [2019] we could repeat the experiments on the same hardware for comparison. This allows a fair comparison of their approach with our memetic algorithm. We note here that the variant we use in this section slightly differs from the version that Winter et al. [2019] used for their experiments. Thus results may be different from literature results. The results of the repeated experiments are used in many tables and plots in the following sections. They are marked with the identifier *SA* (or *SA_G*, if the greedy construction was executed as well).

5.2.3 Parameters

There are various parameters for our memetic algorithm and the local improvement step of the algorithm. The memetic algorithm can be controlled via the following parameters:

- Population size p: A small population size can lead to a less diverse population and not enough memetic material. However, with a big population size fewer generations will be generated.
- Construction strategy c: Sets the strategy to construct the initial population. The different strategies are presented in Section 4.3.4. All of them have different strengths and weaknesses.
- K-tournament competitors k: The number of competitors for the k-tournament in the selection phase of the algorithm. Must be in the interval $[1, p]$.
- Individual learning frequency ilf: This parameter controls the number of individuals to be improved in the local search phase of the algorithm. It lies in the interval $[0, 1]$. To get the number of individuals which are improved, it is multiplied with the population size p.
- Individual learning intensity ili: Determines how long local search is conducted for each selected individual. Specified in seconds.
- Mutation frequency mf: The share of individuals to be mutated each generation is specified by this parameter. Must be in the interval $[0, 1]$.
- Vertical crossover frequency vcf: Specifies the crossover frequency for the vertical crossover operator. It is in the interval $[0, 1]$. The sum of all crossover operator frequencies must be 1 ($vcf + hcf + cdcf = 1$).
- Horizontal crossover frequency hcf: Specifies the crossover frequency for the horizontal crossover operator. It is the interval $[0, 1]$.
- Costs and demand crossover frequency $cdcf$: Specifies the crossover frequency for the costs and demand crossover operator. It is the interval $[0, 1]$.

- Crossover population size *cp*: This parameter controls the population size generated by application of crossover operators - in other words the amount of new offspring which is generated. It must be greater or equal to the population size ($cp \geq p$). For a detailed description on this parameter's influence on the shrinking part of the algorithm, see Section 4.3.1.

For the local improvement step, we use the simulated annealing approach that uses an additional tabu list as it was configured by Winter et al. [2019].

Therefore, the local search default parameters used in our benchmarks were the following:

- Move acceptance method: simulated annealing.
- Move exploration method: min-conflicts.
- Neighbor selection method: tabu search.
- Initial temperature for simulated annealing: 0.025
- Relative tabu list length: 0.001
- Cooldown rate for simulated annealing: 0.95

5.3 Automated Parameter Tuning

Finding efficient parameters for an algorithm is very challenging if a large number of parameters are available. This is because the resulting combinatorial space of parameter settings is extremely large. Parameter tuning, if performed manually by hand, is tedious work. And if the number of parameters grows, the combinatorial space of parameter settings is too large to be explored manually.

There are several tools in literature to solve this algorithm configuration problem, like irace (López-Ibáñez et al. [2016]), SPOT (Bartz-Beielstein [2010]), ParamILS (Hutter et al. [2009]), and spearmint (Snoek et al. [2012]). Hutter et al. [2011] proposed SMAC (sequential model-based algorithm configuration). SMAC is one of the state-of-the-art automated parameter tuning algorithms and is applicable for general algorithm configuration problems.

SMAC constructs a random forest model to predict the algorithm's performance for different parameter configurations. With that model, promising configurations are selected. The random forest is created by sub-sampling the data T times and fitting T regression trees. For each of those trees, a prediction for the configuration is made. Finally, the empirical mean and variance across all predictions are taken.

This selection process uses an approach the authors call "aggressive racing". Racing stands for executing few runs for poor and many runs for good configurations. Poor configurations are aggressively rejected - this can often already be done after a single run.

The selected parameters are then passed to the algorithm. On completion, the selected configuration is compared to the incumbent, and the incumbent is updated if needed. The models are also updated after each run.

SMAC has further been developed, and we use the latest version SMAC3 Lindauer et al. [2017]. SMAC has to be supplied with the following data about the algorithm and its parameters:

- The executable.
- A list of problem instances.
- A list of parameters. This list includes the parameter type (nominal, ordinal, integer, float), and the corresponding allowed values or ranges.
- A default configuration that SMAC uses as a starting point to explore the parameter configuration space.

5.3.1 SMAC configuration

Our algorithm can be supplied with many continuous parameters, which would lead to a large parameter configuration space (PCS). To reduce the complexity of the PCS, we declare the continuous parameters as ordinal parameters and pass a sequence of permitted reasonable values. Those values, as well as the default configuration, are selected by manual tuning.

Our parameter configuration is shown in Table 5.1.

Parameter	**Scale of measure**	**Value sequence**	**Default value**
Population size p	Ordinal	[2, 5, 10, 15, 30]	10
Crossover population size cp*	Ordinal	[1.0, 1.2, 1.4, 2]	1.4
K tournament competitors k*	Ordinal	[0.0, 0.15, 0.35, 0.5]	0.15
Individual learning frequency ilf	Ordinal	[0.0, 0.15, 0.35, 0.5]	0.15
Individual learning intensity ili	Ordinal	[3, 8, 20, 45]	20
Mutation frequency mf	Ordinal	[0.0, 0.1, 0.2, 0.4]	0.2
Crossover probabilities cop*	Nominal	[<1.0, 0.0, 0.0>, <0.0, 0.0, 1.0>, <0.4, 0.3, 0.3>, <0.5, 0.0, 0.5>, <0.5, 0.25, 0.25>, <0.25, 0.25, 0.5>]	<0.5, 0.0, 0.5>

Table 5.1: Parameter configuration space (PCS) supplied to SMAC.

Since no dependencies between parameters can be modeled (e.g. $cp \geq p$), several parameters had to be adapted (tagged with *). We add an option for SMAC to supply cp as a multiplier for p, thus avoiding invalid parameter configurations. k is also defined as multiplier for p, since the dependency $k < p$ cannot be modeled. The sum of all crossover

probabilities must be 1. Therefore we defined them as a 3-tuple with the order: vertical crossover, horizontal crossover, cost and demand crossover.

The benchmarks were executed on one of the machines of the cluster described in Section 5.2.2. Since SMAC supports parallel execution, we were able to utilize all 24 threads of the machine concurrently.

As runtime, we defined 30 minutes for each call of our algorithm. We performed parameter tuning for the small and the big instance set separately. Each of those had a runtime of one week - which means that SMAC was able to execute 8064 runs in that time.

For the small instance set, we used random construction. The greedy construction strategy was used for big instances. This was done for two reasons. Firstly, to reduce the PCS. Secondly, because valid solutions cannot be attained within time constraints when starting with random solutions for the biggest instances. The results obtained by different construction strategies are presented in Section 5.4.1.

5.3.2 SMAC results

Rank	**ID**	p	cp	k	ilf	ili	mf	cop	**Estimated costs**
1	P15_R1_S	15	1.0	0.5	0.15	8	0.0	0.5, 0.0, 0.5	2221
2	P2_R2_S	2	2.0	0.0	0.35	45	0.2	0.5, 0.0, 0.5	2272
3	P10_R3_S	10	1.0	0.5	0.35	8	0.2	0.5, 0.0, 0.5	2314

Table 5.2: Best three incumbents found by SMAC for the set of small instances.

Table 5.2 depicts the three best incumbent configurations found by SMAC for the small instance set. The estimated costs column depicts SMAC's performance measure for this configuration, based on our cost function.

Vertical crossover and cost and demand crossover are the preferred memetic operators. When examining the configurations, we can see that all three configurations use local search on few individuals and for a short time. The second configuration has a population of 2. Therefore this configuration minimizes crossover operators and maximizes local search time.

Rank	**ID**	p	cp	k	ilf	ili	mf	cop	**Estimated costs**
1	P2_R1_B	2	1.0	0.15	0.5	45	0.6	0.5, 0.0, 0.5	10657626190750
2	P2_R2_B	2	1.2	0.0	0.5	45	0.4	0.25, 0.25, 0.5	10707040649979
3	P2_R3_B	2	1.0	0.0	0.5	45	0.4	1.0, 0.0, 0.0	10786331184502

Table 5.3: Best three incumbents found by SMAC for the set of big instances.

The best three incumbents for big instances are shown in Table 5.3. All three configurations have a population size of 2. This has multiple reasons:

- As we will see in Section 5.4.1, a run with a bigger population may yield solutions with better fitness, but sometimes fails to yield feasible solutions for the largest instances. Runs with a smaller population often don't have this problem of not finding a feasible solution. Because they are able to produce more generations, the local improvement step is also executed more often. Hence we assume that local search excels in making solutions viable. It does so in a shorter time than the memetic operators, which get executed more often for solutions with a bigger population.

- If solutions violate any hard constraints, a severe penalty is added to their costs. The penalty added for each hard constraint violation is equal to a solution's maximum possible costs for the current instance. Thus a configuration achieving mediocre costs, but never validating a hard constraint violation, is ranked far better than a configuration achieving good costs but violating hard constraints once.

- As stated earlier, each run for a certain instance and parameter configuration has a limited runtime of 30 minutes. For large instances, the greedy construction strategy is used. This strategy takes a relatively long time. For an instance with 7 rounds, it takes 2 minutes. For the largest instance with 200 rounds, it takes almost 30 minutes. Thus, for some instances, the majority of time is consumed by the construction, and not enough time is left to make the solutions valid. This also explains the large cost values in the column "Estimated costs".

For the reasons stated above, the configurations with the least hard constraint violations win for large instances.

The configurations ranked first and third for small instances are able to achieve competitive solutions - for both small and big instances. There are some configurations in the list of incumbents for big instances, which have a population size of 10 or 15. These configurations are very similar to those found for small instances, in the way that they have a small individual learning frequency and a small individual learning intensity. Those configurations also mostly use either vertical crossover, cost and demand crossover, or a mix of both.

For future work, we could rerun SMAC parameter tuning with a longer runtime.

Detailed Results for the Incumbent Parameter Configurations

Table 5.4 depicts the results of the three incumbent parameter configurations for small instances, and compares them to the simulated annealing approach by Winter et al. [2019]. As stated in Section 5.1, the table contains averaged costs over 10 runs.

Instance	SA	P15_R1_S	P2_R2_S	P10_R3_S
7R-small	899.7	802.1	860.5	783.9
7R-HC-small	963.3	873.5	925.1	853.4
20R-small	996.4	1014.4	1014.7	1009.5
20R-HC-small	1106.8	1043.9	1045.1	965.1
50R-small	591.1	711.0	647.6	681.5
50R-HC-small	883.6	934.2	917.2	922.6
70R-small	1056.7	1391.3	1143.0	1295.2
70R-HC-small	2314.3	1912.5	1934.9	1885.7
100R-small	2362.9	1841.3	1888.6	1729.7
100R-HC-small	1156.2	1192.5	1154.0	1157.5
200R-small	5749.8	4288.5	4993.5	4692.8
200R-HC-small	5699.1	4997.6	5935.7	5159.5

Table 5.4: Average costs of the three incumbent parameter configurations for small instances. Costs in column *SA* are from the simulated annealing experiments executed on the same hardware.

Method	**#feasible**	**#best**	**Average std. dev. σ**
SA	120/120	4/12	36293.9
P15_R1_S	120/120	2/12	23020.6
P2_R2_S	120/120	0/12	24591.1
P10_R3_S	120/120	6/12	17985.6

Table 5.5: Comparison of results of the three incumbent parameter configurations for small instances. The number of feasible solutions yielded by each configuration is shown in column #feasible. Column #best depicts for how many of the 12 instances the configuration achieved the best solution.

For many instances, the configurations *P15_R1_S* and *P10_R3_S* perform better than *P2_R2_S* or *SA* (the simulated annealing experiments executed on the same hardware). The main difference between configurations *P15_R1_S*, *P10_R3_S* and *P2_R2_S* is the population size. A bigger and diverse population can be crucial to escape local optima. We will analyze the population size's influence on the solution quality in Section 5.4.1. A possible reason for the selection of an incumbent configuration with a population of 2 is, as mentioned above, the lower runtime of 30 minutes for SMAC runs.

Figure 5.1 depicts a box plot of the incumbents' RDI values. We can observe that the means of *P15_R1_S* and *P10_R3_S* are lower than the means of the other methods. This shows that they are competitive. Table 5.5 also shows the competitiveness of those two configurations. The average standard deviation σ is lower for more reliable methods, i.e. the 10 repeated experiments per instance yield solutions with more similar costs.

Instance	SA_G	P2_R1_B	P2_R2_B	P2_R3_B	P10_R3_S	P15_R1_S
7R	109556.6	114334.6	106799.1	107067.7	95310.8	91742.4
7R-HC	137216.0	131741.1	182162.2	128012.9	116966.9	106817.7
20R	281695.6	431467.8	371706.9	372942.5	328070.0	268998.8
20R-HC	328517.6	569607.4	568695.5	402294.1	325420.7	299419.8
50R	670416.9	910490.8	981004.1	867114.9	682281.4	647324.6
50R-HC	993294.6	1215625.8	1219386.9	1275473.2	928278.5	853311.8
70R	970832.7	1178815.4	1497888.5	1259019.4	976475.3	914746.7
70R-HC	1447172.7	1799967.7	1740843.1	1819032.7	1441404.4	1323317.0
100R	1714292.1	1915701.0	1840881.1	1814413.3	1586410.8	1494116.8
100R-HC	2602394.9	2587197.3	2610775.5	2549148.4	2106735.0	2071116.4
200R	3027137.3	3100780.3	3123140.0	3095737.5	3853381.5	-
200R-HC	4231962.1	4299495.9	4419954.5	4178882.1	-	-

Table 5.6: Average costs of different parameter configurations for big instances. Costs in column *SA_G* are from the simulated annealing + greedy construction experiments executed on the same hardware. The first three parameter settings are incumbents configurations found by SMAC. The two configurations with population 10 and 15 are incumbent configurations from the small instance set.

Method	**#feasible**	**#best**	**Average std. dev. σ ***
SA	120/120	1/12	44378095687.7
P2_R1_S	120/120	0/12	117254390700.1
P2_R2_S	120/120	0/12	113645887621.0
P2_R3_S	120/120	1/12	89950322274.4
P10_R3_S	110/120	0/12	2627687699.9
P15_R1_S	100/120	10/12	2020348345.6

Table 5.7: Comparison of results of the three incumbent parameter configurations and two configurations with a bigger population size for big instances. (*) Standard deviation for the instances 200R and 200R-HC is not taken into account, since not all methods yield valid solutions for these instances.

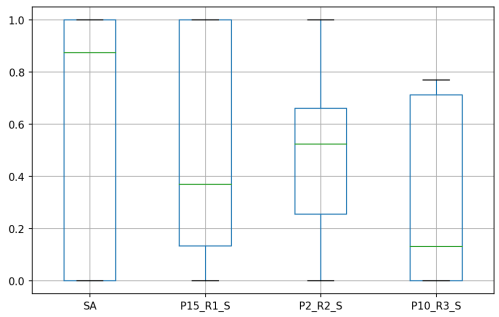

Figure 5.1: RDI values of incumbents configurations for the small instance set.

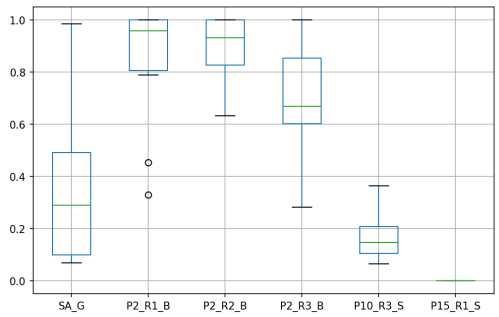

Figure 5.2: RDI values of incumbents configurations for the big instance set without the instances 200R and 200R-HC.

Table 5.6 shows the mean solution costs for incumbent configurations on the big instance set. All three incumbents found by SMAC have a population of two, and therefore rely mostly on local search. To gain insights on how a memetic algorithm with a proper population performs on this instance set, we also executed the experiments with two of the incumbent configurations found by SMAC for the small instance set (*P10_R3_S* and *P15_R1_S*).

The three incumbent configurations *P2_R1_B*, *P2_R2_B*, and *P2_R3_B* perform worse than Winter et al.'s local search algorithm. Interestingly, the two configurations *P10_R3_S* and *P15_R1_S* manage to outperform the three incumbent configurations as well as the local search algorithm on all instances bar 200R and 200R-HC. For those two instances, the configurations struggle to find valid solutions. That is also why SMAC treats configurations with population size of 2 preferential for this instance set.

RDI values for the different configurations are shown in Figure 5.2. They also show that the two configurations taken from the small instance set outperform the other four configurations. Note that the instances 200R and 200R-HC were excluded when calculating the RDI values.

We also tested the statistical significance between the results achieved with different parameter configurations. The Mann-Whitney-Wilcoxon test using a confidence level of 0.95 showed that our memetic algorithm yields significantly improved results with configurations P15_R1_S and P10_R3_S, when compared to the results of the three configurations with population 2 (P2_R1_B, P2_R2_B, P2_R3_B).

5.4 Manual parameter tuning

In this section, we assess the performance of some of the parameters presented in Section 5.2.3. SMAC incumbent parameter configurations are used as a starting point

Configuration	p	k	m	vcf	hcf	$cdcf$	ilf	ili
P10_*	10	5	0.4	0	0	1	0.15	8
P15_*	15	7	0.0	0.5	0	0.5	0.15	8
P45_*	45	7	0.0	0.5	0	0.5	0.05	3
P100_*	100	7	0.0	0.5	0	0.5	0.02	3
P200_*	200	7	0.0	0.5	0	0.5	0.01	3

Table 5.8: Base parameter configurations with different population sizes used to compare construction strategies.

Instance	SA	P2_G	P10_G	P15_G	P45_G	P100_G	P200_G
7R-small	899.7	820.9	782.5	799.3	787.9	785.3	778.5
7R-HC-small	963.3	875.1	851.2	871.7	847.3	843.7	844.2
20R-small	996.4	1022.7	1018.2	1019.4	1024.6	1013.6	992.7
20R-HC-small	1106.8	1054.9	967.5	993.1	994.1	973.2	947.7
50R-small	591.1	653.7	644.3	676.2	681.2	695.4	812.5
50R-HC-small	883.6	918.0	915.1	922.8	928.4	922.9	1171.7
70R-small	1056.7	1142.8	1259.4	1355.2	1371.5	1369.8	2187.7
70R-HC-small	2314.3	1910.4	1842.5	1898.6	1788.9	1907.4	3984.1
100R-small	2362.9	1846.3	1762.0	2229.2	1902.2	1845.2	4988.5
100R-HC-small	1156.2	1153.0	1154.0	1157.2	1168.2	1162.0	1504.3
200R-small	5749.8	4892.0	4141.5	3599.9	2878.5	2375.7	16268.9
200R-HC-small	5699.1	6643.5	5655.1	5388.2	4523.0	3751.1	23377.0

Table 5.9: Average solution costs for the small instance set when using *greedy* construction.

for the parameter tests. The experiments in this section are conducted to find even better parameter configurations and to gain insights on the impact of parameters on the solutions' fitness.

5.4.1 Construction Strategies and Population

The different available construction strategies are presented in Section 4.3.4. When analyzing SMAC's results, we could not find feasible results for all problem instances. Therefore we decided to implement a new construction strategy - *warm start*. The new construction strategy is a combination of the greedy construction strategy and local search. Local search is conducted on the solution generated by the greedy construction until this solution is valid. Only then, a population of solutions is generated, and the memetic algorithm is started.

One of the most influential parameters of a memetic algorithm is the population size. For a population of 2 - as found by SMAC - most of the algorithm's time gets spent by

Instance	SA	P2_RG	P10_RG	P15_RG	P45_RG	P100_RG	P200_RG
7R-small	899.7	839.6	781.1	805.8	791.4	782.5	781.9
7R-HC-small	963.3	908.3	850.6	868.0	873.7	864.1	855.9
20R-small	996.4	1023.8	1005.1	1012.2	1005.4	999.2	990.6
20R-HC-small	1106.8	1081.2	966.9	989.6	1017.0	1015.8	976.4
50R-small	591.1	673.3	682.4	674.7	702.7	703.0	701.5
50R-HC-small	883.6	916.1	921.2	910.9	932.6	923.2	917.1
70R-small	1056.7	1141.6	1313.3	1276.5	1434.2	1399.7	1411.8
70R-HC-small	2314.3	2193.9	1996.2	1900.1	2021.4	2081.2	1893.5
100R-small	2362.9	2199.3	1966.5	1755.5	2328.3	2366.0	2340.1
100R-HC-small	1156.2	1153.5	1156.6	1154.4	1185.0	1154.9	1132.6
200R-small	5749.8	4779.6	3793.9	4013.2	2512.1	2202.9	2157.7
200R-HC-small	5699.1	6796.8	5869.5	5550.2	3774.9	2933.8	2374.7

Table 5.10: Average solution costs for the small instance set when using *random greedy* construction.

Instance	SA	P2_R	P10_R	P15_R	P45_R	P100_R	P200_R
7R-small	899.7	860.5	783.9	802.1	797.5	782.7	791.0
7R-HC-small	963.3	925.1	853.4	873.5	864.7	856.2	858.3
20R-small	996.4	1014.7	1009.5	1014.4	1021.2	1006.6	1007.1
20R-HC-small	1106.8	1045.1	965.1	1043.9	1047.2	1023.0	1006.7
50R-small	591.1	647.6	681.5	711.0	710.5	701.7	700.3
50R-HC-small	883.6	917.2	922.6	934.2	931.3	923.8	923.4
70R-small	1056.7	1143.0	1295.2	1391.3	1364.4	1381.7	1395.6
70R-HC-small	2314.3	1934.9	1885.7	1912.5	1925.8	1932.8	1897.7
100R-small	2362.9	1888.6	1729.7	1841.3	1864.7	1878.3	1887.1
100R-HC-small	1156.2	1154.0	1157.5	1192.5	1171.7	1150.2	1145.5
200R-small	5749.8	4993.5	4692.8	4288.5	3255.0	2571.1	2425.7
200R-HC-small	5699.1	5935.7	5159.5	4997.6	4007.2	3712.1	4124.6

Table 5.11: Average solution costs for the small instance set when using *random* construction.

Instances	SA_G	P2_R1_R	P2_R1_RG	P10_R	P10_RG
Instance-7R	109556.6	2236287.4	2150328.9	2212757.2	2106256.8
Instance-7R-HC	137216.0	2333154.8	2288932.0	2285751.7	2249209.4
Instance-20R	281695.6	8720835.7	8556915.2	9015126.1	8591958.3
Instance-20R-HC	328517.6	8971367.2	8891543.0	9240520.4	8997510.5
Instance-50R	670416.9	24730978.8	24612496.1	-	24955543.0
Instance-50R-HC	993294.6	25403611.4	25575222.8	-	26620642.5
Instance-70R	970832.7	36094142.3	35518433.8	-	37527371.5
Instance-70R-HC	1447172.7	36703137.2	36933203.8	-	39487911.2
Instance-100R	1714292.1	-	53098478.2	-	-
Instance-100R-HC	2602394.9	-	55642887.5	-	-
Instance-200R	3027137.3	-	-	-	-
Instance-200R-HC	4231962.1	-	-	-	-

Table 5.12: Average solution costs for the big instance set when using *random* and *random greedy* construction. Larger populations were omitted because they yield fewer feasible solutions than configurations *P10_R* and *P10_RG*.

Instances	SA_G	P2_R1_G	P10_G	P15_G	P45_G	P100_G
Instance-7R	109556.6	114334.6	95310.8	91742.4	90281.3	99649.1
Instance-7R-HC	137216.0	131741.1	116966.9	106817.7	108663.4	122050.6
Instance-20R	281695.6	431467.8	328070.0	268998.8	240431.7	251082.7
Instance-20R-HC	328517.6	569607.4	325420.7	299419.8	278706.4	283974.6
Instance-50R	670416.9	910490.8	682281.4	647324.6	645720.8	727030.1
Instance-50R-HC	993294.6	1215625.8	928278.5	853311.8	813142.0	1095052.7
Instance-70R	970832.7	1178815.4	976475.3	914746.7	891067.7	-
Instance-70R-HC	1447172.7	1799967.7	1441404.4	1323317.0	1454055.1	-
Instance-100R	1714292.1	1915701.0	1586410.8	1494116.8	1816630.7	-
Instance-100R-HC	2602394.9	2587197.3	2106735.0	2071116.4	-	-
Instance-200R	3027137.3	3100780.3	3853381.5	-	-	-
Instance-200R-HC	4231962.1	4299495.9	-	-	-	-

Table 5.13: Average solution costs for the big instance set when using *greedy* construction.

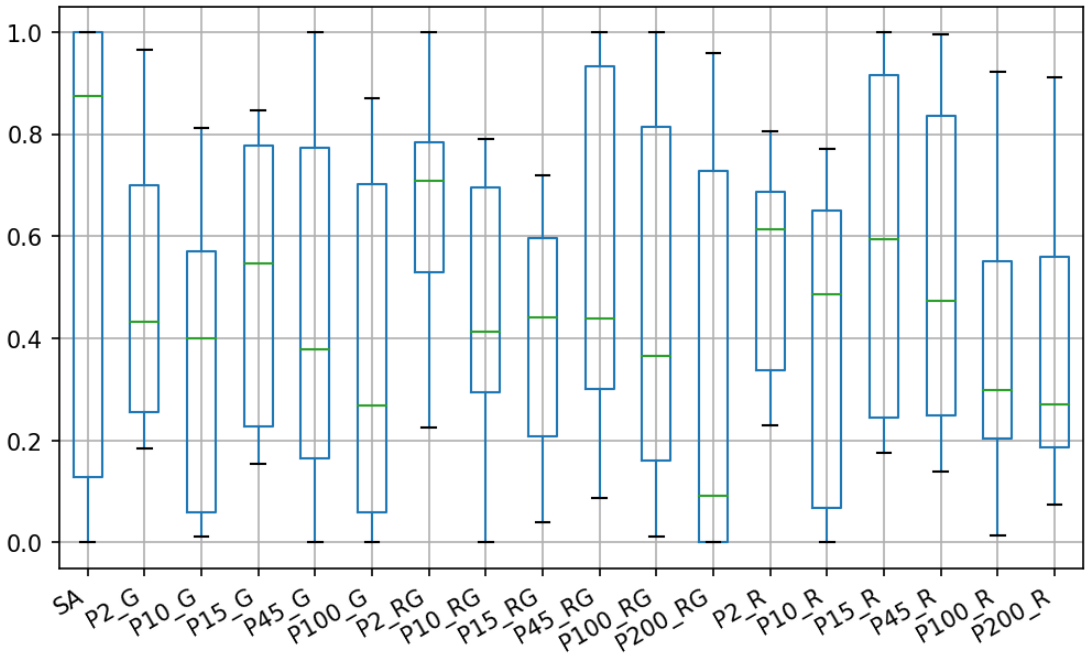

Figure 5.3: RDI values of parameter configurations with various construction strategies for the small instance set.

local improvement and not by memetic operators. We want to test the impact of larger population sizes on the solutions' quality.

The parameter configurations compared in this chapter are based on SMAC's incumbent configurations. Table 5.8 lists base configurations for the different construction strategies. The suffix of the configuration name depicts the construction strategy (greedy = G, warm start = WS, random = R, random greedy = RG).

For small instances, the differences between the construction strategies are very small, as can be seen in Tables 5.9, 5.10, and 5.11. For some small instances, the greedy construction strategy yields slightly worse results than random construction. We assume that's because the resulting population is less diverse. Interestingly, there are some instances (50R-small, 70R-small) where the local search approach from Winter et al. outperforms our algorithm, regardless of the construction strategy or population size. For other instances (200R-small, 200R-HC-small) the memetic algorithm outperforms the local search approach. When analyzing RDI values from Figure 5.3, we can see that all population sizes greater or equal to 15 are competitive.

For the big instance set, either the greedy or the warm start construction strategy is necessary to achieve competitive and valid solutions. Otherwise, when starting from a random solution, for most big instances, there is simply not enough time to fulfill all hard constraints, as can be seen in Table 5.12. Even though the algorithm may spend

Instances	SA_G	P2_R1_WS	P10_WS	P15_WS	P45_WS	P100_WS
Instance-7R	109556.6	133731.6	93810.4	88741.6	89551.1	96930.9
Instance-7R-HC	137216.0	163854.9	110059.4	105752.8	107845.0	119882.9
Instance-20R	281695.6	326556.0	255032.0	265804.9	267976.6	287578.3
Instance-20R-HC	328517.6	377448.8	292760.6	282509.3	302009.0	314397.9
Instance-50R	670416.9	752485.7	654113.0	657234.2	670240.7	702853.2
Instance-50R-HC	993294.6	1075940.1	959462.7	931522.9	957272.4	1060115.6
Instance-70R	970832.7	1224215.9	1107169.9	1132818.2	1111575.6	1290850.3
Instance-70R-HC	1447172.7	1576620.1	1405283.2	1392171.5	1416751.5	1638643.1
Instance-100R	1714292.1	1717952.2	1576961.2	1548580.0	1595445.3	1886488.5
Instance-100R-HC	2602394.9	2692828.2	2433705.3	2480865.7	2613455.8	3122616.2
Instance-200R	3027137.3	4028357.1	3159701.4	3849725.8	4342757.7	4526237.1
Instance-200R-HC	4231962.1	5280047.0	4246989.5	5064482.0	5276129.5	5813151.3

Table 5.14: Average solution costs for the big instance set when using *warm start* construction.

up to 20 minutes in the construction phase when using the greedy construction strategy, it is still vital for the solutions' quality. Tables 5.13 and 5.14 show the results for the greedy and the warm start construction strategies.

Figure 5.4 depicts RDI values for the various configuration strategies on the big instance set. Interestingly, the greedy construction strategy yields slightly better results if a feasible solution is found. The larger the population and the bigger the instance, the harder it is for the configurations with the greedy construction strategy to generate feasible solutions. Population sizes between 10 and 45 yield good results.

Table 5.15 compares the number of generations, which could be generated by our algorithm within the time constraints, for the different parameter configurations. We can clearly see that the number of generations for large instances and for larger population sizes is very small. We assume that this is the cause for the worse performance of the configuration P100_WS for almost all instances, and of configuration P45_WS for the biggest instances. Note that the number of generations for configuration P2_WS is quite consistent between the different instances. This is due to the low population size, as well as the high individual learning intensity (local search gets executed for 45 seconds after each generation on one individual).

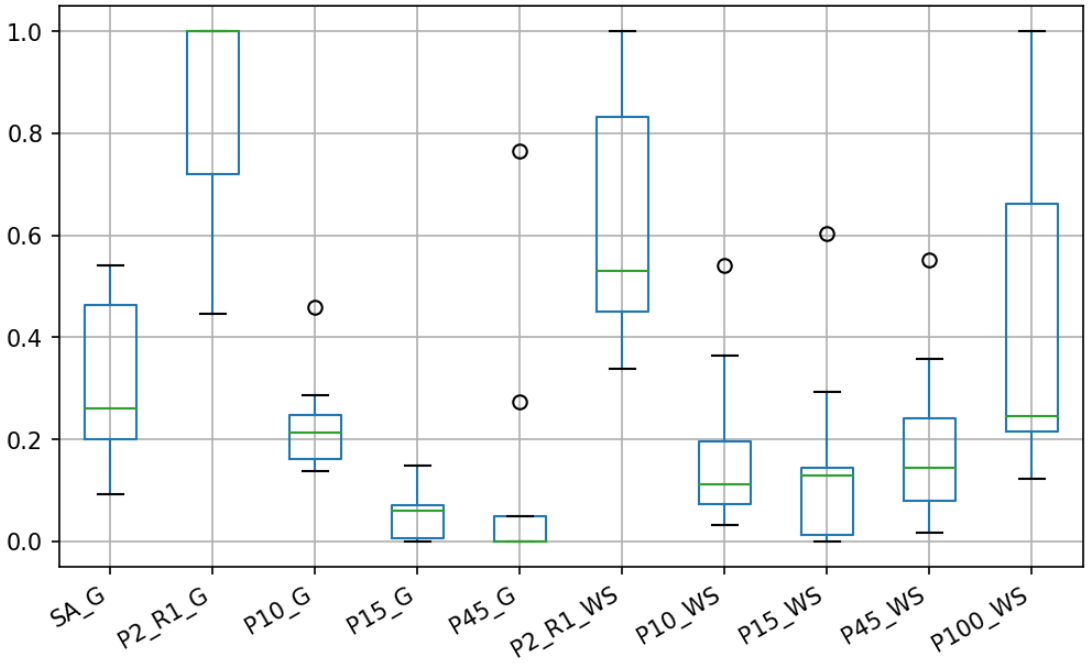

Figure 5.4: RDI values of parameter configurations with various construction strategies for the big instance set. Note that the instances 200R, 200R-HC, and 100R-HC were omitted since, for some parameter settings, no valid solutions could be found for those instances.

Instance	P2_WS	P10_WS	P45_WS	P100_WS
7R	77	383	326	210
7R-HC	76	380	313	201
20R	70	280	160	97
20R-HC	70	274	161	89
50R	63	232	92	33
50R-HC	63	207	85	30
70R	61	186	60	21
70R-HC	59	172	55	20
100R	57	167	48	18
100R-HC	55	153	41	15
200R	45	67	18	6
200R-HC	40	62	13	3

Table 5.15: Number of generated generations for different configurations.

5.4.2 Genetic Algorithm (ilf = 0)

An interesting research question is how the algorithm performs without local search, making it a genetic algorithm. Table 5.16 illustrates the different parameter configurations. All of the configurations have a mutation rate of 20%. Otherwise, no new genetic material would be produced during the evolutionary process.

Some instances are harder to solve than others for the genetic algorithm, as can be seen in Table 5.17. For the instances 70R-HC-small and 100R-small, none of the runs yield any feasible solutions. Figure 5.5 shows the corresponding RDI values. The population size does not have a big impact on the results' costs. The cost and demand crossover yields the best results, either alone or combined with the vertical crossover. In conclusion, the genetic algorithm struggles to find feasible solutions for some of the instances. If feasible solutions are found, the results are competitive when compared to the local search approach, albeit worse than the solutions found by our memetic algorithm.

For the big instance set, the greedy construction strategy was used for our experiments. The average solution costs can be seen in Table 5.18. The results are quite similar to the results of the small instance set, as the same crossover operator combinations yield the best results. Only crossover combinations, which include the cost and demand crossover, yield feasible solutions for instances with a planning horizon of more than 7 rounds. It seems like a little bit of local search is needed, albeit just in the form of the repair, which is done by our cost and demand crossover operator. The configurations *P15_C* and *P15_VC* yield quite competitive results, although they are unable to find feasible solutions for instances with a planning horizon of 200 rounds. The larger the population size, the harder it is for the genetic algorithm to find feasible solutions within time constraints.

Configuration	p	k	m	vcf	hcf	$cdcf$	ilf
P15_VH	15	3	0.2	0.5	0.5	0	0
P15_VC	15	3	0.2	0.5	0	0.5	0
P15_C	15	3	0.2	0	0	1	0
P30_VH	30	5	0.2	0.5	0.5	0	0
P30_VC	30	5	0.2	0.5	0	0.5	0
P30_C	30	5	0.2	0	0	1	0
P45_VH	45	7	0.2	0.5	0.5	0	0
P45_VC	45	7	0.2	0.5	0	0.5	0
P45_C	45	7	0.2	0	0	1	0
P100_VH	100	7	0.2	0.5	0.5	0	0
P100_VC	100	7	0.2	0.5	0	0.5	0
P100_C	100	7	0.2	0	0	1	0
P200_VH	200	7	0.2	0.5	0.5	0	0
P200_VC	200	7	0.2	0.5	0	0.5	0
P200_C	200	7	0.2	0	0	1	0

Table 5.16: Parameter configurations without local search, effectively resulting in a genetic algorithm

Instance	SA	P15_C	P15_VH	P15_VC	P30_C	P30_VH	P30_VC	P45_C	P45_VH
7R-small	899.7	784.4	802.0	785.6	811.2	878.9*	785.4	786.5	936.7*
7R-HC-small	963.3	860.1	910.3	854.8	859.7	959.6*	866.2	889.0	911.8
20R-small	996.4	991.5	983.2	1015.0	1002.8	1044.8	1033.0	1013.7	1093.9
20R-HC-small	1106.8	974.7*	1036.2*	989.2*	975.4*	1046.1*	971.4*	974.3*	1107.3*
50R-small	591.1	726.2	1142.7	752.3	735.0	1303.4	789.8	750.3	1428.3
50R-HC-small	883.6	951.8	1070.5	966.0	961.6	1166.4	1008.6	980.5	1238.3
70R-small	1056.7	1693.5	3559.8	1905.8	1774.6	3769.7	1921.7	1737.2	3799.1
70R-HC-small	2314.3	-	-	-	-	-	-	-	-
100R-small	2362.9	-	-	-	-	-	-	-	-
100R-HC-small	1156.2	1254.6	2401.2	1302.9	1224.6	2642.9	1291.0	1255.7	2935.1
200R-small	5749.8	4290.6	-	5549.0	3635.7	-	5491.4	-	-
200R-HC-small	5699.1	5990.4	-	8582.7	4505.1	-	7680.0	-	-

Instance	SA	P45_VC	P100_C	P100_VH	P100_VC	P200_C	P200_VH	P200_VC
7R-small	899.7	790.5	792.2	883.1	791.3	793.4	872.3*	791.5
7R-HC-small	963.3	927.1	869.6	958.2*	898.5	863.0	923.4*	882.6
20R-small	996.4	1013.9	1007.5	1113.0	1008.3	989.2	1120.6	999.8
20R-HC-small	1106.8	1007.7*	977.4*	1170.2*	989.8*	1000.6*	1161.7*	1000.6*
50R-small	591.1	795.4	735.5	1389.0	758.1	741.6	1392.2	753.8
50R-HC-small	883.6	1011.3	983.1	1232.8	1009.5	980.7	1238.5	1010.6
70R-small	1056.7	1921.6	1699.7	3805.3	1905.2	1678.2	3968.3	1915.2
70R-HC-small	2314.3	-	-	-	-	-	-	-
100R-small	2362.9	-	-	-	-	-	-	-
100R-HC-small	1156.2	1276.7	1206.8	2989.4	1271.4	1174.5	3198.5	1253.4
200R-small	5749.8	5007.1	-	-	4800.0	-	-	4613.5
200R-HC-small	5699.1	7810.1	-	-	8058.9	-	-	10793.3

Table 5.17: Average solution costs of different parameter configurations without local search for the small instance set.

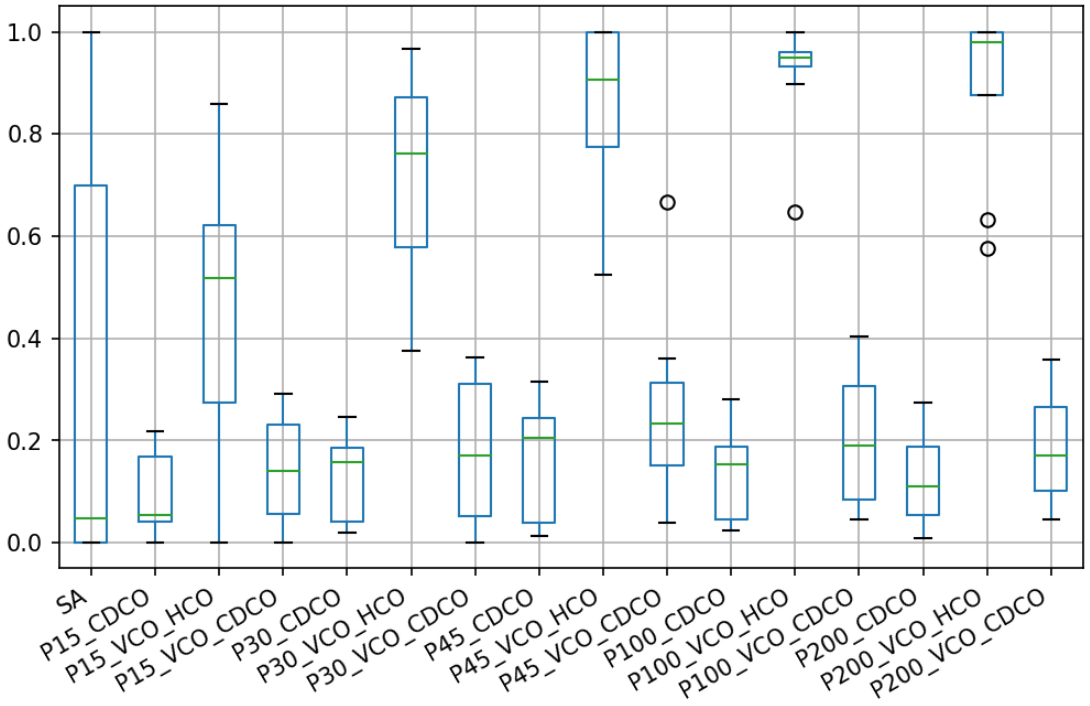

Figure 5.5: RDI values of parameter configurations without local search for the small instance set. Note that the instances 200R, 200R-HC, 100R, 70R-HC were omitted, since for some parameter settings no valid solutions could be found for those instances.

Instance	SA_G	P15_C	P15_VH	P15_VC	P30_C	P30_VH	P30_VC	P45_C	P45_VH
7R	109556.6	95151.8	153028.0	101946.5	97122.0	153130.3	105840.3	96082.9	152662.0
7R-HC	137216.0	112101.8	185812.1	124943.6	115584.9	184921.9	126863.0	117209.5	183910.6
20R	281695.6	234967.2	-	259506.9	231014.5	-	239301.0	228162.8	-
20R-HC	328517.6	254619.9	-	276750.1	254856.3	-	261157.5	255127.7	-
50R	670416.9	577941.6	-	582197.9	659541.5	-	593720.1	669332.2	-
50R-HC	993294.6	680482.5	-	700410.2*	814730.4	-	710528.7*	874762.7*	-
70R	970832.7	792308.6	-	819109.4*	942587.7*	-	880101.7	1005156.0*	-
70R-HC	1447172.7	1096920.1	-	1123514.0	-	-	1317347.6	-	-
100R	1714292.1	1424454.4*	-	1432806.5*	-	-	1617781.4*	-	-
100R-HC	2602394.9	2033819.2*	-	2035732.6*	-	-	2289221.5*	-	-
200R	3027137.3	-	-	-	-	-	-	-	-
200R-HC	4231962.1	-	-	-	-	-	-	-	-

Instance	SA_G	P45_VC	P100_C	P100_VH	P100_V_C	P200_C	P200_VH	P200_VC
7R	109556.6	103409.4	105544.3	152727.6	108223.4	116815.0	152825.6	118440.2
7R-HC	137216.0	125606.5	127802.6	185946.5	132798.5	141271.1	184746.6	148290.4
20R	281695.6	237256.9	235716.9	-	240985.8	254353.6	-	248576.4
20R-HC	328517.6	260334.4	259323.8	-	258791.1	296305.5	-	278941.3
50R	670416.9	614266.4	761657.5	-	728574.2	-	-	-
50R-HC	993294.6	749935.0	-	-	958620.0	-	-	-
70R	970832.7	942675.2 *	-	-	-	-	-	-
70R-HC	1447172.7	-	-	-	-	-	-	-
100R	1714292.1	-	-	-	-	-	-	-
100R-HC	2602394.9	-	-	-	-	-	-	-
200R	3027137.3	-	-	-	-	-	-	-
200R-HC	4231962.1	-	-	-	-	-	-	-

Table 5.18: Average solution costs of different parameter configurations without local search for the big instance set.

5.4.3 Crossover Operators

In this section, we compare the performance of our three novel crossover operators. Different combinations of the crossover operators mixed with different population sizes are tested.

Table 5.19 depicts the results for the small instance set. We can observe that the horizontal crossover operator performs the worst, especially for larger populations. The vertical crossover operator yields competitive results, while the best results are generated by the cost and demand crossover operator. Figure 5.6 shows the corresponding RDI values. Configuration P10_CDCO has the lowest median of all configurations. Interestingly, a bigger population can lead to improved solution quality for the small instance set's larger instances, as can be seen by configuration P45_CDCO and P100_CDCO.

For the large instance set, we tried all different combinations of crossover operators. The results are shown in Table 5.20. All configurations used the warm start construction strategy - that's why all configurations yielded feasible solutions. Looking at the RDI values in Figure 5.7, three combinations yield the best results: vertical crossover only, cost and demand crossover only, and a combination of both.

As we could already observe in Section 5.4.1, a population larger than 45 yields worse results for this instance set.

Instance	SA	P10_HCO	P10_VCO	P10_CDCO	P15_HCO	P15_VCO	P15_CDCO	P45_HCO	P45_VCO
7R-small	899.7	795.5	793.0	789.8	806.9	842.6	802.9	803.1	839.8
7R-HC-small	963.3	861.5	882.5	855.5	858.1	886.8	872.2	860.7	910.1
20R-small	996.4	1044.1	1037.6	1012.3	1073.9	1022.9	1028.2	1082.8	1033.0
20R-HC-small	1106.8	1020.5	1047.8	962.5	1092.7	1117.7	1022.3	1038.4	1109.2
50R-small	591.1	732.3	704.6	694.1	808.7	726.0	713.1	909.8	730.7
50R-HC-small	883.6	950.8	937.2	930.5	997.3	942.8	933.9	1075.0	944.1
70R-small	1056.7	1380.0	1311.5	1286.5	1809.9	1378.1	1347.2	2180.8	1464.5
70R-HC-small	2314.3	2036.2	1994.7	1955.4	1962.1	2046.2	1869.3*	2349.4	2023.7
100R-small	2362.9	1871.8	1891.8	1690.6	2100.0	1912.9*	1667.6*	3960.0	1918.1*
100R-HC-small	1156.2	1265.1	1199.1	1166.2	1463.1	1227.9	1182.3	1761.4	1233.0
200R-small	5749.8	5910.4	5145.0	4695.4	8061.5	5320.0	3487.7	13198.0	5305.3
200R-HC-small	5699.1	8439.6	6424.6	5483.9	13744.7	7164.5	4387.8	25979.8	6565.9

Instance	SA	P45_CDCO	P100_HCO	P100_VCO	P100_CDCO	P200_HCO	P200_VCO	P200_CDCO
7R-small	899.7	797.6	808.5	848.4	797.2	806.1	837.2	798.9
7R-HC-small	963.3	872.2	854.9	879.6	871.3	858.1	880.5	856.6
20R-small	996.4	1021.6	1119.6	1032.9	1011.7	1115.5	1019.5	1001.1
20R-HC-small	1106.8	1069.8	1046.5	1127.4	1063.5	1099.9	1116.7	1144.6
50R-small	591.1	718.5	1055.1	746.8	703.0	1344.4	734.0	689.3
50R-HC-small	883.6	932.1	1167.7	948.7	924.3	1299.3	947.1	923.1
70R-small	1056.7	1403.0	2693.3	1440.0	1398.7	2913.6	1506.5	1378.9
70R-HC-small	2314.3	2054.1*	3963.0*	2023.7*	1934.6	10818.0*	1999.6	1832.3
100R-small	2362.9	3261.0*	17840.0*	1861.8*	-	-	2071.2*	-
100R-HC-small	1156.2	1159.2	2514.0	1249.4	1139.1	3654.4	1256.2	1125.0
200R-small	5749.8	2700.4	20590.3	5264.2	2310.2	31102.8	5491.0	2542.3
200R-HC-small	5699.1	3177.9	38840.6	7087.2	3168.1	58164.1	8047.5	14269.4

Table 5.19: Average solution costs of different parameter configurations with different crossover rates for the small instance set.

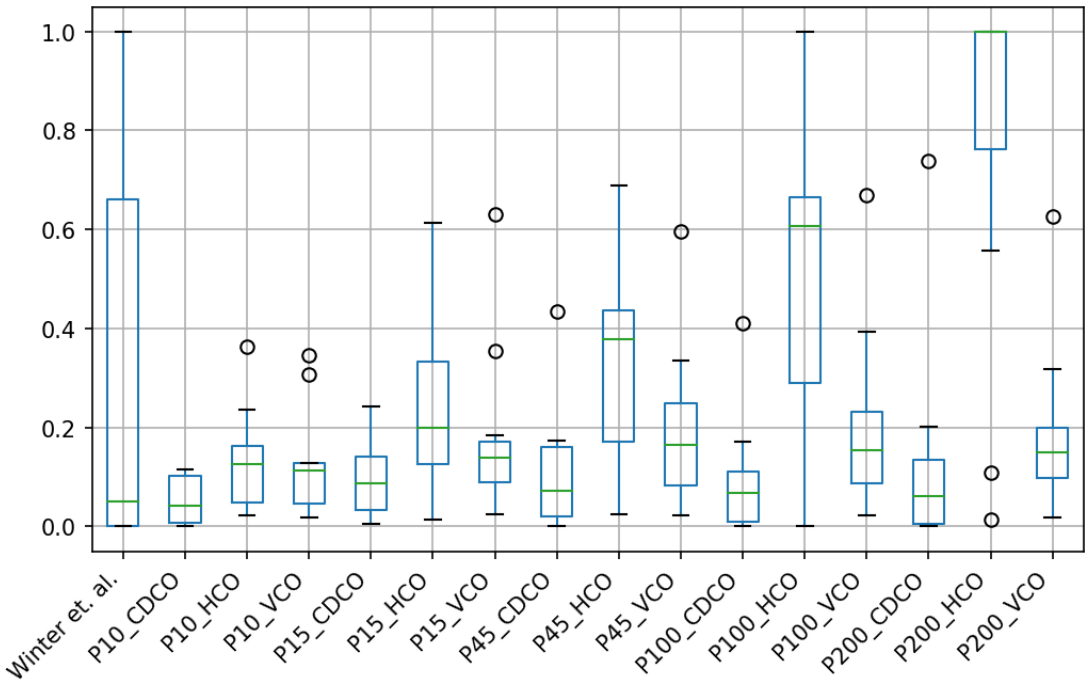

Figure 5.6: RDI values of parameter configurations with various crossover combinations for the small instance set. Note that the instance 100R was omitted since, for some of the parameter settings, no feasible solutions could be found for those instances.

Instance	SA_G	P10_HCO	P10_VCO	P10_CDCO	P10_VCO_HCO	P10_HCO_CDCO	P10_VCO_CDCO	P10_ALL_CO
7R	109556.6	135642.2	93105.9	93152.5	110468.3	104585.1	93810.4	97165.0
7R-HC	137216.0	167003.3	110987.2	110494.7	138360.0	132110.9	110059.4	127252.0
20R	281695.6	328693.4	261162.7	255376.4	302999.6	281116.5	255032.0	276357.4
20R-HC	328517.6	377825.9	302050.3	289103.5	344110.0	333667.4	292760.6	324622.4
50R	670416.9	733951.5	669079.2	654702.5	720228.4	694859.5	654113.0	690538.2
50R-HC	993294.6	1024209.0	957397.6	968996.0	1040150.5	982231.8	959462.7	1023048.0
70R	970832.7	1304480.3	1212560.6	1265530.8	1052881.9	985031.6	1107169.9	981908.4
70R-HC	1447172.7	1501652.9	1421057.1	1403349.3	1533526.6	1527990.8	1405283.2	1477004.5
100R	1714292.1	1661287.6	1589935.4	1579101.4	1668809.3	1692507.3	1576961.2	1684134.3
100R-HC	2602394.9	2608581.0	2495065.8	2628438.3	2597431.6	2442762.9	2433705.3	2542330.8
200R	3027137.3	3503080.1	3137690.2	3179761.1	3219991.0	3296270.1	3159701.4	3219566.5
200R-HC	4231962.1	4910667.4	4444151.8	5205527.1	4738519.2	4522788.7	4246989.5	4435883.4

Instance	SA_G	P45_HCO	P45_VCO	P45_CDCO	P45_VCO_HCO	P45_HCO_CDCO	P45_VCO_CDCO	P45_ALL_CO
7R	109556.6	152475.2	90164.4	96947.8	104317.2	98157.1	89551.1	95862.3
7R-HC	137216.0	184911.8	109788.7	112059.0	134653.0	128654.8	107845.0	121311.4
20R	281695.6	339813.4	269510.4	274122.3	296303.3	281794.0	267976.6	271063.9
20R-HC	328517.6	390865.5	310270.2	316762.5	346898.2	339625.5	302009.0	317877.3
50R	670416.9	875924.0	694628.4	671001.7	725505.6	697110.5	670240.7	695690.4
50R-HC	993294.6	1239611.6	968045.8	954519.6	1017658.7	1035095.6	957272.4	985147.2
70R	970832.7	1501527.6	1106568.1	1232144.8	1036757.3	1021789.1	1111575.6	998564.4
70R-HC	1447172.7	1846585.6	1468425.7	1392912.5	1575908.3	1554761.2	1416751.5	1503233.5
100R	1714292.1	1986713.8	1610142.1	1645387.8	1724689.1	1857470.2	1595445.3	1698460.5
100R-HC	2602394.9	3442106.3	2666641.8	2647671.5	2813913.1	2724071.6	2613455.8	2710678.9
200R	3027137.3	4525252.0	4387568.8	4328163.6	4417552.8	4381828.3	4342757.7	4362582.2
200R-HC	4231962.1	6238531.3	5473032.0	5244989.6	6087839.1	5673768.6	5276129.5	5343972.9

Instance	SA_G	P100_HCO	P100_VCO	P100_CDCO	P100_VCO_HCO	P100_HCO_CDCO	P100_VCO_CDCO	P100_ALL_CO
7R	109556.6	155931.2	99123.5	103970.6	115968.6	107622.4	96930.9	101828.1
7R-HC	137216.0	188316.6	121068.3	123659.3	148927.0	138601.7	119882.9	130068.7
20R	281695.6	372682.9	286167.8	288563.8	308775.4	288852.4	287578.3	299929.9
20R-HC	328517.6	432128.3	332108.0	318858.7	358282.3	328627.1	314397.9	337041.7
50R	670416.9	963243.0	702439.5	732589.3	785873.0	789500.2	702853.2	734749.8
50R-HC	993294.6	1593627.6	1014651.3	1012751.2	1216248.5	1150727.3	1060115.6	1179139.2
70R	970832.7	1685128.1	1374260.8	1273585.2	1217511.2	1175248.4	1290850.3	1141051.2
70R-HC	1447172.7	1961685.3	1660917.0	1695412.4	1875432.6	1857122.6	1638643.1	1765103.5
100R	1714292.1	2060625.2	1880970.6	1872636.2	2012602.2	2015313.9	1886488.5	1897992.2
100R-HC	2602394.9	3484677.5	3165547.8	3195025.0	3164394.6	3179831.2	3122616.2	3108416.3
200R	3027137.3	4510174.4	4551343.6	4500226.5	4517915.6	4528130.5	4526237.1	4505046.4
200R-HC	4231962.1	5809987.0	5789046.2	6121776.5	5912601.4	5535146.8	5813151.3	5715534.5

Table 5.20: Average solution costs of parameter configurations with different crossover rates for the big instance set.

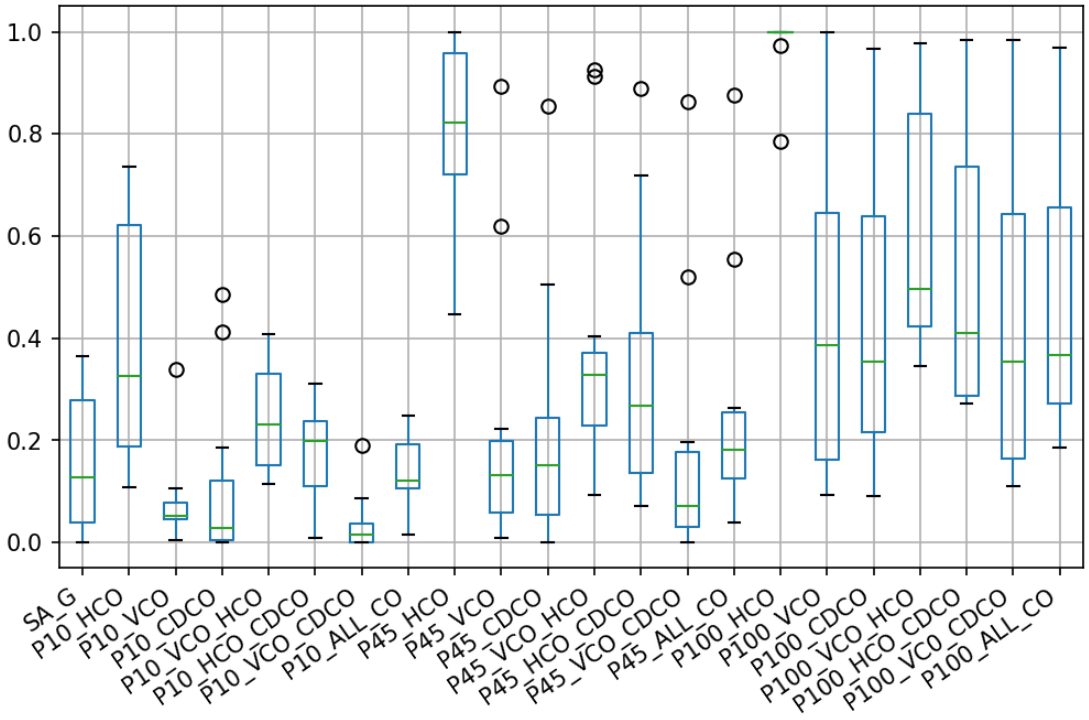

Figure 5.7: RDI values of parameter configurations with various crossover combinations for the big instance set.

5.5 Comparison to Literature Results

Configuration	p	cp	k	m	vcf	hcf	$cdcf$	ilf	ili	c
P100_G	100	1.0	7	0.0	0.5	0	0.5	0.01	3	Greedy
P200_RG	200	1.0	7	0.0	0.5	0	0.5	0.02	3	Random Greedy
P10_WS	10	1.0	5	0.1	0.5	0	0.5	0.15	8	Warm Start
P45_G	45	1.0	7	0.0	0.5	0	0.5	0.05	3	Greedy

Table 5.21: Best parameter configurations found by manual tuning.

This section compares the results obtained by runs of our algorithm with the best parameter configurations to literature results. We take our best result out of 10 runs for each instance for the comparison. Table 5.21 depicts parameter values of our best configurations.

We note that, in this section, we use a slightly modified greedy construction strategy that yields slightly better results than the strategy used in previous sections. There is a small difference in the configuration of the greedy construction strategy controlling color change constraints. Thus, our results for the big instance set may be different from the previous sections' results.

Results for the columns *LS*, *LS/G* and *LS/G/T* are taken from Winter et al. [2019]. Those are the results of different configurations for their simulated annealing local search approach. They use the same runtime limit of 1 hour. The CPU used by the authors for their experiments is an Intel Xeon E5345. Publicly available CPU benchmarks [2] show that our CPU's single-thread performance is about 1.7 times faster. Results from the constraint programming approach from Winter and Musliu [2019a] are shown in column *CP*. With constraint programming, only instances from the small instance set could be solved. For their experiments, they used a time limit of 6 hours.

5.5.1 Small Instance Set

The results from literature approaches and our best configurations for the small instance set are shown in Table 5.23. Our algorithm is able to set new upper bounds for the instances 200R-small and 200R-HC-small. Interestingly, when comparing our results to the simulated annealing approaches, our algorithm achieves similar results for most instances apart from those two. We suspect that the large population sizes are beneficial for the small instance set, especially for those two problem instances.

Figure 5.8 illustrates RDI values for this comparison. We can see that the medians of the configurations *P200_RG* and *P100_G* are smaller than the other approaches' medians.

[2] https://www.cpubenchmark.net/compare/Intel-Xeon-E5-2650-v4-vs-Intel-Xeon-E5345/2797vs1230

As can be seen in Table 5.22, the population size's influence on the number of generations is quite small for this instance set - which means that memetic operators are fast to compute. This comparison is particularly interesting when comparing it to the big instance set in Table 5.24, as for this instance set, the population size has a big influence on the number of generations.

Instance	P100_G	P200_RG	Relative Change
7R-small	499	541	-7.8%
7R-HC-small	502	561	-10.5%
20R-small	497	549	-9.5%
20R-HC-small	463	495	-7.9%
50R-small	465	501	-7.2%
50R-HC-small	478	512	-6.6%
70R-small	438	448	-2.2%
70R-HC-small	456	455	2.2%
100R-small	417	424	-1.7%
100R-HC-small	420	419	-22.0%
200R-small	327	302	-10.6%
200R-HC-small	270	224	46.0%

Table 5.22: Average number of generated generations for our two best configurations for the big instance set. The relative change is the percentage difference between the number of generations from P100_G and P200_WS. The number of generations for P100_G is lower for some instances because of the greedy construction strategy's setup time.

5.5.2 Big Instance Set

For the big instance set, we compare the results in Table 5.25. We are able to set new upper bounds for six of the instances: 7R, 7R-HC, 20R, 20R-HC, 50R-HC, and 70R-HC. Analyzing the cost differences between our two best configurations, P45_G only performs better for the smallest instance. The gap between the two configurations gets bigger for increasing instance sizes. For the largest four instances, our algorithm performed significantly worse than the local search approaches.

RDI values of the various methods for the big instance set are shown in Figure 5.9. The median for our configuration P10_WS is smaller than the literature approaches' median, which indicates very good performance.

To gain insights on the causes of the differences in performance between our two best configurations, we took a look at the number of generations that could be generated within the limited time, as can be seen in Table 5.24. It seems that the computation of memetic operators for large problem instances is too slow. A sufficient number of generations, which seems to be needed for good results, can only be achieved with a smaller population. The relative difference in generations between our two configurations

Instance	LS	LS/G	LS/G/T	CP	P100_G	P200_RG
7R-small	1028	844	882	**775***	781	776
7R-HC-small	868	932	927	**842***	842	844
20R-small	990	992	994	**961***	995	976
20R-HC-small	1016	975	1050	**918***	962	937
50R-small	616	593	599	**530***	655	672
50R-HC-small	887	891	895	**842***	909	906
70R-small	1084	1088	1137	**844***	1272	1353
70R-HC-small	1871	1834	2553	**1237***	1683	1657
100R-small	1767	1735	2421	**975***	1500	2230
100R-HC-small	1262	1243	1269	**964**	1137	1113
200R-small	6298	**5476**	6439	-	2240	**2070**
200R-HC-small	**5723**	7916	8274	-	3172	**2069**

Table 5.23: Comparison of literature results with results from our algorithm for the small instance set. The best result achieved out of 10 runs is shown in columns *P100_G* and *P200_RG*. Columns with a *, denote proven optimal solutions. **Bold** values are upper bounds from literature. **Bold and underlined** values are new upper bounds found by our memetic algorithm.

gets bigger with increasing instance size. For the biggest problem instance, the number of generations is very small. Thus the worse results of our memetic algorithm compared to the local search approaches are not surprising.

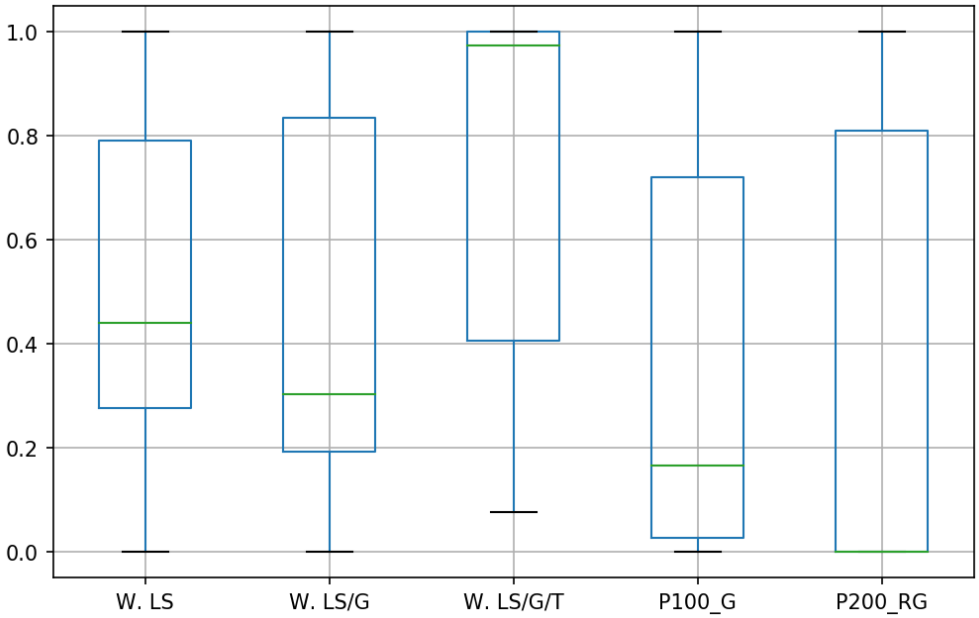

Figure 5.8: RDI values of the best configurations and literature results for the small instance set. Methods are only plotted if they are able to provide feasible solutions for all small instances.

Instance	P10_WS	P45_G	Relative Change
7R	380	317	19.9%
7R-HC	371	310	19.7%
20R	286	161	77.6%
20R-HC	254	154	64.9%
50R	223	81	175.3%
50R-HC	211	69	205.8%
70R	183	63	190.5%
70R-HC	129	41	214.6%
100R	164	40	310.0%
100R-HC	118	29	306.9%
200R	72	16	350.0%
200R-HC	54	13	315.4%

Table 5.24: Average number of generated generations for our two best configurations for the big instance set. The relative change is the percentage difference between the number of generations from P45_G and P10_WS.

Instance	LS	LS/G	LS/G/T	P10_WS	P45_G
7R	2097235	**116235**	123830	80121	**<u>79526</u>**
7R-HC	1985513	**118628**	130552	**<u>116504</u>**	120946
20R	8159361	180863	**172679**	**<u>162012</u>**	178251
20R-HC	8621490	**262252**	262897	**<u>254127</u>**	266444
50R	23320626	**421777**	455321	509188	538880
50R-HC	23947097	**581021**	606917	**<u>535031</u>**	546846
70R	34294393	**555829**	576225	576450	595859
70R-HC	34713814	930564	**927822**	**<u>870889</u>**	916365
100R	-	**917955**	957854	1031698	1087649
100R-HC	-	**1128716**	1142530	1450234	1680627
200R	-	1889804	**1884125**	2209354	2711918
200R-HC	-	**2086450**	-	2312213	2822476

Table 5.25: Comparison of literature results with results from our algorithm for the big instance set. The best result achieved out of 10 runs is shown in columns *P10_WS* and *P45_G*. **Bold** values are upper bounds from literature. **<u>Bold and underlined</u>** values are new upper bounds found by our memetic algorithm.

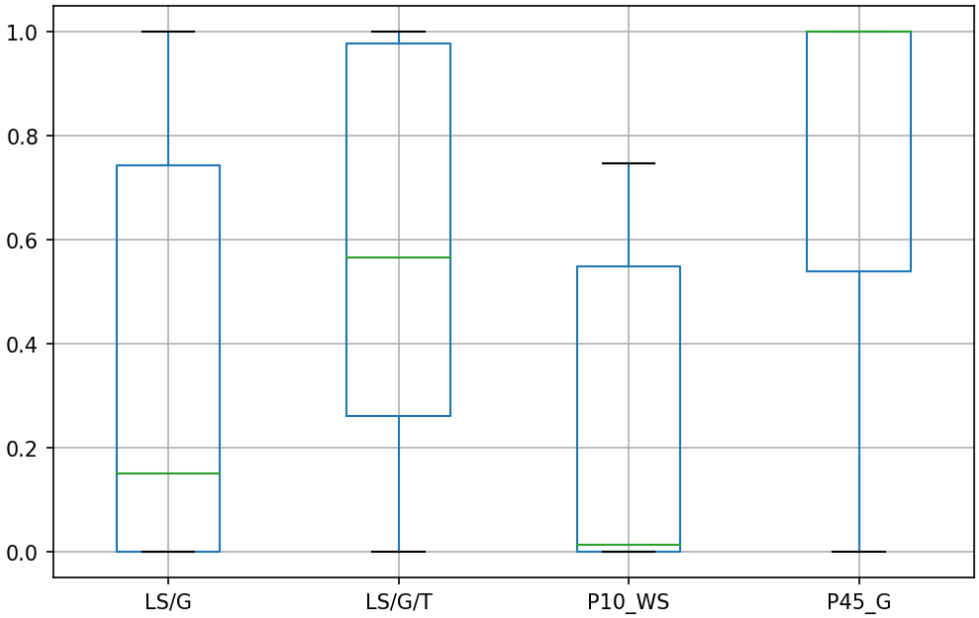

Figure 5.9: RDI values of the best configurations and literature results for the big instance set. Without instance 200R-HC, since for this instance not all configurations yield feasible solutions.

CHAPTER 6

Conclusions and Future Work

In this thesis, we proposed a memetic algorithm to solve the recently introduced Paint Shop Scheduling Problem. We first designed a memetic representation and proposed different population construction strategies. As the algorithm is based on memetic evolution, we designed memetic operators for selection, mutation, and three novel crossover operators, which consider problem-specific knowledge.

The different steps of our memetic algorithm are highly parameterizable. To explore the parameter configuration space effectively, we use SMAC, a state-of-the-art parameter tuning algorithm. Based on SMAC's incumbent parameter configurations, we conducted manual parameter tuning to increase our algorithm's performance further and gain insights on different parameters' influence.

The insights of these experiments were the following:

- A large population yields good results for the small instance set, while smaller population sizes are preferred for the big instance set.
- A combination of the two crossover operators vertical crossover and cost and demand crossover is preferential for most instances.
- Construction strategies for the initial population have a massive influence on the result. For the big instance set, sophisticated construction strategies are needed to generate feasible results within time constraints.
- The genetic algorithm performs worse than the memetic algorithm. Feasible solutions can still be obtained for many instances, but the costs are significantly higher.

We compared the results we obtained by using the best parameter configuration to the best literature results, using a set of publicly available real-life instances. Our memetic

algorithm's results are competitive for the small instance set, as well as for small instances of the big instance set. We managed to obtain new upper bounds for 8 of the 24 instances.

Our experiments highlighted the correlation between the initial population's quality and the result's quality for the set of big instances. Therefore, more sophisticated construction strategies could still improve the results for those problem instances. Another interesting research topic would be the design and evaluation of different memetic representations. For example, there could be two chromosomes - one representing the color and one representing the carrier sequence. Memetic operators could operate on those chromosomes, thus only improving one of these aspects of a solution. Our evaluation demonstrates that our algorithm struggles to generate many generations for bigger population sizes for large problem instances. A variable population size could help to achieve more competitive results by starting with a smaller population and increasing population size once solutions reach a better quality. Since the generation of instances, as well as most of the algorithm's steps done in the generational loop, are independent, parallelization would be an easy way to increase the algorithm's performance.

List of Figures

2.1 A typical layout of an automotive paint shop. 6
2.2 Schemtic representation of three carriers. 6
2.3 A PSSP schedule in tabular form for three rounds. 6
2.4 Three ways to reuse carriers in consecutive rounds. 6

4.1 Solution representation 19
4.2 Vertical crossover 21
4.3 Horizontal crossover 22
4.4 Cost and demand crossover 23

5.1 RDI values of incumbents configurations for the small instance set. . . . 40
5.2 RDI values of incumbents configurations for the big instance set without the instances 200R and 200R-HC. 40
5.3 RDI values of parameter configurations with various construction strategies for the small instance set. 44
5.4 RDI values of parameter configurations with various construction strategies for the big instance set. 46
5.5 RDI values of parameter configurations without local search for the small instance set. 50
5.6 RDI values of parameter configurations with various crossover combinations for the small instance set. 53
5.7 RDI values of parameter configurations with various crossover combinations for the big instance set. 55
5.8 RDI values of the best configurations and literature results for the small instance set. 59
5.9 RDI values of the best configurations and literature results for the big instance set. 61

List of Tables

5.1 Parameter configuration space (PCS) supplied to SMAC. 35
5.2 Best three incumbents found by SMAC for the set of small instances. . . 36
5.3 Best three incumbents found by SMAC for the set of big instances. . . . 36
5.4 Average costs of the three incumbent parameter configurations for small instances. 38
5.5 Comparison of results of the three incumbent parameter configurations for small instances. 38
5.6 Average costs of different parameter configurations for big instances. . . . 39
5.7 Comparison of results of the three incumbent parameter configurations and two configurations with a bigger population size for big instances. 39
5.8 Base parameter configurations with different population sizes used to compare construction strategies. 41
5.9 Average solution costs for the small instance set when using *greedy* construction. 41
5.10 Average solution costs for the small instance set when using *random greedy* construction. 42
5.11 Average solution costs for the small instance set when using *random* construction. 42
5.12 Average solution costs for the big instance set when using *random* and *random greedy* construction. 43
5.13 Average solution costs for the big instance set when using *greedy* construction. 43
5.14 Average solution costs for the big instance set when using *warm start* construction. 45
5.15 Number of generated generations for different configurations. 46
5.16 Parameter configurations without local search, effectively resulting in a genetic algorithm 48
5.17 Average solution costs of different parameter configurations without local search for the small instance set. 49
5.18 Average solution costs of different parameter configurations without local search for the big instance set. 51
5.19 Average solution costs of different parameter configurations with different crossover rates for the small instance set. 52

5.20 Average solution costs of parameter configurations with different crossover rates for the big instance set. 54
5.21 Best parameter configurations found by manual tuning. 56
5.22 Average number of generated generations for our two best configurations for the small instance set. 57
5.23 Comparison of literature results with results from our algorithm for the small instance set. 58
5.24 Average number of generated generations for our two best configurations for the big instance set. 59
5.25 Comparison of literature results with results from our algorithm for the big instance set. 60

List of Algorithms

4.1 Memetic base algorithm 20
4.2 Material scarcity calculation 24
4.3 Cost and demand crossover cost calculation 25
4.4 Random greedy construction 28

Bibliography

Scott Atran. *In gods we trust: The evolutionary landscape of religion*. Oxford University Press, 2002.

Thomas Bartz-Beielstein. Spot: An r package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. *arXiv preprint arXiv:1006.4645*, 2010.

Una Benlic and Jin-Kao Hao. A multilevel memetic approach for improving graph k-partitions. *IEEE Trans. Evol. Comput.*, 15(5):624–642, 2011.

Susan Blackmore and Susan J Blackmore. *The meme machine*, volume 25. Oxford Paperbacks, 2000.

Nils Boysen and Malte Fliedner. Comments on "solving real car sequencing problems with ant colony optimization". *Eur. J. Oper. Res.*, 182(1):466–468, 2007. doi: 10.1016/j.ejor.2006.07.012. URL https://doi.org/10.1016/j.ejor.2006.07.012.

Edmund K. Burke, James P. Newall, and Rupert F. Weare. A memetic algorithm for university exam timetabling. In *PATAT*, volume 1153 of *Lecture Notes in Computer Science*, pages 241–250. Springer, 1995.

Sara Bysko and Jolanta Krystek. Follow-up sequencing algorithm for car sequencing problem 4.0. In Roman Szewczyk, Cezary Zielinski, and Malgorzata Kaliczynska, editors, *Automation 2019 - Progress in Automation, Robotics and Measurement Techniques, outcomes of the international conference AUTOMATION 2019, 27-29 March, 2019, Warsaw, Poland*, volume 920 of *Advances in Intelligent Systems and Computing*, pages 145–154. Springer, 2019. doi: 10.1007/978-3-030-13273-6_15. URL https://doi.org/10.1007/978-3-030-13273-6_15.

Xianshun Chen and Yew-Soon Ong. A conceptual modeling of meme complexes in stochastic search. *IEEE Trans. Syst. Man Cybern. Part C*, 42(5):612–625, 2012.

Carlos Cotta, Luke Mathieson, and Pablo Moscato. Memetic algorithms. In *Handbook of Heuristics*, pages 607–638. Springer, 2018.

Charles Darwin. *On the origin of species, 1859*. Routledge, 1859.

Richard Dawkins. *The selfish gene*. Oxford university press, 1976.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002.

Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *ECAI*, pages 290–295. Pitmann Publishing, London, 1988.

Jan Dörmer, Hans-Otto Günther, and Rico Gujjula. Master production scheduling and sequencing at mixed-model assembly lines in the automotive industry. *Flexible Services and Manufacturing Journal*, 27(1):1–29, 2015.

Guiliang Gong, Qianwang Deng, Raymond Chiong, Xuran Gong, and Hezhiyuan Huang. An effective memetic algorithm for multi-objective job-shop scheduling. *Knowl. Based Syst.*, 182, 2019.

S. M. Kamrul Hasan, Ruhul A. Sarker, Daryl Essam, and David Cornforth. Memetic algorithms for solving job-shop scheduling problems. *Memetic Comput.*, 1(1):69–83, 2009.

Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Intell. Res.*, 36:267–306, 2009.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011.

Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Sci.*, 220(4598):671–680, 1983.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. Smac v3: Algorithm configuration in python. `https://github.com/automl/SMAC3`, 2017.

Bo Liu, Ling Wang, and Yihui Jin. An effective pso-based memetic algorithm for flow shop scheduling. *IEEE Trans. Syst. Man Cybern. Part B*, 37(1):18–27, 2007.

Bo Liu, Juan-Juan Xu, Bin Qian, Jian-Rong Wang, and Yan-Bin Chu. Probabilistic memetic algorithm for flowshop scheduling. In *Memetic Computing*, pages 60–64. IEEE, 2013.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Zhipeng Lü and Jin-Kao Hao. A memetic algorithm for graph coloring. *Eur. J. Oper. Res.*, 203(1):241–250, 2010.

Brad L. Miller and David E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol. Comput.*, 4(2):113–131, 1996.

Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

Pablo Moscato and Carlos Cotta. An accelerated introduction to memetic algorithms. In *Handbook of Metaheuristics*, pages 275–309. Springer, 2019.

Rafael Nogueras and Carlos Cotta. An analysis of migration strategies in island-based multimemetic algorithms. In *PPSN*, volume 8672 of *Lecture Notes in Computer Science*, pages 731–740. Springer, 2014.

Matthias Prandtstetter and Günther R. Raidl. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *Eur. J. Oper. Res.*, 191(3):1004–1022, 2008.

Jakob Puchinger, Günther Raidl, and Martin Gruber. *Cooperating memetic and branch-and-cut algorithms for solving the multidimensional knapsack problem.* na, 2005.

Nikos Angelos Salingaros and Michael W Mehaffy. *A theory of architecture.* UMBAU-VERLAG Harald Püschel, 2006.

Vincent M Sarich and Allan C Wilson. Generation time and genomic evolution in primates. *Science*, 179(4078):1144–1147, 1973. Publisher: JSTOR.

JE Smith. The co-evolution of memetic algorithms for protein structure prediction. In *Recent advances in memetic algorithms*, pages 105–128. Springer, 2005.

Jim E. Smith. Coevolving memetic algorithms: A review and progress report. *IEEE Trans. Syst. Man Cybern. Part B*, 37(1):6–17, 2007.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2960–2968, 2012.

Christine Solnon, Van-Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the roadef'2005 challenge problem. *Eur. J. Oper. Res.*, 191(3):912–927, 2008.

Kristina Yancey Spencer, Pavel V. Tsvetkov, and Joshua J. Jarrell. A greedy memetic algorithm for a multiobjective dynamic bin packing problem for storing cooling objects. *J. Heuristics*, 25(1):1–45, 2019.

Sven Spieckermann, Kai Gutenschwager, and Stefan Voß. A sequential ordering problem in automotive paint shops. *International journal of production research*, 42(9):1865–1878, 2004. Publisher: Taylor & Francis.

Y Sugimori, K Kusunoki, F Cho, and SJTIJOPR UCHIKAWA. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The international journal of production research*, 15(6):553–564, 1977.

Junwen Wang, Jingshan Li, and Ningjian Huang. Optimal vehicle batching and sequencing to reduce energy consumption and atmospheric emissions in automotive paint shops. *International Journal of Sustainable Manufacturing*, 2(2-3):141–160, 2011.

Magdalena Widl and Nysret Musliu. An improved memetic algorithm for break scheduling. In *Hybrid Metaheuristics*, volume 6373 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2010.

Magdalena Widl and Nysret Musliu. The break scheduling problem: complexity results and practical algorithms. *Memetic Comput.*, 6(2):97–112, 2014.

Felix Winter and Nysret Musliu. Constraint Based Modeling for Scheduling Paint Shops in the Automotive Supply Industry (under submission). Technical report, TU Wien, 2019a. URL `https://dbai.tuwien.ac.at/staff/winter/cd-tr-2019-1.pdf`.

Felix Winter and Nysret Musliu. Exact Methods for a Paint Shop Scheduling Problem from the Automotive Supply Industry. 2019b. URL `https://dbai.tuwien.ac.at/staff/winter/cpaior_2019_full.pdf`.

Felix Winter, Nysret Musliu, Emir Demirovic, and Christoph Mrkvicka. Solution approaches for an automotive paint shop scheduling problem. In *ICAPS*, pages 573–581. AAAI Press, 2019.

Yuan Yuan and Hua Xu. Multiobjective flexible job shop scheduling using memetic algorithms. *IEEE Trans Autom. Sci. Eng.*, 12(1):336–353, 2015.