TU WIEN Informatics

Solving the Production Leveling Problem with Memetic Algorithms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Patrick Malik, BSc
Matrikelnummer 11776819

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Professor Dr. techn. Nysret Musliu
Mitwirkung: Projektass.in Dipl.-Ing. Dr.in techn. Marie-Louise Lackner

Wien, 4. September 2023

Patrick Malik	Nysret Musliu

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

TU WIEN Informatics

Solving the Production Leveling Problem with Memetic Algorithms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Patrick Malik, BSc
Registration Number 11776819

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Professor Dr. techn. Nysret Musliu
Assistance: Projektass.in Dipl.-Ing. Dr.in techn. Marie-Louise Lackner

Vienna, 4th September, 2023

Patrick Malik	Nysret Musliu

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

Erklärung zur Verfassung der Arbeit

Patrick Malik, 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, 4. September 2023

Patrick Malik

Danksagung

Zunächst möchte ich meinem Betreuer Associate Professor Dr. techn. Nysret Musliu danken, der mich nicht nur während der kompletten Arbeit, sondern insbesondere beim Abschließen, tatkräftig unterstützt hat. Auch Projektass.in Dipl.-Ing. Dr.in techn. Marie-Louise Lackner die besonders anfangs eine große unterstützende Rolle gespielt hat möchte ich danken. Nicht nur habe ich in ihnen Unterstützung in all meinen Vorhaben gefunden, zudem wurde mir anfangs ein Ruck in die richtige Richtung gegeben und gegen Ende, unter anderem auch aus dem Urlaub, geholfen, auch die letzten Schritte noch rechtzeitig und sinnvoll abzuschließen. Ich wurde unterstützt mich im Zuge der Arbeit mit vielen Dingen auseinandersetzen, die mir schon lange am Herzen lagen und die ein neugefundenes Interesse am Thema in mir wachsen haben lassen. Vielen Dank für all die Unterstützung und Beratung die ich in den letzten Monaten erfahren durfte!

Ich möchte aber auch meinem persönlichen Umfeld danken. Meinen Eltern, die mich seit 26 Jahren bei allem das ich mir vornehme unterstützen und über die Jahre einiges zu hören bekommen haben, wenn es einmal nicht so lief. Meinem besten Freund, ohne den ich im ersten Semester vermutlich nicht einmal in die Vorlesung gefunden hätte, mit dem ich aber auch seit Jahren die Aufs und Abs unserer (oft gar nicht so guten) Entscheidungen teile, aber auch meinem weiteren Freundeskreis, der die letzten Jahre zu einer unvergesslichen Zeit gemacht hat und mich zu dem gemacht hat, der ich heute bin. Abschließend möchte ich auch meiner Freundin danken, die ständig meinen Horizont erweitert, mich immer bei allem unterstützt und immer für mich da ist, auch an Tagen, an denen sie selbst zu kämpfen hat.

Danke, ohne euch wär ich nicht hier!

Kurzfassung

In dieser Diplomarbeit präsentieren wir sowohl einen Memetischen als auch Genetischen Algorithmus, um das kürzlich präsentierte Production Leveling Problem erstmals evolutionär zu lösen. Das Production Leveling Problem ist ein NP-schweres kombinatorisches Optimierungsproblem, das sich mit der Zuordnung von Aufträgen zu Produktionsperioden beschäftigt. Das Ziel ist die Aufträge gleichmäßig über die Perioden und bestimmte Ressourcen zu verteilen, während Perioden- und Ressourcen-Limits und Prioritäten eingehalten werden. Das Problem liegt im Bereich der mittelfristigen Planung, ein Gebiet, in dem die Bestellungen erst gruppiert werden, um dann während der kurzfristigen Planung konkret für die Periode geplant zu werden.

Wir präsentieren eine Lösungsdarstellung basierend auf der bisherigen Definition und verwenden die entsprechende Fitness-Funktion mit problemspezifischen Optimierungen. Sieben Konstruktionsheuristiken werden implementiert, diskutiert und bezüglich Qualität und Diversität der erzeugten Lösungen analysiert. Während der Diversitätsanalyse wurde neben den bereits etablierten Metriken der ”allele coverage” und ”solution equality”, der Begriff des ”Extended Jaccard Index” eingeführt, um die Diversität der Mengen beschreiben zu können, die durch die Heuristiken erzeugt werden. Weiters werden drei ”Selection” Operatoren, fünf ”Crossover” Operatoren und vier ”Mutation” Operatoren implementiert und besprochen. Vier ”Local Search” Operatoren werden für die Verwendung im Memetischen Algorithmus vorgestellt. Schließlich werden noch drei ”Replacement” Ansätze diskutiert.

Nachdem die Operatoren und deren Potential besprochen wurden, werden die vielversprechendsten für das Parameter-Tuning übernommen. Um während des Tunings gute Lösungen zu finden, ohne die Algorithmen auf die Testdaten anwenden zu müssen, werden 2000 Trainingsdaten unter Verwendung einer Heuristik aus vorherigen Arbeiten erzeugt. Für einen faireren Vergleich, der aufgrund von Systemunterschieden erschwert ist, wird ein ”Simulated Annealing” Ansatz implementiert und getuned.

Der Hauptfokus der Arbeit liegt auf dem Vergleich und der Bewertung des Memetischen und des Genetischen Algorithmus. Dafür vergleichen wir zuerst die Algorithmen, die im Zuge dieser Arbeit entstanden sind und dann die Ergebnisse mit denen aus der Literatur.

Die Ergebnisse zeigen, dass der Genetische Algorithmus am besten abschneidet, dicht gefolgt vom Memetischen Algorithmus. Die beiden evolutionären Algorithmen liefern

für das Set der realistischen Instanzen Ergebnisse innerhalb einer 5% "optimality gap". Zusätzlich lösen sie alle der optimal lösbaren Instanzen ohne "hard constraint"-Verstoß und sogar 4 der 50 optimal, beides Leistungen die, soweit uns bewusst ist, bisher nicht mit Metaheuristiken erreicht wurden.

Abstract

In this thesis we develop and present a memetic and genetic algorithm for solving the Production Leveling Problem introduced recently, providing the first evolutionary approach to this problem. The Production Leveling Problem is an NP-hard combinatorial optimisation problem about assigning orders to production periods in such a way that their demand across those periods and certain resources is balanced while also taking into account demand limits per period and per resource. It is a problem situated in the realm of medium-term planning, an area where orders are first grouped to then be scheduled more concretely during short-term planning.

We propose a solution representation based on the previous problem definition and implement the fitness function accordingly, introducing problem specific optimisations. Seven construction heuristics are implemented and discussed and further analysed in terms of quality and diversity of the solutions they create. When discussing diversity, in addition to the already established measures of allele coverage and solution equality, the notion of the Extended Jaccard Index is introduced to gain insight into how diverse the sets are that the heuristics create. Furthermore, three selection operators, five crossover operators and four mutation operators are implemented, reviewed and discussed. Four local search strategies are presented for the use in the memetic algorithm. Lastly, we look at three different approaches for replacement.

After reviewing the operators and their potential they are reduced to a more compact set from which the best configurations are then chosen during hyperparameter tuning. In order to find good solutions without applying the algorithm to the test data provided in existing literature, a set of 2000 training instances is generated based on a random instance generator discussed in earlier work. For the sake of comparison and due to system differences, we also implement and tune a simulated annealing approach.

The main focus of the thesis is the comparison and review of the memetic and the genetic algorithm developed. For that we first compare the algorithms introduced and tuned during this thesis amongst themselves and then compare their results with the findings presented in the literature.

We find that the genetic algorithm performs best amongst the algorithms developed during this thesis, with the memetic algorithm trailing only slightly behind. On the set of realistic instances both evolutionary approaches manage to mostly find solutions within a

5% optimality gap. Additionally, they manage to solve the entire set of perfectly solvable test instances without hard constraint violations, and even solve four of the 50 instances to optimality, both feats that, to the best of our knowledge, have not been achieved thus far using metaheuristic approaches.

Contents

CHAPTER 1

Introduction

In a production setting optimisation grows in importance day by day. Costs need to be as low as can be to stay afloat in a world that grows more and more competitive. Supply chains are designed to be as tight as possible since storing materials is expensive, making planning evermore crucial. A lot of planning has changed from being done manually to automatically over the last decades. Planning is done on various levels. So-called medium-term-planning, lodged right between long- and short-term planning, is in our context about creating sets of orders for production periods that are then scheduled more concretely during short-term planning[VLM20].
In this thesis we work on the Production Leveling Problem (PLP), a rather novel combinatorial optimisation problem that has been introduced and defined by Vass et al. in [VLM20] in collaboration with an industrial partner. Such a problem consists of a set of orders that are of a product type and have a production demand and priority. A problem instance also specifies an amount of periods with maximum production capacities in which all these orders must be planned. In addition every product type has a separate maximum capacity. The scheduling algorithm tries to assign orders to each period while not breaking the maximum capacity limit of each period and of each product type. The goal is to eventually have a solution that not only adheres to all the capacity restrictions and keeps orders in order of their priority but also levels the cumulative production demand of each period.
This problem has been shown to be NP-hard by an NP-Completeness proof via reduction from Bin Packing in [VLM$^+$22]. Vass et al. presented exact and metaheuristic approaches solving the Production Leveling Problem that yield good results not only on the instances provided by the industrial partner but also on randomly created much harder instances. In this thesis we want to extend the literature on this problem by introducing a memetic as well as a genetic algorithm. We discuss various operators and analyse their applicability, find good configurations using hyperparameter tuning and compare the results amongst the algorithms developed during this thesis as well as with the solutions found by Vass

et al. in [VLM20].

Memetic and genetic algorithms are considered population based evolutionary algorithms and follow principles found in nature making them easy to understand and digest, while they are complex enough to warrant exhaustive studying if desired. memetic algorithms can be considered an extension or hybridisation of genetic algorithms, incorporating local search into the process of genetic algorithms. This thesis proposes a solution representation based on the problem definition by Vass et al. in [VLM20] and implements their fitness function while introducing problem specific optimisations. It implements and discusses seven construction heuristics that are analysed in depth, gathering information on the quality and diversity of the solutions they create. In order to better evaluate their diversity, the notion of the Extended Jaccard Index is introduced, making a comparison between sets of sets possible. Additionally, three selection operators, five crossover operators, four mutation operators and three local search strategies are implemented that are then compared and reduced to a smaller set of viable operators for parameter tuning. Lastly, we consider three different approaches to replacement.
After finding the best configuration of a memetic and genetic algorithm they are applied to the test instances. Since there is a considerable difference in hardware, language and setup between this work and that of Vass et al. [VLM+22] that makes a direct comparison difficult, we also implement and tune a simulated annealing approach to have a fairer comparison with.

1.1 Aims of the thesis

This thesis aims to develop a memetic and genetic algorithm for solving the Production Leveling Problem metaheuristically. We want to gather and provide valuable information on the roles of each operator, as well as the various implementations on a more theoretical basis with a particular focus on construction heuristics. In addition a fair comparison between a simulated annealing approach and the evolutionary algorithms is necessary. This work further aims to find the best configuration of each algorithm using the parameter tuning framework SMAC3 [LEF+22]. To validate the findings of Vass et al. and assist the tuning process, the implementation of their random instance generator is necessary and important. Last but not least this work seeks to find which algorithm developed during this thesis performs best on the testing data and how they compare to the results presented in [VLM20].

1.2 Contributions

This thesis not only provides two evolutionary approaches to solving the Production Leveling Problem, it investigates and compares the effects different types of operators have and discusses why some implementations are better suited for this type of problem than others with a particularly in depth analysis of construction heuristics. This thesis further introduces the notion of the Extended Jaccard Index, a way of measuring equality

of sets of sets used for comparing our solution representations. During this thesis we implemented the instance generation procedure initially introduced in [VLM20] to provide us with valuable training data that is employed to find the best configurations of the algorithms we developed. Given the change in hardware, programming language and framework, implementing the simulated annealing approach and tuning its parameters was deemed necessary. This thesis compares the performance of the genetic and memetic algorithm as well as the simulated annealing approach we developed and finds that the genetic algorithm performs best overall, slightly outperforming the memetic algorithm. We also find that the system change lead to a strong decrease in performance when it comes to the simulated annealing approach. While the direct comparison of the best results found in this thesis and the solutions presented in [VLM20] is difficult given the differences in setup, we discuss the difference in results and find that the evolutionary algorithms both achieve results mostly within 5% optimality gap when looking at the set of realistic instances. In addition, however, they perform very well when looking at the set of randomly generated instances that are perfectly solvable, not only solving the entire set of 50 instances without a hard constraint violation but also solving four instances to optimality, both feats that, to the best of our knowledge, have not been achieved thus far using metaheuristics.

1.3 Structure of the thesis

The thesis is structured as follows. Chapter 2 introduces the problem and presents related work. Chapter 3 introduces the algorithm and different operators of the algorithm. It presents and compares the various implementations of those operators and goes into detail on why an approach would be chosen over another. Section 3.3 discusses extensively the various implemented construction heuristics and their diverse set of merits and evaluates those heuristics based on both quality and diversity. Section 4 presents the problem instances used and how they were generated, discusses the hyperparameter tuning setup and findings of each algorithm and examines and compares the results achieved. The results are first compared amongst each other and later compared with the solutions found by Vass et al. in [VLM20]. In addition some claims from 3.3 are investigated and confirmed. Last but not least the optimality gap on a subset of the set of realistic instances is calculated and juxtaposed.

CHAPTER 2

Problem Statement and Related Work

The Production Leveling Problem has been introduced and defined in [VLM20]. In the following we present those definitions and discuss related work as has been done in [VLM20]. Additionally, we briefly introduce memetic algorithms and related problems, where they have been employed.

2.1 Problem Statement

The recently introduced "Production Leveling Problem"[VLM20][VLM+22] (PLP) deals with assigning orders to production periods. An order has a demand, priority value and product type. Each period has a maximum demand capacity for each product type as well as the sum of all product types. The goal is to assign orders to those periods balancing the production volume between them, i.e. minimise the sum of deviations between the periods, while keeping the orders in the order their priorities imply.
Problem-instances are provided both by an industrial partner as well as a random instance generator created in [VLM20] which is also re-implemented during development of this thesis to create training data.
In [VLM20] a mixed integer programming (MIP) formulation, variable neighbourhood descent and simulated annealing algorithm were developed and compared. While the MIP approach was able to solve most of the small instances, orders with more than 300 orders are considered to be too hard by [VLM20]. The simulated annealing algorithm, while having no guarantee for optimality, reliably finds very good solutions in under 5 five minutes also for very large instances. In comparison to simulated annealing, the variable neighbourhood descent approach performs worse. Based on those results this thesis will consider the results of the simulated annealing approach during comparison. We will also implement this version of simulated annealing to show reproducibility and

have a fair comparison, given that the environment changed considerably not only in hardware but also in framework and programming language.
A Problem related to the Production Leveling Problem, the so-called "Production Leveling Problem with Order-Splitting and Resource Constraints" has been introduced in [VMW20][VLM+22] and also been solved by using Mixed-Integer-Programming and simulated annealing. There is currently no approach using evolutionary algorithms to solve the PLP.

2.1.1 Mathematical Formulation

The problem has been first defined in [VLM20][VLM+22], hence we will use the mathematical definition introduced there and cite it directly. We will start with the input parameters, go over the variables and hard constraints and eventually define the fitness function.

Input parameters (from section 2.1.2 in [VLM20])

$K \subseteq \mathbb{Z}^+$	Set of orders $\{i \in \mathbb{Z}^+ \mid 1 \leq i \leq k\}$, where k is the number of orders
$M \subseteq \mathbb{Z}^+$	Set of product types $\{i \in \mathbb{Z}^+ \mid 1 \leq i \leq m\}$, where m is the number of product types
$N \subseteq \mathbb{Z}^+$	Set of periods $\{i \in \mathbb{Z}^+ \mid 1 \leq i \leq n\}$, where n is the number of periods
$a_i \in \mathbb{R}^+$	for each objective function component $i \in \{1, 2, 3\}$ the associated weight
$c \in \mathbb{R}^+$	the maximum overall production volume per period
$c_t \in \mathbb{R}^+$	for each product type $t \in M$ the maximum production volume per period
$d_j \in \mathbb{Z}^+$	for each order $j \in K$ its associated demand
$p_j \in \mathbb{Z}^+$	for each order $j \in K$ its associated priority
$t_j \in \mathbb{Z}^+$	for each order $j \in K$ the product type
$d^* \in \mathbb{Z}^+$	the target production volume per period, i.e. $\frac{1}{n}\sum_{j \in K} d_j$
$d_t^* \in \mathbb{Z}^+$	the target production volume per period for each product type $t \in M$, i.e. $\frac{1}{n}\sum_{j \in K \mid t_j = t} d_j$

Variables (from section 2.1.2 in [VLM20])

- For each order the production period for which it is planned:

$$y_j \in N \quad \forall j \in K$$

- The production volume for each period (helper variable):

$$w_i = \sum_{j \in K : y_j = i} d_j \quad \forall i \in N$$

- The production volume for each product type and period (helper variable):

$$w_{i,t} = \sum_{j \in K : y_j = i \wedge t_j = t} d_j \quad \forall i \in N, \forall t \in M$$

Hard constraints (from section 2.1.2 in [VLM20])

- The limit for the overall production volume is satisfied for each period:

$$\forall i \in N \quad w_i \leq c$$

- The limit for the production volume of each product type is satisfied for each period:

$$\forall i \in N, t \in M \quad w_i, t \leq c_p$$

Objective function

The objective (or fitness) function comprises three objectives. These three objectives are each represented by a corresponding function and are taken as is from [VLM20]:

$$f_1 = \sum_{i \in N} |d^* - w_i| \tag{2.1}$$

$$f_2 = \sum_{t \in M} \left(\frac{1}{d_t^*} \cdot \sum_{i \in N} (|d_t^* - w_{i,t}|) \right) \tag{2.2}$$

$$f_3 = |\left\{(i,j) \in K^2 : y_i > y_j \text{ and } p_i > p_j\right\}| \tag{2.3}$$

Equation 2.1 calculates the sum of deviations across all periods disregarding the product types. The demands of each order planned for a period are summed up and compared to the target value w_i. This deviation is calculated for every period and added up. Similarly equation 2.2 calculates the sum of deviations across all periods while considering product types. Each product type has a per period target value. All orders of the same product type, planned for the same period are added up and compared with their respective target value. The deviations across the periods are summed and normalised for each product type by dividing by the target value of the relevant product type. Those normalised deviations of each product type are then added up for the f_2 value. Lastly, equation 2.3 counts the amount of priority inversions in the solution. A priority inversion is present if an order with a higher priority is planned after an order with a lower priority.

Those three functions will be combined into a single fitness function after a normalisation step as presented in [VLM20]:

$$g_1 = \frac{1}{n \cdot d^*} \cdot f_1 \tag{2.4}$$

$$g_2 = \frac{1}{n \cdot m} \cdot f_2 \tag{2.5}$$

$$g_3 = \frac{2}{k \cdot (k-1)} \cdot f_3 \tag{2.6}$$

In [VLM20] they argue that: *"The normalisation ensures that g_1 and g_2 stay between 0 and 1 with a high probability. Only for degenerated instances, where even in good solutions the target is exceeded by factors ≥ 2 higher values are possible for g_1 and g_2. The value of g_3 is guaranteed to be ≤ 1 because the maximum number of inversions in a permutation of length k is $k \cdot (k-1)/2$."*

Those normalised values are then weighted to represent their severance according to the industrial partner and added up to make up the final objective function 2.7. In [VLM20] they further investigated what impact a quadratic objective function would have but came to the conclusion that there are advantages and disadvantages to both and none of them is as such better. They further used the non-quadratic objective function throughout their work which is why we will use it as well for the sake of comparison. For the weights they decided to use $a_1 = 1, a_2 = 1, a_3 = \frac{1}{3}$ in cooperation with the industrial partner [VLM20].

$$\textbf{minimize } g = a_1 \cdot g_1 + a_2 \cdot g_2 + a_3 \cdot g_3 \tag{2.7}$$

In section 3.2 we will further investigate how the calculation of the fitness function can be sped up both by introducing delta evaluation and the concrete way of implementing those calculations.

2.1.2 Hard Constraint Violations & Penalties

As stated above this problem comprises two types of hard constraints and therefore two hard constraints that can be violated. During development it was necessary to decide how to deal with hard constraint violations. Whether we were to entirely disallow them in our algorithm by designing our operators around this restriction or to penalise their presence. In [VLM20] they opted to penalise hard constraint violations. Given the difficulty of the problem and the complexity of creating solutions that have no hard constraint violations, forbidding illegal solution candidates is a difficult endeavour. There also exist many instances in our training and test sets for which we don't know whether legal solutions exist. Both this and our goal of comparing our solutions with the results of [VLM20] lead to the implementation of a penalty system.

Every violation of a hard constraint is counted and, if required, added to the fitness value. For each violation 1 is added. A violation exists not for every order that violates a hard constraint, since this would be ambiguous, but for every period that violates the total capacity and for every period that violates a product type capacity. If a period violates multiple product type capacities it of course introduces multiple hard constraint violations.

Since it is difficult for a legal solution to have a fitness value of > 1 a candidate with hard constraint violations is most likely to have a higher fitness value so a lower overall fitness.

2.2 Related Work

The Production Leveling Problem [VLM20][VLM+22] and related problems like the Production Leveling Problem with Order-Splitting and Resource Constraints [VMW20][VLM+22] and the Fixed Order Production Leveling Problem [LV19] have been introduced fairly recently and are therefore not exhaustively researched. The papers currently available focus on simulated annealing [VLM20][VMW20][VLM+22], variable neighbourhood descent [VLM20][VLM+22] and mixed integer programming [VLM20][VMW20][VLM+22]. In [LV19] Lackner et al. also proposed a dynamic programming algorithm for a derivative of the problem.
A better studied set of problems closely related to the Production Leveling Problem can be found under the term *Balancing Problems* [VLM20]:

- *Balanced Academic Curriculum Problem (BACP)*
  The Balanced Academic Curriculum Problem is very similar in idea to the Production Leveling Problem. It is about balancing courses that have a workload across periods and minimising large deviations from the mean workload per period. This very much aligns with the demand a PLP order has and the leveling of the cumulative demand per period. In addition a course in the BACP has prerequisite courses that enforce an order in a way that is not present in the PLP where priority inversion are similar, but not hard constraints. The BACP also has no property that is similar to the product type mechanic of the PLP. In [LPR+13] the problem was approached using Ant Colony Optimisation, [CM01] presented an Integer Linear Programming algorithm as well as a constraint programming solution. In [LCM+05] they combine genetic algorithms with Constraint Programming for this problem showing that a combination outperforms both approaches on their own in results and speed. In a later paper by the same team [LCMS06] they went into detail about the hybridisation approach. It is made clear that this is not a memetic algorithm where propagation is used as a local search algorithm but rather a hybrid approach where domain reduction functions, split and a short genetic algorithm run are employed side by side. It is also documented how the genetic algorithm has a minor role at the beginning of the run, where constraint propagation reduces the search space but grows in relevance later in the run when it is about finding an optimal solution.

- *Nurse Scheduling Problem (NSP)*
  The Nurse Scheduling Problem is a problem about assigning nurses workloads. In [ML02] this problem is defined for Nurses in neonatal care. Where each nurse is assigned a set of infants. Each infant, based on the severity of their condition, might need more or less care, resulting in more or less workload. The goal is to balance

the workload across the nurses. Additionally, one nurse exists that receives less workload because they are considered an 'admit' nurse, that has to deal with infants that are admitted during the shift. Another aspect is that nurseries are often split into multiple rooms with different room sizes that can affect the workload leading to nurses only being assigned infants in one room (or zone in [ML02]). There are also maximum and minimum constraints. The problem really comes down to finding groups of infants with similar workloads rather than actually assigning nurses. This is in essence similar to the Production Leveling Problem while being a considerably different set of constraints.
In [ML02] they developed both an Integer Programming approach and heuristic approaches The Integer Programming approach, while having issues finding optimal solutions in the required time frame, generally found good solutions. The heuristic approaches are a problem specific heuristic that assigns zone by zone, potentially losing some optimality if the nurses are not assigned correctly to the zones, that achieved promising results in the required amount of time, and a COMSOAL approach.
A different team [SHR09] was able to develop a Constraint Programming model based on the findings of [ML02] that was capable of solving even very complex instances of the problem presented in a short amount of time.

- *Simple Assembly Line Balancing Problem (SALBP)*
  The Simple Assembly Line Balancing Problem is a very well studied problem with an extensive pool of used approaches and derivations. It is about the assigning of tasks to workstations that are generally arranged in sequence. Some tasks might require the work of other tasks beforehand introducing precedence relations. Each task has some processing time. The so called cycle time specifies how much time is spent at each workstation. If the goal is to find an assignment of tasks to a set of workstations while reducing the idle time (and hence workstations) as much as possible, the problem is called SALBP-1[BFS07][Bay86]. Some problems might specify the amount of workstations making the objective the reduction of the cycle time. Those would be called SALBP-2 [BFS07][Bay86]. Making both cycle time and number of stations variable makes a problem SALB-E [BFS07] and lastly making both fixed makes a problem SALB-F [BFS07].
  For the various SALB problems a plethora of approaches is documented in [BFS07] ranging from mathematical models, over dynamic programming and branch and bound to metaheuristic approaches like genetic algorithms, ant colony optimisation, simulated annealing and lots more.

Memetic algorithms [NC12][NOK07] are a form of evolutionary algorithms, more specifically a genetic algorithm [KCK21][GH88] that makes use of local search to improve a solution. genetic algorithms are a population based approach where first an initial population, a set of possible solutions, is generated. Those individuals are then evaluated and assigned a fitness value that represents how well they perform. During *selection* a subset of individuals is chosen for *recombination*, based on the previously assigned

fitness-value. During *recombination* new individuals are created by combining those previously selected, creating new individuals that are then changed slightly based on chance. This process is called *mutation*. From the old population and the set of new individuals a new population is created with which the whole process starts anew until a termination condition is met.
When using memetic algorithms, local search can then be used to either obtain better performing individuals or to repair an invalid solution that conflicts with hard constraints [Ber19]. How often and long this local search takes place and how many individuals are improved is subject to fine-tuning.
Evolutionary algorithms have been used extensively over the last decades and have also been shown to work well with scheduling and balancing problems. In [WMW21] a memetic algorithm has been used for the Paintshop Scheduling Problem, [WM10] used memetics for a Break Scheduling Problem and [BFS07] shows a number of papers making use of genetic algorithms for solving Simple Assembly Line Balancing Problems.

The current state of the art only provides a small set of heuristic techniques for solving the Production Leveling Problem. Memetic algorithms have only been used for related problems. This work will fill this gap and provide a first memetic approach for the PLP.

CHAPTER 3

Genetic and Memetic Algorithms for the Production Leveling Problem

A memetic algorithm is a type of evolutionary algorithm introduced by Moscato in [Mos89]. More specifically a genetic algorithm with additional local search capabilities or as Moscato puts it "They combine a very fast heuristic to improve a solution with a recombination mechanism that creates new individuals."[Mos89]. Genetic algorithms were first introduced by John Holland and properly presented in an early book in 1975 [Hol75]. They are based on the idea of Darwinian evolution and abstractly model the processes found in nature. Every genetic algorithm features a set of operators that each can be implemented in various ways based on the problem at hand. Every algorithm needs a way of representing a solution, often called solution representation or in the case of population based algorithms individual. Each individual has a concrete encoding of the solution they are representing called chromosome in the language of genetic and memetic algorithms. A set of such individuals is called a population. To evaluate and compare a solution, the algorithm needs a fitness function. This fitness function differs for each problem and is what is eventually going to be optimised. After a population of various different individuals is created, a subset of the population is chosen for reproduction. They then reproduce via the crossover operator, where a set of (mostly two) individuals creates another set of individuals based on the properties of its parents. They are then mutated by slightly changing bits of the solution. A memetic algorithm additionally employs a local search. Where this local search is placed is not specified. Moscato in [Mos89] introduced local search right at the beginning of the loop arguing that the solutions are to be improved before they are allowed to interact. In the same paper he presents different implementations that introduce local search at the end of the loop, before starting a new generation. This generally leads to the same sequence of operators

except that having it at the beginning also local searches the initially created population. Those steps are then repeated until some stopping criterion is met.
In this chapter we will first discuss the choice of solution representation, then consider and compare various ways of constructing such a solution for our starting population. We will then talk about the fitness function and delta evaluation. Next the various operators will be discussed starting with selection and crossover followed by the mutation operator. Afterwards we will examine different ways of Local Search and last but not least will talk about differing approaches when it comes to replacement strategies.

3.1 Solution Representation

The Solution Representation is arguably the most important part of a genetic and memetic algorithm [Koz93][Dav90]. It maps the search space to the chromosome space [Rad91]. Choosing the right solution representation can affect the algorithm in many ways. It can steer the search and help make meaningful neighbourhood moves, it can make it easier to find and keep building blocks and formae and it might enable and affect different types of operators. Another important part when designing the chromosome can be how understandable it is for human readers. A good representation can help understand and interpret the problem [Gol91].
Solution Representations can come in any form. Initially, in early genetic algorithms, they were developed to be fixed length binary strings with each binary character representing some state or parameter of the problem [Hol75]. Parameters of the problem that are closely related were also expected to be spatially close. This was necessary for the genetic algorithm to converge based on the schema theorem. The schema theorem is the formal groundwork on which genetic and memetic algorithms are built. It is a kind of template similar to a regular expression. A schema specifies some fixed gene values of a chromosome and some wildcards and can be arbitrarily long. Depending on the schema there typically are multiple solution representations that match it. By calculating the fitnesses of all the solutions matching a schema one can calculate the average fitness of a schema. If a schema has a higher than average fitness, it is generally expected to survive longer within the population. Given the simple operators that were used for early genetic algorithms, schemata that have a long defining length, so schemata that span over the whole chromosome, specifying a fixed gene assignment at the beginning and around the end, tend to be broken up more easily through crossover. Initially the simple crossover, or one-point crossover, was being used [Rad92][Hol75]. This obviously tends to tear apart long schemata leading to good schemata needing to be short so they were not to be broken up and would indeed survive multiple generations of the genetic algorithm, prompting the definition of the building block hypothesis, arguing that genetic algorithms perform well when working with schemata with short defining length and low order, i.e. few fixed positions that are close to each other, with a higher than average fitness. It was further argued that this theory is what makes genetic algorithms perform as well as they do.
When evaluating a solution candidate one evaluates a plethora of schemata. Solutions

with high performing schemata are more likely to survive meaning in turn that high performing schemata are more likely to survive. The fact that a solution representation can match a large number of schemata and evaluating one solution evaluates all those schemata was called "intrinsic parallelism" [Hol75] (often called implicit parallelism in the literature). The crossover operator is an important operator when it comes to making use of this implicit parallelism since it recombines large amounts of schemata and helps sampling solutions for schemata further identifying strong building blocks. Since the emergence of the schema theorem and the development of simple genetic algorithm a lot of work has been done to further expand the realm of genetic algorithms.
The solution representation used in this thesis is a non-binary encoding. [HLV98] talks about why non-binary representations were argued against in the early stages. On the one hand because of the argued reduced implicit parallelism as compared to binary alphabets. Goldberg states that: "The binary alphabet offers the maximum number of schemata per bit of information." [Gol91]. A chromosome can contain a higher number of schemata if the alphabet used is binary. Therefore more schemata are being processed when processing a chromosome. [HLV98] goes on to argue that there are however other issues that are more prominent with smaller alphabets, like deception and that the reasoning for lower implicit parallelism is only true for the classical schema theorem and is shown not to be a problem for different approaches to the schema theorem [Ant89]. They even go on to argue that the implicit parallelism is higher than for binary alphabets based on how the wildcard was interpreted [Ant89][Rad92]. On the other hand the size of the alphabet directly affects the size of the search space that is covered by one chromosome and hence the population needed to cover a large percentage of the alleles, or gene values. Meaning a larger alphabet needs a larger population independent of chromosome length, as is shown in [TS93].
[Rad91] moves away from the original definition of schemata and defines equivalence relations or formae. They are a more generalised version of the schema theory, supporting the definition of non-binary solution representations. In [JD96] formae are described as follows with C being the Chromosome space: "It is clear that if $C = \{0, 1\}^n$, then formae reduce to schemata, so a forma is a generalisation of the concept of a schema.". They also talk about the role of the different operators and their interaction with formae, also allowing more problem specific but generally new crossover-types, as long as they "respect" and "properly assort" formae [Rad91]. They also go on to show that such a non-binary representation can be advantageous in certain search spaces and that those representations will be implicitly (or intrinsically) parallel and not suffer from the initially assumed slow down as long as the operators "respect" and "properly assort" the formae and are not too disruptive.

The solution representation used in this work is a non-binary string, whose alphabet consists of period numbers. The chromosome contains as many genes as the problem instance contains orders. More formally, given the set of periods N and the set of orders K, the alphabet, or possible gene values are equivalent to N, and the chromosome is of

length $|K|$, or as defined in [VLM20] "For each order the production period for which it is planned" or:

$$y_j \in N \quad \forall j \in K$$

This solution representation comes with both advantages and disadvantages, but is certainly a very intuitive and interpretable encoding. It is one of two very natural such representations, with the other being the opposite, a list of sets that contain the orders that are in each period. In comparison, the chosen solution representation is a lot easier to work with, since no nested data structures are needed. Simple operators like a simple mutation and simple crossover are also trivial, particularly compared to the second option, where a simple crossover would not have been possible without having to repair the solutions afterwards in most cases. There are some obvious, more problem specific crossovers, that could have been applied more easily with the second encoding, this, however, can be rectified easily and will be addressed in the section about our crossover operators. Of course there are also options to encode the problem in a binary alphabet, in much the same way the adjacency matrix does for the Travelling Sales-rep Problem. Given the sheer size of a lot of our problem instances however, this would not only have been problematic in terms of space complexity, it would have also introduced issues in terms of crossover, similar to the earlier alternative representation. Enumerating the combinations is of course also out of the question for similar reasons.
When looking at the guidelines the original schema theorem presented, our encoding is a bad choice, since, given the lack of relatedness of neighbouring genes, short building blocks are not encouraged. When looking at formae and our choice of crossover operators, however, the encoding appears to be a valid choice.
A problem our solution representation has is the large amount of epistasis present. High epistasis means, that looking at a gene value in isolation says very little about the fitness contribution of this allele, i.e. there is strong interdependence among the genes [Dav90]. This is of course, to a degree, inherent to the problem definition. With the representation we chose to use, one cannot gauge the impact a gene has, since this impact of an order being in a certain period, very much depends on every other gene that has the same value. If the target limit is already reached, adding an order to a period has a very different effect then when the period is empty.
In [BBM93a] and [Dav90] it is argued that genetic algorithms, and in turn also memetic algorithms, are a particularly good choice when it comes to epistatic problems, as long as the epistasis is not too high. They argue that low-epistasis problems are easily solvable by hill-climb algorithms and don't need the tool set provided by genetic algorithms. Yet epistasis leads to larger building blocks since more genes need to be considered, which in turn is detrimental to a genetic or memetic algorithms performance. [Dav90] discusses the role of solution representations and their strong effect on epistasis. It appears that epistasis is mostly an issue based on the assumptions made for the classical schema theorem. Large building blocks, which are a consequence of epistatic problems, have a negative effect on a genetic or memetic algorithms performance, since their survivability

is low when looking at the original operators. This can be mitigated, however, if operators are chosen in a way that preserves those building blocks. This appears to be in line with the reason why the choice of solution representation has a large effect on the epistasis of a problem and should reduce the negative effects of highly epistatic problems. The crossover operators chosen in this thesis appear to address that issue to a certain degree as will be mentioned at the relevant section.

3.2 Fitness Function

The fitness function represents the viability of a solution. It is this function that is optimised and hence capturing each change in value is necessary for the algorithm to move in the right direction. This necessitates regular and highly frequent recalculations. Each change in the solution has to be measured and evaluated requiring constant updates of the fitness function. The speed at which this value is calculated is therefore crucial for the performance of the algorithm.
This evaluation can be sped up in two important ways: firstly by making the calculation of the entire function as quick as possible and secondly by introducing delta evaluation. Delta evaluation is a technique where the fitness delta introduced by a change is calculated and then added (or subtracted) to the previous fitness value. This is particularly useful for small and regular changes where the calculation of the delta can vastly outspeed the recalculation of the entire function.

Given the nature of a memetic algorithm both types of speedup are vital. Mutations and the local search introduce many small iterative changes that can capitalise on the performance improvements introduced by delta evaluations. The crossover operator however regularly creates large changes that facilitate entire recalculations of the fitness function. To support both approaches every solution keeps a state for each part of the fitness function. This state is made to be easily modifiable such that small changes can be made that are introduced by the delta evaluation. Based on this state the entire fitness value can be then calculated quickly while parts of it can be used to calculate the fitness delta for the delta evaluations.

3.2.1 f_1 - support state

Each solution candidate keeps a vector of the cumulative demand for each period. When a delta evaluation asks for moving an order from one period to the other, the demand of this order can be simply subtracted from the *from*-period and added to the *to*-period. Subtracting the target value yields the new deviation for this single period. This value can then be compared to the deviation present before this change resulting in the delta value of f_1. For multiple moves the changes to each affected period have to be added up and can then be calculated in much the same way. To recalculate the entire f_1 value we can simply subtract the target demand per period d^* from each value then take the absolute of the results and create the sum of the vector. Using numpy's matrix operations

this operation is considerably quicker than calculating those deviations programmatically.

This vector also has the benefit of making it just as easy to count hard constraint violations. For this, one simply subtracts the capacity limit of each period w_i and counts the remaining values > 0.

3.2.2 f_2 - support state

Given the more complex nature of f_2 its support state is also somewhat more complicated. The idea is similar to that of f_1. Each solution candidate keeps a matrix of demands per period and product type. A row represents one period, a column a product type yielding a $n \times m$ matrix. Instead of subtracting a single value from each cell of a vector, this time a row vector is subtracted from each row in the matrix. The vector that is subtracted being the target demands of the product types. f_2 further requires the division by d_t^* which is done after the absolute deviations of each period are summed up to result in the deviations per product type. After division, this row is then also added up producing the f_2-value of this solution. Since this can be done by matrix operations again, recalculating the fitness value is fairly quick. For the delta evaluation the calculation is akin to the calculation of f_1 with the additional division being necessary.

Hard constraint violations of the second type are also easily supported by this structure. A vector of product type maximum capacities has to be subtracted from the matrix after which, again, all values > 0 can be counted.

3.2.3 f_3 - support state

Counting priority inversions is entirely more difficult than the two calculations prior. A lot of development went into finding a fast way of calculating this component of the fitness function. The support structure used is again a matrix. The rows represent the periods, the columns the priorities yielding a $n \times p_{max}$ matrix. Each cell represents the amount of orders with this priority in the given period. To calculate the f_3-value from this matrix we first have to calculate the cumulative sum along axis 1 resulting in a same sized matrix where each element of a row counts how many orders are smaller than or equal to its priority in this period, i.e. if we look at the third row and the fourth cell we see how many orders of priority 4 or less (or 3 or less, depending on whether priorities start at 0 or 1) are in this period. On this basis another cumulative sum is built along the other axis. This calculates a similar thing but across all previous periods, i.e. this same example of the third row 4th cell now counts all orders of priority 4 (or three) or less that have been planned in this, the first or the second period. With this information one can now plan an order for a period and check how many orders with a priority less than itself have been planned earlier resulting in the priority inversions this order creates. To receive the entire f_3 value we multiply the initial structure counting the appearance of a priority per period with the matrix we received after creating the cumulative sums. The matrices cannot be multiplied directly but have to be shifted since a priority inversion is

only present for orders that are planned earlier and have less priority, not in the same period or with the same priority. Meaning the order count for a (period, priority) combo must be multiplied with the (period − 1, priority − 1) combo of the cumulative sum matrix. This creates a matrix with the amount of priority inversions each period and priority combination creates which only needs to be added up to receive f_3. This entire calculation can be achieved using numpy's tensordot function with axes=2 resulting in the following, surprisingly fast code:

```
def f3(self, priority_dict: ndarray) -> float:
    priority_dict_summed_to_use = np.cumsum(
        np.cumsum(priority_dict, axis=1)
        , axis=0
    )

    return float(
        np.tensordot(
            priority_dict[1:, 1:],
            priority_dict_summed_to_use[:-1, :-1]
        )
    )
```

The calculation of the f_3-delta is a bit more involved too. First it must be determined how the order moves. If it moves from an earlier period up into a later period or vice versa. Depending on the direction the amount of inversions must be in- or decreased by the amount of orders that are affected. Since a delta evaluation can contain multiple moves (e.g. an order swap might contain two) the changing amount of orders per period and priority must be considered as well. The sum of priority inversion changes is then the resulting delta.

3.3 Construction Heuristics

Construction heuristics are an essential part of a memetic algorithm. There are a few important aspects such a heuristic has to fulfil. Given that a lot of solutions have to be created in order to provide a population at the start of a run, the construction heuristic must be fairly quick. This will be discussed in subsection 3.3.2. Genetic algorithms and memetic algorithms find new good solutions by putting together building blocks or as has been discovered more general formae [Rad91]. They are combined during crossover to form potentially better solutions. Early on it was argued that they have to be initially provided during the construction of the population. Mutation was considered more of a security policy to reintroduce prematurely deleted building blocks [Gol89]. [TS93] argues that this is mostly true for low cardinality alphabets. High cardinality alphabets, particularly combinatorial problems with high cardinality alphabets, are simply too vast to introduce all of the possible building blocks in a population. They conclude that a high mutation rate might be necessary to further explore the search space. This is in a way also supported by [Rad92] which argues that in k-ary string representations the

mutation operator is a way to "keep the gene-pool well stocked"[Rad92].
Memetic algorithms additionally have the stage of local search which might, for one, restore previous building blocks that have been prematurely removed from the population, fulfilling the role Goldberg had for mutation, it might however as well help find new building blocks, even though only within the neighbourhood and not in a particularly exploratory way.
Another important facet of construction heuristics is that they provide solutions that steer the algorithm in the right direction. [SG90] shows non-random construction heuristics and argues that the diversity of the initial population is of course important and gives the algorithm a lot of diverse building blocks, random initialisations also create a lot of nonsensical candidates, however, that just lengthen the search process. In our problem this could be reflected by a solution candidate that puts all of the orders into one period. This would of course be a possible solution, also part of the search space, but would generally not put us anywhere near a good solution.
With this being said, the construction heuristic needs to provide candidates that point in the direction of good and valid solutions while still creating a diverse population that contains a large number of different building blocks.
For this thesis we created different construction heuristics. Which construction heuristic fits the best for our algorithm and problem will be eventually decided during hyperparameter tuning. First, however, we will analyse our options and potentially remove inadequate algorithms. In the following we will first explain the algorithms, then discuss the quality of the solutions based on a few metrics. We will then introduce and evaluate the algorithms on another set of metrics for measuring the diversity of the solutions created by the algorithms, also establishing the notion of the Extended Jaccard Index. Based on those comparisons we will compare them and decide which construction heuristics will be considered during hyperparameter tuning.

3.3.1 Algorithms

This section explains how the different construction heuristics work and what their focus is. Their expected results in terms of population diversity and which parts of the fitness function they cater to is discussed.

First-Fit Construction Heuristic

The First-Fit construction heuristic is a fairly quick and easy construction heuristic. The idea is to assign orders to the first period they fit into without violating hard constraints. To accommodate the priority inversion part of the fitness function the orders are first sorted by priority. This clearly harms the diversity of the produced solutions and therefore the amount of provided formae. It is, however, one of the easiest ways to ensure some compliance with the priority order. Some diversity remains, since generally a lot of orders have the same priority. Those ties are broken randomly, yet they will clearly be spatially close, given the essence of the algorithm. The sorted orders are then, like stated above, iterated through and assigned to the first period that can accommodate this order

without violating a hard constraint. Any remaining orders are assigned a random period. Given the nature of the algorithm, there generally is a bit of skew towards the earlier periods, meaning that earlier periods usually have higher cumulative demand, while later periods tend to stay emptier. No concrete step is taken to match the target limit, the target limit being the cumulative demand each period tries to reach as closely as possible, which certainly affects the demand balance across the periods, affecting the f_1 and f_2 parts of the fitness function negatively. Given the focus of the heuristic, it is expected to do fairly well when it comes to hard constraint violations.
Summarily the First-Fit construction heuristic is a simple and quick algorithm that tries to honour the priority order. It fills up earlier periods first leading to an unbalanced solution while still trying to fulfil hard constraints as long as possible. The produced solution candidates are also not expected to be particularly diverse given the ordering that takes place in the beginning of the heuristic and the way those orders are distributed. Solutions created by the First-Fit construction heuristic are expected to lack performance when it comes to f_1 and f_2 but excel in f_3. They are also expected to have a fairly low amount of constraint violations.

First-Fit-with-Target-Limit Construction Heuristic

The First-Fit-with-Target-Limit construction heuristic addresses some issues the previous algorithm had. The idea is the same: orders are sorted by priority and assigned to the earliest accommodating period. The definition of whether a period can accommodate an order, however, has changed. In this heuristic orders are only accepted into a period if the demands don't exceed the target limit of the instance. In comparison, the First-Fit construction heuristic only checks for validity. Any remaining orders will then be assigned by the original rules, allowing orders only if they do not violate hard constraints. Lastly, all the orders still not assigned are assigned at random.
It is important to note that of course both, f_1 and f_2 target limits, so overall period target limits and target limits per product type and period are considered. This leads to a much more balanced solution candidate, while introducing a lot more priority inversions, given that lower priority orders are what remains after the first step, which are then assigned, starting at earlier periods. In terms of hard constraint violations, this heuristic should perform similarly to the First-Fit construction heuristic.

Vass-et-al Construction Heuristic

The Vass-et-al construction heuristic is a heuristic from the thesis originally introducing the Production Leveling Problem [VLM20], from the authors Vass et al., hence the name. This is a more involved construction heuristic trying to not only honour the priority order as well as possible but also find the best fitting period. This generally leads to good solutions while being a lot slower than the other heuristics. The algorithm iterates through the periods, calculates the delta cost of adding each order to the current period and adds an order to a list of suitable orders if adding it would improve the solution. This is done until either there are no orders left or a set amount of suitable orders have

been found. Another parameter, the selection size r, then further restricts this set of suitable orders to the best r options, i.e. the r orders with the best fitness improvement. One of these orders is then randomly chosen to be added to the period. This is done until the algorithm finds no suitable orders, i.e. the period is close to the target limit, after which the next period is considered. If there are any orders remaining, they are then, one by one, assigned to the period with the greatest remaining capacity.
This is the most considerate construction heuristic of this thesis, while also taking a lot longer to execute. It considers the priority orders in a very similar way to the previous two, while focusing more on balancing the periods and enforcing the hard constraints. While they originally state that by controlling the selection size r the diversity of the population can be increased, diversity is expected to be a problem with this implementation still. This comes as no surprise, however, since the construction heuristic was originally created for a simulated annealing and variable neighbourhood descent approach, where only one initial solution needed creating. Given the amount of care put into choosing the best match for an order, the heuristic is expected to do fairly well when it comes to hard constraint violations and fitness in general.

Random Construction Heuristic

The Random construction heuristic is a very simple construction heuristic assigning each order to a random period. The orders are shuffled randomly and then split into a number of sets corresponding to the periods. This leads to a solution that has roughly the same amount of orders per period, while completely disregarding any fitness concerns. The algorithm provides a good benchmark for comparing different construction heuristics, particularly when looking at diversity. While in early, non-combinatorial, low cardinality alphabet problems random solutions were the go-to for population creation, the sheer size of the search space commands some steering in the right direction. The number of hard constraint violations introduced in this construction heuristic will be a good benchmark, indicating when there is a certain tendency in one direction or whether there is none at all.

By-Demand Construction Heuristic

The By-Demand construction heuristic was created purely to create a very balanced solution, dealing with hard to accommodate orders early. The orders are sorted descending by their demand. The algorithm then walks through the periods one by one, moving back down when reaching the last period and assigning one order at a time. This leads to a very balanced solution, completely disregarding priorities and product types. The diversity of the created solution is also expected to be fairly low given the initial sorting of the orders. Depending on the problem instance there can be more or less orders with the same demand, so the diversity can differ from instance to instance. A diverse population would be, however, merely a coincidence and not at all inherent to the heuristic. Solutions created by this construction heuristic are expected to do exceptionally well when it comes to f_1, they are expected to do okay when looking at f_2 but will do about as well when it

comes to f_3 as the Random construction heuristic. It is expected to do very well when it comes to violating the per period capacity constraint. Which is expected to affect the per product type and period constraint violations in a positive way.

Next-Fit Construction Heuristic

The Next-Fit construction heuristic is a heuristic very similar to the First-Fit construction heuristic. Orders are first sorted by priority, even though this effect will be marginal given the rest of the algorithm. Then it tries to find the next period in which this order still fits without violating hard constraints. The main difference to the First-Fit construction heuristic is the starting point from which the algorithm looks for an accommodating period. The First-Fit algorithm always starts at the first period, the Next-Fit algorithm starts from the first period after the last assignment, i.e. the next period. E.g. If one iteration assigns period 2 to an order, the first period that is checked for the next order would be period 3, as compared to period 1 which would be checked when applying the First-Fit heuristic. It tries to approach the balancing issue the First-Fit construction heuristic has from a different angle, but creates a lot more priority inversions. Diversity wise this approach is expected to do slightly better than the First-Fit construction heuristic, since small changes should lead to larger changes down the line with this heuristic.
Solutions created by the Next-Fit construction heuristic are expected to do better in f_1 and f_2 than First-Fit, yet worse when it comes to f_3. In terms of violations, results should be comparable to the First-Fit construction heuristic. The results should be similar to First-Fit-with-Target-Limit when it comes to f_1 and f_2 and worse for f_3.

By-Demand-Next-Fit Construction Heuristic

The By-Demand-Next-Fit construction heuristic is, as the name implies, a combination of both, the By-Demand and the Next-Fit construction heuristic. Given the marginal benefits of sorting by priority in the Next-Fit construction heuristic and the obvious disadvantages of completely disregarding two thirds of the fitness function, this construction heuristic tries to combine the perks of the two. Orders are sorted by demand and should therefore help balancing the solution early on, yet they are tested for compliance with hard constraints. This, again, does not consider priority inversions. It is expected to be less diverse than the Next-Fit construction heuristic, since there are generally more unique demand values than priority values in the problem instances, which leads to a stricter order of orders and therefore less randomness when choosing the next order. It is expected to be more diverse than the plain By-Demand construction heuristic, since compliance must be considered which can be affected by small changes earlier in the algorithm, i.e. small randomness within the ordering of orders goes a longer way when checking whether an order fits or not, since the product types may be distributed differently and might therefore be full at different times, which might affect the period that is chosen next, which affects the period afterwards, etc. since this uses the Next-Fit heuristic. It is also expected to be more diverse than the First-Fit construction heuristic, for the

same reasons the Next-Fit construction heuristic is expected to be more diverse, which is very much in line with the reasoning about why it is more diverse than the By-Demand heuristic. It has to be noted that sorting the orders by demand instead of priority works against the diversity, as stated earlier, this is, however, expected to reduce the diversity only marginally, as compared to the reasons stated for Next-Fit vs. First-Fit. In terms of fitness this heuristic is expected to do well when it comes to f_1 and f_2. It is expected to outperform First-Fit, Next-Fit and First-Fit-with-Target-Limit when it comes to f_1. For f_2 better results than First-Fit and Next-Fit are anticipated. f_3 is of course expected to perform on par with Next-Fit and considerably worse than First-Fit and First-Fit-with-Target-Limit.

3.3.2 Runtime

This section discusses the differences and results when looking at the runtime of the various construction heuristics. The runtime is a measure that is not crucial to which algorithm will be chosen for parameter tuning, it can however disqualify a construction heuristic if the runtime is not accommodable. Constructing a population heuristically, while not being integral to the notion of a memetic algorithm, is an important part kick-starting the process.

instance name	# orders	# periods	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	2019	10	0.192962	0.278277	0.04847	0.04532	0.123304	0.121345	19.00843
958	888	5	0.055028	0.038168	0.012564	0.00939	0.014951	0.013543	6.55470
987	2706	4	0.218398	0.268908	0.061272	0.052801	0.112029	0.109302	67.01908
small_0001	86	8	0.256754	0.356063	0.129709	0.061948	0.153359	0.155395	0.03772
small_0003	34	7	0.003864	0.004927	0.005516	0.00163	0.002221	0.002636	0.00681
957	885	34	0.027551	0.035951	0.012892	0.010744	0.017223	0.015926	1.62304
966	975	22	0.024665	0.032291	0.011653	0.010574	0.019025	0.020056	2.77193
967	2527	31	0.104902	0.150392	0.032317	0.029904	0.080617	0.071563	13.8639
987	1725	26	0.057846	0.076989	0.020989	0.021413	0.044829	0.042186	6.25021
980	1449	23	0.047484	0.060679	0.018242	0.01669	0.034059	0.030369	4.84471
994	3360	37	0.152414	0.238769	0.044644	0.042904	0.115706	0.114404	22.03195
956	3583	50	0.181366	0.283796	0.044877	0.040892	0.116534	0.114389	18.44896
960	733	61	0.029476	0.035756	0.010081	0.008844	0.014441	0.013029	1.1809
962	3086	77	0.441327	0.255496	0.040578	0.038306	0.099578	0.09532	10.31609
964	3856	69	0.2445	0.327717	0.051597	0.048233	0.139232	0.136788	21.23846
970	112	57	0.003586	0.004438	0.001937	0.002265	0.00221	0.001885	0.02429

Table 3.1: This table shows the mean runtimes in seconds for each of the construction heuristics of a few chosen instances with increasing amount of orders, since that has been the main parameter affecting the runtime. Each value is the average of 50 runs. More specific data can be found in the appendix A.1.

When looking at the runtimes in Table 3.1 there exists one very obvious outlier with the Vass-et-al construction heuristic. The algorithm design explains the poor runtime, since for each period, every order is considered multiple times until the period is full. While this is of no particular concern when creating one solution for a simulated annealing run, it is problematic when such a construction has to be executed up to a couple hundred times. Therefore the Vass-et-al construction heuristic will not be considered during hyperparameter tuning but will be further analysed nonetheless, since it might provide

some insights in what makes a high quality construction heuristic for the problem at hand.

3.3.3 Solution Quality

As discussed earlier we expect each construction heuristic to create solutions that are strong in some aspects an weak in others. In the following we analyse whether those expectations are fulfilled and if there are any construction heuristics that create outstandingly strong solution candidates as well as the opposite and can be sorted out. We first look at the overall fitness average of a population with 100 individuals created by each construction heuristic. We then look at the various parts that make up this fitness value on their own. Lastly, we will discuss the prevalence of hard constraint violations in the created solutions.

Fitness - overall

The overall solution fitness consists of the three normalised and weighted sub-fitness values f_1, f_2 and f_3 that are explained in more detail in chapter 2. In addition to those weighted values, hard constraint violations are added. For each hard constraint violation 1 is added to the fitness value. The fitness value of a solution without hard constraint violation is typically below 1, making hard constraint violations particularly weighty.
Table 3.2 shows the average fitness values of 100 solutions created by a construction heuristic for a given instance. The best averages of an instance are highlighted in bold. Most of the best fitnesses are achieved by three construction heuristics: By-Demand-Next-Fit, Vass-et-al and First-Fit. The By-Demand-Next-Fit heuristic ranks highest most often but also performs very well when not in first. The heuristic is in terms of rank the best performing construction heuristic of the bunch. There are also no outliers except for instance 962 where it only places 4^{th}. First-Fit and Vass-et-al are close. With First-Fit performing a bit worse, yet more consistent and Vass-et-al having more first places while having a few more bad placements as well. The Random heuristic performs as expected, worse than any other heuristic in every single instance. The By-Demand heuristic performs surprisingly badly and is generally a lot closer to the Random construction heuristics results than expected, except for instance 970, for which it beats all the other heuristics. This, again, must be an outlier, where the sorting of orders by demand corresponds well to a good solution. The performance of the First-Fit-with-Target-Limit construction heuristic is surprising too, in that it is considerably worse than the First-Fit construction heuristic. This would have not been surprising were it only marginal. It seems, however, that the heuristic introduces a lot more constraint violations. The Next-Fit construction heuristic performs very average. Similarly to First-Fit-with-Target-Limit, slightly worse performance than the First-Fit construction heuristic would not have been surprising, particularly with the introduction of more priority inversions. That there are this many more hard constraint violations introduced, however, is surprising.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	1.20984	5.48576	39.36117	31.69154	2.70177	**0.4113**	1.79531
958	0.3159	1.35334	18.23439	12.9018	0.94774	**0.22198**	0.92303
987	0.85052	2.47204	16.53666	7.6801	1.91491	**0.11124**	2.59134
small_0001	**0.23092**	2.73015	8.99624	7.45302	1.10126	0.2428	0.27771
small_0003	**0.38874**	2.24078	6.90951	5.62134	2.55777	0.51536	0.62839
957	6.13806	22.24932	100.6481	99.26041	12.58682	**0.8428**	8.64692
966	0.41853	2.17289	31.26933	25.00536	0.86415	0.38763	**0.29489**
967	14.74353	40.24608	184.32559	172.9424	25.55374	9.94495	**9.90171**
978	2.2602	23.17795	112.88342	100.49048	6.99952	**0.68265**	7.23193
980	2.01215	4.78539	56.87753	37.43377	2.80417	2.24437	**1.18184**
994	0.57071	17.99127	146.34361	101.517	4.98209	**0.52554**	2.49298
956	0.35794	1.37748	50.62482	56.87084	0.30545	0.23857	**0.10086**
960	**0.84092**	7.30709	39.6401	3.87745	0.90094	0.88606	3.91812
962	0.36333	**0.00629**	24.84062	0.17022	0.29274	0.18792	0.02391
964	0.3174	5.00921	70.96599	33.33036	4.54743	0.21442	**0.17154**
970	1.32102	13.91778	44.23045	**0.25408**	10.55117	0.2793	2.04602

Table 3.2: This table shows the mean fitness values of 100 solutions created by the corresponding construction heuristics. The first block shows a sample of problem instances with small alphabets (number of periods), the middle shows medium alphabets and the bottom block shows large alphabets. Each fitness value is calculated by the formula for a solutions fitness. For each hard constraint violation present, 1 is added. The best fitness value of an instance is highlighted in bold.

Fitness - g_1

The fitness value g_1 is the normalised f_1 value that represents the overall deviations from the target demand per period. For this value, no product types are considered, it is just checked how far off the target of each period is from the cumulative demand per period. The numbers shown in the table have been normalised by dividing the absolute f_1 values by the target demand per period times the number of periods.
The results are not particularly surprising. The best value of each instance is highlighted in bold. The By-Demand construction heuristic is in all but one cases the best heuristic when it comes to balancing the demand across periods. This came as no surprise since that is the only strength of this particular heuristic. For instance 962 the First-Fit-with-Target-Limit construction heuristic performed better. By-Demand-Next-Fit is arguably the second best construction heuristic for this metric. Nine out of 16 times it placed second, four times it placed third, one time fourth and once fifth. The other heuristics performed somewhat worse with First-Fit performing the worst overall, most of the time even worse than the Random construction heuristic. The First-Fit construction heuristic was expected to be deficient when it comes to g_1, yet the degree of deficiency comes as a surprise. The changes made to improve this heuristic in the form of the First-Fit-with-Target-Limit construction heuristic appear to have had the desired effect. It outperforms the First-Fit construction heuristic in almost every instance and generally by a lot. The Next-Fit construction heuristic was created as an improvement for First-Fit heuristic and is an alternative to First-Fit-with-Target-Limit. It was expected to outperform First-Fit when it comes to f_1 and be comparable to First-Fit-with-Target-Limit. The heuristic

indeed outperforms First-Fit by a lot. When compared to First-Fit-with-Target-Limit, it is at times similar, can even deliver better results, yet in 10 out of those 16 cases it performs a bit worse. Vass-et-al performs not particularly well in this regard placing in the midfield with First-Fit-with-Target-Limit and Next-Fit. The comparison with By-Demand seems almost unfair given the otherwise general ineffectiveness of this construction heuristic.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0.16617	0.01165	0.0537	**0.00126**	0.03475	0.0079	0.03752
958	0.15149	0.01352	0.0414	**0.00039**	0.0293	0.01688	0.02971
987	0.14599	0.0093	0.01909	**0.0001**	0.01665	0.00198	0.00116
small_0001	0.10826	0.05908	0.17626	**0.01204**	0.05353	0.03679	0.05397
small_0003	0.07716	0.11192	0.16737	**0.0149**	0.10542	0.07607	0.05877
957	0.15344	0.10787	0.0903	**0.00535**	0.0639	0.054264	0.06552
966	0.15602	0.02667	0.08352	**0.00757**	0.06111	0.02661	0.07697
967	0.13758	0.02839	0.06006	**0.00061**	0.03205	0.02578	0.04535
978	0.13702	0.06559	0.06375	**0.00131**	0.03841	0.02251	0.04542
980	0.16366	0.01402	0.06338	**0.00291**	0.04299	0.01357	0.04173
994	0.15715	0.01903	0.05938	**0.00074**	0.0353	0.03657	0.03632
956	0.17736	0.00306	0.05855	**0.00148**	0.05137	0.00872	0.04282
960	0.12963	0.0611	0.15398	**0.00586**	0.05298	0.03202	0.03072
962	0.18047	**0.00154**	0.07881	0.00346	0.06542	0.01205	0.00897
964	0.16358	0.01294	0.05437	**0.00262**	0.05448	0.00655	0.03379
970	0.10981	0.20374	0.38371	**0.04887**	0.18715	0.05715	0.07139

Table 3.3: The table shows the mean g_1 values of 100 solutions created by the corresponding construction heuristics. The first block shows a sample of problem instances with small alphabets (number of periods), the middle shows medium alphabets and the bottom block shows large alphabets. The best fitness value of an instance is highlighted in bold.

Fitness - g_2

The g_2 value is the normalised f_2 value. f_2 represents the deviation from the target demand per period and product type. The value is normalised by dividing it by the number of periods times the number of products. In Table 3.4 the average g_2 values of 100 solutions created by the different construction heuristics are displayed. The best are highlighted in bold.
The picture this table paints is a lot less clear than the previous ones. First-Fit-with-Target-Limit has most of the time the best results, with 9 of 16 values being the best and another 3 being second best, while also having a few results near the bottom with 3 fifth and one sixth place. By-Demand-Next-Fit and Vass-et-al perform about as well as each other. All in all it is very hard to judge which construction heuristic performs best when looking at the g_2 values. It is, however, fairly easy to judge the worst contenders. Both Random and By-Demand perform very badly, with By-Demand having one exception, where it comes first. Next-Fit and First-Fit perform okay but all in all just very average. The First-Fit-with-Target-Limit construction heuristic outperforms

the First-Fit construction heuristic in a lot of the instances, as was predicted. The magnitude at which this outperforming happens is a lot less significant though, than it has been for f_1. Next-Fit outperforms First-Fit in most cases, yet does considerably worse when compared to First-Fit-with-Target-Limit. The performance of By-Demand is unexpectedly bad. The balancing work that goes into balancing for f_1 was expected to lead to an okay performance for f_2 and even though it outperformed the Random construction heuristic, it generally did only by little.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0.32633	**0.25246**	0.50361	0.40671	0.28309	0.26035	0.25724
958	0.15761	**0.04924**	0.26644	0.22647	0.11142	0.09935	0.06382
987	0.13345	**0.04086**	0.19503	0.14796	0.0957	0.04684	0.07238
small_0001	0.1073	0.07038	0.3067	0.29707	0.08162	0.07653	**0.05447**
small_0003	0.28896	0.32709	0.5202	0.47138	0.32474	0.27246	**0.26402**
957	0.6528	**0.61468**	0.87846	0.85004	0.62602	0.63025	0.62639
966	0.24632	**0.15865**	0.44081	0.35154	0.20883	0.20308	0.19244
967	0.29067	0.24633	0.54644	0.45398	0.25283	**0.23115**	0.27583
978	0.53562	0.53952	0.73186	0.6978	0.50444	**0.50122**	0.53426
980	0.23819	**0.09459**	0.38753	0.28685	0.16508	0.14259	0.11486
994	0.39951	0.3187	0.5438	0.46606	0.34729	0.3291	**0.02616**
956	0.17397	**0.00657**	0.19496	0.18723	0.09369	0.06831	0.04616
960	0.70014	0.72599	0.76331	0.70832	**0.68797**	0.6933	0.86683
962	0.18047	**0.00154**	0.07881	0.00346	0.06542	0.01205	0.00897
964	0.14884	**0.02669**	0.23887	0.1641	0.09297	0.04547	0.04806
970	0.10981	0.20374	0.38371	**0.04887**	0.18715	0.05715	0.07139

Table 3.4: In this table the mean g_2 values of 100 solutions created by the corresponding construction heuristics are displayed. The three blocks show small, medium and large alphabets. The best fitness value of an instance is indicated in bold.

Fitness - g_3

Table 3.5 shows the mean g_3 values of 100 solutions created by each construction heuristic for each instance. The g_3 value is the normalised f_3 value, calculated by multiplying f_3 with $\frac{2}{k*(k-1)}$ with k being the number of orders of the instance. A high g_3 value means a high amount of priority inversions, meaning that orders with low priority are often scheduled before orders with high priority.
This table is the very opposite of the g_2-table. The picture could not be clearer. The First-Fit construction heuristic delivers the best results for each instance except one and generally by a lot. This makes sense, since it is the heuristic with the most focus on priority and the heuristic that keeps this order the strictest. It was also expected to excel when it comes to f_3, as has been discussed in 3.3.1. The second and third best heuristics when it comes to g_3 are not as obvious, with First-Fit-with-Target-Limit being the second best in most instances and Vass-et-al being the third best in most instances. The Random construction heuristic performs a bit below the 0.5 mark which is expected. Also both, By-Demand and By-Demand-Next-Fit, were anticipated to do about as well as Random does, since there was no effort put into reducing priority inversions. Unsurprisingly the results are at times almost identical.

First-Fit-with-Target-Limit was expected to perform worse than First-Fit, yet the viability of solutions constructed by the First-Fit-with-Target-Limit construction heuristic is surprising. Next-Fit performs unsurprisingly worse than both First-Fit and First-Fit-with-Target-Limit, very close to random. The difference between First-Fit-with-Target-Limit and Next-Fit, however, was expected to be a lot smaller. This is mostly because of the unexpectedly good performance of First-Fit-with-Target-Limit. The Vass-et-al construction heuristic performing worse than the two heuristics that adhere a lot more closely to the priority order comes as no surprise. Yet just putting it in third place appears to be underselling the actual results when it comes to fitness. The solutions created by Vass-et-al are quite a bit worse than the solutions created by the best two in most cases, yet they are distinctly better than the rest of the solutions when it comes to priority inversions.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	**0.02201**	0.03494	0.43153	0.43068	0.43176	0.42913	0.12163
958	**0.0204**	0.03175	0.31964	0.31484	0.32104	0.31722	0.11849
987	**0.00322**	0.00563	0.18762	0.18611	0.18769	0.18726	0.05336
small_0001	**0.04606**	0.09204	0.42981	0.43172	0.40832	0.38839	0.11778
small_0003	**0.06787**	0.12527	0.42579	0.40518	0.38281	0.50048	0.13679
957	**0.06542**	0.17031	0.478	0.49504	0.4707	0.47486	0.105
966	**0.04854**	0.08268	0.46498	0.46873	0.46261	0.4738	0.07642
967	**0.04582**	0.06408	0.38727	0.38341	0.38654	0.38404	0.06154
978	**0.05265**	0.09853	0.47341	0.4541	0.46997	0.47677	0.09674
980	**0.03088**	0.05032	0.46984	0.462	0.46829	0.47459	0.07571
994	**0.04212**	0.07059	0.48128	0.48061	0.47848	0.47961	0.06323
956	**0.01981**	0.02353	0.48393	0.48637	0.48117	0.48461	0.0356
960	**0.03342**	0.05997	0.48839	0.48979	0.47995	0.48223	0.06169
962	**0.00715**	0.00962	0.489	0.48987	0.48571	0.49145	0.01792
964	**0.01491**	0.02875	0.48823	0.49089	0.47992	0.48717	0.02906
970	0.1842	0.21091	0.4891	0.46898	0.35059	0.495	**0.12967**

Table 3.5: This table shows the mean g_3 values of 100 solutions created by their corresponding construction heuristic. The three blocks show small, medium and large alphabets. The best fitness values of the instance are highlighted in bold.

Fitness - hard constraint violations

There are two types of hard constraint violations and they very much fit the concepts of f_1 and f_2. The first type of hard constraint violation occurs when the cumulative demand of a period is larger than c and the second hard constraint is violated whenever the orders in a period of a product type cumulatively exceed their c_t.
Each such violation is counted separately. A solution with 10 periods could technically only have 10 violations of the first hard constraint and if this same instance has 5 product types, it could have 5 hard constraint violations per period, so 50 violations when considering the second hard constraint and 60 altogether.
Table 3.6 displays the average number of hard constraint violations for 100 solutions created by the corresponding construction heuristic right after construction. Table 3.7 shows the number of hard constraint violations of the first type and table 3.8 shows the

hard constraint violations of the second type. It is not certain whether all of the 16 instances have valid configurations, i.e. can have 0 hard constraint violations.
By-Demand-Next-Fit delivers the best results when it comes to hard constraint violations. Most of the time it manages to not introduce any violations at all, while it still keeps them lower than the other heuristics in the other cases. All the violations introduced are created through the second hard constraint, which is a pattern that occurs through the rest of this section. It creates the least violations in all of the instances, while being tied in all but two, mostly with the First-Fit construction heuristic and Vass-et-al. The First-Fit heuristic performs best, tied with By-Demand-Next-Fit, in 11 out of the 16 instances and is in a firm second place. This also created most violations for the product type hard constraint, yet the difference to the first hard constraint is not as prominent here. Vass-et-al places third with 9 ties for the first place. The remaining heuristics perform a lot worse. The Random construction heuristic introduces the most hard constraint violations in all but one instance, closely followed by the By-Demand construction heuristic which tends to create the second worst results in most cases. It performs even worse than Random in one case but performs okay for a few other instances. All of this happens, while the results for the per period hard constraint are excellent. The By-Demand construction heuristic creates no hard constraint violations when it comes to the first constraint and performs on par with By-Demand-Next-Fit as best heuristic in this regard, yet it introduces a lot of violations when it comes to violations of the second type, looking at table 3.8, sometimes performing worse than Random. This development was unexpected in a way, since the performance when it comes to the second constraint was expected to be influenced by the performance when it comes to the first, which appears not to be the case.
Interestingly instance 962 is constructed without hard constraint violations by all construction heuristics but the Random construction heuristic. The remaining heuristics all perform okay.
The results of the Next-Fit construction heuristic are somewhat surprising. In terms of violations, the heuristic was expected to perform about as well as the First-Fit construction heuristic. Yet, while performing okay, the results are notably worse, with most of the difference introduced in the per product type and period capacity constraint. This might be explained by the better balancing the Next-Fit construction heuristic does from the start. A period is not, one after another, filled until there is no capacity left but all are filled somewhat evenly. This leads to an even remaining capacity, i.e. each period has about the same space left. That is not the case with the First-Fit construction heuristic, since earlier periods are filled first. Therefore the later periods have larger spaces, making it possible for orders of larger demands to be assigned as well, which might not be possible for solutions created by the Next-Fit construction heuristic. This leads to more orders being unassigned, that are then assigned randomly and clearly breaking capacity limits. This line of thinking also explains the even worse results of First-Fit-with-Target-Limit. Since the balancing is done twice, first with the target limit, second with the maximum capacity, it disqualifies larger demand orders even earlier and when reconsidering them there already is only the limited space left between the target and maximum capacity.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	**0**	5	38	31	2	**0**	1
958	**0**	1	17	12	**0**	**0**	**0**
987	**0**	2	16	7	1	**0**	2
small_0001	**0**	2	8	7	**0**	**0**	**0**
small_0003	**0**	1	6	5	2	**0**	**0**
957	5	21	99	98	11	**0**	7
966	**0**	1	30	24	**0**	**0**	**0**
967	14	39	183	172	25	**9**	**9**
978	1	22	111	99	6	**0**	6
980	**1**	4	56	36	2	**1**	**1**
994	**0**	17	145	100	4	**0**	2
956	**0**	1	50	56	**0**	**0**	**0**
960	**0**	6	38	3	**0**	**0**	3
962	**0**	**0**	24	**0**	**0**	**0**	**0**
964	**0**	4	70	33	4	**0**	**0**
970	1	13	43	**0**	10	**0**	1

Table 3.6: This table shows the mean number of hard constraint violations of 100 solutions created by the corresponding construction heuristic for each instance. The rows are partitioned into small, medium and large alphabets. The best results are highlighted in bold.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0	0	0	0	0	0	0
958	0	0	0	0	0	0	0
987	0	0	0	0	0	0	0
small_0001	0	0	2	0	0	0	0
small_0003	0	0	2	0	0	0	0
957	2	6	6	0	2	0	0
966	0	0	3	0	0	0	0
967	3	0	3	0	0	0	0
978	0	2	2	0	0	0	0
980	0	0	2	0	0	0	0
994	0	0	3	0	0	0	0
956	0	0	4	0	0	0	0
960	0	4	18	0	0	0	0
962	0	0	11	0	0	0	0
964	0	0	5	0	0	0	0
970	0	6	22	0	5	0	0

Table 3.7: This table shows the mean number of violations of the maximum capacity per period hard constraint of 100 solutions created by the corresponding construction heuristic for each instance. The rows are partitioned into small, medium and large alphabets.

Fitness - conclusion

In the following the construction heuristics are evaluated purely from a fitness point of view. Diversity will be addressed in the next section and is another important part when choosing a construction heuristic for memetic algorithms. When looking at overall fitness, the worst contenders are the Random and By-Demand construction heuristic.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0	5	38	30	2	0	1
958	0	1	18	12	0	0	0
987	0	2	16	7	1	0	2
small_0001	0	1	6	7	0	0	0
small_0003	0	0	3	5	0	0	0
957	3	15	92	98	9	0	7
966	0	1	27	24	0	0	0
967	10	39	181	173	24	9	9
978	1	19	109	99	6	0	6
980	1	4	53	37	2	1	1
994	0	17	141	100	4	0	2
956	0	1	46	56	0	0	0
960	0	1	21	3	0	0	3
962	0	0	11	0	0	0	0
964	0	4	65	33	4	0	0
970	0	6	22	0	5	0	0

Table 3.8: This table shows the mean number of violations of the maximum capacity per period and product type hard constraint of 100 solutions created by the corresponding construction heuristic for each instance. The rows are partitioned into small, medium and large alphabets.

Random was never expected to do particularly well when it comes to fitness. It is a well needed benchmark for comparison with other construction heuristics and will have better results when it comes to diversity. The By-Demand construction heuristic does very well when it comes to balancing the solutions, yet this good performance when it comes to f_1 did not translate well to f_2, including the corresponding hard constraints. The amount of violations this construction heuristic produces is unexpectedly high. It is just not a very good construction heuristic for the problem at hand. Vass-et-al performed about as expected. The runtime of this heuristic makes it unviable when it comes to using it for a memetic algorithm, yet in terms of fitness the heuristic performs amongst the best. There are next to no hard constraint violations introduced, performing slightly worse than By-Demand-Next-Fit and First-Fit. While Next-Fit and First-Fit-with-Target-Limit were both introduced to improve on the idea of the First-Fit construction heuristic, they perform worse when it comes to overall fitness, mostly on the basis of hard constraint violations. Both considerably improve the performance of the First-Fit construction heuristic when it comes to f_1. They both do better in terms of f_2 even though the difference is a lot smaller and while both are expected to lose out on f_3, which they do, the performance loss is a lot less considerable than assumed. The amount of hard constraint violations introduced, however, thwarts the apparent gains. Lastly, the two most promising construction heuristic when it comes to fitness while having reasonable runtimes need discussing. The First-Fit construction heuristic and By-Demand-Next-Fit construction heuristic are two very different approaches. The First-Fit construction heuristic creates solutions with a focus on priority, while keeping constraint violations low. Yet sacrificing levelling of the periods which reflects in their f_1 and f_2 values. The By-Demand-Next-Fit construction heuristic is very different in that way, since it

completely disregards priority. This can be seen when looking at the results of f_3. It does outstandingly well when it comes to hard constraint violations and produces passable results when it comes to levelling the orders.
Combining both, First-Fit and By-Demand-Next-Fit to produce an initial population could prove useful and is something that should be considered at a later point.

3.3.4 Solution Diversity

The diversity of a population is a key component to a functional memetic algorithm. The population created at the start needs to provide a diverse set of building blocks or formae to enable the algorithm to find good solutions [TS93][Rad91][SG90][Sud18]. Evaluating the diversity of a population, however, can be difficult. Sometimes Hamming-distance is used for measuring diversity of a population [CYTH14][Zhu03] (and the problems with using only that will be addressed later) but other than that there seems to be no standard way of measuring how diverse a population is [BGK04]. For a simple non-combinatorial problem where each gene acts somewhat independent of one another, allele coverage can be a helpful tool. Allele coverage shows how many of the alleles, or assignable gene values, are covered within a population. This is a tool introduced in [TS93] with the notion of expected allele coverage. They further discuss how large a population must be to have enough allele coverage when using a Random construction heuristic. With high epistasis this tool becomes less useful. Section 3.3.4 will go into detail on how allele coverage and expected allele coverage work and whether using it can still be of benefit to our problem. Another very simple tool that will be discussed in the following section, is what we call equality ratio. This simply checks for orders that are assigned the same period. This too will be further expanded on in the next section. Both those tools lack a certain sophistication given the complexity of the problem at hand. We introduce the extended Jaccard index to help further describe the diversity of a population created for our problem. This metric builds upon the Jaccard index and will be further explained in the corresponding part.

Equality ratio

The equality ratio is a very quick and simple way of evaluating a populations diversity. A similar notion was used in [Spe00] for measuring population homogeneity. It represents the average equality per gene and is arguably the opposite of the normalised hamming distance. A gene is equal to another gene in the same position of another solution candidate if the value is the same:

$$x = [\underline{1}, 0, 3, 2, \underline{2}, 0], \; y = [\underline{1}, 3, 2, 0, \underline{2}, 1]$$

$$D_{H,norm}(x, y) = \frac{4}{6}$$

$$equ(x, y) = \frac{2}{6} = 1 - D_{H,norm}(x, y)$$

If two solutions have the same value in every gene, their equality ratio is 1. If two solutions share no gene value, their equality ratio equals 0. The amount of equalities are added up and divided by the number of genes, producing the equality ratio. If a construction heuristic creates each solution in the same way, e.g. by just assigning each order a period in an ascending way, the construction heuristics diversity will be obviously low, displayed by a high equality ratio, in this case, 1.
While a low equality ratio does not necessarily equate a high diversity, a high equality ratio certainly gives strong indication of low diversity.

Table 3.9 shows the results of our heuristics. The Random construction heuristic is a valuable reference point in this endeavour, since it is often very instance dependent how likely a high equality ratio is. It is also the best performing construction heuristic when it comes to equality ratio, which is unsurprising. What is surprising, however, is that the Next-Fit construction heuristic comes very close for most instances, and even performs slightly better for instance 967. The results are notably bad whenever there are not a lot of orders. Which makes sense, since the start of the Next-Fit construction heuristic is very similar for each solution. This explains the results for instances ”small_0001”, ”small_0003” and ”970”. The lack of performance when it comes to ”962” and ”964” is hard to explain, but might be due to the high number of periods both of those instances have, leaving lots of capacity in every period which discourages diversity since no period has to be found that would allow a previously non-fitting order.
The First-Fit, First-Fit-with-Target-Limit and Vass-et-al construction heuristics, perform poorly yet understandably so. The high equality of First-Fit is inherent to the algorithm. The heuristic sorts the orders first by priority and then tries to assign them to the first period that can accommodate it. This, of course, leads to the orders being put in the same period until it is full, removing any benefit in terms of diversity that the little randomness provided within each priority class had. First-Fit-with-Target-Limit performs even worse at times. This arguably comes from the ”restarting” of the algorithm once it filled up the periods to its target limit. The First-Fit construction heuristic is able to introduce some randomness after some initial filling up, which is completely taken away from the First-Fit-with-Target-Limit heuristic, which starts again, filling up the periods, starting at period 1.
Vass-et-al was used with a random selection size of 10. The general high equality can be explained by the very careful choice of orders for each period and their initial sorting. Since this heuristic was not particularly viable from the start, given its runtime, the algorithm wasn’t tested with different values for r. This equality ratio is therefore not very meaningful when it comes to assessing the diversity of the algorithm. It also has to be noted, again, that the algorithm was not designed with diversity in mind, given that it was designed for a simulated annealing and variable neighbourhood descent algorithm.

The By-Demand construction heuristic lies in the midfield of the algorithms. The heuristic was expected to perform rather badly, given the strict order that is enforced in the beginning of the algorithm. The way orders are assigned periods, appears to positively affect the diversity of the created solutions.
The By-Demand-Next-Fit construction heuristic performs well. As a hybrid of By-Demand, which performs okay, and, Next-Fit, which excels when it comes to the measure of equality ratio, the results are about as expected. The heuristic performs well and even close to random at times, yet also has a few outliers. The order introduced with By-Demand seems to generally worsen the results in terms of diversity, yet the Next-Fit approach appears to introduce a lot of randomness.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0.82583	0.83053	0.10003	0.16403	0.10009	0.13753	0.83089
958	0.71274	0.77516	0.19971	0.36236	0.19998	0.26289	0.74147
987	0.47991	0.47532	0.24977	0.28775	0.24991	0.26158	0.84179
small_0001	0.78058	0.66201	0.12555	0.4735	0.21725	0.43252	0.31537
small_0003	0.66767	0.69844	0.14355	0.76368	0.30214	0.76342	0.32624
957	0.65389	0.74595	0.0294	0.27829	0.04559	0.11709	0.26428
966	0.73852	0.71577	0.04527	0.10806	0.04728	0.08871	0.58695
967	0.1684	0.1612	0.03227	0.15506	0.03215	0.05587	0.2994
978	0.72879	0.74029	0.03837	0.19966	0.03968	0.09872	0.5052
980	0.77778	0.77096	0.04359	0.11197	0.04483	0.0835	0.57972
994	0.73309	0.72636	0.02701	0.07842	0.02786	0.05775	0.59237
956	0.71909	0.69923	0.02	0.04081	0.02138	0.03122	0.69636
960	0.74286	0.73426	0.01645	0.10004	0.16747	0.09133	0.33372
962	0.79322	0.78784	0.01299	0.01932	0.04622	0.01432	0.7102
964	0.72959	0.72197	0.01454	0.02496	0.02171	0.01937	0.69164
970	0.40146	0.46882	0.0178	0.16592	0.28939	0.16107	0.20169

Table 3.9: This table shows the equality ratio (smaller is better) of each construction heuristic on a given instance. The instances are blocked by alphabet size, starting with a small alphabet size, then medium, then large.

Extended Jaccard Index

Since the solution representation is highly non-linear, the value of a single gene, all by itself, says little about the fitness of the solution. The fitness is much more a product of how the orders are grouped together. This, of course, has an effect on how building blocks or formae work for our problem. We will use building blocks and formae interchangeably from here on out unless otherwise stated, since the idea of formae is better conveyed through the ”building blocks” name. If order 1 and 2 are in period 3 together and perform well in terms of fitness for this period, it is less relevant that they are both in period three, than that they are in a period together. Whether that’s period 3, 2 or 1, can of course impact the f_3 value, but is, as such, of little relevance to the quality of the solution, particularly when it comes to building blocks. Building blocks and formae in this problem, are much more the set of orders that are planned together and work well together.
As mentioned before, one of the jobs of a construction heuristic, is to provide a diverse

set of solutions with many different building blocks. This means in our case, that it is important, that solutions provide a lot of different combinations of orders for a given period. The equality ratio is not capable of recognising this type of diversity if orders are not put into the same period together.

The following should paint a clearer picture of why this is an issue: Assume there are two solution candidates for a problem instance with 3 periods and 6 orders. Solution 1 looks as follows: $[0, 0, 1, 1, 2, 2]$. So order 0 and 1 are assigned to period 0, order 1 and 2 are scheduled for period 1 and order 2 and 3 are planned to be handled in period 2. Solution 2 looks different but somewhat recognisable: $[1, 1, 2, 2, 0, 0]$. These two solutions very much differ when it comes to f_3, the amount of priority inversions, at least assuming that priority behaves as normal and is not the same for all, etc.. When looking at hard constraint violations and f_1 and f_2, however, both solutions are the exact same. Illustrating the earlier point about the irrelevance of period and relevance of which orders are put together. This is a dynamic the equality ratio fails to capture. The equality ratio of the two solutions above would be 0, implying there is no relationship whatsoever. Yet it is this diversity that is crucial for how well a genetic or memetic algorithm performs. Measuring this equality proofed harder than expected. To the best of our knowledge there is no metric available that would capture the similarity of two sets of sets. There is a very well known metric for measuring the similarity of two sets in the Jaccard index. The Jaccard index is defined as follows:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

The size of the intersection of two sets divided by the size of the union of two sets. This will come in very handy, yet is not particularly applicable in this form. To compare two solutions and how similar they are, the overlap of each period needs to be calculated. For this the Jaccard index of each period of one with each period of the other solution is calculated creating n values for each period of one solution, with n being the number of periods, and therefore n^2 values for the comparison of two solutions. This matrix makes a comparison difficult, particularly if more than two solutions are to be compared. Some consolidation is necessary.
We looked at two of several methods of capturing this information. First averaging the values across all the periods for each period was tried. This dilutes the values and makes it hard to see differences. The other option considered, was to take the biggest overlap of each period with another period. This represents strong overlaps very well, yet has issues capturing more sophisticated distributions. This is further illustrated in an example below. Eventually it was decided that the issues of averaging can be rectified much more easily than the issues of taking the maximum, by simply normalising. For this, the minimum and maximum values possible need to be calculated. The quickest way of finding these values was to enumerate the possibilities of small examples and simply find the minimum and maximum values the extended Jaccard index calculation could produce. The maximum value possible was found to be $\frac{1}{n}$, the minimum value possible,

for every problem with more orders than periods, was found to be $\frac{1}{n^2}$. Next it is shown how the minimum and maximum values possible are derived from the formulas. The Jaccard index divides the size of the intersection by the size of the union. The set of union is always at least as large as the set of intersection $|X \cup Y| \geq |X \cap Y|$. Hence $\frac{|X \cap Y|}{|X \cup Y|} \leq 1$. Therefore no single Jaccard index can be larger than 1. It is important to notice, that an order can only be assigned to one period, meaning an order that already intersected, cannot intersect again. Also, every order has to intersect at one point, since every order must be assigned. These bits of information are useful to show the minimum and maximum values the index can reach.
The next part illustrates why the following formula is true:

$$\frac{1}{n} \geq \frac{\sum_{X \in N_{S1}} \frac{\sum_{Y \in N_{S2}} \frac{|X \cap Y|}{|X \cup Y|}}{n}}{n} \geq \frac{1}{n^2}$$

With n being the amount of periods and hence period sets, and N_{S1} and N_{S2} being the period sets for solution 1 and solution 2. Further it is important to note that $|N_{S1}| = |N_{S2}| = n$. First we show that $\frac{1}{n^2}$ is the lower bound of this index. To get the smallest possible value from the inner fraction, the union of sets must be as large as possible. This is the case if all of the orders of the problem are in the union, meaning, $|X \cup Y| = k$, with k being the number of orders. With this, the inner part and eventually the entire numerator can be written as:

$$\sum \frac{|X \cap Y|}{k} = \frac{1}{k} \sum |X \cap Y| \implies \frac{1}{k} \sum_{X \in N_{S1}} \sum_{Y \in N_{S2}} |X \cap Y|$$

Since each order has to be intersected once, the sum of intersections across all period combinations between solutions 1 and 2 is k, simplifying to $\frac{1}{k}k = 1$. Leaving only the two divisions by n: $\frac{\frac{1}{n}}{n} = \frac{1}{n^2}$.
Along very similar lines we can show why $\frac{1}{n}$ constitutes the upper bound of our index. The maximum overlap a period can have is 1. Since every order can only intersect once, the sum of intersections of one period of solution 1 with the periods of solution 2, is equal to the orders in the period of solution 1, so:

$$\sum_{Y \in N_{S2}} |X \cap Y| = |X|; X \in N_{S1}$$

If this period is only overlapped with subsets, the size of union is always equal to the size of the period set of solution 1:

$$|X \cup Y| = |X| \text{ iff } Y \subseteq X, X \in N_{S1}, Y \in N_{S2}$$

This leads to

$$\frac{1}{|X|} \sum_{Y \in N_{S2}} |X \cap Y| = \frac{|X|}{|X|} = 1; X \in N_{S1} \text{ iff } \forall Y \in N_{S2} : Y \subseteq X$$

Assuming all this, the formula looks like follows:

$$\frac{\sum_{X \in N_{S1}} \frac{1}{n}}{n}$$

Since $|N_{S1}| = n$ we get $\sum^{n} \frac{1}{n} = 1$, yielding our $\frac{1}{n}$ as best possible result.

With the minimum and maximum value now calculatable, we can go on and normalise the extended Jaccard index results trivially by

$$\frac{x - x_{min}}{x_{max} - x_{min}}$$

Calculating the similarity of a whole population this way is computationally somewhat expensive, yet nothing that has to happen regularly, which makes it acceptable. Every solution has to be compared with every other solution in the population, eventually averaging those results as well.
The following tries to clarify the idea and struggles of the Extended Jaccard Index with an example.

This clarification uses the example from above.
To reiterate: solution 1: $[0, 0, 1, 1, 2, 2]$*, solution 2:* $[1, 1, 2, 2, 0, 0]$*.*
Translated into the relevant period sets: solution 1: $\{\{0, 1\}, \{2, 3\}, \{4, 5\}\}$ *and solution 2:* $\{\{4, 5\}, \{0, 1\}, \{2, 3\}\}$
The overlap of this example is rather obvious and should be reflected as such. The Jaccard index for each combination of period 0 of solution 1 with solution 2 would be: $\frac{0}{4}$*,* $\frac{2}{2}$ *and* $\frac{0}{4}$*. Since period 0 of solution 1 overlaps entirely with period 1 of solution 2 and not at all with periods 0 and 2. When averaged, this leads to a value of* $\frac{0+1+0}{3} = \frac{1}{3}$*. If a best case, diversity wise, came around:* $[0, 0, 0, 0, 0, 0]$*, the period sets would look like the following:* $\{\{0, 1, 2, 3, 4, 5\}, \{\}, \{\}\}$*, leading to Jaccard indices of* $\frac{\frac{2}{6}}{3}$*,* $\frac{\frac{2}{6}}{3}$ *and* $\frac{\frac{2}{6}}{3}$ *and hence* $\frac{\frac{1}{9}+\frac{1}{9}+\frac{1}{9}}{3} = \frac{1}{9}$*. When using the maximum value instead, the first example of maximum overlap would show up as 1 and this example of minimum overlap shows up as* $\frac{1}{3}$*. If we now go on to normalise the values of the average results, our maximum example produces an index of:* $\frac{\frac{1}{3}-\frac{1}{9}}{\frac{1}{3}-\frac{1}{9}} = \frac{\frac{2}{9}}{\frac{2}{9}} = 1$ *and our minimum example:* $\frac{\frac{1}{9}-\frac{1}{9}}{\frac{1}{3}-\frac{1}{9}} = 0$

With the Extended Jaccard Index explained, the results of the tested construction heuristics can be discussed. In table 3.10 the extended Jaccard indices of each heuristic and instance combination are presented. The Random construction heuristic somewhat confirms the validity of the normalised index, with most of the results being around 0.5. The first very clear outliers are both First-Fit construction heuristics: First-Fit and First-Fit-with-Target-Limit. Both are among the worst performing heuristics for most of the instances. The poor performance comes as no surprise given the nature of this heuristic. Both were expected to perform badly when it comes to diversity and they do indeed. Vass-et-al comes next, also finding itself consistently in the last third. The way solutions are created leaves little room for diversity. Both By-Demand and By-Demand-Next-Fit

perform surprisingly well, with the latter achieving a bit better results. As mentioned earlier the stricter order was expected to result in less diverse solutions, yet the small deviations that exist seem to have a larger effect. The By-Demand-Next-Fit construction heuristic performs a bit better when it comes to diversity, which makes sense, given that a large part of the heuristic is taken from the Next-Fit construction heuristic, which performs outstandingly. The Next-Fit construction heuristic is by far the best heuristic when it comes to diversity, particularly for small alphabets it seems. The results are generally very close to Random. Next-Fit is by far the best construction heuristic of the bunch when it comes to the Extended Jaccard Index.

instance name	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
953	0.73401	0.77702	0.47502	0.47823	0.0711	0.47579	0.7688
958	0.63835	0.69277	0.44592	0.46823	0.13121	0.44841	0.6543
987	0.50691	0.524	0.42898	0.43031	0.15221	0.42912	0.72201
small_0001	0.73855	0.6546	0.49245	0.55798	0.16327	0.53629	0.5097
small_0003	0.71571	0.69106	0.52203	0.76891	0.22383	0.75652	0.54473
957	0.66164	0.71836	0.50246	0.53782	0.50157	0.50741	0.53544
966	0.68214	0.69949	0.49417	0.49734	0.4933	0.49521	0.61189
967	0.49804	0.51427	0.49492	0.50489	0.49418	0.49493	0.53378
978	0.69974	0.71233	0.49405	0.51071	0.49331	0.49592	0.59021
980	0.74372	0.73833	0.49297	0.49949	0.49235	0.49466	0.61952
994	0.68134	0.70805	0.49595	0.50008	0.49546	0.49704	0.63043
956	0.64996	0.69526	0.49848	0.50035	0.49792	0.49875	0.6845
960	0.6889	0.73233	0.51772	0.52734	0.53749	0.52509	0.56676
962	0.69699	0.76181	0.50307	0.50358	0.50739	0.50315	0.70192
964	0.67859	0.71203	0.50088	0.50162	0.50047	0.50087	0.68369
970	0.86654	0.84529	0.66381	0.69831	0.73333	0.71275	0.7792

Table 3.10: This table shows the extended Jaccard Index (smaller is better) for each construction heuristic and instance. The instances are partitioned into small, medium and large alphabets.

Expected Allele Coverage

Allele coverage and Expected Allele coverage are a notion introduced in [TS93]. Chromosomes have a number of genes and those genes have a number of possible values, also called alleles. The cardinality of the alphabet used is equal to the number of alleles per gene. If a chromosome has k genes, it can cover k alleles. With a binary alphabet, in theory, two solutions could cover the entire space, as long as they are disjoint. It helps a genetic algorithm if as many alleles as possible are covered. For a linear problem it could be argued that those two intentionally disjoint solutions could suffice. Yet since most applications of genetic or memetic algorithms are non-linear, it helps if some combinations of alleles are present in a few solutions, to help with finding appropriate and powerful schemata. Initially those starting populations were created randomly. [TS93] talks about how for simple binary encodings random generation of an initial population is feasible but how the need for larger populations grows with the cardinality of the alphabet used and the non-linearity present in the problem. They introduce the concept of Expected

Allele Coverage that, based on the cardinality and population size, shows how much of the alleles are covered if a population is created randomly [TS93]:

$$p_m(N,K) = \frac{1}{K^N}\binom{K}{m}\sum_{v=0}^{K-m-1}(-1)^v\binom{K-m}{v}(K-m-v)^N$$

$$E[ac] = \frac{1}{K}\sum_{m=1}^{K-1} m\ p_m(N,K)$$

with N being the population size and K the cardinality of the alphabet. Even though the initial paper presents $E[ac]$ to be the Expected Allele Coverage, the result of the above formula must be subtracted from 1 to receive the actual $E[ac]$ which appears to be an oversight in the initial paper:

$$E[ac] = 1 - \frac{1}{K}\sum_{m=1}^{K-1} m\ p_m(N,K)$$

They then go on and show like in table 3.11 for different cardinalities what size a population needs to be to cover more than 99% of the alleles [TS93]:

Alphabet Size	Required Population Size	Expected Coverage (%)
2	7	99.22
4	17	99.25
8	35	99.07
16	72	99.04
32	146	99.03

Table 3.11: This table from [TS93] shows the population size needed to reach at least 99% of allele coverage for a given cardinality when initialising a population at random.

For validation the following plot (Figure 3.1) shows the expected coverage versus the actual allele coverage of a random population for a given cardinality with the population size growing along the x-axis. The two plots show the results for a small and large alphabet, more plots can be found in the appendix Figure B.1, generally behaving very similarly. Notable is however, that for large cardinality alphabets the population needs a certain size for the formula to produce realistic results. Generally far below the cardinality and hence far below any reasonable population size. To avoid scaling issues and confusion those early populations will be omitted in later plots.

Since our problem encoding is highly epistatic and can have a fairly large alphabet size, depending on the number of periods, allele coverage is only helpful to a small extent. It cannot really be used as a metric to evaluate whether our populations are diverse enough, yet it can give an indication of whether our construction heuristics generate a

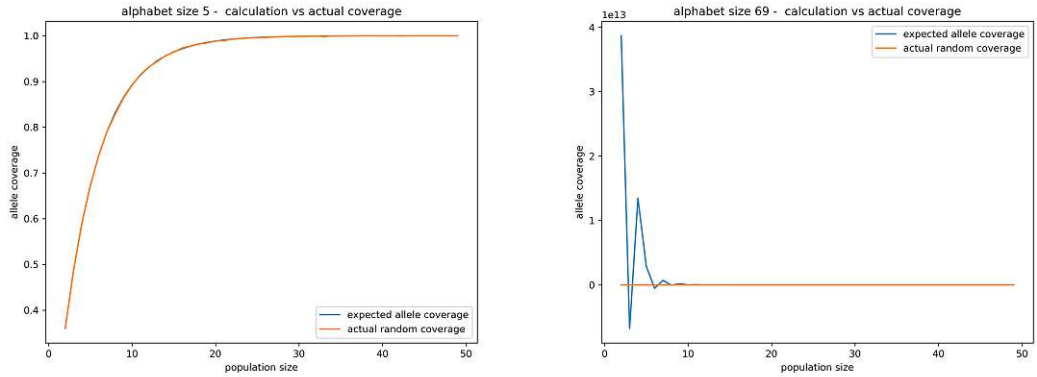


Figure 3.1: The expected and actual allele coverage values for small and large cardinalities based on the calculations from [TS93] and its change along increasing population sizes.

lot of similar solutions, very similar to the equality ratio.
In the following we will calculate the expected allele coverage based on the population size and cardinality and compare it to the actual allele coverage the construction heuristics produce.

instance small_0001 - 8 periods

heuristic / population size	10	20	50	100	200
expected	0.73692	0.93079	0.99873	0.99999	1
First-Fit	0.19767	0.20494	0.20784	0.20784	0.20784
First-Fit-with-Target-Limit	0.24854	0.27761	0.29505	0.29505	0.29651
Random	0.72674	0.94186	1	1	1
By-Demand	0.38662	0.41279	0.42877	0.43023	0.43023
Next-Fit	0.61191	0.71511	0.76308	0.78343	0.78343
By-Demand-Next-Fit	0.43313	0.5029	0.52034	0.5218	0.5218
Vass-et-al	0.48982	0.58866	0.73401	0.78197	0.83284

Table 3.12: This table shows the development of the allele coverage (bigger is better) based on population size for each construction heuristic on a small instance with 8 periods.

The three tables 3.12, 3.13 and 3.14 show how different sized alphabets represented by three instances behave based on population size. Further examples are given in the appendix (B.1, B.2, B.3). The tables show very clearly how the allele coverage grows with rising population sizes, yet that there are differing limits each construction heuristic converges towards. This shows that while for small populations it might be viable to use one construction heuristic, its diversity might be limited if the population size requirements grow. The data shows that both First-Fit and First-Fit-with-Target-Limit not only start a lot lower in coverage but also seem to converge to a fairly low limit.

instance 978 - 26 periods					
heuristic / population size	**10**	**20**	**50**	**100**	**200**
expected	0.32443	0.54361	0.85928	0.98019	0.9996
First-Fit	0.07021	0.07674	0.08374	0.08836	0.09108
First-Fit-with-Target-Limit	0.06958	0.07638	0.08782	0.09204	0.09638
Random	0.32437	0.54283	0.85841	0.98013	0.99946
By-Demand	0.21346	0.29168	0.36013	0.37687	0.37855
Next-Fit	0.32372	0.53917	0.85039	0.96784	0.98862
By-Demand-Next-Fit	0.27567	0.43023	0.62169	0.70956	0.74053
Vass-et-al	0.11121	0.13634	0.17112	0.19835	0.22296

Table 3.13: This table shows the development of the allele coverage based on population size for each construction heuristic on a medium instance with 26 periods.

instance 964 - 69 periods					
heuristic / population size	**10**	**20**	**50**	**100**	**200**
expected	-	-	-	0.51806	0.94605
First-Fit	0.02544	0.02776	0.02985	0.02989	0.03223
First-Fit-with-Target-Limit	0.02661	0.02932	0.0325	0.03255	0.03589
Random	0.13599	0.25332	0.5183	0.51902	0.94596
By-Demand	0.12986	0.23341	0.4397	0.43871	0.70912
Next-Fit	0.1318	0.24012	0.45201	0.45296	0.72911
By-Demand-Next-Fit	0.13308	0.2444	0.48401	0.48464	0.70924
Vass-et-al	0.0304	0.03664	0.04664	0.05469	0.06238

Table 3.14: This table shows the development of the allele coverage based on population size for each construction heuristic on a large instance with 69 periods.

Random behaves as expected very much in line with the calculated expected allele coverage. Next-Fit performs amongst the best again and is generally a very strong heuristic when it comes to diversity. Vass-et-al achieves a high allele coverage for the small population but cannot keep up when the population size grows. This could be due to it's parameter controlling randomness. The random selection size r is set to 10 across all these experiments. If this parameter was to increase with population size, the diversity would surely be able to keep up. Since the construction heuristic is not relevant for further use, given the unviable runtime, this will not be investigated further. By-Demand and By-Demand-Next-Fit lie somewhere in between the best and worst contenders. While it seems that By-Demand-Next-Fit outperforms By-Demand in most cases, on closer look the difference seems to vary less with more periods. While investigating this peculiarity we found that it does not depend on the amount of orders and also not on the amount of orders per period. The only correlation that could be found, while very weak, is between periods and difference. This correlation varies with different population sizes. When calculating the Pearson correlation coefficient for period amount and difference between

By-Demand and By-Demand-Next-Fit a fairly strong negative correlation of -0.526944491 could be found for population 10, this strongly reduced for population 50 to -0.216800649, for a population size of 100 we calculated even less with -0.182506019 and for our largest test population of 200 the correlation apparently grew again to -0.263338734. When plotting the allele coverages it seems that the instances of medium size produce more outliers which By-Demand-Next-Fit seems to be able to handle better. The single spike early in the graph can be explained by the low amount of orders instance "small_0003" has. Results for this instance have regularly been shown to be an outlier, particularly when it comes to diversity. This seems to show that there is no relevant correlation between the amount of periods and the difference in performance of the By-Demand and By-Demand-Next-Fit construction heuristic, yet it appears that By-Demand-Next-Fit not only generally performs slightly better but is also more robust.

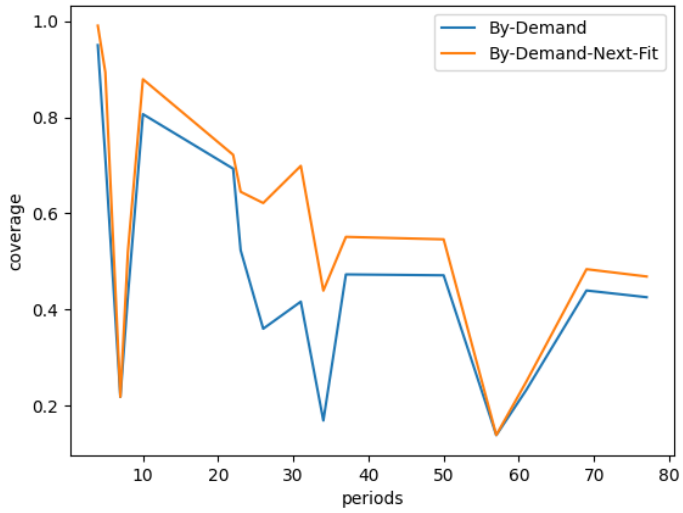


Figure 3.2: This graph shows the coverage versus the amount of periods represented by a single instance per period and the measured coverage of the By-Demand and By-Demand-Next-Fit construction heuristic when producing a population of size 50.

3.3.5 Diversity - conclusion

In conclusion we can say that as expected the Random construction heuristic performs best in terms of diversity. Followed closely by the Next-Fit construction heuristic which at times even outperforms the Random construction heuristic. First-Fit and First-Fit-with-Target-Limit both perform unsatisfactory and are regularly last when it comes to our diversity metrics. Even though the Vass-et-al heuristic will not be considered further it has to be mentioned again that the results are not particularly representative. The heuristic does well when it comes to expected allele coverage for small alphabets. The selection size chosen is also small. It can be expected that with growing selection size

the results for larger alphabets improve too. Still, Vass-et-al achieves in both equality ratio and Extended Jaccard Index fairly bad results that do not appear to improve with smaller alphabets. Hence, even adjusting the selection size might not make the heuristic particularly diverse, even though it certainly helps to an extent. Last but not least By-Demand and By-Demand-Next-Fit perform well in most diversity, and also fitness, regards. By-Demand-Next-Fit outperforms the By-Demand heuristic in every category and for most instances. Both algorithms are regularly amongst the best heuristics tested. When it comes to equality ratio they are slightly worse than Random and similarly matching the Random construction heuristic in most cases when it comes to the Extended Jaccard index. This means that both heuristics not only perform well in relation to other heuristics but also in terms of absolute numbers.

Conclusion - overall

The best overall results are achieved by the By-Demand-Next-Fit construction heuristic. It's constructions not only perform well when it comes to fitness and hard constraint violations, they are also diverse. The First-Fit construction heuristic lacks when it comes to diversity, yet produces great results in terms of fitness. Vass-et-al is too slow for a population based heuristic but generally creates great solutions. First-Fit-with-Target-Limit performs well in terms of fitness but introduces entirely too many hard constraint violations. By-Demand and Next-Fit are greats when it comes to their specialities, f_1 and diversity respectively but cannot translate those to other areas.
During the experiment we will consider creating a population made up of both By-Demand-Next-Fit and First-Fit solutions. The lack in diversity can be balanced out by only creating a small amount of First-Fit solutions yet the quality of those solutions should not be overlooked. Given the very different types of solutions created by First-Fit and By-Demand-Next-Fit mixing both types of solutions will, rather ironically, create an even more diverse population. Whether or not this actually achieves better results will be discussed in chapter 4.

3.4 Selection

Selection is the part of the algorithm that steers the search towards better solutions. Here it is decided which individuals to pick for reproduction. In this step the search can be guided strongly towards better solutions, called high selective pressure, or not so strongly, low selective pressure. While high selective pressure generally leads to fast improvements it makes it difficult to break out of local optima and leads to premature convergence since it only allows very good candidates to be part of the population. Hence it is important to find a balance that keeps good solutions in the population while also allowing less well performing solutions that might be within the basin of attraction of another optimum.
Initially selection was introduced as fitness proportionate, or Roulette Wheel, selection [Hol75]. Which we will shortly look at in more detail. Over the years a few more

approaches of selecting the next individuals emerged. In addition to fitness proportionate selection we will discuss Rank Selection [BBM93b][GD90][Whi89] and Tournament Selection [BBM93b][GD90][Whi89] and why Tournament Selection is our tool of choice. We will also discuss the scaling problem and how it affects our decision.

3.4.1 Fitness Proportionate Selection

Fitness proportionate or Roulette Wheel Selection selects the individuals in relation to their fitness values [Hol75][GD90]. An individual with twice as good a fitness, will be chosen twice as likely while an individual with nine tenths of the fitness of another will be chosen 10 percent less likely. This fact while being subtle perfectly describes the scaling problem and why it becomes an issue at later stages of the search. As the solutions get better their fitness converges. Where in the beginning there generally are a wide range of fitnesses, towards the end of the search the fitnesses of the individuals in the population differ less and less. When using fitness proportionate selection this means that in the beginning when there are large differences between the fitnesses, there is high selection pressure, while towards the end this pressure shrinks dramatically [Whi89]. This can be addressed by scaling the fitness when selecting or by choosing a different type of selection like rank selection.

It should be noted that for fitness proportionate selection we cannot use our fitness calculation out of the box since we are trying to minimise the fitness value. To address this we simply divide 1 by our fitness value leading to larger values when the fitness is smaller.

3.4.2 Rank Selection

Rank Selection is a selection method that addresses the scaling problem by ranking the solutions by fitness and then choosing them randomly with probabilities according to their rank [BBM93b][Whi89][GD90]. This means that it does not matter how the fitness behaves towards the end of the search. When using linear ranking a solution one rank better than another will be chosen twice as likely and a solution two ranks higher will be selected three times as likely. This selection can of course also be non-linear to increase or decrease selection pressure [BBM93b]. This type of selection makes it possible to tune selective pressure very precisely while keeping it steady throughout the search. [BBM93b] further argues that [Whi89] and [Bak85] show ranking "to be superior to fitness scaling"[BBM93b].

For the thesis we implemented the pseudo code from [Whi89] appendix 1.

3.4.3 Tournament Selection

Tournament selection is the third and last selection method implemented for this thesis. The idea is rather simple. k individuals are sampled randomly and the best of those k is

chosen for the next population [Whi89][BBM93b][GD90]. By increasing k the selective pressure increases. One clear disadvantage tournament selection has compared to rank selection is that k is a natural number which means that tuning the selective pressure can only be done coarsely [FL10]. In [HH08] tournament selection is investigated for relations with polynomial ranking methods. Amongst other results it is shown that a binary tournament selection is equivalent to linear ranking without the need of sorting a population. [GD90] has shown this as well earlier and also argues to use tournament selection for its speed and control of selective pressure.

3.4.4 Conclusion

Based on the information gathered, the experiments will be conducted using Tournament Selection with the tournament size as tunable parameter. If the tournament size is found to be 2 we will reconsider using Rank Selection to more finely downtune the selection pressure, for all other intents and purposes Tournament Selection should be the right choice.

3.5 Crossover

The crossover operator is one of the unique features of evolutionary algorithms. During recombination multiple individuals of a population are combined to produce offspring based on their own properties. It is here where good features of one solution can be combined with good features of another solution, creating a new, better, solution than existed before. The crossover operator is next to the mutation operator one of the most important tools of exploration of a genetic algorithm [ECS89]. How much of the population of the next generation is created by recombination depends on the crossover rate. The missing amount of individuals is then generated from solutions of the previous generation. Sometimes, as it is the case with our algorithm, the currently best known solution is reintroduced in the population as well.
Crossover methods reach from historically studied implementations that are not specific to a problem to problem specific operators that make use of peculiarities of a concrete problem formulation. The main goal of both is, however, to explore the search space and create better individuals from the good bits of other solution candidates.
”In order to explore they must disrupt some of the schemata on which they operate” [ECS89]. This disruption means that children lose membership of some schemata their parents are in. This is of course true for every change made to a chromosome. Since every gene is part of a large number of schemata, changing a single allele introduces disruption. Higher disruption means lower probability of survival for a schema. While disruption is important and necessary for the algorithm to find new and better solution candidates, it must not be too strong, since too much disruption, and hence exploration, might just overwhelm the exploitation efforts of the selection operator [ECS89].
The way disruption works, however, is a lot more subtle than this section might have let on. For one, disruption can come from various places. It is for example affected by

biases. Crossover operators can have multiple biases. [ECS89] introduced the notion of the positional and distributional bias. A crossover method has positional bias if the probability of disrupting a schema depends on the position of a gene. In other words, if the defining length of a schema has influence on the survivability, a crossover operator has positional bias. This is particularly present for the One-Point crossover operator. An operator has distributional bias if the number of bits changed is non-uniform. Confusingly the Uniform crossover is an example of an operator with high distributional bias. The number of bits exchanged for each crossover operation is binomially distributed around a mean. The One-Point crossover is the exact opposite. The number of bits exchanged is uniformly distributed, meaning it is just as likely to exchange one bit as it is to exchange the whole chromosome. Both of those operators and a few more will be discussed shortly.
In his dissertation [Spe00] Spears considers population diversity an important part of disruption. This is of course a natural evolution of the idea, since recombination combines parts of a population. If a population is homogeneous a recombination operator's ability of exploration is limited. Throughout the generations of an evolutionary algorithm the population converges and grows more homogeneous meaning that in later stages recombination grows less and less disruptive. In [JS90] they argue along the same lines that so called crossover productivity diminishes with the the population growing more homogeneous.
Based on the idea of [Sys89] Spears considers not only disruptive but also constructive aspects of recombination. Construction is about crossover operators creating instances of new, higher order, schemata by recombination from parents with lower order schemata parts. It is generally found that the probability of construction depends on different schema properties depending on the crossover operator and the population homogeneity. There also seems to be some correlation between distributional bias and construction, since [Spe00] shows that Uniform Crossover is most constructive around its 0.5 parameter (so half of the chromosome is swapped) and N-Point crossover becomes more constructive with growing N. That is when both crossovers have the highest distributional bias. It is important to mention here that while higher disruption lead to higher construction during recombination, this is not the case for mutation operators [Spe00].
In [Spe00] Spears found that the survival of a schema does, on average, not depend on the type of recombination operator. Loss of survivability by disruption is always made up by the gain in construction. It is merely dependent on population homogeneity and of course order. He further confirmed that recombination is better at constructing higher order building blocks from lower order building blocks than mutation. "[...] the largest constructive advantage for recombination occurs when both lower-order building blocks have order roughly 1/2 of k (the order of H_k)"[Spe00] with H_k being the schema of order k. "This paints the picture that recombination will be most useful when high-fitness building blocks of relatively high order [...] can be combined into higher-order building blocks (H_k) that are also of high fitness."[Spe00]. Generally supporting the idea of the recombination operator being the most important tool when it comes to recombining building blocks into better solutions.
In [JS90] Spears and De Jong show that for a search space that is too complex for its

population size higher disruption leads to better results. Both [TS93] and [Rad92] made a similar statement about mutation and that for complex search spaces high mutation rates, and hence high disruption, are necessary for exploration. This suggests that it is indeed the disruption and not the construction of disruptive recombination operators that is relevant.

3.5.1 Recombination Strategies

We will now present and discuss the various crossover operators that were implemented during this thesis, their advantages and disadvantages and why only a hand full of operators were considered during parameter tuning.

One-Point and N-Point Crossover

Even though One-Point and N-Point crossover have already been discussed we will quickly summarise the most important details about the operators. One-Point crossover is one of the earliest operators, initially called Simple Crossover, already mentioned in [Hol75]. The idea is simple. Two chromosomes are split into two parts each at a randomly chosen point. Two new chromosomes are created by combining one part of one chromosome with the other part of the other chromosome. This operator has very high positional bias [ECS89]. Genes close to each other are more likely to be moved together. This means that it is necessary to model the solution representation in a way that puts genes closer together if they are related. Whether genes are related or not can sometimes be not very clear [ECS89]. In comparison One-Point crossover has very low distributional bias. The amount of genes exchanged ranges uniformly from one to all.
The logical next step is increasing the amounts of points at which the chromosome is split. 2-Point crossover and further N-Point crossover were an early addition to the pool of crossover operators in both the literature and this thesis. With even N the operator now takes out a chunk of the solution and transplants it into another chromosome. This is a lot less disruptive to schemata of high defining length than one-point crossover was. The positional bias for $N = 2$ is slightly lower, while the distributional bias remains unchanged. With higher N, however, both distributional and positional bias move away from One-Point crossover [ECS89].

While both operators were implemented during this thesis they are not particularly promising. Positional bias is of little use with the type of solution representation employed. It is not possible to know which orders are interacting when encoding the problem. This means that the position of genes is random and cannot make use of the positional bias introduced by the crossover operators. Given the complexity of our problem it also appears that high disruption is and necessary [JS90] and we also think that more sophisticated operators are in order.

Uniform Crossover

The Uniform Crossover operator has also been discussed in the introduction of this chapter. It has been presented by Syswerda in [Sys89] and has since been a staple when it comes to crossover operators. The concept is rather simple. For each gene in a chromosome it is decided separately whether it is part of the first or second new solution. This happens by chance and is controlled via a parameter. When this parameter is set to 0.5 the chance of each gene to be part of the first new chromosome is 0.5, meaning half of the genes are part of one chromosome and half of the other. This is also where the distributional bias of the operator is at its peak. The amount of data exchanged is binomially distributed. The positional bias is for every value of the parameter 0, since membership of one or the other chromosome is decided on a gene by gene basis. In [Sys89] it is argued that Uniform Crossover performs better than One- and 2-Point Crossover. They explain it both theoretically and present empirical evidence. The theoretical argument is that One- and Two-Point crossover performs better for schemata of shorter defining length and Uniform crossover has better results when it comes to schemata of higher defining length. Since there are a lot more schemata of higher defining lengths however, Syswerda argues that Uniform crossover is the better choice. The experiments done mostly support that claim. Another investigation by [ECS89] certainly supports that uniform crossover is generally a better choice than One- and Two-Point crossover, yet it is also regularly outperformed by N-Point crossovers with $N > 2$. Additionally, One- and Two-Point crossover both require a certain adherence to the Building Block Hypothesis to work well, given their positional bias. Uniform Crossover has no such restriction. The high disruption introduced by Uniform crossover certainly comes in handy when working with too small a population [BBM93a].

Uniform crossover is expected to be a better choice than N-Point crossover. Its high disruption and low positional bias seem like a good fit for the problem at hand. We think, however, that the algorithm needs more help with maintaining and recombining promising building blocks than what Uniform crossover can deliver. The operator will be considered during hyperparameter testing, yet is not expected to perform particularly well.

Period Uniform Crossover

Period Uniform Crossover is the first of three custom crossover operators introduced in this thesis. The idea is conceptually simple. Instead of orders, periods are recombined. This requires transformation of the solution representation into the representation suggested in section 3.1. For this, the solution is transformed from a list of orders to a list of sets of orders that correspond to the order assignment of each period. This makes it possible to recombine periods. Orders that are in periods together are interacting hence orders that fit well together could be considered building blocks. Recombination on the basis of periods seems a lot more natural for this problem then recombination based on arbitrary orders.

To recombine periods, the algorithm iterates through the number of periods and considers for each period the order sets of both parents. It then decides by chance which set is put into which child. This most likely introduces duplicate orders, so orders that are in two periods at the same time, while also discarding orders, not making them part of the solution at all. This is simply dealt with by removing any duplicate order from the period with higher cumulative demand and reintroducing missing orders. Missing orders are first sorted by demand and then, largest demand first, added to the period with the currently lowest cumulative demand.
Analysing the biases of this operator turns out to be fairly difficult. On the one hand because of the transformation that takes place, on the other, because of the random aspect of redistributing orders that have not been assigned. Given the transformation, this crossover operator has no positional bias when considering the initial solution representation. The distributional bias however is fairly high. Period Uniform crossover is very dependent on the other operators, since this crossover operator alone introduces relatively little randomness, because entire sets of orders are recombined. Orders that are together in a period are unlikely to be removed from each other during recombination. This suggests some kind of positional bias when looking at the transformed solution representation, even though the bias does not come from the position of the gene in the chromosome but from the position in the transformed chromosome which comes from the value of the gene in the initial solution representation. It is not positional bias in the traditional sense, especially since it does not depend on the defining length of the schema.

The operator is created with the idea of not disrupting building blocks, while reconfiguring those into better solutions. This disruption appears to be limited to schemata of order greater than $\frac{k}{n}$, so the average number of orders per period and to schemata that overlap periods. Additionally, there is some disruption of schemata smaller than this through the redistribution when orders are duplicated. This generally suggests that there is not a lot of exploratory power behind this operator in the typical sense, but some exploration similar to what mutation does.

Simple Period Crossover

Simple-Period crossover is another custom operator based on the Period-Uniform crossover operator. This operator also works on the transformed solution representation and recombines periods rather than orders. To improve the results the performance of each period is considered.
The process is similar to the process of Period-Uniform crossover. They simply differ when deciding which child receives its period from which parent. Where the Period-Uniform operator decided by chance, this operator considers the fitness of the periods. One child will receive all periods that have better fitness and the other child will receive all periods that remain. Subsequently the removal of duplicates and redistribution of missing orders is handled.

The biases are similar to the prior operator, yet the algorithm seems more directed, certainly introducing more biases and further removing exploration capabilities. In [VC99] this new bias is called *directional bias*. "This bias determines the direction the GA is likely to converge to. Directional bias benefits GA search as it pushes the GA towards fitter regions in the search space."[VC99]. It is difficult to judge whether this additional steering is too much restriction in terms of exploration, particularly for a problem and a search space that requires a lot of it.

Product Type Crossover

The third and last custom operator we introduced is Product-Type crossover. It too makes use of transforming the solution representation, yet in a different way than our other operators. This operator recombines orders based on their product type. During recombination a set of half the product types is chosen randomly. One individual gets all orders that are of the product types from the set from one parent and the remaining orders from the other parent. This requires no handling of duplicate or missing orders, yet it also means that all the orders of a product type are always moved together. This introduces another bias similar to the positional bias while not depending on the position in the chromosome but the product type that is initially assigned by the instance. This restriction is stronger than the period restriction earlier, since the period based crossovers introduced some randomness when duplicates appear and also move different sets of orders based on the disruption by the mutation operator. This operator always moves the same set of orders together regardless of what happens between recombination cycles. The distributional bias is fairly high since around half the product types are chosen.

This operator appears to be too strict and not exploratory enough. The never changing set of orders that is moved together seems to be problematic. The operator is, however, considered during parameter tuning.

3.5.2 Conclusion

During hyperparameter tuning the configuration will contain Uniform, Period-Uniform, Simple-Period and Product-Type crossover as choices for the crossover operator. One- and N-Point crossover have been decided to be disregarded during those experiments due to their nature and how they seem inappropriate for this solution representation. Additionally, we cannot make use of the positional bias with the problem encoding at hand. We expect Period-Uniform crossover to come out on top based on the natural design of the operator and the relatively high exploration when comparing it with the other custom operators.

3.6 Mutation

The mutation operator is the second operator responsible for exploration. This operator mutates individuals most of the time directly after recombination. This mutation introduces much needed randomness. How a solution is mutated can vary. How much of a chromosome is mutated depends on the mutation rate of the algorithm, another tunable parameter.
Mutation is often considered a secondary or background operator [Gre86] ([Jon85] via [BBM93a] and was initially called a security policy for reintroducing already lost alleles. The introduction of randomness that does not depend on other solutions in the population is particularly important when it comes to genetic drift and exploration towards the end of the search when the population has largely converged. Genetic drift happens when some gene value starts appearing more frequently than others by pure chance, which is then amplified by the mechanisms of the algorithm.
While recombination is seen as the "main force leading to a thorough search of the problem space"[BBM93a], crossover productivity starts declining towards the end of the search. One advantage of the crossover operator is that generally only alleles are introduced that have been tested to a degree, since they already are in the population. This is the reason for the diminishing crossover productivity. Once the population loses diversity, introducing new gene values through crossover becomes more difficult [Spe00]. This is not the case for mutation. While for high mutation rates this is equatable to random search, low mutation rates can be particularly helpful towards the end of the search, introducing new gene material to the pool. Or as [BBM93a] argues that [Con91] points out: "mutation becomes more productive, and crossover less productive, as the population converges".

Disruption and construction are important properties when it comes to search operators. [Spe00] analyses disruption and construction rates of both operators in depth. While mutation can be as disruptive as recombination and more, recombination introduces construction at the same rate as it disrupts, yet for mutation, construction decreases with growing disruption. Hence mutation has no role in constructing higher order building blocks but is the more viable exploration operator [Spe92]. This leads to an important interaction between the two operators. Recombination helps combining promising parts of different solutions while removing diversity from the population and mostly recombining what is already there. Mutation helps exploring the search space and slows down premature convergence while providing the recombination operator with new material. This lead Spears in [Spe92] to state: "It is not clear that the current distinction between crossover and mutation is necessary, or even desirable, although it may be convenient".

When choosing a mutation rate, it is important that it is high enough to provide enough disruption and exploration, yet it must also be low enough so that it does not overpower the selection pressure applied. Too much mutation and it degrades into random search, too little and premature convergence is a given.
In [Müh92] it is stated: "In most of the cases the optimal mutation rate is proportional to the length of the chromosome". Given the instance-based problem size our algorithm

faces, we adapted the parameter. Instead of a mutation rate, so a probability a gene is mutated, we opted to tune something we call the mutation dividend. This value is then divided by the length of the chromosome, in our case the number of orders, to provide the mutation rate. This makes it possible to tune the value for instances of varying size.

Last but not least it is necessary to mention that adaptive mutation behaviour is something that is frequently mentioned in the literature. This makes it possible to adapt the mutation rate across the duration of the search. Given the large amount of variables already part of this evaluation it was decided to not consider the use of adaptive mutation behaviour.

3.6.1 Mutation Strategies

In the following we present the different mutation strategies that have been implemented in this thesis.

Move Mutation

The Move mutation operator is a very simple operator similar to the early mutation operators that simply flipped a binary bit. This is the extension to larger alphabets but just as simple. The operator calculates the number of mutations based on the mutation probability and the length of the chromosome. Then the according amount of Move-neighbourhood moves is generated. For each move an order is chosen at random and for each order a new period is chosen randomly as well. This can be any possible period except the one the order is already assigned to.
Creeping mutation is a design choice that is regularly mentioned in the literature, this would allow only values close to the currently assigned gene value. While moving an order only to neighbouring periods would certainly be more gentle to f_3 the operator would lose a considerable amount of exploratory power. This is why it was decided against using creeping mutation.

Swap Mutation

Swap mutation is another rather simple operator. For this mutator the amount of mutations is calculated and correspondingly drawn from the Swap-Neighbourhood. For one such move two genes are selected at random and their values are swapped. If the genes have the same value, another gene is chosen. This always makes orders swap their periods. Here too, creeping mutation was considered but decided against.

Neighbourhood-Switching Mutation

Neighbourhood-Switching mutation combines the two operators just introduced. A tunable parameter controls the probability of choosing the move- or swap-neighbourhood. When mutating an individual it is first decided by chance whether this mutation is done by

moving or swapping genes. Afterwards the individual is then mutated as described above. This operator was inspired by [VLM20] and their chance-based choice of neighbourhood.

Violation-based Mutation

This operator is a bit more sophisticated than the previous two. With this strategy, genes that are violating a hard constraint, so are within a period that is violating one of the two capacity limits, are more likely to be mutated. The increase in likelihood is influenced by a parameter acting as a factor. When mutating a solution, each gene is checked for mutation based on the mutation rate configured. If an order is violating a constraint, the mutation probability is multiplied by the given factor. This leads to a higher likelihood of mutation when an order is in a violating period. The mutation applied is then either a move or swap mutation decided randomly based on a parameter like for the Neighbourhood-Switching mutation.

3.6.2 Conclusion

During hyperparameter tuning we will only consider the Violation-based mutator. Both Move and Swap mutation can be emulated by tuning the parameter of the Neighbourhood-Switching strategy if necessary and the Neighbourhood-Switching strategy can be emulated by changing the parameter of the Violation-based mutator to 1.

3.7 Local Search

Memetic algorithms distinct themselves by introducing local search to the concept of evolutionary algorithms. In [Mos89] they were first labelled as such and have since only grown in relevance. He presents algorithms that he deems "examples of what [he] calls memetic algorithms."[Mos89] and argues that they qualify because they are population based approaches that "combine a very fast heuristic to improve a solution with a recombination mechanism that creates new individuals"[Mos89]. While giving a rough idea, this also shows that memetic algorithms are not as strictly laid out as other types of algorithms. In his article he presents two approaches in ASPARAGOS [Müh89] and SAGA [HB91]. In [Müh89] the initial population is improved by local search and after applying the typical genetic operators the offspring is improved as well. Those improved individuals are then considered for the next generation. This application of the local search after applying the genetic operators is common [Poo09][WM10][NOK07]. There are also approaches that improve the entire population at the start of a generation so before applying the genetic operators [LC18]. While this difference might appear trivial since the operators are applied in a loop, there is a subtle change to which individuals are searched. When local searching the entire population while a crossover rate $< 100\%$ is configured, individuals that have already been improved are searched again. With crossover rates generally being less than 100% this happens frequently. While this is a small difference, it is worth mentioning.
For this thesis we decided to stick with the more common approach and apply the local

search after the genetic operators have been employed and only consider the offspring during local search. Based on [NOK07] it was decided to make the frequency of the local search and the amount of individuals that are improved tunable. It is possible to specify a cadence in terms of generations, i.e. a cadence of 5 means every 5 generations local search is performed, and also how much of the offspring is considered during search, while this always considers the stronger individuals first, i.e. a parameter of 10% configures the algorithm to search the top 10% of the offspring. Additionally, the amount of iterations per local search can be specified.

Lastly, the type of local search needs to be decided. During the development of this thesis multiple local search options have been implemented and will now be presented and discussed. We will further argue which options are considered during parameter tuning.

3.7.1 Simple Local Search

Move- and Swap-Neighbourhood Local Search

Similar to the mutation operator, a local search operator employing the move- and swap-neighbourhood was implemented. With a very simple hill-climb algorithm it checks whether the randomly chosen move from the neighbourhood improves the current solution. If that is the case, the move is applied and another iteration is started from this new, improved solution. Once no move in the neighbourhood can improve the solution or the amount of iterations specified is reached, the search stops. For the same reasons as in mutation, it was decided to consolidate the two neighbourhoods into one strategy with a probability of choosing one or the other. This probability can be set as parameter. The chosen neighbourhood can change between iterations. While being simple, this operator has no intentional bias towards any parts of the fitness function. This makes it possible to explore the neighbourhood without considering problem specific knowledge, introducing random exploration similar to the mutation operator, while exploiting good solutions as starting point.

3.7.2 Problem Specific Local Search

A set of more problem specific local search approaches have been implemented as well, each catering to a certain part of the fitness function.

Priority-Inversion-Fixing Local Search

The Priority-Inversion-Fixing local search operator addresses the concepts assessed by f_3. The local search operator tries to remove priority inversions, i.e. orders that are planned before other orders with higher priorities.
Given the maximum priority and the number of periods of the instance, one can calculate the average priority per period p_{avg}. Dividing the priority value of an order by this

newly calculated p_{avg} yields a period reference for each order, i.e. a rough estimate for the period this order should be in to create as little priority inversions as possible. Subtracting this value from the period the order is currently planned in shows how far away the order is from its reference period. The bigger the difference, the more likely it introduces a lot of priority inversions. The heuristic now iterates through the orders, starting with the order with the biggest deviation. Next it creates a sorted list of the periods it should rather be in based on this reference value. Creating this list is more tricky than it first appears, e.g. if an order is planned for period 6 and its reference value states 2.3, the algorithm must consider 2 first, then 3, 1, 4, 5, 6, ... This can be easily handled by taking a list of period numbers, subtracting the optimal period value, taking the absolute and ranking those periods by their value. The periods are then considered, one after the other. For each period a swap with all of the orders in this period is considered and rejected if the constraint violations increase or the f_3-value is not actually improved. If no improvement can be found for this order and period combination, the next period is chosen and the procedure repeats until a move is found or no periods are left. Subsequently the next order is considered until there are no iterations or orders left.
This clearly biases the search towards fixing f_3. During development it was found that generally when no hard constraint violations remained the biggest and certainly easiest to fix fitness cost came from f_3. While this is certainly no search that should be used on its own, it can be a helpful tool to improve a solution.

Product-Type-Hard-Constraint-Fixing Local Search

In a similar vein the Product-Type-Hard-Constraint-Fixing local search addresses f_2 and the corresponding hard constraint violations. The algorithm first checks which product types are currently penalised because of hard constraint violations. It then chooses one of these product types and uses a modified move and swap neighbourhood in which only orders are considered that are of the chosen product type. If no product type is violated, all product types are considered. The neighbourhood is chosen at random. The algorithm then looks for a move that improves f_2 and does not introduce any new hard constraint violations. If no such move is found it considers another product type. If such a move is found, it restarts at choosing a product type to fix until no iteration is left. The bias towards repairing f_2 is clear. Since the biggest part of the hard constraint violation can come from the product type capacity violations and balancing the product type levels helps in balancing the overall levels, focusing on that part of the problem seemed apparent. This search, however, really only focuses on that part of the problem and might be best combined with other searches.

Behaviour-Switching Local Search

The Behaviour-Switching local search operator combines all of the earlier introduced operators. If a solution has product type hard constraint violations, the Product-Type-Hard-Constraint-Fixing local search operator is applied. If the normalised f_3-value g_3 is

the largest part of the current fitness, the Priority-Inversion-Fixing operator is applied. If neither of those are the case, the simple local search switching randomly between Move- and Swap-neighbourhood is applied. If one of the cases applies, all of the iterations are done using this local search.

3.7.3 Conclusion

During hyperparameter tuning the two consolidated operators and the Priority-Inversion-Fixing local search will be considered. The simpler neighbourhood switching operator might be able to introduce more new, unseen solutions that have not been considered by more sophisticated operators. Particularly when other operators of the algorithm employ problem specific knowledge. This might help with exploration while exploiting already good solutions. This in turn means, however, that the search is a lot less directed and probably slower in improving the solutions. The Behaviour-Switching local search, employing both of the problem specific and the simple search algorithms, has a very clear bias towards solutions that are in more problem specific neighbourhoods. Since while product type capacity violations exist only the product-type-fixing operator is applied, the other aspects might not be considered at all, particularly when it is not possible to resolve the violations. Last but not least, it is certainly worth considering the priority-inversion-fixing operator. While it only focuses on one aspect of the function, there are usually a lot of priority inversions to fix. If the simpler local search is being chosen during parameter tuning, we expect a higher amount of iterations to be tuned as well than would be necessary for the other local searches.

3.8 Replacement Strategy

The replacement strategy of a memetic algorithm decides which Individuals are taken to the next Generation. Often, when talking about genetic algorithms, the algorithm is generational, meaning that all individuals of the earlier generation are replaced. There is the notion of elitism, which would ensure that the best solution appears in the next generation at least once. [SV98] talks about different Replacement strategies in steady state genetic algorithms. It presents Replace Worst, Replace Random, Elitist Replace Random, Kill Tournament, Elitist Kill Tournament and Conservative Replacement and discusses their takeover times. For Kill Tournament, much like the related Tournament Selection operator, a tournament is held, eliminating the worst contender. This tournament is repeated until the population is small enough. Elitist Recombination is another interesting idea. There, a tournament is held between the parents and the offspring and the two fittest individuals of the family are chosen [Thi97].
In this thesis Replace Worst, Kill Tournament and Generational Replacement are implemented and discussed. When investigating the effects of the various crossover-operators it became clear that sometimes crossover operators can produce a very good individual and one bad candidate that is worse than the parents. This could of course be an opportunity for exploration, however it also poses the question whether it might be better to consider

the parents as well for the following generation. During hyperparameter tuning Kill Tournament and Generational Replacement are considered. Replace Worst can simply be emulated by Kill Tournament given a large enough tournament size. Kill Tournament has great potential selection pressure which might help steer the algorithm. When not tuned properly, however, it might hinder the much needed exploration efforts the problem requires.

CHAPTER 4

Experimental Evaluation

In this chapter we discuss the experiments done in this thesis. First we talk about the problem instances used. Next we quickly discuss the simulated annealing approach from [VLM20] and how and why it was implemented again here. We then present the settings used for our algorithms that have been found through parameter tuning. Afterwards we compare and talk through the results. All experiments were run on an *Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz* and 16GB DDR4 RAM.

4.1 Problem Instances

This thesis aims to make a direct comparison with the results of [VLM20] possible. This requires our tests to run on the same test-instances. Those test-instances have been provided. Additional training data is of course necessary for the hyperparameter tuning step. In [VLM20] they were provided with 27 realistic instances by their industrial partner which entirely went into the testing set. They further described a way of creating additional instances based on those 27 realistic examples. In this thesis we recreated the instance generator to generate 2000 instances of training data.

4.1.1 Instance Generator

The instance generator presented in [VLM20] was implemented to generate our own training instances. We created both perfect and purely random instances. During implementation we came across a few minor problems that we tried to clear up. Firstly we want to mention the integer partitioning algorithm. There are conditions for the input parameters that have to be met for the algorithm to produce the right results. In the first line of the algorithm they specify:

$array \leftarrow$ an array consisting of $n - (k \cdot minV)$ zeros;

This of course requires $n - (k \cdot minV)$ to be at least 0. The integer partitioning happens several times during the creation of a perfect instance and just once for purely random instances. When creating a perfect instance, so an instance whose fitness is known to be able to be zero, it happens once in the very beginning using the input parameters. The same is true for the one occurrence when creating random instances. For those calls we can control the input and restrict the provided values (number of orders, number of periods and number of products) to only those that allow the array to be of correct size. During the creation of a perfect instance, however, the function is called multiple times, usually based on other integer partitions. This means that we cannot ensure that the input parameters conform with the necessary restrictions. The result of an integer partitioning algorithm has two very easy to check properties: It must be split into the specified amount of partitions and its sum must equal the integer that needed partitioning. When creating instances we regularly check whether these two requirements hold. If this is not the case we scrap the instance creation and try again.
When creating a perfect instance we further check whether the fitness actually equals zero and scrap the instance if this is not the case. This rarely happens, yet can happen due to floating point errors.

With those considerations we created 1000 perfectly solvable and 1000 purely random instances. In table 4.1 we compare the minimum, maximum, mean and standard deviation of the parameters of all the relevant sets. R_1 is the set of 27 realistic instances provided by the industrial partner through [VLM20]. R_2 is the set of 1000 perfectly solvable training instances created during development of this thesis. R_2^* consists of 50 perfectly solvable instances created by [VLM20]. R_3 is a set of 1000 purely random instances created for this thesis. R_3^* is made up of 50 purely random instances created in [VLM20]. R_4 is a set of 10 randomly generated small instances created by Vass et al. in [VLM20]. Instance sets marked with * are sets that are part of the test set. It is interesting to note that even though the sets R_2 and R_2^*, and R_3 and R_3^* are created by the same algorithm their distributions do not align. Particularly the difference in mean value for R_2 and R_2^* are notable.

As done in [VLM20] we want to show the distribution of our generated instances. Figure 4.1 shows the scatter plot of our 1000 perfect instances in R_2. Supported by the histograms of each property on its own in Figure 4.2 we can say that the instances are fairly well distributed. The number of instances with low orders is distinctly low and the number of products shows a very clear dip around 9 that we cannot properly explain. It most certainly comes from the restrictions on the input parameters.

The scatter plot of our random instances is similar to the results of the perfect instances. The histograms also mostly align. A key difference is the amount of low order instances, where we can see no distinct dip in occurrences for our purely random instances. The peculiarity of about 9 products is present too. Number of products is, however, one of the controlled parameters that are checked for validity due to the integer partitioning algorithm. That the dip exists for both types of instances would suggest that it indeed

Parameter	Instance Set	min	max	mean	std
k	R_1^* [VLM20]	79	1585	307.19	412.56
	R_2	119	3994	2179.766	1049.15053
	R_2^* [VLM20]	105	3896	1595.86	954.08971
	R_3	101	3997	2008.894	1100.41202
	R_3^* [VLM20]	112	3991	2076.76	1207.01978
	R_4^* [VLM20]	34	98	61.20	19.70
m	R_1^* [VLM20]	4	8	6.93	1.24
	R_2	1	19	9.672	5.43874
	R_2^* [VLM20]	1	19	8.82	5.50209
	R_3	1	19	9.833	5.4331
	R_3^* [VLM20]	1	19	9.04	5.47707
	R_4^* [VLM20]	1	4	2.80	1.03
n	R_1^* [VLM20]	20	20	20.00	0.00
	R_2	2	79	39.078	22.25612
	R_2^* [VLM20]	4	78	39.5	22.48287
	R_3	2	80	39.87	22.67093
	R_3^* [VLM20]	4	77	39.26	22.03782
	R_4^* [VLM20]	7	18	10.90	4.04

Table 4.1: This table shows the minimum, maximum, mean and standard deviation of the three parameters used when creating instances. Instance Sets marked with * are part of the test-set and taken from [VLM20].

comes from the controlling of the parameters to allow for a valid partitioning.

In summary we created 1000 perfectly solvable and 1000 purely random instances with the algorithm presented in [VLM20]. Those 2000 instances make up our training data. Another 50 perfectly solvable and 50 purely random instances have been generated by [VLM20] and are, next to the 27 realistic and 10 small instances also provided by [VLM20], our test data.

4.2 Implementation

The entirety of the project has been implemented in Python 3.10.0 with numpy 1.22.3 as one of the core libraries. During development a lot of optimisation techniques have been tried. We considered using numba, PyJion, pypy and cython. With PyJion the amount of extra work that was needed made us decide to discard it as an option. PyPy turned out to be a lot slower since our algorithm made use of lots of numpy operations which appear to not work as well in combination with PyPy as compared to native python. Numba did not work for us since it apparently had issues with the object oriented design of our code.

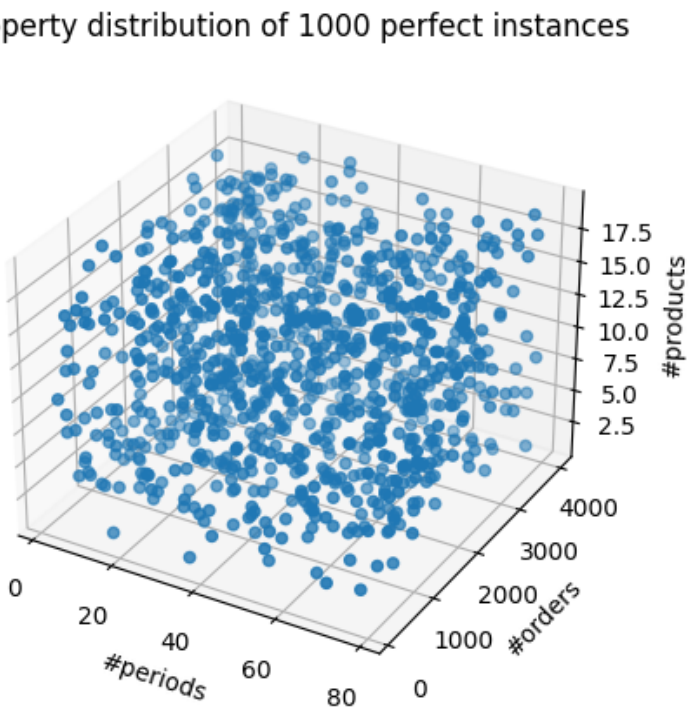


Figure 4.1: This scatter plot shows the distribution of R_2 across their properties: number of orders, number of periods and number of products.

It was opted to focus on improving the performance by using numpy and just generally better designed code.
In [VLM20] the results were achieved using C# which makes a fair comparison difficult anyways. To allow for a comparison nevertheless, we decided to implement the proposed simulated annealing algorithm on the basis of our PLP-framework, using the same solution representation, fitness function implementation etc..

4.3 Hyperparameter Tuning

In the following we discuss the configurations, setup and results of our hyperparameter tuning runs. With the different framework and hardware we decided to tune the parameters of the simulated annealing implementation as well as our memetic and genetic algorithm. All three algorithms were tuned for a runtime of 5 minutes with runtimes exceeding those by more than 10% considered illegal. The tuning was done using SMAC3 for a fixed amount of trials.

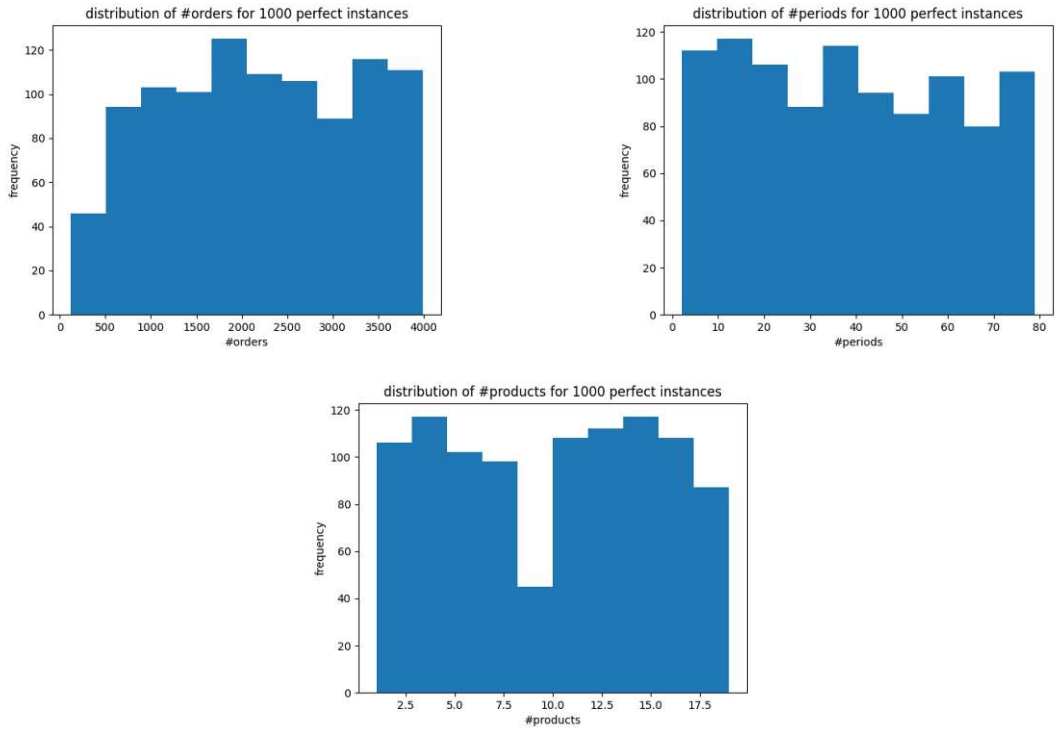


Figure 4.2: The histograms show how the number of different properties of R_2 are distributed.

4.3.1 Simulated Annealing

To find whether the results of the Vass et al. thesis [VLM20] could be replicated simply by increasing the runtime the configuration was applied to test runs with runtimes of 300, 600 seconds. Some results for 300 and 600 seconds are documented in Table 4.3. After finding that higher runtimes on a small subset did not improve the performance dramatically they were not further tested and are therefore not documented. It was then decided to re-tune the parameters, leading to better results. To tune the simulated annealing algorithm a similar setup like in [VLM20] was used. The parameters *initial temperature* and *iterations per temperature* were tuned. It was decided against tuning the *move percentage* and *cooling rate* for multiple reasons. The *cooling rate* was shown to have a similar slope at different rates with tuned *iterations per temperature*. This means that tuning the iterations with a set cooling rate of 0.95 should be sufficient and is what was done in [VLM20]. The *move percentage* has also been tuned in [VLM20] and found to be best at 40%, meaning that the neighbourhood is a move-neighbourhood 40% of the time and a swap-neighbourhood 60% of the time. While the *initial temperature* and *iterations per temperature* values strongly depend on the runtime of the algorithm and the amount of iterations that get done during this runtime, tuning them in this new setup seems necessary. Tuning the *move percentage*, however, appears to be independent

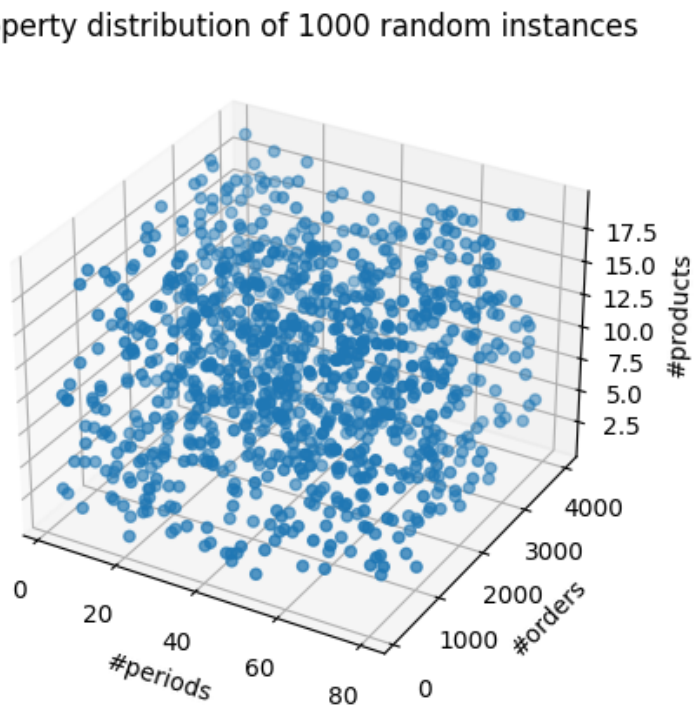


Figure 4.3: This scatter plot shows the distribution of R_3 across their properties: number of orders, number of periods and number of products.

of the setup and will be taken as is from the earlier thesis.
The tuning limits were set to be the same as in [VLM20]. The limits and results can be seen in Table 4.2

Parameter	Type	Min	Max	Tuning
Iterations per temperature	integer	10^3	10^6	19078
Initial Temperature	real	0.1	10	0.389

Table 4.2: The tuning limits and results for tuning the simulated annealing algorithm on 2000 instances with 10.000 trials using SMAC3

The parameters were tuned for 10.000 trials on 2000 training instances generated as has been described in section 4.1. While the *initial temperature* is with 0.389 higher than the 0.22 found in [VLM20], the *iterations per temperature* are with 19078 considerably lower than the 252.000 found in [VLM20]. This can be explained by the slower system. Since less iterations are available, the algorithm has to make more daring moves leading to a higher initial temperature. The lower amount of iterations per temperature is a clear

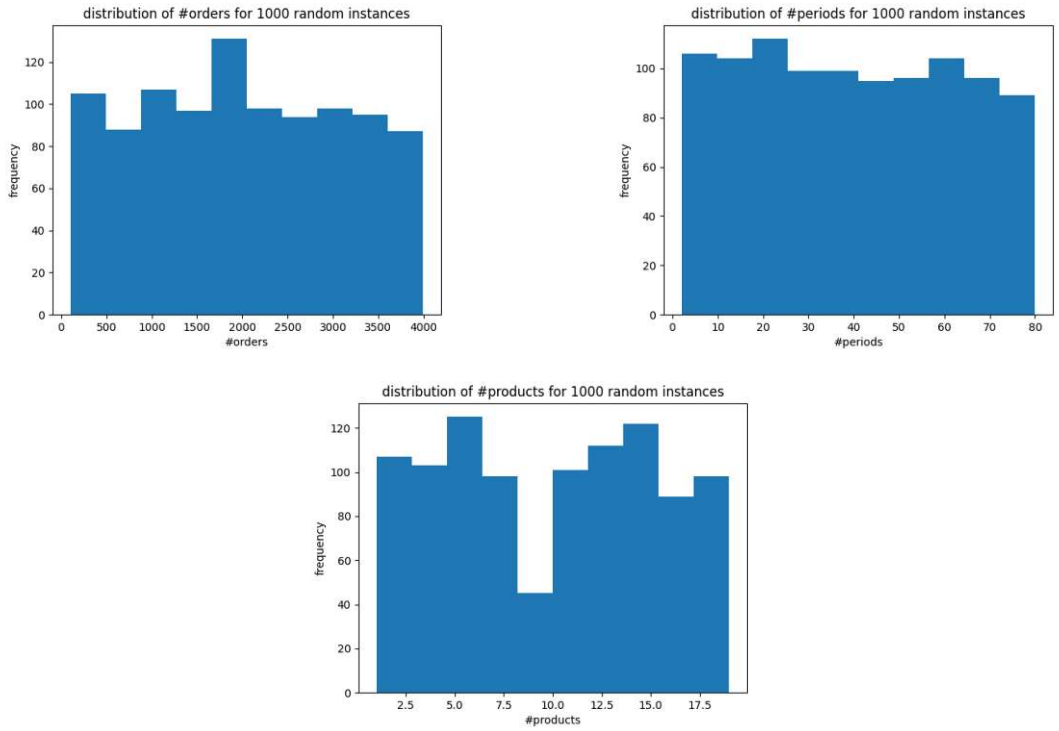


Figure 4.4: The histograms show how the number of different properties of R_3 are distributed.

consequence of that.
In Table 4.3 a few fitness values are shown to display the results achieved by the different tunings. Column one specifies the instance name, column two lists the results reported by [VLM20]. Column three and four show the results achieved by using the tunings reported in [VLM20] with 5 minutes and 10 minutes runtime on the hardware used during this thesis. The last column shows the results achieved by using the parameters found during hyperparameter tuning in this thesis.

While the results reported by [VLM20] still outperform all the simulated annealing results achieved by our algorithm, our own parameters perform significantly better than the parameters from [VLM20]. Doubling the runtime of the configuration taken from [VLM20] increases the results of the tuning somewhat, while generally still not beating the results of our own tuning. When ranking the results and averaging this rank we find that in total, across all testing instances, the tuning found during this thesis ranks 1.52, while the configuration found in [VLM20] only ranks 2.3 for the 5 minute runtime and 2.06 for the 10 minute runtime. Table **??** shows the average ranks split between testing subsets. While most results are self explanatory, it should be mentioned that on the set of perfectly solvable instances R_2^* both tunings of Vass et al. perform slightly better

instance name	[VLM20]	[VLM20] tune 5min	10min	own tune
realistic_instance_01	1.0203	1.19221	1.20298	1.13472
realistic_instance_02	1.1734	1.34517	1.33427	1.26852
realistic_instance_03	0.9894	1.07747	1.15742	1.08214
realistic_instance_04	1.0128	2.21124	1.17736	1.09785
realistic_instance_05	1.1575	1.23212	1.29193	1.21835
randomly_perfect_random_0951	0.0467	0.09296	0.10095	0.09442
randomly_perfect_random_0952	0.0391	1.03104	2.03136	1.02858
randomly_perfect_random_0953	0.0227	0.1368	0.13498	0.0975
randomly_perfect_random_0954	0.0278	0.03079	1.0361	2.0378
randomly_perfect_random_0955	0.005	0.00574	0.00556	0.00573
randomly_generated_0951	0.3754	3.50871	3.62064	0.54686
randomly_generated_0952	1.9065	9.31275	7.21181	6.10842
randomly_generated_0953	0.2255	1.32146	0.39841	0.36389
randomly_generated_0954	0.127	1.2198	1.20938	1.21767
randomly_generated_0955	0.0322	1.1404	1.1356	1.13086
randomly_generated_small_0001	0.0303	0.1297	0.13235	0.03378
randomly_generated_small_0002	0.4499	1.5385	2.59995	0.4685
randomly_generated_small_0003	0.2715	0.33422	0.33735	0.27422
randomly_generated_small_0004	0.3541	0.43878	0.407	0.35464
randomly_generated_small_0005	0.79	1.82128	0.82337	0.80014

Table 4.3: This table shows the median fitness of three trials on five instances per testing subset by different tunings and the results reported in [VLM20]

than the results of our simulated annealing configuration. While the tuning found in this thesis ranks first only 12 times, which is less often than the other two candidates with 21 and 23 for the 5 and 10 minute configurations respectively, it ranks second on 24 out of 50 instances as compared to 13 and 10. Also it is last least often with only 14 third places versus 16 and 17. This suggests that the tuning found in [VLM20] indeed works better on longer runtimes.
While the comparison with a 10 minute run is not particularly fair, the tuning found still beats it most of the time showing that it works well for this setup.

4.3.2 Memetic Algorithm

Tuning the memetic algorithm is a much bigger endeavour. During this experiment multiple tuning runs with restrictions were executed that have been adjusted run after run. The first hyperparameter tuning run of 4000 trials on 2000 instances was started with a large amount of possible parameters and generous limits. Table 4.5 lists the 16 parameters that have been tuned. Four of those were only relevant if certain parameters were active.

	average rank		
Instance Set	own tune 5 minutes	[VLM20] 5 minutes	[VLM20] 10 minutes
R_1^* *[VLM20]*	1.07407	2.70370	2.22222
R_2^* *[VLM20]*	2.04	1.9	1.88
R_3^* *[VLM20]*	1.4	2.38	2.22
R_4^* *[VLM20]*	1	2.6	2.4

Table 4.4: The average rank per subset of the simulated annealing algorithm tuned in this chapter and the configuration found in [VLM20] running 5 and 10 minutes.

parameter	**condition**	**values**
Population Size		100 .. 500
By Demand Next Fit Ratio		0. .. 1.
Tournament Size		1 .. 10
Crossover Behaviour		Period-Uniform Crossover, Uniform Crossover, Simple-Period Crossover, Product-Type Crossover
Crossover Rate		0. .. 1.
Mutation Move %		0. .. 1.
Violating Probability Factor		0. .. 10.
Mutation Dividend		0.01 .. 10.
LS Behaviour		Behaviour-Switching, Priority-Inversion-Fixing, Neighbourhood-Switching
LS Move %	Neighbourhood-Switching, Behaviour-Switching	0. .. 1.
LS Iterations		0 .. 200
LS Cadence		0 .. 200
LS %		0. .. 1.
LS Inversion Fixing %	Behaviour-Switching	0. .. 1.
Replacement Strategy		Generational, Kill-Tournament
Kill-Tournament Size	Kill-Tournament	1 .. 10

Table 4.5: This table lists the parameters and possible values and conditions of each parameter for the first tune with 4000 trials.

In the following each parameter is briefly described:

- **Population Size**
  Controls the size of the population

- **By Demand Next Fit Ratio**
  The ratio of individuals of the population that are created by using the By-Demand-Next-Fit construction heuristic. The remaining individuals are constructed using the First-Fit heuristic.

- **Tournament Size**
  The tournament size used during the Tournament Selection process.

- **Crossover Behaviour**
  The crossover behaviour chosen.

- **Crossover Rate**
  The crossover rate specifies how many of the selected individuals are chosen for recombination and in turn how many newly created individuals are considered for the next generation.

- **Mutation Move %**
  During mutation it is frequently chosen between the move- and swap-neighbourhood. This value specifies how often the move-neighbourhood is chosen, the rest of the times the swap-neighbourhood is used. (e.g. 0.4 means 40% of the time move and 60% of the time swap is used)

- **Violating Probability Factor**
  During mutation when a gene is considered to be mutated, the probability of mutation is multiplied with this factor if the order is part of a violating period or period-product-type combination.

- **Mutation Dividend**
  The amount of genes that are supposed to be mutated in an individual. Usually mutation rate is used for the probability of mutating a gene. Since chromosome length is variable with this problem encoding, longer chromosomes would be mutated more often. To combat this the mutation dividend value specifies the amount of genes that are supposed to be mutated. This value is then divided by the chromosome length, yielding the mutation rate and probability of mutation per gene.

- **Local Search (LS) Behaviour**
  The local search behaviour used during the local search step.

- **LS Move %**
  Similarly to the Mutation Move % this parameter specifies how likely it is that the move-neighbourhood is chosen over the swap-neighbourhood. Only two of three local search behaviours use those neighbourhoods, therefore this parameter is only active if either the Neighbourhood-Switching or Behaviour-Switching LS behaviour is active.

- **LS Iterations**
  This parameter specifies the amount of iterations done on each individual that is searched.
- **LS Cadence**
  The local search cadence specifies how often individuals are local searched. A cadence of 0 turns off local search, a cadence of 1 applies local search on every generation, a higher cadence makes local search rarer.
- **LS %**
  This parameter specifies the fraction of the population that is searched. The population is first sorted by fitness. E.g. if the parameter is set to 0.1, the top 10 percent of the population are local searched.
- **LS Inversion Fixing %**
  When using the Behaviour-Switching local search Behaviour, part of the local search is a heuristic that fixes priority inversions. How likely this local search is used is specified by this parameter.
- **Replacement Strategy**
  This parameter specifies the replacement Strategy used.
- **Kill-Tournament Size**
  If the Kill-Tournament replacement strategy is active, the size of the tournament is controlled by this parameter.

In this first tuning run the parameters shown in table 4.6 have been found to be the best configuration after 4.000 trials on 2000 problem instances. Many of the parameters are not only unexpected, they are also suboptimal at best. Most of the parameters like By-Demand-Next-Fit Ratio, population size and crossover behaviour are unsurprising. What is surprising, however, is that the found configuration turns off local search entirely by setting the cadence to 0. Additionally, the Kill-Tournament replacement strategy was chosen with a very large tournament size of 10, introducing a large amount of selection pressure. The crossover rate and hence the amount of recombined, mutated and searched individuals is very low with about 5 percent. This makes the choice of mutation and crossover parameters almost irrelevant.
With those results it is clear that the tuning configuration has to be reconsidered and has been too loosely defined for the amount of trials to achieve useful results. Hence the configuration was strongly restricted. Since local search was turned off entirely, considering a purely genetic algorithm configuration is strongly suggested. To give both approaches a fair chance, we will first consider tuning configurations that force local search to be applied by adjusting the limits and then tune a purely genetic algorithm by turning off local search entirely. In this subsection we will further discuss different memetic algorithm tunings. In subsection 4.3.3 we will focus on the tuning of a genetic algorithm. To allow further restrictions and restarting of the tuning process the amount

of trials was reduced to 1000 and the amount of instances to 200 with 100 instances of the randomly generated set R_2^* and 100 instances of the perfectly solvable randomly generated set R_2^*.

parameter	**tuned value**
Population Size	377
By Demand Next Fit Ratio	0.8917
Tournament Size	3
Crossover Behaviour	ProductTypeCrossover
Crossover Rate	0.0534
Mutation Move %	0.8152
Violating Probability Factor	0.2836
Mutation Dividend	7.9143
LS Behaviour	BehaviourSwitching
LS Move %	0.9981
LS Iterations	-
LS Cadence	0
LS %	-
LS Inversion Fixing %	-
Replacement Strategy	KillTournament
Kill-Tournament Size	10

Table 4.6: This table shows the results from the first parameter tuning run described in 4.5. Since the LS cadence was set to 0, the other parameters were not further considered.

For the second tuning run a few changes were made. The parameters can be found in table 4.7. Firstly it was made impossible to turn off local search. It is still an option to increase the local search cadence to a degree that makes local search rare and reduce iterations so that the local search step is short. This decision was made to make sure that local search takes place and is considered fairly during the tuning process. Since the Kill-Tournament replacement strategy has a strong effect on selection pressure, therefore generally lowering the crossover rate in return, which disregards a lot of the other operators and parameters, the replacement strategy was set to *Generational*. This ensures that the tune produces a more traditional memetic algorithm. Additionally, the crossover rate minimum was raised to 0.4.
This new tuning setup displayed in 4.8 was found to perform better on every single instance tested. This time the By-Demand-Next-Fit ratio prefers the First-Fit construction heuristic. The crossover rate for the Period-Uniform crossover operator is on the lower end with around 43%. Population size is rather high, which is expected. The mutation dividend is fairly low and about where a typical mutation rate would be expected with a high factor for violating genes of about 8. The most notable part of the configuration is that local search is avoided again. While the Neighbourhood-Switching local search is already a fast operator with little complexity and the most moderate of the options, its cadence was set to 84 generations, its iterations are set to 1 and the percentage of

individuals that is searched is found to be around 1%, making local search rare and brief. The complete disregard for the local search operator comes as a surprise. While we cannot explain that phenomenon the first thought was that maybe diversity was reduced notably leading to less gene material and hence hurting the performance of the algorithm. After a short investigation into that concern it was found that this has not been the case. The operator does the opposite. Not only does the average fitness of the population improve, the allele coverage does too and population equality is reduced as well. All in all local search seems to be a good addition. Another reason could be the way the tuning process works. As mentioned earlier, trials that exceed the 5 minute runtime limit by more than 10% are considered illegal. A high amount of iterations, low cadence and a high percentage will inevitably lead to exceeding this runtime limit. Many of the configurations that use local search will therefore be considered illegal. This might lead to the tuner considering local search to be unviable.

parameter	condition	values
population size		100 .. 500
By Demand Next Fit ratio		0. .. 1.
tournament size		1 .. 10
Crossover Behaviour		PeriodUniformCrossover, UniformCrossover, SimplePeriodCrossover, ProductTypeCrossover
crossover rate		0.4 .. 1.
mutation move %		0. .. 1.
violating probability factor		0. .. 10.
mutation dividend		0.01 .. 10.
LS Behaviour		BehaviourSwitching, PriorityInversionFixing, NeighbourhoodSwitching
LS move %	NeighbourhoodSwitching, BehaviourSwitching	0. .. 1.
LS iterations		1 .. 200
LS cadence		1 .. 200
LS %		0. .. 1.
LS Inversion Fixing %	BehaviourSwitching	0. .. 1.

Table 4.7: This table lists the parameters, conditions and values possible for the second memetic algorithm parameter tuning run.

To investigate this second theory another tuning run was done reducing the upper limits of the local search parameters to reduce the amounts of illegal solutions and just generally removing a few more parameters. This is to check whether it was illegal solutions and how the tuning process works, or the general unviability of the local search of this implementation.

parameter	tuned value
Population Size	296
By Demand Next Fit Ratio	0.31856
Tournament Size	7
Crossover Behaviour	PeriodUniformCrossover
Crossover Rate	0.4376
Mutation Move %	0.3203
Violating Probability Factor	8.1672
Mutation Dividend	1.0905
LS Behaviour	NeighbourhoodSwitching
LS Move %	0.6019
LS Iterations	1
LS Cadence	84
LS %	0.0121
LS Inversion Fixing %	-

Table 4.8: This table displays the results found during the tuning of the parameters described in 4.7. With the Neighbourhood-Switching LS Behaviour, no LS-Inversion-Fixing % is present.

The new tuning scenario and best found configuration can be found in table 4.9 and 4.10 respectively.

This configuration comes a lot closer to the expected values of each local search parameter. Four iterations per individual is not a lot but with a cadence of 18 generations frequent, especially since the algorithm tends to take several hundred generations for most instances. The local search percentage is with 16% fairly low but otherwise unremarkable. The remaining configuration is not particularly special. The By-Demand-Next-Fit-Ratio is with not even 10% a lot lower than expected.
When testing this configuration it was found that it improves on the first configuration found but does not achieve the results the previous configuration did. A sample of the performance of the three tested configurations is shown in Table 4.11.
This finding suggests that it is indeed the implementation of local search that is at fault. Forcing more local search does not improve the results. While we do not know why this is the case, it might be that while the local search operator requires quite a bit of resources, the improvements are small and local, also strongly affected by the epistasis at hand. Those small improvements, while expensive, are than mostly removed by the remaining operators. While the local search operator does in fact improve solutions, the time needed for the improvement gained might be used better elsewhere. Finally in subsection 4.3.3 we investigate whether turning off local search entirely leads to better results.

parameter	condition	values
population size		150 .. 300
By Demand Next Fit ratio		0. .. 1.
tournament size		1 .. 10
Crossover Behaviour		PeriodUniformCrossover, UniformCrossover, SimplePeriodCrossover, ProductTypeCrossover
crossover rate		0.4 .. 0.8
mutation move %		0.4
violating probability factor		0. .. 10.
mutation dividend		0.5 .. 10.
LS Behaviour		BehaviourSwitching, PriorityInversionFixing, NeighbourhoodSwitching
LS move %	NeighbourhoodSwitching, BehaviourSwitching	0.4
LS iterations		1 .. 50
LS cadence		1 .. 50
LS %		0.1 .. 1.
LS Inversion Fixing %	BehaviourSwitching	0. .. 1.

Table 4.9: This table shows the further restricted MA tuning configuration. Local search iterations, percentage and cadence were further restricted to enable regular use and less illegal configurations. Additionally, the population size, crossover rate and mutation dividend values were restricted to a smaller range. The two move percentage values for mutation and local search have also been fixed to 0.4 based on the results from [VLM20].

4.3.3 Genetic Algorithm

Based on the experiments from the previous section, a separate tune of a purely genetic algorithm is sensible. The parameters that are tuned, their value ranges and the tuning results can be found in table 4.12. The population size is notably smaller than with the configuration found for the memetic algorithm. The tournament size is with 8 fairly high. Product-Type crossover was a viable candidate from the start and is not a particularly surprising choice. The crossover rate is with about 55% a bit lower than expected, yet still within a reasonable range. A surprising parameter choice is the very low By-Demand-Next-Fit-Ratio with only about 5%, clearly preferring the First-Fit construction heuristic. Lastly, when looking at mutation, an interesting picture is painted, with a fairly low mutation dividend of only about 0.7 and a higher violating probability factor of 2.9. This suggests that the mutation operator tries to keep non-violating genes in place while moving orders that are violating the hard constraints. The move-% parameter for the mutation operator is with 55% about where it was expected to be.

parameter	tuned value
Population Size	177
By Demand Next Fit Ratio	0.0971
Tournament Size	6
Crossover Behaviour	ProductTypeCrossoverBehaviour
Crossover Rate	0.42137
Mutation Move %	0.4
Violating Probability Factor	3.59097
Mutation Dividend	3.6775
LS Behaviour	NeighbourhoodSwitching
LS Move %	0.4
LS Iterations	4
LS Cadence	18
LS %	0.16782
LS Inversion Fixing %	-

Table 4.10: The table shows the configuration found to be best for the tuning configuration in table 4.9

.

	fitness median		
instance name	**initial tune**	**second tune**	**third tune**
realistic_instance_01	1.17423	1.04929	1.11176
realistic_instance_02	1.27884	1.18529	1.23207
realistic_instance_03	1.1514	1.00376	1.07758
realistic_instance_04	1.1928	1.0231	1.10796
realistic_instance_05	1.297	1.18685	1.24192

Table 4.11: This table shows a sample of the three tunings described in section 4.3.2. While it is only a sample of 5 instances, the results behave similarly on every instance tested. The initial tuning has the worst results across the board, always outperformed by tuning three while the second tuning always achieves the best results with a wide margin.

4.4 Construction Heuristic Experiments

In section 3.3 it was claimed that the First-Fit and By-Demand-Next-Fit construction heuristics both produce good results when it comes to fitness and hard constraint violations with the By-Demand-Next-Fit heuristic being more diverse and hence probably a better choice. Further the First-Fit construction heuristic was mainly included in the tuning to provide a more diverse set of solutions.
In this section we want to discuss the configurations found during tuning and support our claims with experiments while also exploring why the found configurations don't necessarily agree with what has been claimed.

parameter	values	tuned
population size	100 .. 500	155
By Demand Next Fit ratio	0. .. 1.	0.0437
tournament size	1 .. 10	8
Crossover Behaviour	PeriodUniformCrossover, UniformCrossover, SimplePeriodCrossover, ProductTypeCrossover	ProductTypeCrossover
crossover rate	0. .. 1.	0.5529
mutation move %	0. .. 1.	0.54724
violating probability factor	0. .. 10.	2.91607
mutation dividend	0.01 .. 10.	0.67895

Table 4.12: This table shows the parameters, possible values and tuned values of the genetic algorithm hyperparameter tuning process.

4.4.1 Diversity

In section 3.3 it was claimed that the First-Fit construction heuristic produces a less diverse set of solutions that the By-Demand-Next-Fit construction heuristic. If that is the case a genetic algorithm with a population of First-Fit constructed solutions requires more exploration than a configuration using By-Demand-Next-Fit as construction heuristic. To test this a simple tuning setup was created. A genetic algorithm is being tuned to simplify the process, hence local search was turned off entirely. To minimise the amount of influences the set of tunable parameters was reduced to only *mutation dividend* and *tournament size*. The other parameters have been set to values that are near the values found to do well during tuning of the genetic algorithm in Table 4.12. The algorithm used a *population size* of 150, a *crossover rate* of 0.5, the *Crossover Behaviour* was set to the *Product-Type-Crossover* and for simplicity the *Mutation Behaviour* was changed from the *Violation-Based-Mutator* to the *Neighbourhood-Switching* mutation behaviour. The *mutation move percentage* was set to 0.4 based on the findings in [VLM20]. The *tournament size* is considered during tuning to be able to still get competitive results since increasing exploration without increasing the selection pressure is most likely not sufficient. The experiment was then started, tuning the *mutation dividend* and *tournament size* with the *By-Demand-Next-Fit ratio* set to 0 and 1 meaning a population made from only First-Fit or only By-Demand-Next-Fit constructed solutions.
The tuning process was allowed 300 trials using 200 training instances.
For our claim to be true we expect a higher *mutation dividend* for the configuration with the *By-Demand-Next-Fit ratio* set to 0.

The tuning results are displayed in Table 4.13 and indeed support our claim. The *mutation dividend* found to work best for the population made up of First-Fit constructed solutions is with ~ 1.32 about 25% higher than the ~ 1.05 of the By-Demand-Next-Fit

parameter	First-Fit	By-Demand-Next-Fit
mutation_dividend	1.32711	1.05127
tournament_size	8	5

Table 4.13: This table displays the tuning results of the experiment validating our claims made in section 3.3.

constructed population. The *tournament size* has also been adjusted accordingly.

4.4.2 Tuning results

This begs the question why every configuration found to perform well during hyperparameter tuning seems to find low values to work best for the By-Demand-Next-Fit ratio. To answer this we applied the two configurations found in the previous subsection to 5 instances of each test subset.

instance	First-Fit	By-Demand-Next-Fit
randomly_generated_0951	**0.40921**	0.50918
randomly_generated_0952	**1.94424**	2.01522
randomly_generated_0953	**0.25370**	0.32899
randomly_generated_0954	1.17224	**0.24751**
randomly_generated_0955	1.08022	**0.11663**
randomly_generated_small_0001	0.04569	**0.03994**
randomly_generated_small_0002	**0.47228**	0.48503
randomly_generated_small_0003	0.28797	**0.27903**
randomly_generated_small_0004	**0.35643**	0.37791
randomly_generated_small_0005	**0.79293**	1.80612
randomly_perfect_random_0951	**0.07459**	0.16725
randomly_perfect_random_0952	**0.07484**	0.1939
randomly_perfect_random_0953	**0.01692**	0.07219
randomly_perfect_random_0954	**0.03816**	0.1556
randomly_perfect_random_0955	**0.00102**	0.07536
realistic_instance_01	**1.05166**	1.05867
realistic_instance_02	**1.18803**	1.20494
realistic_instance_03	**1.00135**	1.03567
realistic_instance_04	2.02735	**1.05293**
realistic_instance_05	**1.17951**	1.18763

Table 4.14: This table displays the fitness values achieved on a few instances by the two tunings found during the diversity experiment.

Table 4.14 shows the fitness values found during testing. The configuration using only the First-Fit construction heuristic outperforms the other configuration often. This

first configuration even delivers some of the best results found during this thesis, while introducing more hard constraint violations than usual in others. Generally the results seem to suggest that the solutions created by the First-Fit construction heuristic are better suited for processing by a genetic or memetic algorithm. It might be that the structure of the solution is better suited for improvements or that diversity plays not as big a role for the population sizes that are being used.
Diversity is expected to play a bigger role the bigger the population gets but appears to have been overestimated in section 3.3. When considering the configuration found for the memetic algorithm in Table 4.8 and the configuration of the genetic algorithm in Table 4.12 the latter uses a smaller By-Demand-Next-Fit ratio while also having a smaller population size. To find whether the need for diversity increases with population size another experiment is conducted in a similar fashion. This time in addition to the *mutation dividend* and *tournament size* the *By-Demand-Next-Fit ratio* is tunable. For the first tuning the population size is kept at 150, for the second tuning the population size is increased to 400. If the need for diversity indeed increases with population size, the *By-Demand-Next-Fit ratio* is expected to grow as well.

parameter population size	**150**	**400**
mutation_dividend	0.67453	2.18489
tournament_size	2	6
by_demand_next_fit_ratio	0.39357	0.73478

Table 4.15: This table shows the tuning results for the experiment testing whether the By-Demand-Next-Fit ratio value increases with larger population size.

And grow it does. In Table 4.15 the results of the tuning process are displayed and confirm our expected results. For a smaller population size of 150 the *By-Demand-Next-Fit ratio* was set to about 40% while it was increased to 73% for a population of size 400.

4.5 Evaluating the best Configurations

In this section we will consider and compare the testing results using our best tunings for the three different approaches discussed in this chapter so far. The simulated annealing approach will use the configuration shown in Table 4.2. The memetic algorithm uses the configuration shown in Table 4.8 and the genetic algorithm is configured with the parameters displayed in Table 4.12. While the evaluation of the three configurations is done considering all test instances, only samples will be shown in this chapter. The results of each configuration on every test instance can be found in the appendix.
As shown in Table 4.1 the test set is split into 4 different subsets. A set of 50 randomly generated instances, 50 randomly generated perfectly solvable instances that are known to have their optimum at fitness 0.0, 27 realistic instances and 10 randomly generated small instances.
When looking at the performance of the entire test set, the genetic algorithm is the clear winner. On 69 of the 137 test instances it ranks first, on 54 second and on 14 last.

The memetic algorithm comes second with 29 first places, 72 second places and 36 last places. Lastly, the simulated annealing approach comes last, while having a higher first place count than the memetic algorithm with 43 first places. It comes second 7 times and places last on 87 of the test instances. Looking at the results of each subset gives a bit more insight. The average ranks of each algorithm for each subset can be seen in Table 4.16. While the simulated annealing approach comes last overall, it ranks best in 9 out of 10 small instances of set R_4^* and comes last in every single one of the 27 realistic instances of set R_1^*. While it ranks last on the two sets of randomly generated instances, the results are a lot less clear ranking 2.32 on average on the randomly generated subset R_3^* and 2.18 on the perfectly solvable subset R_2^*.
The memetic and genetic algorithm have a clearer ranking. The memetic algorithm is on average worse in every subset than the genetic algorithm. While the memetic algorithm happens to rank better than the genetic algorithm on a few instances, this is generally not the case. On the subset R_3^* it averages 2.14 as compared to 1.54 of the genetic algorithm. On the small instance subset R_4^* both rank with 2.5 and 2.3 similarly and considerably worse than the simulated annealing approach with 1.2. On the set of perfectly solvable instances R_2^* the memetic algorithm averages a rank of 2.12, barely outperforming the 2.18 of the simulated annealing approach and beaten by an average rank of 1.62 of the genetic algorithm. Lastly, on the set of realistic instances both approaches perform well with results close to each other. This is also represented by the similar average rank. The memetic algorithm ranks 1.59 while the genetic algorithm ranks 1.40 on average. Another notable peculiarity is that only the two evolutionary approaches manage to find an optimal solution to four of the perfectly solvable instances.

	average rank		
Instance Set	**SA**	**MA**	**GA**
R_1^* *[VLM20]*	3	1.5925	1.4074
R_2^* *[VLM20]*	2.18	2.12	1.62
R_3^* *[VLM20]*	2.32	2.14	1.54
R_4^* *[VLM20]*	1.2	2.5	2.3

Table 4.16: Here the average rank of the simulated annealing (SA), memetic algorithm (MA) and genetic algorithm (GA) on each subset of the test set is displayed.

While ranking can give a good overview of what model performs best in most cases, it gives very little information about the actual performance. While Table 4.17 shows a sample of each subset, Table 4.18 shows a summary of the performance of each approach. While the ranks indicate that the genetic algorithm does best most of the time, Table 4.18 shows that the results are not as clear cut. For the perfectly solvable set R_2^* the statistics are clear. Not only does the genetic algorithm have a lower mean and median, the results are also more robust considering it has the lowest standard deviation. Additionally, the highest fitness value for this set is the lowest of the three. The set of realistic instances R_1^* is not as clear. While the average fitness is smaller for the genetic algorithm, the median and standard deviation of the memetic algorithm is smaller. Yet both, minimum

	median fitness		
instance	**SA**	**MA**	**GA**
randomly_generated_0951	0.54686	0.5164	**0.50251**
randomly_generated_0952	6.10842	**1.94685**	2.02057
randomly_generated_0953	0.3639	0.33652	**0.33044**
randomly_generated_0954	1.21767	0.25902	**0.23438**
randomly_generated_0955	1.13086	0.12494	**0.12473**
randomly_generated_small_0001	**0.03378**	0.04261	0.03775
randomly_generated_small_0002	**0.46851**	0.49274	0.48977
randomly_generated_small_0003	**0.27422**	0.28193	0.28313
randomly_generated_small_0004	**0.35464**	0.36444	0.35812
randomly_generated_small_0005	0.80014	0.79373	**0.79213**
randomly_perfect_random_0951	0.09442	0.08673	**0.07115**
randomly_perfect_random_0952	1.02858	0.07428	**0.07177**
randomly_perfect_random_0953	0.0975	0.0702	**0.04977**
randomly_perfect_random_0954	2.0378	**0.03843**	0.0398
randomly_perfect_random_0955	0.00573	**0.00138**	0.00441
realistic_instance_01	1.13472	1.04929	**1.04357**
realistic_instance_02	1.26852	**1.18529**	1.19578
realistic_instance_03	1.08214	**1.00376**	1.00689
realistic_instance_04	1.09785	**1.0231**	1.02378
realistic_instance_05	1.21835	1.18685	**1.18372**

Table 4.17: This table shows the median fitness of the simulated annealing (SA), memetic algorithm (MA) and genetic algorithm (GA) configuration on 5 instances of each test subset. The best result is highlighted in bold.

and maximum value of the genetic algorithm show better performance. A similar picture paints itself for set R_3^* of the random instances. This time the average fitness value is lower for the memetic algorithm while the median is higher. The memetic algorithm has lower standard deviation and finds a smaller minimum fitness, still beat by the simulated annealing approach, but reaches a slightly higher maximum value. While the genetic and memetic algorithm both clearly outperform the simulated annealing approach in sets R_1^*, R_2^* and R_3^*, as indicated by the ranks it is the clear winner on the set of small instances R_4^* achieving better values in every column.

Another insight can be gained when looking at the difference between the fitness values of the memetic and genetic algorithm. Table 4.19 displays them as absolute difference subtracting the fitness of the genetic algorithm from that of the memetic algorithm. A positive difference hence means that the genetic algorithm performed better, a negative value indicates the opposite.

Without going into too much detail, the values paint a clear picture. While the average difference of every test set except the set of random instances R_3^* is in favour of the

		average	median	std.dev	min	max
R_1^*	**SA**	1.07283	1.11567	0.35739	0.54032	2.22541
	MA	0.96408	1.0231	0.34092	0.44161	2.16071
	GA	0.96059	1.02378	0.34441	0.4239	2.15999
R_2^*	**SA**	0.30215	0.07006	0.48011	0.00417	2.03780
	MA	0.08346	0.07466	0.06	0	0.22259
	GA	0.07272	0.06853	0.04914	0	0.19988
R_3^*	**SA**	2.30846	1.19638	3.08682	0.02346	15.90542
	MA	1.63904	0.48159	2.51242	0.04307	11.50414
	GA	1.64845	0.40476	2.6002	0.06183	11.45467
R_4^*	**SA**	0.43842	0.41157	0.28757	0.01355	0.80014
	MA	0.55019	0.60457	0.3625	0.0198	1.2397
	GA	0.54857	0.60318	0.362349	0.02227	1.23697

Table 4.18: This table shows the average, median, standard deviation, minimum and maximum fitness of the test subsets.

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^*	0.00348	0.00187	0.00758	-0.01049	0.02158
R_2^*	0.01073	0.00818	0.02696	-0.0602	0.10911
R_3^*	-0.00941	0.00648	0.32005	-1.01045	1.05585
R_4^*	0.00161	0.00217	0.00303	-0.0029	0.00632

Table 4.19: This table shows the absolute difference between the memetic and genetic algorithm on the three test subsets.

genetic algorithm, the median values support that even more conclusively.

In summary, given that the genetic algorithm performs best on most instances, places last on the least and shows its ability when compared directly in terms of absolute difference, the genetic algorithm can be considered the best configuration found during this thesis.

4.6 Comparison with Related Work

In this section we want to compare our results with those of related work, i.e. the paper of Vass et al. [VLM20]. The direct comparison is difficult. Not only does the hardware differ, there is also a difference in programming language and framework. As mentioned earlier this is the reason for tuning our implementation again instead of executing it with the parameters from [VLM20].

In order to understand the difference in setup we discuss the difference in results of our simulated annealing configuration and the configuration of Vass et al.. With this

relationship in mind we will subsequently review and compare the performance of the memetic and genetic algorithm.

4.6.1 Simulated Annealing

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^*	0.14215	0.09273	0.20443	0.06085	1.16844
R_2^*	0.18586	0.02767	0.46875	-0.98712	2.01
R_3^*	1.47515	0.17747	2.45378	-0.82033	14.25492
R_4^*	0.00645	0.00499	0.00531	0.00054	0.01861

Table 4.20: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the simulated annealing tuning found in subsection 4.3.1 and the results from Vass et al.[VLM20] for each testing subset.

Table 4.20 shows the absolute difference between the simulated annealing configuration found in subsection 4.3.1 and the results from Vass et al.[VLM20]. The fitness values compared include hard constraint violations. This leads to rather high values for solutions that differ in the amount of hard constraint violations. The values give a good indication that in most cases the solution presented by Vass et al. performs better, yet it also shows that there are instances in R_2^* and R_3^* that produce better results using the simulated annealing approach developed in this thesis. The average difference of R_1^* and R_2^* show small but significant increases in fitness. The median values are considerably smaller. While the smallest difference of R_1^* is with 0.06 small but positive, the minimum value of absolute differences found in R_2^* is negative, indicating that at least one instance is solved more efficiently by the simulated annealing approach developed in this thesis. The largest difference for instances of the set of realistic problems is with around 1.16 considerable and suggests the introduction of an additional hard constraint violation in at least one instance. The maximum difference of the set R_2^* is even bigger with a value slightly above 2. The set of random instances R_3^* shows that while there are some instances that improve on the results from Vass et al. by up to -0.82, there are also instances that perform significantly worse with an absolute difference of 14.25 as the worst example of the entire testing set. The difference on the set of small instances R_4^* is tiny but positive throughout.

As mentioned before, the number of hard constraint violations is part of the fitness values compared. Since their representation as fitness value is somewhat arbitrary and makes a fair comparison of fitness values difficult the information has been gathered again after removing any instances that differ in hard constraint violations. The difference in hard constraint violations is additionally displayed in table 4.21. The table shows that the simulated annealing algorithm presented by Vass et al. has one instance in R_3^* for which it creates more hard constraint violations than the configuration presented in this thesis. Our own simulated annealing implementation introduces considerably

more hard constraint violations than the algorithm presented in [VLM20]. While no instances in R_4^* and only one instance in R_1^* contains more hard constraint violations, R_2^* and R_3^* introduce more such violations in 8 and 23 instances respectively. While for the 8 instances in R_2^* only 9 more violations are introduced, the 23 instances of R_3^* that violate more hard constraints do so by violating additional 72. This severely affects the performance of our simulated annealing configuration.

	instance set	R_1^*	R_2^*	R_3^*	R_4^*
Vass	# instances	0	1	1	0
	# violations	0	1	1	0
SA	# instances	1	8	24	0
	# violations	1	9	72	0

Table 4.21: This table shows the number of instances with hard constraint violations that are present for only one of the two implementations. In addition the number of violations is shown as well.

When removing those values from the data set, the absolute difference figures appear considerably less severe. Table 4.22 shows the absolute difference on each cleaned subset. Given the amount of outliers in set R_3^*, this is were the biggest changes are to be expected. Here the average changed from 1.47 to 0.08 while the median changed from 0.177 to 0.07. While the clean up naturally had the biggest impact on the maximum value, changing it from an absolute difference of 14.25 to 0.18, the minimum value has also been affected, changing it from a negative -0.82 to a positive 0.0001 leaving no instance for which the simulated annealing algorithm improved on the results of this subset.

	absolute difference				
instance set	avg	median	std.dev	min	max
R_1^* **cleaned**	0.10268	0.09256	0.03653	0.06085	0.1889
R_2^* **cleaned**	0.02329	0.02209	0.02093	-0.01134	0.0748
R_3^* **cleaned**	0.08224	0.07717	0.06026	0.00014	0.18348
R_4^* **cleaned**	0.00645	0.00499	0.00531	0.00054	0.01860

Table 4.22: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the simulated annealing tuning found in subsection 4.3.1 and the results from Vass et al.[VLM20] for each testing subset after removing instances that differ in hard constraints.

In summary the simulated annealing approach presented in [VLM20] vastly outperforms the simulated annealing approach developed and tuned during this thesis. The choice of programming language, framework and hardware while still maintaining a runtime limit of 300 seconds lead to a more aggressive tune that while producing reasonable results most of the time still struggles to remove many hard constraint violations.

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^*	0.0334	0.02817	0.02968	0.00555	0.12452
R_2^*	-0.03283	0.03269	0.24925	-0.89645	0.16329
R_3^*	0.80573	0.13065	1.67485	-0.91588	9.18753
R_4^*	0.11822	0.01511	0.30182	0.00373	1.0231

Table 4.23: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the memetic algorithm tuning found in subsection 4.3.2 and the results from Vass et al.[VLM20] for each testing subset.

4.6.2 Memetic Algorithm

Table 4.23 shows the absolute difference values between the memetic algorithm and the simulated annealing approach presented in [VLM20]. Starting with the set of realistic instances R_1^* the average difference is with 0.033 considerably lower than that of the simulated annealing approach presented in the previous section. This is true for every value of this set. The median is found to be slightly smaller than the average. In addition a standard deviation of about 0.03 shows stable difference values on this test subset. While the minimum and maximum values are small, the minimum value remains positive, meaning that while the results are better than our simulated annealing approach, they fall short of the results shown by Vass et al.. The set of perfect instances R_2^* shows a negative average that cannot be backed up by the small but positive median of about 0.033. The minimum value is negative, showing an improvement in at least one instance of the set when compared to the results of [VLM20]. The maximum value is with an increase of 0.16 larger than for the previous set but considerably smaller than the corresponding value of our simulated annealing approach in Table 4.20. The set of random instances R_3^* has the by far biggest average difference amongst the sets with a 0.8 increase in fitness value. The median is notably lower at about 0.13 suggesting some strong outliers with considerably higher difference which is confirmed by the large maximum difference of above 9. The standard deviation is higher than for the other subsets with about 1.67. The smallest value of the set improves the results of Vass et al. with a difference of around -0.91. All of the values calculated for R_3^* are significantly lower than those found for the simulated annealing approach in Table 4.20. On the set of small instances R_4^* the memetic algorithm not only performs worse than the simulated annealing approach of Vass et al. it is also outperformed by the simulated annealing algorithm developed in this thesis. The average difference is about 0.11, with a lower median of around 0.015. In addition the standard deviation is with about 0.3 considerably larger than the almost 0.005 of the simulated annealing result in Table 4.20 indicating a less stable set of fitness values. The smallest difference found on this rather small test set of only 10 instances is with a fitness increase of about 0.0037 only marginally bigger when compared to the results in [VLM20]. It is considerably larger than the difference of our simulated annealing approach. The biggest absolute difference found is slightly above 1, giving an indication of why the standard deviation is higher than expected and the introduction of a hard constraint violation.

As done in the previous subsection we will again look at the difference in hard constraint violations.

	instance set	R_1^*	R_2^*	R_3^*	R_4^*
Vass	**# instances**	0	4	1	0
	# violations	0	4	1	0
MA	**# instances**	0	0	14	1
	# violations	0	0	37	1

Table 4.24: This table shows the number of instances with hard constraint violations that are present for only one of the two implementations. In addition the number of violations is shown as well.

Table 4.24 shows the number of instances and corresponding hard constraint violations that are present for one approach but not the other. While there is no difference in hard constraint violations in either direction for the set of realistic instances R_1^*, our memetic algorithm removes one violation each in four instances of the set of perfect instances R_2^*. Those four instances are now devoid of any violations making the entire set non-violating. There are no additional violations introduced by the memetic algorithm in set R_2^*. The set R_3^* paints the exact opposite picture. While one hard constraint violation in one instance is solved by the memetic algorithm, it introduces 37 additional violations in 14 of the 50 instances. The results of Vass et al. show no hard constraint violations in R_4^*, hence there are no improvements possible. The memetic algorithm introduces, however, one violation in one instance on the set of small instances. While this result strongly improves on the violations introduced by our simulated annealing approach and even manages to make an entire set violation-free, the amount of additional violations introduced, particularly in the set of random instances R_3^*, is problematic. In the following we will also look at the statistical descriptors of the test sets after removing the instances on which hard constraint violations differ.

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^* **cleaned**	0.0334	0.02817	0.02968	0.00555	0.12452
R_2^* **cleaned**	0.03944	0.03466	0.04687	-0.04311	0.16329
R_3^* **cleaned**	0.08846	0.09274	0.05115	-0.01609	0.17497
R_4^* **cleaned**	0.01768	0.01231	0.01136	0.00373	0.04284

Table 4.25: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the memetic algorithm tuning found in subsection 4.3.1 and the results from Vass et al.[VLM20] for each testing subset after removing instances that differ in hard constraints.

Table 4.25 shows the absolute difference again after cleaning the data. As expected, they normalised considerably after removing the strongest values. Since there were no

differences in violations for the set of realistic instances R_1^* the values for this set remain unaltered. Since the memetic algorithm removed the violations of four instances, those improvements had to be removed, raising the value from an average improvement on the fitness to an average increase in fitness. The value changed from about negative -0.03 to positive about 0.04. Removing four of the lowest values also raised the median to 0.034. Accordingly the standard deviation shrank from about 0.25 to about 0.05. The minimum value, while showing less improvement, remained negative, while the maximum value did not change. The biggest effect is again to be seen on set R_3^*. By removing 15 of the 50 instances from the data, the values changed drastically. The average shrank to about 0.088, the median changed from about 0.13 to around 0.092, the standard deviation changed from including an entire hard constraint violation with above 1.67 to a considerably more stable 0.05 and while the minimum value remained barely negative at about -0.01, the maximum value naturally shrank from a difference in fitness value of above 9 to around 0.175.

4.6.3 Genetic Algorithm

In the following we will discuss the difference in fitness values for the genetic algorithm described in Table 4.12. Table 4.26 shows the absolute difference on each testing subset between the genetic algorithm and the results achieved by the simulated annealing approach presented in [VLM20]. Starting with set R_1^* the average is with 0.0299 low but positive, indicating that on average this approach performs worse than the simulated annealing approach. Both the average and the median are, however, still a bit lower than those values of the memetic algorithm, showing some improvement over the results of the memetic algorithm for the set of realistic instances. The minimum value is with 0.0078 positive and larger than the smallest difference of the Memetic Algorithm, the maximum value is with 0.1 smaller. On the set of perfect instances R_2^* the genetic algorithm performs better than the memetic algorithm on all accounts but the standard deviation. While the average is negative at -0.04, the median is positive at 0.024. The standard deviation lies around 0.26. The minimum value improves the fitness value of an instance by -0.95, while the biggest fitness increase compared to the results of Vass et al. lies around 0.14. While the average absolute difference on the set R_3^* is with 0.81 slightly bigger than the average of the memetic algorithm, the median is with 0.11 smaller than the 0.13 of the memetic algorithm in Table Table 4.23. The standard deviation is with 1.75 even higher than that of the memetic algorithm. The minimum value reaches an improvement of -0.92, a value smaller than that of the memetic algorithm, while the maximum also improves to the still large value of about 9.18. On the set of small instances R_4^* the genetic algorithm achieves on average a 0.11 higher fitness value, which is slightly improving on the results of the memetic algorithm. So does the median of 0.013. The standard deviation lies at 0.3. While the minimum value shows no improvement at all on the results of Vass et al., the smallest fitness increase remains negligible. The maximum difference of above 1 still suggests at least one instance for which another hard constraint violation is introduced.

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^*	0.02991	0.02362	0.02331	0.00786	0.10294
R_2^*	-0.04357	0.02409	0.2594	-0.9555	0.14058
R_3^*	0.81514	0.11348	1.75558	-0.92276	9.13807
R_4^*	0.1166	0.01383	0.30146	0.00213	1.02037

Table 4.26: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the genetic algorithm tuning found in subsection 4.3.2 and the results from Vass et al.[VLM20] for each testing subset.

Looking at the hard constraint violations that have been removed and introduced in Table 4.27 the table looks barely any different to that of the memetic algorithm (Table 4.24). While there are no violation changes in R_1^*, the genetic algorithm also removes the 4 remaining violations of set R_2^*. In set R_4^* the genetic algorithm introduces a violation in one instance. Set R_3^* is, again, the set with the worst results. While one violation that has been present in [VLM20] is removed, 37 new violations are introduced. This time, however, with only 13 instances affected, one instance less than the memetic algorithm. This means that in turn another instance appears to have received an additional violation.

	instance set	R_1^*	R_2^*	R_3^*	R_4^*
Vass	**# instances**	0	4	1	0
	# violations	0	4	1	0
GA	**# instances**	0	0	13	1
	# violations	0	0	37	1

Table 4.27: This table shows the number of instances with hard constraint violations that are present for only one of the two implementations. In addition the number of violations is shown as well.

	absolute difference				
instance set	**avg**	**median**	**std.dev**	**min**	**max**
R_1^* **cleaned**	0.02991	0.02362	0.02331	0.00786	0.10294
R_2^* **cleaned**	0.03199	0.02679	0.04079	-0.04439	0.14058
R_3^* **cleaned**	0.08664	0.09694	0.04471	-0.01846	0.1521
R_4^* **cleaned**	0.01619	0.0135	0.01183	0.00213	0.03987

Table 4.28: This table shows the average, median, standard deviation, minimum and maximum absolute difference between the genetic algorithm tuning found in subsection 4.3.1 and the results from Vass et al.[VLM20] for each testing subset after removing instances that differ in hard constraints.

Table 4.28 shows the average, median, standard deviation, minimum and maximum value

of the absolute difference between the testing results of the genetic algorithm and the simulated annealing results presented in [VLM20] by Vass et al. after removing instances for which there are differing hard constraint violations to give a better indication of the actual difference in fitness values. Since there has been no change in data, the values for the set of realistic instances have not changed for the genetic algorithm. When looking at the set of perfectly solvable instances R_2^*, however, the average moved from an improvement to a fitness increase of 0.03. The median also rose slightly to 0.026. Both values indicate better performance than the cleaned values of the memetic algorithm in Table 4.25. The standard deviation lies around 0.04. The best improvement on an instance of the set of R_2^* without considering violations reduces fitness by -0.044, while the worst change, while not being affected by the cleaning process, increases fitness by 0.14. Testing subset R_3^* is affected most by the removal of differing instances. Its average fitness difference decreases to 0.086, a value lower than the memetic algorithm's. The median is, however, slightly above both the average and the median of the memetic algorithm. The standard deviation measures 0.044. The minimum shows an improvement of -0.01846, while the maximum difference increases fitness by 0.1521. Both values are below the corresponding minimum and maximum numbers for the memetic algorithm. On the set of small instances R_4^* the average and median shrank slightly to 0.016 and 0.0135 respectively. While the genetic algorithm fails to improve on any of the instances of R_4^*, the smallest difference found is with 0.00213 the smaller of the two evolutionary approaches. So is the maximum difference of 0.04.

4.6.4 Optimality Gap

In [VLM20] an optimality gap analysis was done for the metaheuristic and exact techniques employed in their work. For this gap the lower bound of the MIP solver was used. The values are provided in the corresponding thesis. They found that for most of the set of realistic instances their simulated annealing algorithm manages to find solutions within a 3% optimality gap. For a similar analysis of the results gathered in this thesis the optimality gaps for those same 19 out of 27 instances for which a reasonable lower bound has been found are calculated and shown in Figure 4.5. The instances that are not shown are not displayed because they argue that only lower bounds for which the MIP optimality gap is 10% or lower should be considered. The figure shows clearly that the simulated annealing results presented in [VLM20] are performing considerably better than the results found during this thesis, also within the 3% stated. The simulated annealing configuration found in subsection 4.3.1 performs considerably worse than the other algorithms. While the set of realistic instances is not necessarily representative for the entire problem the plot nicely visualises in a way that is otherwise difficult the relation of the algorithms while slightly understating the performance of the genetic algorithm. While the optimality gap of 3% cannot be reached by our evolutionary approaches, most are within 5%.

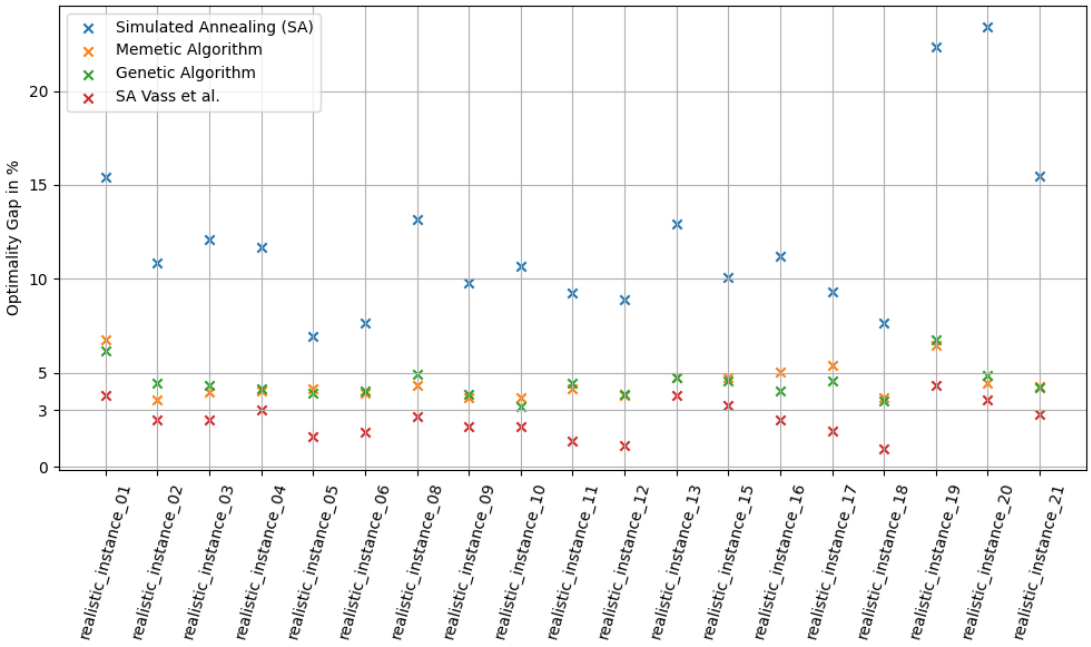


Figure 4.5: This figure shows the optimality gap for the simulated annealing approach by Vass et al. [VLM20] and the three approaches that have been developed in this thesis.

CHAPTER 5

Conclusion

In this thesis we presented a genetic and memetic approach to solving the Production Leveling Problem. We first introduced the problem and fitness function, then went on to present our solution representation and various implementations of the different genetic and memetic operators. We discussed and analysed those operators with a particular focus on construction heuristics, for which we looked at the quality and diversity of the solutions they create. In order to measure equality between sets of our solution representations, the notion of the Extended Jaccard Index was introduced. Based on the analysis we manually disregarded some options to reduce the set of operator implementations to a more manageable size. We implemented and applied a random instance generator presented in earlier work to generate 2000 random instances for use in the tuning process. Using a hyperparameter tuning framework we then tuned both the memetic and genetic algorithm to find their best configurations. To make a fairer comparison possible and to accommodate changes in hardware, programming language and general framework we implemented a simulated annealing approach and tuned its *learning rate* and *initial temperature*. During the experiments we found evidence supporting the diversity claims made during this investigation while it was also shown that there are additional properties that influence the suitability of a construction heuristics that need further investigation. Using their best configuration the simulated annealing, genetic and memetic algorithm were applied to the test data provided in the literature. We first compare the results amongst the algorithms implemented in this thesis and additionally compare them with the results of previous work.
The main outcomes of this work are:

- Diversity plays an important role for genetic algorithms, requiring more diverse construction heuristics with growing population sizes.
- The genetic algorithm performed slightly better than the memetic algorithm which additionally applies local search.

- Although the evolutionary algorithms could not quite match the results found in the literature on the set of perfectly solvable instances, they managed to solve the entire set without hard constraint violations, as well as 4 instances to optimality, which, to the best of our knowledge, has not been achieved thus far using metaheuristics.
- On the set of realistic instances both evolutionary approaches find solutions mostly within a 5% optimality gap.

There are various avenues for future work opened by this thesis. The further investigation of the Production Leveling Problem is certainly an interesting topic. With the difference in circumstances it could not be resolved whether evolutionary algorithms perform overall better or worse than simulated annealing on the PLP. Further work could investigate that by testing the algorithms found in this thesis with similar hardware and implemented in a similarly fast programming language as done in other work. For this another tuning process would have to take place since, as also shown in this thesis, different hardware requires different parameters, particularly with a runtime limit in place. Another approach that is not as dependent on hardware would be to increase the runtime used. The same experiments could be conducted on 10 or 20 minute runtimes to give the algorithm more time to properly execute. This, too, would require additional tuning. Independently of the problem, the role of construction heuristics could be further investigated. The applicability of the Extended Jaccard Index needs to be explored and further validated. This thesis also showed that while Solution Quality and Diversity are an important measure when it comes to construction heuristics, there are other properties that we have not been able to capture within this work that need discussing.

APPENDIX A

Runtime information on Construction Heuristics

	953				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.19296	0.18499	0.16697	0.29276	0.02681
First-Fit-with-Target-Limit	0.27827	0.2727	0.25955	0.35237	0.01728
Random	0.04846	0.04494	0.04108	0.07406	0.0075
By-Demand	0.04532	0.04213	0.038	0.06927	0.00763
Next-Fit	0.1233	0.12137	0.11374	0.14383	0.00787
By-Demand-Next-Fit	0.12134	0.11879	0.11361	0.13809	0.00668
Vass-et-al	14.76951	14.25532	13.98852	18.47919	1.37648

	958				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.05502	0.04251	0.03141	0.15357	0.02824
First-Fit-with-Target-Limit	0.03816	0.03752	0.03578	0.0526	0.00276
Random	0.01256	0.0106	0.00898	0.02576	0.00411
By-Demand	0.00938	0.00891	0.00783	0.02929	0.00298
Next-Fit	0.01495	0.0142	0.01257	0.03063	0.0031
By-Demand-Next-Fit	0.01354	0.01296	0.01227	0.03048	0.0027
Vass-et-al	0.61506	0.61667	0.595	0.62607	0.00957

	987				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.21839	0.20766	0.18059	0.30073	0.03041
First-Fit-with-Target-Limit	0.2689	0.26163	0.23432	0.36638	0.02753
Random	0.06127	0.05428	0.03796	0.12156	0.01908
By-Demand	0.0528	0.0495	0.03538	0.11195	0.01621
Next-Fit	0.11202	0.1047	0.09024	0.15022	0.01625
By-Demand-Next-Fit	0.1093	0.10538	0.09148	0.151	0.01498
Vass-et-al	6.57144	6.41616	6.32205	7.11303	0.26413

	small__0001				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.25675	0.24955	0.22654	0.3206	0.02514
First-Fit-with-Target-Limit	0.35606	0.34435	0.33047	0.46466	0.02942
Random	0.1297	0.10397	0.05438	0.49834	0.0839
By-Demand	0.06194	0.05761	0.04256	0.10784	0.01591
Next-Fit	0.15335	0.14826	0.13035	0.21004	0.01939
By-Demand-Next-Fit	0.15539	0.14478	0.13057	0.25107	0.02826
Vass-et-al	12.49558	12.39268	12.14817	13.83533	0.48143

	small__0003				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.00386	0.00363	0.00338	0.00635	0.00065
First-Fit-with-Target-Limit	0.00492	0.00457	0.00405	0.00875	0.00102
Random	0.00551	0.00608	0.00229	0.01091	0.00211
By-Demand	0.00163	0.00144	0.00138	0.00425	0.00054
Next-Fit	0.00222	0.00207	0.0019	0.00479	0.00045
By-Demand-Next-Fit	0.00263	0.00227	0.00172	0.00627	0.00105
Vass-et-al	0.01959	0.01874	0.01794	0.0273	0.00278

	957				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.02755	0.02601	0.02411	0.04614	0.00436
First-Fit-with-Target-Limit	0.03595	0.03533	0.03422	0.05514	0.00298
Random	0.01289	0.01207	0.0101	0.02657	0.00295
By-Demand	0.01074	0.00977	0.00874	0.02695	0.00285
Next-Fit	0.01722	0.01674	0.01547	0.02834	0.00189
By-Demand-Next-Fit	0.01592	0.01541	0.01484	0.02872	0.00198
Vass-et-al	1.53761	1.4393	1.38894	2.39079	0.22329

	966				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.02466	0.02419	0.02283	0.03681	0.00196
First-Fit-with-Target-Limit	0.03229	0.03166	0.03033	0.04458	0.00246
Random	0.01165	0.01137	0.01044	0.02427	0.00187
By-Demand	0.01057	0.01011	0.00933	0.02407	0.00209
Next-Fit	0.01902	0.01832	0.01648	0.03006	0.00262
By-Demand-Next-Fit	0.02005	0.01833	0.01628	0.0305	0.00438
Vass-et-al	2.22708	2.16485	2.06907	3.05381	0.18503

	967				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.1049	0.0986	0.09159	0.14214	0.0136
First-Fit-with-Target-Limit	0.15039	0.14385	0.13593	0.20566	0.01679
Random	0.03231	0.03118	0.02822	0.05443	0.00503
By-Demand	0.0299	0.02823	0.02617	0.05684	0.00548
Next-Fit	0.08061	0.07452	0.06664	0.1476	0.01707
By-Demand-Next-Fit	0.07156	0.06893	0.06455	0.10831	0.00764
Vass-et-al	12.40396	11.89147	11.57874	22.42138	1.63738

	978				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.05784	0.05488	0.05171	0.08325	0.00701

	980				
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.04748	0.04462	0.03934	0.07845	0.00735

First-Fit-with-Target-Limit	0.07698	0.0769	0.0741	0.08106	0.00162
Random	0.02098	0.02053	0.01909	0.02516	0.0012
By-Demand	0.02141	0.02017	0.01743	0.03467	0.00375
Next-Fit	0.04482	0.04231	0.03714	0.06608	0.00759
By-Demand-Next-Fit	0.04218	0.0387	0.03661	0.0742	0.00784
Vass-et-al	5.87972	5.82007	5.75282	6.60861	0.19345

First-Fit-with-Target-Limit	0.06067	0.05758	0.05402	0.08407	0.00721
Random	0.01824	0.0173	0.0161	0.03331	0.00283
By-Demand	0.01668	0.01587	0.0146	0.03622	0.00317
Next-Fit	0.03405	0.03118	0.02856	0.0569	0.00688
By-Demand-Next-Fit	0.03036	0.02949	0.02805	0.05141	0.00338
Vass-et-al	4.32943	4.20477	4.10103	5.20406	0.26543

994					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.15241	0.14649	0.14147	0.19902	0.01234
First-Fit-with-Target-Limit	0.23876	0.23083	0.21845	0.30872	0.02288
Random	0.04464	0.04218	0.03809	0.07186	0.00746
By-Demand	0.0429	0.03804	0.03506	0.09734	0.01243
Next-Fit	0.1157	0.10911	0.09994	0.15382	0.01412
By-Demand-Next-Fit	0.1144	0.10893	0.1012	0.15068	0.0129
Vass-et-al	17.04793	16.51025	16.04884	24.17448	1.52648

956					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.18136	0.18089	0.16568	0.21257	0.01102
First-Fit-with-Target-Limit	0.28379	0.26936	0.24709	0.4108	0.03829
Random	0.04487	0.04258	0.04042	0.06146	0.00552
By-Demand	0.04089	0.03873	0.03643	0.06977	0.00645
Next-Fit	0.11653	0.11395	0.11041	0.14101	0.00674
By-Demand-Next-Fit	0.11438	0.11184	0.10994	0.13546	0.006
Vass-et-al	14.01949	13.60102	13.39229	15.81247	0.8548

960					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.02947	0.02877	0.02677	0.04901	0.00408
First-Fit-with-Target-Limit	0.03575	0.03483	0.03421	0.05234	0.00287
Random	0.01008	0.00967	0.0085	0.02471	0.00228
By-Demand	0.00884	0.00843	0.00765	0.02478	0.00237
Next-Fit	0.01444	0.01327	0.01204	0.03212	0.00314
By-Demand-Next-Fit	0.01302	0.01266	0.01191	0.02923	0.00239
Vass-et-al	0.57803	0.5765	0.56509	0.59333	0.00822

962					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.44132	0.24365	0.16462	3.11136	0.52174
First-Fit-with-Target-Limit	0.25549	0.25207	0.23246	0.32551	0.02244
Random	0.04057	0.03798	0.03629	0.06252	0.00627
By-Demand	0.0383	0.03476	0.03243	0.05783	0.00719
Next-Fit	0.09957	0.09373	0.08834	0.14844	0.01325
By-Demand-Next-Fit	0.09532	0.09141	0.08722	0.12101	0.00864
Vass-et-al	6.37136	6.29784	5.93559	7.35394	0.38579

964					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.24449	0.23585	0.21344	0.33271	0.02597
First-Fit-with-Target-Limit	0.32771	0.32087	0.31309	0.40986	0.01819
Random	0.05159	0.04786	0.04433	0.07745	0.00852
By-Demand	0.04823	0.04566	0.04202	0.0658	0.00679
Next-Fit	0.13923	0.13332	0.12511	0.1888	0.01466
By-Demand-Next-Fit	0.13678	0.13226	0.12508	0.17349	0.01209
Vass-et-al	12.15971	11.97158	11.88331	13.36876	0.44723

970					
	mean	**median**	**min**	**max**	**std. dev.**
First-Fit	0.00358	0.00346	0.0033	0.00606	0.0004
First-Fit-with-Target-Limit	0.00443	0.00428	0.00402	0.00696	0.00047
Random	0.00193	0.00168	0.00158	0.00456	0.00054
By-Demand	0.00226	0.00177	0.00137	0.00719	0.00123
Next-Fit	0.0022	0.00206	0.00186	0.00543	0.00052
By-Demand-Next-Fit	0.00188	0.00172	0.00166	0.00512	0.00051
Vass-et-al	0.02006	0.01982	0.01833	0.02234	0.00128

	953	**958**	**987**	**957**	**small_1**	**small_3**
orders	2019	888	2706	885	86	34
periods	10	5	4	34	8	7

	978	**980**	**994**	**962**	**956**	**960**
orders	1725	1449	3360	3086	3583	733
periods	26	23	37	77	50	61

Table A.1: This table shows the mean, median, min, max and standard deviation of the runtimes in seconds for each construction heuristic on a set of files. The instances chosen are not necessarily representative, they are, however, chosen with a certain variation in number of periods and number of orders in mind. All the results have been calculated from 50 runs of each combination.

APPENDIX B

Expected and actual allele coverage for multiple examples of both small and large cardinalities

population 20									
instance name	periods	expected	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
987	4	**0.996829**	0.601349	0.598577	0.996674	0.945307	0.996582	0.983925	0.380081
958	5	**0.988471**	0.368919	0.385135	0.988514	0.709009	0.986937	0.876577	0.383784
small_0003	7	**0.954179**	0.315126	0.340336	0.970588	0.218487	0.62605	0.218487	0.642857
small_0001	8	**0.930791**	0.204942	0.277616	0.94186	0.412791	0.715116	0.502907	0.588663
953	10	**0.878423**	0.150619	0.15478	0.878058	0.723824	0.877811	0.784151	0.168648

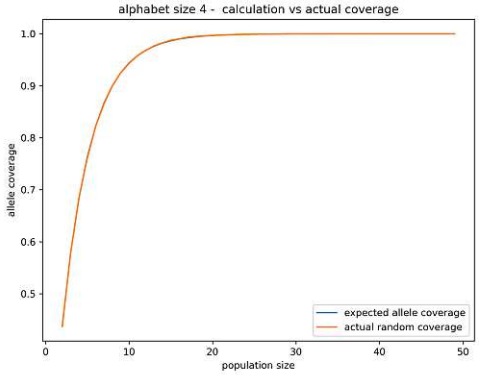
Table B.1: This table shows the expected allele coverage for the given periods of small size and a population of 20 and the results the different construction heuristics produced.

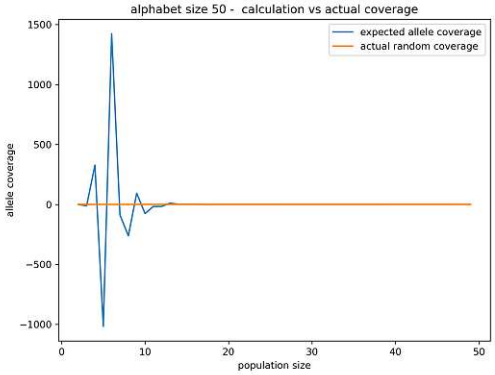
population 100									
instance name	periods	expected	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
966	22	**0.990458**	0.103823	0.112168	0.990256	0.758228	0.976317	0.791515	0.203263
980	23	**0.988265**	0.086776	0.088427	0.987488	0.552735	0.979116	0.700183	0.200108
978	26	**0.9802**	0.088361	0.09204	0.980134	0.376878	0.967848	0.709565	0.19835
967	31	**0.962333**	0.247916	0.28528	0.962163	0.464072	0.961615	0.830131	0.223011
957	34	0.949475	0.082918	0.076604	0.948388	0.169791	0.77338	0.523662	0.249053
994	37	**0.935423**	0.060039	0.067704	0.936583	0.54782	0.928209	0.653016	0.113127

Table B.2: This table shows the expected allele coverage for the given periods of medium size and a population of 100 and the results the different construction heuristics produced.

population 200									
instance name	periods	expected	First-Fit	First-Fit-with-Target-Limit	Random	By-Demand	Next-Fit	By-Demand-Next-Fit	Vass-et-al
956	50	**0.982412**	0.047234	0.050008	0.981931	0.6247	0.949813	0.624516	0.087273
970	57	**0.970984**	0.065163	0.107613	0.968828	0.141604	0.239192	0.141604	0.222431
960	61	**0.963332**	0.038915	0.049315	0.962941	0.245902	0.206383	0.245902	0.150158
964	69	**0.946053**	0.032233	0.035897	0.945968	0.709123	0.729114	0.709247	0.062387
962	77	**0.926789**	0.027291	0.028987	0.926097	0.72059	0.305519	0.720796	0.049234

Table B.3: This table shows the expected allele coverage for the given periods of large size and a population of 200 and the results the different construction heuristics produced.





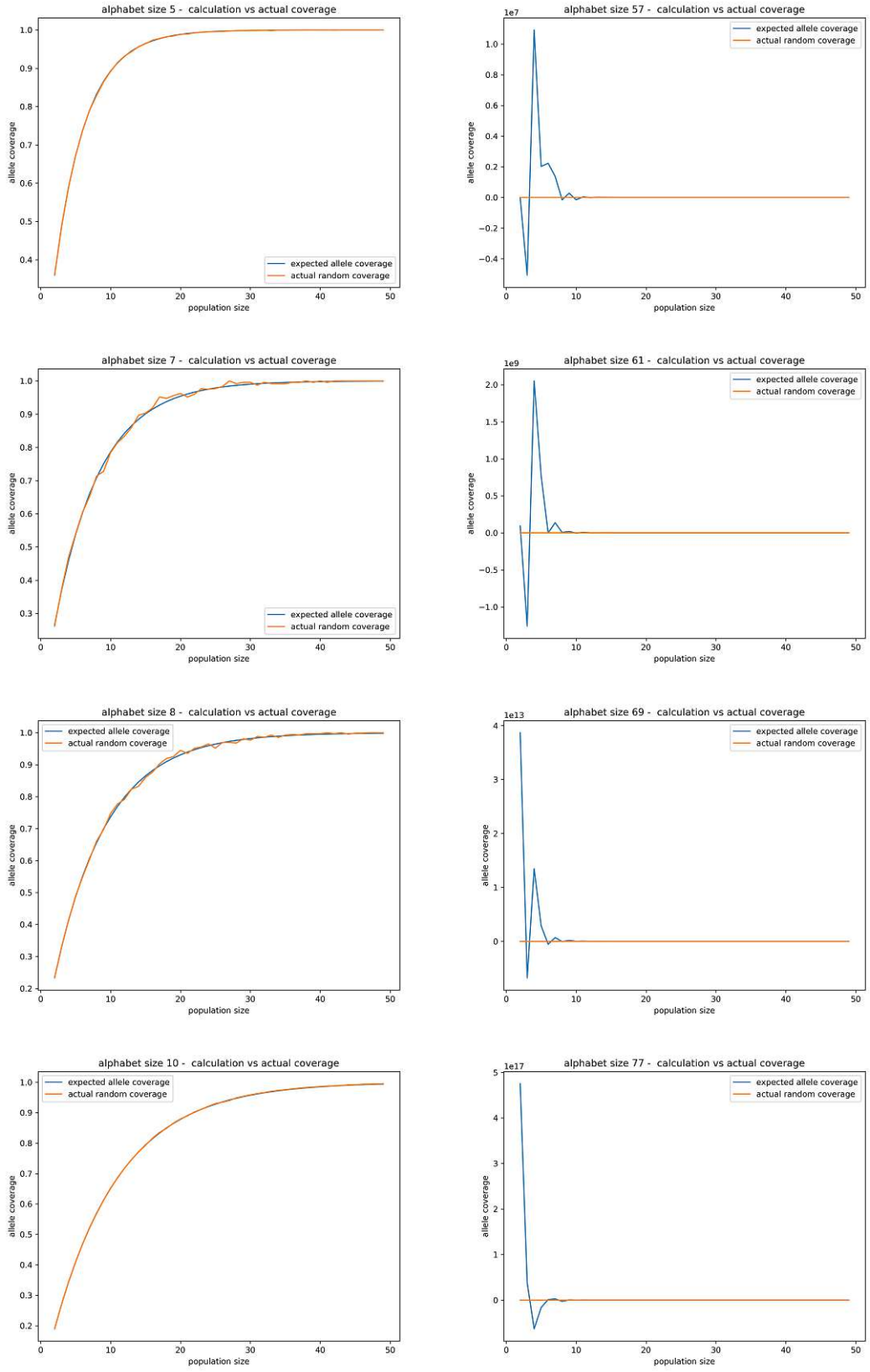


Figure B.1: The expected and actual allele coverage values for small and large cardinalities based on the calculations from [TS93] and its change along increasing population sizes.

APPENDIX C

Results on test instances

C.1 Simulated Annealing

instance	average	median	std.dev	min	max
randomly_generated_0951	0.844961	0.546864	0.496755	0.4431	1.544919
randomly_generated_0952	4.731234	6.108424	1.959221	1.960493	6.124784
randomly_generated_0953	0.335662	0.3639	0.044813	0.272407	0.370677
randomly_generated_0954	1.20836	1.217676	0.822351	0.196565	2.21084
randomly_generated_0955	1.10831	1.130868	0.031955	1.063119	1.130944
randomly_generated_0956	0.079454	0.078334	0.001786	0.078054	0.081974
randomly_generated_0957	2.35159	2.726958	0.535963	1.593635	2.734176
randomly_generated_0958	0.091784	0.112771	0.03329	0.044793	0.117788
randomly_generated_0959	0.094826	0.113566	0.028113	0.055089	0.115822
randomly_generated_0960	0.79452	0.830231	0.052221	0.720682	0.832647
randomly_generated_0961	3.161625	2.526766	0.974556	2.419642	4.538468
randomly_generated_0962	0.02354	0.023469	0.002168	0.020921	0.026231
randomly_generated_0963	2.041689	2.095619	0.875904	0.942982	3.086465
randomly_generated_0964	0.086457	0.087009	0.00217	0.083566	0.088795
randomly_generated_0965	0.075886	0.075732	0.000229	0.075717	0.07621
randomly_generated_0966	0.263837	0.29996	0.054713	0.186517	0.305035
randomly_generated_0967	5.626459	6.301157	0.983681	4.235531	6.34269
randomly_generated_0968	1.095401	0.467371	0.972963	0.349147	2.469684
randomly_generated_0969	4.35696	5.360877	2.162018	1.353883	6.35612
randomly_generated_0970	0.116756	0.123537	0.015417	0.09542	0.131312
randomly_generated_0971	2.214861	2.509834	0.431169	1.605202	2.529546
randomly_generated_0972	0.841263	1.175086	0.473221	0.172028	1.176675
randomly_generated_0973	0.218481	0.250558	0.045813	0.153693	0.251191
randomly_generated_0974	0.859208	0.863451	0.010261	0.845069	0.869105
randomly_generated_0975	0.035477	0.035077	0.000584	0.035052	0.036303
randomly_generated_0976	1.270249	1.2964	0.039687	1.214165	1.300181
randomly_generated_0977	4.347572	5.366804	2.186418	1.309813	6.366099
randomly_generated_0978	1.579282	1.613171	0.866095	0.501999	2.622677
randomly_generated_0979	5.379998	4.769315	1.760598	3.59492	7.77576
randomly_generated_0980	1.158204	1.172371	0.023216	1.125469	1.176771
randomly_generated_0981	4.452382	5.458934	1.426374	2.435186	5.463026
randomly_generated_0982	0.060685	0.059147	0.007666	0.052161	0.070748
randomly_generated_0983	5.440618	5.498602	0.842812	4.380618	6.442634
randomly_generated_0984	0.154734	0.162113	0.020939	0.126209	0.17588
randomly_generated_0985	0.037624	0.037704	0.003518	0.033276	0.041893
randomly_generated_0986	9.513173	11.51588	2.839112	5.498066	11.52558
randomly_generated_0987	0.06538	0.080444	0.023291	0.032481	0.083216
randomly_generated_0988	0.469882	0.139032	0.471838	0.133461	1.137154
randomly_generated_0989	2.917032	3.959862	1.479454	0.824771	3.966463
randomly_generated_0990	0.02884	0.029931	0.002225	0.025739	0.03085
randomly_generated_0991	1.403585	1.434622	0.84753	0.350406	2.425726

randomly_generated_0992	2.653952	3.358829	1.001134	1.238139	3.364889
randomly_generated_0993	4.472721	4.841025	0.524958	3.730326	4.846812
randomly_generated_0994	1.728637	2.40032	0.955258	0.377705	2.407886
randomly_generated_0995	1.882673	2.216411	0.473686	1.212781	2.218826
randomly_generated_0996	13.19415	15.90542	6.091396	4.757242	18.91979
randomly_generated_0997	5.357349	6.08226	1.02629	3.905956	6.08383
randomly_generated_0998	1.223063	1.225967	0.006003	1.214703	1.22852
randomly_generated_0999	1.905952	1.277286	0.973309	1.159822	3.280748
randomly_generated_1000	0.082105	0.096325	0.029328	0.041253	0.108738

Table C.1: This table shows the results of the simulated annealing algorithm using the configuration found in this thesis on the randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
randomly_generated_small_0001	0.035934	0.033787	0.006692	0.029024	0.04499
randomly_generated_small_0002	0.466009	0.46851	0.004199	0.460094	0.469423
randomly_generated_small_0003	0.275406	0.274226	0.001752	0.274108	0.277883
randomly_generated_small_0004	0.355067	0.354648	0.001728	0.353191	0.357362
randomly_generated_small_0005	0.799201	0.800143	0.002972	0.795183	0.802278
randomly_generated_small_0006	0.69001	0.690531	0.002079	0.687244	0.692255
randomly_generated_small_0007	0.01368	0.01355	0.001982	0.01132	0.016169
randomly_generated_small_0008	0.22123	0.228276	0.010921	0.205806	0.22961
randomly_generated_small_0009	0.793097	0.792984	0.000396	0.792679	0.793629
randomly_generated_small_0010	0.731647	0.727558	0.007614	0.725065	0.742319

Table C.2: This table shows the results of the simulated annealing algorithm using the configuration found in this thesis on the small randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
randomly_perfect_random_0951	0.093996	0.094427	0.001154	0.092417	0.095144
randomly_perfect_random_0952	1.028674	1.028585	0.815963	0.029371	2.028065
randomly_perfect_random_0953	0.075978	0.097505	0.033689	0.028406	0.102025
randomly_perfect_random_0954	1.390059	2.037804	0.918457	0.091166	2.041207
randomly_perfect_random_0955	0.005717	0.00573	3.39E-05	0.00567	0.00575
randomly_perfect_random_0956	0.062824	0.079768	0.026377	0.025571	0.083132
randomly_perfect_random_0957	0.073116	0.07279	0.001722	0.07119	0.075369
randomly_perfect_random_0958	0.698586	1.02914	0.47284	0.029904	1.036715
randomly_perfect_random_0959	0.392935	0.061875	0.471478	0.057229	1.0597
randomly_perfect_random_0960	0.35572	0.023223	0.471764	0.021043	1.022893
randomly_perfect_random_0961	1.12677	1.179654	0.81822	0.099264	2.101391
randomly_perfect_random_0962	1.086204	1.090062	0.012027	1.069929	1.098621
randomly_perfect_random_0963	0.391375	0.061403	0.471001	0.055261	1.057461
randomly_perfect_random_0964	0.091523	0.087858	0.010443	0.080966	0.105745
randomly_perfect_random_0965	0.078031	0.0787	0.001748	0.075635	0.079759
randomly_perfect_random_0966	1.423751	1.091955	0.469989	1.090882	2.088416
randomly_perfect_random_0967	0.004177	0.004177	0	0.004177	0.004177
randomly_perfect_random_0968	0.093185	0.093114	0.008231	0.08314	0.1033
randomly_perfect_random_0969	0.016998	0.020037	0.004513	0.010618	0.020339
randomly_perfect_random_0970	0.04887	0.050217	0.0043	0.043061	0.053333
randomly_perfect_random_0971	0.095968	0.098589	0.005419	0.088421	0.100894
randomly_perfect_random_0972	0.091492	0.088752	0.010019	0.080823	0.104901
randomly_perfect_random_0973	0.074146	0.074584	0.003546	0.069601	0.078254
randomly_perfect_random_0974	0.03072	0.031412	0.001521	0.028611	0.032138
randomly_perfect_random_0975	0.74719	1.086212	0.480933	0.067049	1.088308
randomly_perfect_random_0976	0.043019	0.044267	0.004176	0.037395	0.047393
randomly_perfect_random_0977	0.080233	0.082292	0.005068	0.073258	0.085148
randomly_perfect_random_0978	0.067712	0.068591	0.010177	0.054833	0.079713

randomly_perfect_random_0979	0.045786	0.044923	0.002879	0.042772	0.049663
randomly_perfect_random_0980	0.057456	0.057896	0.005112	0.050987	0.063486
randomly_perfect_random_0981	0.04224	0.043677	0.003492	0.03743	0.045613
randomly_perfect_random_0982	1.090025	1.090765	0.001283	1.08822	1.091091
randomly_perfect_random_0983	0.070211	0.073871	0.005476	0.062471	0.074292
randomly_perfect_random_0984	1.407333	1.078197	0.469475	1.07254	2.071262
randomly_perfect_random_0985	0.39504	0.063118	0.472792	0.058337	1.063663
randomly_perfect_random_0986	0.381193	0.054922	0.470292	0.04241	1.046247
randomly_perfect_random_0987	1.023533	0.021158	1.417933	0.020647	3.028792
randomly_perfect_random_0988	0.013746	0.014984	0.00175	0.011271	0.014984
randomly_perfect_random_0989	0.081287	0.084944	0.00591	0.07295	0.085966
randomly_perfect_random_0990	0.74408	1.074945	0.469248	0.080464	1.076831
randomly_perfect_random_0991	0.056962	0.056419	0.002679	0.053987	0.060482
randomly_perfect_random_0992	0.015324	0.015324	1.73E-18	0.015324	0.015324
randomly_perfect_random_0993	0.058884	0.057557	0.005103	0.053403	0.06569
randomly_perfect_random_0994	0.057981	0.057385	0.001646	0.056329	0.060228
randomly_perfect_random_0995	0.070479	0.070643	0.002309	0.067574	0.073222
randomly_perfect_random_0996	0.006324	0.005442	0.001336	0.005319	0.008213
randomly_perfect_random_0997	0.395104	0.06247	0.470696	0.062073	1.060768
randomly_perfect_random_0998	1.089718	1.097983	0.822634	0.078095	2.093078
randomly_perfect_random_0999	0.048865	0.048952	0.000198	0.048591	0.049052
randomly_perfect_random_1000	0.065722	0.069491	0.006653	0.056371	0.071303

Table C.3: This table shows the results of the simulated annealing algorithm using the configuration found in this thesis on the randomly perfect generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
realistic_instance_01	1.101344	1.134725	0.049217	1.03176	1.137548
realistic_instance_02	1.241425	1.268523	0.045819	1.176902	1.278851
realistic_instance_03	1.059096	1.08214	0.041263	1.001144	1.094003
realistic_instance_04	1.072543	1.097858	0.03996	1.01613	1.103641
realistic_instance_05	1.203758	1.218353	0.02863	1.163754	1.229167
realistic_instance_06	1.12471	1.14411	0.027803	1.085392	1.144628
realistic_instance_07	1.241362	1.264685	0.043435	1.180488	1.278914
realistic_instance_08	1.195048	1.228737	0.048495	1.126469	1.229937
realistic_instance_09	1.16853	1.191608	0.03963	1.11276	1.201221
realistic_instance_10	1.190898	1.218058	0.039445	1.135122	1.219516
realistic_instance_11	1.093612	1.118947	0.03664	1.0418	1.12009
realistic_instance_12	1.165945	1.1917	0.039061	1.110746	1.19539
realistic_instance_13	1.062368	1.087339	0.040436	1.005329	1.094434
realistic_instance_14	2.204968	2.225414	0.03324	2.158086	2.231403
realistic_instance_15	0.99798	1.012709	0.027124	0.959941	1.021289
realistic_instance_16	1.202372	1.23109	0.043141	1.141395	1.23463
realistic_instance_17	1.037485	1.059153	0.034302	0.989064	1.064239
realistic_instance_18	1.099615	1.115671	0.03312	1.053482	1.129692
realistic_instance_19	0.648838	0.661096	0.042985	0.591144	0.694273
realistic_instance_20	0.709052	0.737539	0.049485	0.639451	0.750168
realistic_instance_21	0.749199	0.77192	0.036392	0.697847	0.777831
realistic_instance_22	0.566579	0.571399	0.007235	0.556352	0.571985
realistic_instance_23	0.634921	0.656509	0.032649	0.58878	0.659474
realistic_instance_24	0.990496	0.673715	0.496831	0.605748	1.692025
realistic_instance_25	0.516552	0.540327	0.037917	0.46304	0.54629
realistic_instance_26	1.757811	1.787146	0.865626	0.683276	2.80301
realistic_instance_27	0.646548	0.676043	0.047104	0.580074	0.683528

Table C.4: This table shows the results of the simulated annealing algorithm using the configuration found in this thesis on the realistic part of the test set. The values are calculated from three trials per instance.

C.2 Memetic Algorithm

instance	average	median	std.dev	min	max

randomly_generated_0951	0.515944	0.516406	0.001503	0.513916	0.51751
randomly_generated_0952	1.94858	1.946854	0.002558	1.946689	1.952197
randomly_generated_0953	0.336306	0.336525	0.000601	0.335486	0.336909
randomly_generated_0954	0.259011	0.259022	0.001647	0.256988	0.261023
randomly_generated_0955	0.125282	0.124945	0.00067	0.124684	0.126218
randomly_generated_0956	0.170736	0.170506	0.001168	0.169435	0.172268
randomly_generated_0957	0.67397	0.67235	0.002736	0.671737	0.677824
randomly_generated_0958	0.090792	0.090185	0.001108	0.089844	0.092347
randomly_generated_0959	0.119887	0.11986	0.000818	0.118898	0.120902
randomly_generated_0960	0.687569	0.688104	0.001206	0.685898	0.688704
randomly_generated_0961	6.496276	6.501339	0.81491	5.495697	7.491791
randomly_generated_0962	0.102884	0.102878	0.000245	0.102587	0.103187
randomly_generated_0963	0.9607	0.960668	0.000706	0.959852	0.96158
randomly_generated_0964	0.178444	0.178578	0.000228	0.178124	0.178632
randomly_generated_0965	0.139798	0.140772	0.001584	0.137564	0.141058
randomly_generated_0966	0.235358	0.235158	0.0019	0.233137	0.237779
randomly_generated_0967	6.344533	6.3472	0.815842	5.344004	7.342395
randomly_generated_0968	0.429716	0.429942	0.001533	0.427736	0.431471
randomly_generated_0969	5.440245	5.439694	0.002851	5.437062	5.44398
randomly_generated_0970	0.097177	0.097137	0.001095	0.095857	0.098538
randomly_generated_0971	3.868595	3.539358	0.466927	3.537498	4.528928
randomly_generated_0972	0.248392	0.247751	0.001208	0.247342	0.250085
randomly_generated_0973	0.220518	0.22009	0.000894	0.219701	0.221762
randomly_generated_0974	0.854724	0.854571	0.000419	0.854305	0.855297
randomly_generated_0975	0.10983	0.11039	0.002347	0.106717	0.112383
randomly_generated_0976	1.25403	1.284095	0.045397	1.189871	1.288125
randomly_generated_0977	1.397164	1.396317	0.001402	1.396035	1.39914
randomly_generated_0978	0.588278	0.587717	0.003462	0.584346	0.59277
randomly_generated_0979	2.999677	2.671986	0.46966	2.663187	3.663857
randomly_generated_0980	1.118104	1.117338	0.00117	1.117217	1.119757
randomly_generated_0981	2.527039	2.52913	0.004988	2.520159	2.531828
randomly_generated_0982	0.141653	0.141883	0.000999	0.140331	0.142746
randomly_generated_0983	12.17025	11.50414	0.943529	11.50202	13.5046
randomly_generated_0984	0.143066	0.142688	0.001758	0.141126	0.145383
randomly_generated_0985	0.059635	0.059048	0.001657	0.057964	0.061893
randomly_generated_0986	10.19814	9.498973	1.743694	8.499778	12.59567
randomly_generated_0987	0.064585	0.064214	0.000797	0.063848	0.065692
randomly_generated_0988	0.138458	0.140329	0.003981	0.132923	0.142122
randomly_generated_0989	0.799778	0.800469	0.002391	0.796566	0.8023
randomly_generated_0990	0.042249	0.043079	0.002708	0.038596	0.045071
randomly_generated_0991	0.366397	0.36807	0.004092	0.360763	0.370357
randomly_generated_0992	0.302013	0.302123	0.001408	0.300235	0.303679
randomly_generated_0993	4.464161	4.796417	0.469895	3.799628	4.796438
randomly_generated_0994	0.446648	0.446774	0.003564	0.442221	0.45095
randomly_generated_0995	2.29736	2.298762	0.002784	2.293474	2.299845
randomly_generated_0996	3.155733	2.824738	0.468692	2.823899	3.818563
randomly_generated_0997	6.002031	6.009767	0.818287	4.995992	7.000333
randomly_generated_0998	1.234945	1.235296	0.000549	1.23417	1.235371
randomly_generated_0999	1.259155	1.259385	0.000917	1.257935	1.260145
randomly_generated_1000	0.100422	0.099153	0.006077	0.093696	0.108419

Table C.5: This table shows the results of the memetic algorithm using the configuration found in this thesis on the randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
randomly_perfect_random_0951	0.085613	0.086731	0.004867	0.079172	0.090935
randomly_perfect_random_0952	0.074168	0.07428	0.0003	0.073758	0.074467
randomly_perfect_random_0953	0.051743	0.070207	0.026712	0.01397	0.071051
randomly_perfect_random_0954	0.038107	0.038431	0.000983	0.036774	0.039116
randomly_perfect_random_0955	0.001947	0.001388	0.000896	0.001243	0.003212
randomly_perfect_random_0956	0.073098	0.074	0.00194	0.070403	0.074891
randomly_perfect_random_0957	0.106661	0.108417	0.002694	0.102854	0.108711
randomly_perfect_random_0958	0.018397	0.018262	0.000678	0.017642	0.019286
randomly_perfect_random_0959	0.204319	0.206646	0.006039	0.196039	0.210271
randomly_perfect_random_0960	0.006737	0.007029	0.000412	0.006155	0.007029
randomly_perfect_random_0961	0.197406	0.196558	0.001436	0.196232	0.199428
randomly_perfect_random_0962	0.076432	0.076645	0.000831	0.075324	0.077326
randomly_perfect_random_0963	0.224463	0.222592	0.004967	0.219535	0.231263

randomly_perfect_random_0964	0.06061	0.060848	0.001588	0.058556	0.062425
randomly_perfect_random_0965	0.115619	0.11646	0.004115	0.110212	0.120186
randomly_perfect_random_0966	0.197974	0.196978	0.001902	0.196307	0.200635
randomly_perfect_random_0967	0	0	0	0	0
randomly_perfect_random_0968	0.051621	0.052643	0.001806	0.049083	0.053136
randomly_perfect_random_0969	0	0	0	0	0
randomly_perfect_random_0970	0.064923	0.065933	0.002994	0.060856	0.067979
randomly_perfect_random_0971	0.024713	0.023586	0.003199	0.021482	0.029071
randomly_perfect_random_0972	0.09121	0.091063	0.001792	0.089093	0.093475
randomly_perfect_random_0973	0.08606	0.085736	0.002015	0.083771	0.088674
randomly_perfect_random_0974	0.077007	0.077522	0.001769	0.074629	0.078868
randomly_perfect_random_0975	0.058252	0.058142	0.000453	0.05776	0.058854
randomly_perfect_random_0976	0.08925	0.089143	0.008781	0.07855	0.100058
randomly_perfect_random_0977	0.075765	0.075044	0.00155	0.074333	0.077918
randomly_perfect_random_0978	0.061364	0.061042	0.001965	0.059136	0.063915
randomly_perfect_random_0979	0.08542	0.085778	0.002016	0.082792	0.08769
randomly_perfect_random_0980	0.008293	0.006842	0.00215	0.006704	0.011333
randomly_perfect_random_0981	0.056884	0.055903	0.001819	0.055316	0.059435
randomly_perfect_random_0982	0.11933	0.153039	0.051218	0.046953	0.157997
randomly_perfect_random_0983	0.115416	0.115488	0.006097	0.107912	0.122847
randomly_perfect_random_0984	0.202052	0.201143	0.001533	0.200802	0.204212
randomly_perfect_random_0985	0.174095	0.174561	0.00258	0.170728	0.176995
randomly_perfect_random_0986	0.092498	0.090698	0.002953	0.090134	0.096661
randomly_perfect_random_0987	0.032967	0.032768	0.000513	0.032461	0.03367
randomly_perfect_random_0988	0	0	0	0	0
randomly_perfect_random_0989	0.057042	0.056115	0.002424	0.054648	0.060365
randomly_perfect_random_0990	0.070132	0.070239	0.000555	0.069405	0.070752
randomly_perfect_random_0991	0.093886	0.069697	0.034338	0.069514	0.142448
randomly_perfect_random_0992	0	0	0	0	0
randomly_perfect_random_0993	0.083172	0.08254	0.00304	0.079806	0.08717
randomly_perfect_random_0994	0.121853	0.120972	0.004818	0.116442	0.128146
randomly_perfect_random_0995	0.101716	0.101767	0.001211	0.100208	0.103172
randomly_perfect_random_0996	0.001659	0.001374	0.000876	0.000756	0.002846
randomly_perfect_random_0997	0.183985	0.187209	0.006989	0.174281	0.190465
randomly_perfect_random_0998	0.162661	0.16255	0.000388	0.162252	0.163182
randomly_perfect_random_0999	0.098893	0.098504	0.001028	0.097874	0.1003
randomly_perfect_random_1000	0.07079	0.070543	0.000637	0.070163	0.071663

Table C.6: This table shows the results of the memetic algorithm using the configuration found in this thesis on the randomly perfect generated part of the test set. The values are calculated from three trials per instance.

instance	**average**	**median**	**std.dev**	**min**	**max**
randomly_generated_small_0001	0.038242	0.042619	0.006212	0.029457	0.04265
randomly_generated_small_0002	0.48938	0.492745	0.008092	0.478226	0.49717
randomly_generated_small_0003	0.281947	0.281934	0.004287	0.276703	0.287204
randomly_generated_small_0004	0.364124	0.364448	0.001193	0.362528	0.365397
randomly_generated_small_0005	0.793137	0.793737	0.001571	0.790985	0.794689
randomly_generated_small_0006	0.716465	0.716398	9.57E-05	0.716398	0.716601
randomly_generated_small_0007	0.02048	0.019804	0.005099	0.014601	0.027036
randomly_generated_small_0008	0.911434	1.239706	0.466427	0.251809	1.242786
randomly_generated_small_0009	0.803744	0.80442	0.003142	0.799602	0.807209
randomly_generated_small_0010	0.749112	0.746131	0.005249	0.744715	0.75649

Table C.7: This table shows the results of the memetic algorithm using the configuration found in this thesis on the small randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
realistic_instance_01	1.046516	1.049296	0.004207	1.040571	1.049682
realistic_instance_02	1.187758	1.185297	0.004675	1.183674	1.194302
realistic_instance_03	1.003595	1.003767	0.000521	1.002888	1.004129
realistic_instance_04	1.032051	1.023103	0.01559	1.019074	1.053976
realistic_instance_05	1.188053	1.18685	0.005095	1.182502	1.194807
realistic_instance_06	1.106118	1.104435	0.003676	1.102699	1.111219
realistic_instance_07	1.211571	1.212041	0.00376	1.206749	1.215924
realistic_instance_08	1.13698	1.133449	0.005207	1.133149	1.144341
realistic_instance_09	1.125378	1.125354	0.002902	1.121836	1.128944
realistic_instance_10	1.139077	1.140684	0.003639	1.134039	1.142507
realistic_instance_11	1.068581	1.066731	0.005859	1.06251	1.076501
realistic_instance_12	1.13554	1.135822	0.002381	1.132492	1.138304
realistic_instance_13	1.007863	1.008407	0.001392	1.005952	1.009231
realistic_instance_14	2.494184	2.16071	0.472118	2.159983	3.161859
realistic_instance_15	0.962395	0.963717	0.002271	0.959199	0.96427
realistic_instance_16	1.16208	1.163288	0.005728	1.154539	1.168414
realistic_instance_17	1.017462	1.021065	0.005526	1.009654	1.021666
realistic_instance_18	1.071092	1.074773	0.00587	1.062808	1.075694
realistic_instance_19	0.575241	0.575197	0.001902	0.572934	0.577592
realistic_instance_20	0.624511	0.624254	0.002025	0.622169	0.62711
realistic_instance_21	0.696562	0.697258	0.001493	0.694488	0.697939
realistic_instance_22	0.5612	0.562426	0.005933	0.553398	0.567775
realistic_instance_23	0.563832	0.564126	0.003772	0.559073	0.568297
realistic_instance_24	0.581547	0.581644	0.003365	0.577377	0.585619
realistic_instance_25	0.443057	0.441618	0.003181	0.440084	0.447467
realistic_instance_26	0.664961	0.664892	0.003305	0.660948	0.669042
realistic_instance_27	0.560937	0.559987	0.00271	0.558195	0.564628

Table C.8: This table shows the results of the memetic algorithm using the configuration found in this thesis on the realistic part of the test set. The values are calculated from three trials per instance.

C.3 Genetic Algorithm

instance	average	median	std.dev	min	max
randomly_generated_0951	0.499349572	0.502514	0.010044	0.485776	0.509759
randomly_generated_0952	1.996945686	2.020572	0.035367	1.946954	2.023311
randomly_generated_0953	0.329161966	0.330441	0.002089	0.326216	0.330829
randomly_generated_0954	0.237958051	0.234388	0.011773	0.225659	0.253827
randomly_generated_0955	0.12591904	0.124732	0.00271	0.123357	0.129668
randomly_generated_0956	0.168461027	0.169223	0.001449	0.166432	0.169728
randomly_generated_0957	0.672990332	0.673795	0.002491	0.669617	0.675559
randomly_generated_0958	0.09005765	0.090259	0.002813	0.086516	0.093398
randomly_generated_0959	0.124399455	0.125295	0.001484	0.122307	0.125596
randomly_generated_0960	0.686968733	0.687074	0.001296	0.685331	0.688501
randomly_generated_0961	6.135057958	6.463896	0.465075	5.477343	6.463935
randomly_generated_0962	0.15335692	0.153375	0.000579	0.152639	0.154057
randomly_generated_0963	0.955889538	0.955836	0.000849	0.954877	0.956956

randomly_generated_0964	0.175790832	0.175459	0.00047	0.175458	0.176455
randomly_generated_0965	0.131876526	0.131663	0.000854	0.130953	0.133013
randomly_generated_0966	0.221125297	0.220921	0.00148	0.219423	0.223032
randomly_generated_0967	5.31789765	5.324661	0.825291	4.303762	6.32527
randomly_generated_0968	0.400075485	0.400927	0.001302	0.398236	0.401064
randomly_generated_0969	5.076265708	5.413582	0.480695	4.396466	5.418749
randomly_generated_0970	0.092835604	0.094165	0.002212	0.089718	0.094623
randomly_generated_0971	4.544601235	4.54981	0.81741	3.540888	5.543106
randomly_generated_0972	0.233643931	0.2338	0.002365	0.230672	0.236459
randomly_generated_0973	0.210316161	0.210922	0.001576	0.208156	0.211871
randomly_generated_0974	0.854112724	0.853989	0.002426	0.851205	0.857144
randomly_generated_0975	0.110627257	0.110294	0.000787	0.109874	0.111714
randomly_generated_0976	1.176189803	1.177379	0.003193	1.171822	1.179368
randomly_generated_0977	0.33558394	0.340466	0.007616	0.324829	0.341457
randomly_generated_0978	0.559599315	0.556713	0.010043	0.548999	0.573086
randomly_generated_0979	2.32230257	2.654797	0.472464	1.654141	2.65797
randomly_generated_0980	1.140093356	1.112325	0.039489	1.112016	1.195939
randomly_generated_0981	2.501721831	2.503855	0.81844	1.498276	3.503034
randomly_generated_0982	0.137546514	0.137559	0.000346	0.137117	0.137964
randomly_generated_0983	11.12373501	11.45467	1.252012	9.451893	12.46464
randomly_generated_0984	0.142135306	0.140146	0.003613	0.139055	0.147205
randomly_generated_0985	0.08472753	0.084781	0.001454	0.082921	0.086481
randomly_generated_0986	9.816346729	10.48414	0.945092	8.479785	10.48512
randomly_generated_0987	0.061611811	0.061836	0.000696	0.06067	0.062329
randomly_generated_0988	0.123314631	0.123752	0.000721	0.122298	0.123894
randomly_generated_0989	0.793765725	0.79306	0.001091	0.79293	0.795307
randomly_generated_0990	0.08130432	0.081785	0.001172	0.07969	0.082438
randomly_generated_0991	0.362953683	0.363902	0.001754	0.360494	0.364465
randomly_generated_0992	0.295397014	0.29524	0.000461	0.294928	0.296023
randomly_generated_0993	4.455309521	4.788225	0.472084	3.787683	4.79002
randomly_generated_0994	0.406078918	0.4086	0.006711	0.396895	0.412742
randomly_generated_0995	2.561994922	2.25942	0.443374	2.237671	3.188894
randomly_generated_0996	3.115861697	2.812536	0.43686	2.801408	3.733641
randomly_generated_0997	6.981019169	6.978928	0.819099	5.978879	7.98525
randomly_generated_0998	1.211331085	1.211116	0.004606	1.2058	1.217077
randomly_generated_0999	1.239461666	1.239736	0.003932	1.234514	1.244135
randomly_generated_1000	0.105316917	0.106269	0.001658	0.102985	0.106696

Table C.9: This table shows the results of the genetic algorithm using the configuration found in this thesis on the randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
randomly_perfect_random_0951	0.070893316	0.071155	0.000877	0.069712	0.071813
randomly_perfect_random_0952	0.070626356	0.071772	0.002445	0.067229	0.072878
randomly_perfect_random_0953	0.045976891	0.049774	0.007228	0.035859	0.052297
randomly_perfect_random_0954	0.038711068	0.039806	0.002525	0.03522	0.041107
randomly_perfect_random_0955	0.004316969	0.004416	0.000203	0.004034	0.004501
randomly_perfect_random_0956	0.044296026	0.046534	0.005423	0.036824	0.04953
randomly_perfect_random_0957	0.087132151	0.086399	0.003842	0.082836	0.092162
randomly_perfect_random_0958	0.018987209	0.019042	0.000226	0.018687	0.019233
randomly_perfect_random_0959	0.170407569	0.169769	0.002803	0.167339	0.174115
randomly_perfect_random_0960	0.006971984	0.007039	0.00046	0.006379	0.007499
randomly_perfect_random_0961	0.088193258	0.087446	0.00526	0.082158	0.094976
randomly_perfect_random_0962	0.092318764	0.095651	0.017541	0.069364	0.111942
randomly_perfect_random_0963	0.198116666	0.199884	0.003628	0.193062	0.201404
randomly_perfect_random_0964	0.049332881	0.050974	0.003286	0.044747	0.052278
randomly_perfect_random_0965	0.089248947	0.087631	0.002455	0.087398	0.092718
randomly_perfect_random_0966	0.105145519	0.1056	0.00149	0.103136	0.1067
randomly_perfect_random_0967	0	0	0	0	0
randomly_perfect_random_0968	0.039302621	0.038874	0.000827	0.038574	0.04046
randomly_perfect_random_0969	7.15001E-05	0	0.000101	0	0.000215
randomly_perfect_random_0970	0.058105919	0.058427	0.002553	0.054831	0.06106
randomly_perfect_random_0971	0.025976064	0.022314	0.006578	0.020402	0.035212
randomly_perfect_random_0972	0.081317337	0.080755	0.002169	0.078987	0.08421
randomly_perfect_random_0973	0.067177045	0.067313	0.001961	0.064711	0.069507
randomly_perfect_random_0974	0.069591904	0.069759	0.001085	0.068187	0.07083
randomly_perfect_random_0975	0.061316887	0.053545	0.012852	0.050973	0.079432
randomly_perfect_random_0976	0.149605255	0.149341	0.000422	0.149275	0.1502

randomly_perfect_random_0977	0.076908356	0.078205	0.004165	0.071284	0.081236
randomly_perfect_random_0978	0.057533498	0.056983	0.000901	0.056814	0.058804
randomly_perfect_random_0979	0.130271487	0.131892	0.002327	0.126981	0.131941
randomly_perfect_random_0980	0.012026204	0.01053	0.00359	0.008574	0.016975
randomly_perfect_random_0981	0.053443243	0.052448	0.002682	0.050771	0.057111
randomly_perfect_random_0982	0.110026388	0.112714	0.00712	0.100279	0.117086
randomly_perfect_random_0983	0.09627884	0.094228	0.003394	0.093547	0.101062
randomly_perfect_random_0984	0.17250318	0.172867	0.00234	0.169473	0.17517
randomly_perfect_random_0985	0.150154256	0.150865	0.001191	0.148476	0.151121
randomly_perfect_random_0986	0.077859725	0.076976	0.001437	0.076717	0.079886
randomly_perfect_random_0987	0.031568185	0.031959	0.000817	0.030431	0.032315
randomly_perfect_random_0988	0	0	0	0	0
randomly_perfect_random_0989	0.061633954	0.061186	0.001893	0.059572	0.064144
randomly_perfect_random_0990	0.063183474	0.061638	0.004956	0.058036	0.069876
randomly_perfect_random_0991	0.065855613	0.064302	0.003383	0.062714	0.070551
randomly_perfect_random_0992	0	0	0	0	0
randomly_perfect_random_0993	0.135880063	0.134855	0.001744	0.134449	0.138336
randomly_perfect_random_0994	0.099166243	0.098578	0.002753	0.096128	0.102793
randomly_perfect_random_0995	0.089363961	0.088269	0.001992	0.087664	0.092159
randomly_perfect_random_0996	0.004121399	0.004295	0.000523	0.003412	0.004657
randomly_perfect_random_0997	0.153618076	0.151572	0.003182	0.15117	0.158112
randomly_perfect_random_0998	0.129101368	0.124605	0.017663	0.11007	0.152629
randomly_perfect_random_0999	0.084050445	0.084025	0.002166	0.081411	0.086716
randomly_perfect_random_1000	0.061021092	0.059878	0.002104	0.059214	0.063971

Table C.10: This table shows the results of the genetic algorithm using the configuration found in this thesis on the randomly perfect generated part of the test set. The values are calculated from three trials per instance.

instance	**average**	**median**	**std.dev**	**min**	**max**
randomly_generated_small_0001	0.038028301	0.037751	0.00517	0.03184	0.044494
randomly_generated_small_0002	0.492090472	0.489777	0.007796	0.483912	0.502582
randomly_generated_small_0003	0.285103649	0.283132	0.003254	0.282489	0.28969
randomly_generated_small_0004	0.358161034	0.35812	0.000177	0.357968	0.358395
randomly_generated_small_0005	0.791647403	0.79213	0.000962	0.790304	0.792508
randomly_generated_small_0006	0.716388368	0.716601	0.001738	0.714162	0.718403
randomly_generated_small_0007	0.022712768	0.022272	0.000953	0.02183	0.024036
randomly_generated_small_0008	1.232759731	1.236972	0.006099	1.224135	1.237172
randomly_generated_small_0009	0.804220411	0.8	0.009123	0.795772	0.816889
randomly_generated_small_0010	0.752181854	0.749032	0.004961	0.748328	0.759186

Table C.11: This table shows the results of the genetic algorithm using the configuration found in this thesis on the small randomly generated part of the test set. The values are calculated from three trials per instance.

instance	average	median	std.dev	min	max
realistic_instance_01	1.044595781	1.043579	0.008595	1.034614	1.055594
realistic_instance_02	1.194188491	1.195786	0.002803	1.190248	1.196531
realistic_instance_03	1.005891023	1.006899	0.001746	1.003435	1.00734
realistic_instance_04	1.025971325	1.023787	0.00372	1.022918	1.031208
realistic_instance_05	1.183083917	1.183724	0.003848	1.178084	1.187444
realistic_instance_06	1.106003358	1.105925	0.000315	1.105662	1.106422
realistic_instance_07	1.209604695	1.208791	0.001666	1.208096	1.211927
realistic_instance_08	1.137826358	1.139783	0.002987	1.133606	1.14009
realistic_instance_09	1.126811279	1.127586	0.002496	1.123441	1.129407
realistic_instance_10	1.137951553	1.135953	0.005886	1.131953	1.145949
realistic_instance_11	1.0700808	1.069545	0.001455	1.068628	1.072069
realistic_instance_12	1.132480111	1.136164	0.005412	1.124828	1.136448
realistic_instance_13	1.008857766	1.008423	0.002085	1.00655	1.0116
realistic_instance_14	2.161467026	2.15999	0.004199	2.157225	2.167187
realistic_instance_15	0.959600558	0.961847	0.006876	0.950284	0.966671
realistic_instance_16	1.154238208	1.151712	0.004995	1.149787	1.161215
realistic_instance_17	1.009883333	1.012789	0.004782	1.003142	1.013718
realistic_instance_18	1.072658925	1.072678	0.001131	1.071264	1.074034
realistic_instance_19	0.57715469	0.576793	0.000695	0.576544	0.578127
realistic_instance_20	0.626174719	0.626568	0.004661	0.62028	0.631676
realistic_instance_21	0.696304963	0.696946	0.000913	0.695014	0.696955
realistic_instance_22	0.541188627	0.540847	0.004031	0.536431	0.546288
realistic_instance_23	0.547575236	0.545415	0.006696	0.540671	0.55664
realistic_instance_24	0.566817423	0.566748	0.009015	0.555812	0.577893
realistic_instance_25	0.417037275	0.423903	0.01676	0.393958	0.433251
realistic_instance_26	0.660432755	0.659894	0.000917	0.65968	0.661724
realistic_instance_27	0.554433419	0.553983	0.001263	0.553162	0.556155

Table C.12: This table shows the results of the genetic algorithm using the configuration found in this thesis on the realistic part of the test set. The values are calculated from three trials per instance.

List of Figures

List of Tables

Bibliography

[Ant89] Jim Antonisse. A new interpretation of schema notation that overtums the binary encoding constraint. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 86–91. Morgan Kaufmann, 1989.

[Bak85] James E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, page 101–111, USA, 1985. L. Erlbaum Associates Inc.

[Bay86] Ilker Baybars. A survey of exact algorithms for the simple assembly line balancing problem. *Management Science*, 32(8):909–932, 1986.

[BBM93a] D Beasley, DR Bull, and RR Martin. An overview of genetic algorithms: Pt 2, research topics. *University Computing*, 15/4:170 – 181, 1993.

[BBM93b] David Beasley, David R. Bull, and Ralph Robert Martin. An overview of genetic algorithms: Pt1, fundamentals. *University Computing archive*, 15:58–69, 1993.

[Ber19] Michal Bereta. Baldwin effect and lamarckian evolution in a memetic algorithm for euclidean steiner tree problem. *Memetic Comput.*, 11(1):35–52, 2019.

[BFS07] Nils Boysen, Malte Fliedner, and Armin Scholl. A classification of assembly line balancing problems. *Eur. J. Oper. Res.*, 183(2):674–693, 2007.

[BGK04] Edmund K. Burke, Steven M. Gustafson, and Graham Kendall. Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Trans. Evol. Comput.*, 8(1):47–62, 2004.

[CM01] Carlos Castro and Sebastian Manzano. Variable and value ordering when solving balanced academic curriculum problems. *CoRR*, cs.PL/0110007, 2001.

[Con91] G. V. Conroy. *Handbook of genetic algorithms* by lawrence davis (ed.), chapman & hall, london, 1991, pp 385, £32.50. *Knowl. Eng. Rev.*, 6(4):363–365, 1991.

[CYTH14] Cheng Chen, Zhenyu Yang, Yuejin Tan, and Renjie He. Diversity controlling genetic algorithm for order acceptance and scheduling problem. *Mathematical Problems in Engineering*, 2014:367152, Feb 2014.

[Dav90] Yuval Davidor. Epistasis variance: Suitability of a representation to genetic algorithms. *Complex Syst.*, 4(4), 1990.

[ECS89] Larry J. Eshelman, Rich Caruana, and J. David Schaffer. Biases in the crossover landscape. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 10–19. Morgan Kaufmann, 1989.

[FL10] Yongsheng Fang and Jun Li. A review of tournament selection in genetic programming. In Zhihua Cai, Chengyu Hu, Zhuo Kang, and Yong Liu, editors, *Advances in Computation and Intelligence - 5th International Symposium, ISICA 2010, Wuhan, China, October 22-24, 2010. Proceedings*, volume 6382 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2010.

[GD90] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990*, pages 69–93. Morgan Kaufmann, 1990.

[GH88] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Mach. Learn.*, 3:95–99, 1988.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[Gol91] David E. Goldberg. Real-coded genetic algorithms, virtual alphabets, and blocking. *Complex Syst.*, 5(2), 1991.

[Gre86] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern.*, 16(1):122–128, 1986.

[HB91] Christopher L. Huntley and Donald E. Brown. A parallel heuristic for quadratic assignment problems. *Comput. Oper. Res.*, 18(3):275–289, 1991.

[HH08] Kassel Hingee and Marcus Hutter. Equivalence of probabilistic tournament and polynomial ranking selection. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China*, pages 564–571. IEEE, 2008.

[HLV98] Francisco Herrera, Manuel Lozano, and José L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artif. Intell. Rev.*, 12(4):265–319, 1998.

[Hol75] John H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. Univ. of Michigan Pr., Ann Arbor, Mich, 1975.

[JD96] Márk Jelasity and József Dombi. Implicit formae in genetic algorithms. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV, International Conference on Evolutionary Computation. The 4th International Conference on Parallel Problem Solving from Nature, Berlin, Germany, September 22-26, 1996, Proceedings*, volume 1141 of *Lecture Notes in Computer Science*, pages 154–163. Springer, 1996.

[Jon85] Kenneth A. De Jong. Genetic algorithms: A 10 year perspective. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985*, pages 169–177. Lawrence Erlbaum Associates, 1985.

[JS90] Kenneth A. De Jong and William M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature, 1st Workshop, PPSN I, Dortmund, Germany, October 1-3, 1990, Proceedings*, volume 496 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 1990.

[KCK21] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multim. Tools Appl.*, 80(5):8091–8126, 2021.

[Koz93] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.

[LC18] Beatrice Luca and Mitica Craus. Local search algorithms for memetic algorithms: understanding behaviors using biological intelligence. 07 2018.

[LCM+05] Tony Lambert, Carlos Castro, Éric Monfroy, María Cristina Riff, and Frédéric Saubion. Hybridization of genetic algorithms and constraint propagation for the BACP. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668 of *Lecture Notes in Computer Science*, pages 421–423. Springer, 2005.

[LCMS06] Tony Lambert, Carlos Castro, Éric Monfroy, and Frédéric Saubion. Solving the balanced academic curriculum problem with an hybridization of genetic algorithm and constraint propagation. In Leszek Rutkowski, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada, editors, *Artificial Intelligence and Soft Computing - ICAISC 2006, 8th International Conference, Zakopane, Poland, June 25-29, 2006, Proceedings*, volume 4029 of *Lecture Notes in Computer Science*, pages 410–419. Springer, 2006.

[LEF+22] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022.

[LPR+13] José Miguel León, Wenceslao Palma, Nibaldo Rodriguez, Ricardo Soto, Broderick Crawford, Fernando Paredes, and Guillermo Cabrera-Guerrero. Solving the balanced academic curriculum problem using the aco metaheuristic. *Mathematical Problems in Engineering*, Volume 2013:1–8, 2013.

[LV19] Marie-Louise Lackner and Johannes Vass. Extended complexity results for the production leveling problem. Technical Report CD-TR 2019/2, 2019.

[ML02] C. Mullinax and Mark Lawley. Assigning patients to nurses in neonatal intensive care. *J. Oper. Res. Soc.*, 53(1):25–35, 2002.

[Mos89] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P Report 826, California Institute of Technology, 1989.

[Müh89] Heinz Mühlenbein. Parallel genetic algorithms, population genetics, and combinatorial optimization. In Jörg D. Becker, Ignaz Eisele, and Friedhelm Mündemann, editors, *Parallelism, Learning, Evolution, Workshop on Evolutionary Models and Strategies, Neubiberg, Germany, March 10-11, 1989, Workshop on Parallel Processing: Logic, Organization, and Technology - WOPPLOT 89, Wildbad Kreuth, Germany, July 24-28, 1989*, volume 565 of *Lecture Notes in Computer Science*, pages 398–406. Springer, 1989.

[Müh92] Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In Reinhard Männer and Bernard Manderick, editors, *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*, pages 15–26. Elsevier, 1992.

[NC12] Ferrante Neri and Carlos Cotta. Memetic algorithms and memetic computing optimization: A literature review. *Swarm Evol. Comput.*, 2:1–14, 2012.

[NOK07] Quang Huy Nguyen, Yew-Soon Ong, and Natalio Krasnogor. A study on the design issues of memetic algorithm. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2007, 25-28 September 2007, Singapore*, pages 2390–2397. IEEE, 2007.

[Poo09] Garg Poonam. A comparison between memetic algorithm and genetic algorithm for the cryptanalysis of simplified data encryption standard algorithm. *International Journal of Network Security & Its Applications*, 1, 04 2009.

[Rad91] Nicholas J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Syst.*, 5(2), 1991.

[Rad92] Nicholas J. Radcliffe. Non-linear genetic representations. In Reinhard Männer and Bernard Manderick, editors, *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*, pages 261–270. Elsevier, 1992.

[SG90] Alan C. Schultz and John J. Grefenstette. Improving tactical plans with genetic algorithms. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence, TAI 1990, Herndon, VA, USA, November 6-9, 1990*, pages 328–334. IEEE Computer Society, 1990.

[SHR09] Pierre Schaus, Pascal Van Hentenryck, and Jean-Charles Régin. Scalable load balancing in nurse to patient assignment problems. In Willem Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*, volume 5547 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2009.

[Spe92] William M. Spears. Crossover or mutation? In L. Darrell Whitley, editor, *Proceedings of the Second Workshop on Foundations of Genetic Algorithms. Vail, Colorado, USA, July 26-29 1992*, pages 221–237. Morgan Kaufmann, 1992.

[Spe00] William M Spears. *Evolutionary algorithms : the role of mutation and recombination ; with 23 tables.* Natural computing series. Springer, Berlin [u.a.], 2000.

[Sud18] Dirk Sudholt. The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses. *CoRR*, abs/1801.10087, 2018.

[SV98] Jim Smith and Frank Vavak. Replacement strategies in steady state genetic algorithms: Static environments. In Wolfgang Banzhaf and Colin R. Reeves, editors, *Proceedings of the Fifth Workshop on Foundations of Genetic Algorithms, Amsterdam, The Netherlands, September 24-28, 1998*, pages 219–234. Morgan Kaufmann, 1998.

[Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 2–9. Morgan Kaufmann, 1989.

[Thi97] Dirk Thierens. Selection schemes, elitist recombination, and selection intensity. In Thomas Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms, East Lansing, MI, USA, July 19-23, 1997*, pages 152–159. Morgan Kaufmann, 1997.

[TS93] David M. Tate and Alice E. Smith. Expected allele coverage and the role of mutation in genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, Urbana-Champaign, IL, USA, June 1993*, pages 31–37. Morgan Kaufmann, 1993.

[VC99] Kanta Vekaria and Chris Clack. Biases introduced by adaptive recombination operators. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1, pages 670–677. Morgan Kaufmann, 1999.

[VLM20] Johannes Vass, Marie-Louise Lackner, and Nysret Musliu. Exact and metaheuristic approaches for the production leveling problem. *CoRR*, abs/2006.08731, 2020.

[VLM+22] Johannes Vass, Marie-Louise Lackner, Christoph Mrkvicka, Nysret Musliu, and Felix Winter. Exact and meta-heuristic approaches for the production leveling problem. *J. Sched.*, 25(3):339–370, 2022.

[VMW20] Johannes Vass, N. Musliu, and Felix Winter. Solving the production leveling problem with order-splitting and resource constraints. In *13th International Conference on the Practice and Theory of Automated Timetabling - PATAT 2021*, pages 261–284, 2020.

[Whi89] L. Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 116–123. Morgan Kaufmann, 1989.

[WM10] Magdalena Widl and Nysret Musliu. An improved memetic algorithm for break scheduling. In Maria J. Blesa, Christian Blum, Günther R. Raidl, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics - 7th International Workshop, HM 2010, Vienna, Austria, October 1-2, 2010. Proceedings*, volume 6373 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2010.

[WMW21] Wolfgang Weintritt, Nysret Musliu, and Felix Winter. Solving the paintshop scheduling problem with memetic algorithms. In Francisco Chicano and Krzysztof Krawiec, editors, *GECCO '21: Genetic and Evolutionary Computation Conference, Lille, France, July 10-14, 2021*, pages 1070–1078. ACM, 2021.

[Zhu03] Kenny Qili Zhu. A diversity-controlling adaptive genetic algorithm for the vehicle routing problem with time windows. In *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA*, pages 176–183. IEEE Computer Society, 2003.