FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Exact and Metaheuristic Approaches for the Production Leveling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Johannes Vass, Bsc.
Matrikelnummer 01327476

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 3. Oktober 2019

Johannes Vass	Nysret Musliu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Exact and Metaheuristic Approaches for the Production Leveling Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Johannes Vass, Bsc.
Registration Number 01327476

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 3rd October, 2019

Johannes Vass	Nysret Musliu

Erklärung zur Verfassung der Arbeit

Johannes Vass, 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, 3. Oktober 2019

Johannes Vass

Acknowledgements

I would like to express my sincere gratitude to my advisor Priv.-Doz. Dr. Nysret Musliu for his continuous and intensive support throughout the whole process of research and writing. Furthermore, I want to thank my colleague Dr. Marie-Louise Lackner, with whom I started researching the Production Leveling Problem together earlier this year. I appreciate the excellent collaboration very much! Last but not least, I would like to thank my colleague Felix Winter for providing valuable feedback many times throughout the whole research process and answering lots of questions.

The work was supported by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development though the Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling. Further financial support was given by the company MCP, the industry partner of the CD Laboratory. I would like to thank Christoph Mrkvicka, our contact person at MCP, for the good cooperation and lots of interesting insights into the production planning and scheduling industry.

Kurzfassung

Im Rahmen dieser Diplomarbeit wird ein neues Problem im Gebiet der mittelfristigen Produktionsplanung eingeführt – das Production Leveling Problem. Dabei besteht die Aufgabe darin, Aufträge zu Produktionsperioden so zuzuteilen, dass die Auslastung der unterschiedlichen Perioden und Produktionsressourcen ausbalanciert wird. Außerdem sollen die Aufträge möglichst in jener Reihenfolge eingeplant werden, die von deren Priorität bestimmt wird. Das Production Leveling Problem ist ein wichtiger Zwischenschritt im mehrstufigen Prozess der Produktionsplanung, weil es dafür sorgt, dass möglichst ausgewogene Auftragsgruppen zum darauffolgenden Scheduling-Schritt gelangen.

In dieser Arbeit wird zunächst ein formales Modell des Problems dargestellt und dessen Komplexität theoretisch analysiert. Als exakte Lösungsverfahren werden Mixed-Integer-Programming und Constraint-Programming untersucht, welche optimale Lösungen für kleine Instanzen sowie untere Schranken beweisen sollen. Um auch große Probleminstanzen lösen zu können, werden in Folge Verfahren auf der Basis von metaheuristischer lokaler Suche erkundet. Dafür warden eine Greedy-Heuristik und zwei Nachbarschaftsstrukturen für die Lokale Suche eingeführt und die Verfahren Variable Neighborhood Descent und Simulated Annealing untersucht. Zur Evaluierung wird ein Set von realistischen Probleminstanzen eines Industriepartners veröffentlicht, sowie zwei zufallsbasierte Instanzgeneratoren. Die Forschungsfrage bezüglich der exakten Methoden ist, wie große Instanzen in einem fixen Zeitraum lösbar sind. Für die metaheuristischen Ansätze soll gezeigt werden, dass kleine Probleminstanzen annähernd optimal gelöst werden können und die Skalierbarkeit bis hin zu sehr großen Instanzen ebenfalls gegeben ist.

Die wichtigsten theoretischen Ergebnisse dieser Arbeit stellen das formale Problemmodell sowie ein Beweis der NP-hardness dar, welche mittels Reduktion von Bin-Packing gezeigt wird. Die Evaluierung der Lösungsmethoden kommt zu dem Resultat, dass mithilfe von Mixed-Integer-Programming Instanzen mit bis zu 250 Aufträgen fast immer gelöst werden können, während bei größeren Instanzen meist keine Lösung gefunden wird. Von den untersuchten metaheuristischen Verfahren produziert Simulated Annealing die besten Resultate. Es wird gezeigt, dass kleine Instanzen, für die untere Schranken bewiesen werden konnten, mit durschnittlich weniger als 3% Optimality Gap gelöst werden können. Auf der anderen Seite sind aber auch die größten Instanzen mit tausenden Aufträgen und Dutzenden von Produktionsperioden in der Regel gut lösbar. Die vorgestellten metaheuristischen Verfahren wurden bereits in der Industrie zum Einsatz gebracht.

Abstract

This thesis introduces the Production Leveling Problem, which is a new problem in the field of mid-term production planning. The task is to assign orders to production periods such that the workload in each period and on each production resource is balanced, capacity limits are not exceeded and the order's priorities are taken into account. The Production Leveling Problem is an important intermediate step between long-term planning and the final scheduling of orders within a production period, as it responsible for selecting good subsets of orders to be scheduled within each period.

A formal model for the Production Leveling Problem is proposed and the theoretical complexity is analyzed. Mixed Integer Programming and Constraint Programming are investigated as exact methods for solving moderately sized instances of the problem. In order to be able to solve also large problem instances, metaheuristic local search is investigated. A greedy heuristic and two neighborhood structures for local search are proposed, in order to apply them using Variable Neighborhood Descent and Simulated Annealing. A set of realistic problem instances from an industrial partner is contributed to the literature, as well as random instance generators and the instances which were generated for evaluation. Regarding exact techniques, the main question of research is, how large instances can be while still being solvable within a fixed amount of time. For the metaheuristic approaches the aim is to show that they produce near-optimal solution for smaller instances, but also scale well to very large instances.

The main theoretical results of this thesis are a formal model of the Production Leveling Problem and the proof of NP-hardness by reduction from Bin Packing. The experimental evaluation conveys that the proposed MIP model works well for instances with up to 250 orders, but soon hits a glass ceiling if they get larger. Out of the investigated metaheuristic approaches, Simulated Annealing achieves the best results. It is shown to produce solutions with less than 3% average optimality gap on small instances and to scale well up to thousands of orders and dozens of periods and products. The presented metaheuristic methods are already being actively used in the industry.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Aims of the thesis 2
1.2 Contributions 2
1.3 Structure of the thesis 3

2 Problem Statement and Related Work **5**
2.1 Problem Statement 5
2.2 Related Work 10

3 Complexity Analysis **13**
3.1 NP-hardness of the Production Leveling Problem 13
3.2 Fixed-Order Production Leveling 15

4 Solution Approaches **21**
4.1 Exact modeling techniques 22
4.2 Metaheuristic local search 28

5 Experimental Evaluation **35**
5.1 Problem Instances 35
5.2 Experimental Setting 38
5.3 Evaluation of exact modeling techniques 40
5.4 Evaluation of metaheuristic local search 45

6 Conclusion **55**

A Computational results of exact modeling techniques **57**

B Computational results of metaheuristic local search **61**

List of Figures **65**

List of Tables **67**

List of Algorithms **69**

Acronyms **71**

Bibliography **73**

CHAPTER 1

Introduction

Production systems are being subject to continuous and radical change in the course of the last decades. The need for productivity improvements is provoking companies to invest heavily in automation on all levels. Production planning and scheduling play a major role in these developments as the replacement of manual work with software-assisted or even autonomous systems can lead to considerable efficiency increases.

We introduce a new problem in the field of production planning which arises from the needs of an industrial partner. It is a combinatorial optimization problem which treats the leveling of production – thus we coin the name Production Leveling Problem (PLP). Stated briefly, it is a medium-term planning problem, i.e. it is intended to be embedded between the long-term planning and the scheduling of the concrete production sequence.

The problem is concerned with assigning orders of certain product types and demand sizes to production periods such that the production volume of each product type is leveled across all periods. Furthermore, the overall amount produced in each period is subject to leveling as well. A solution is feasible if the production volumes to be leveled do not exceed given maximum values. The optimization part consists in minimizing the deviation of the production from the optimal balance, while at the same time making sure that the orders are assigned approximately in the order of their priorities. The idea behind this goal is that considering orders only in the order of decreasing priority, as it is often done, frequently leads to spikes and idle times for certain resources involved in the production process. Leveling these highs and lows results in a smoother production process because a similar product mix is produced in every period. It is important to note, that the solution to the PLP is not a schedule since the orders are only assigned to production periods but the concrete execution sequence and assignment to machines and workers is not part of this problem. The intention is rather so serve as a step between long-term planning and short-term production scheduling.

A variety of optimization problems with a similar balancing objective have been investigated in the past. For example, the Balanced Academic Curriculum Problem (BACP) is concerned with creating curricula so that the student's workload is balanced between the terms [Chi+12], which is essentially the same as leveling the production between production periods. An extensive list of similar balancing problems can be found in the thesis of Pierre Schaus [Sch09]. Usual solution methods for these problems are Constraint Programming (CP) or Mixed Integer Programming (MIP) models and metaheuristic local search techniques. However, there does not seem to exist previous work which combines balancing goals with priorities in the way that we are confronted with in the PLP. The problems in the literature use to enforce some precedence relations as a hard constraint, but in the PLP we have the priorities as a soft constraint, which makes a direct comparison of results impossible.

1.1 Aims of the thesis

The main goals of this work are:

- Provide a formal model for the PLP.
- Determine the theoretical complexity of the problem and investigate tractable cases.
- Investigate exact modeling techniques in order to obtain optimal solutions for moderately sized instances and lower bounds.
- Develop metaheuristic local search strategies which scale to large problem instances.
- Evaluation of the proposed solution methods.

1.2 Contributions

The main contributions of this thesis are:

- A mathematical model for the PLP is provided.
- The associated decision problem is proven to lie in the class of NP-complete problems and thus the optimization problem is NP-hard. Furthermore, the Fixed-Order PLP is introduced and shown to be solvable in polynomial time by providing a dynamic programming algorithm.
- Two exact models are provided: a CP model and a MIP model,
- Two neighborhood structures for local search are introduced and applied using Variable Neighborhood Descent (VND) and Simulated Annealing.
- Realistic problem instances and two random instance generators are provided to the literature.

- We conducted an extensive experimental evaluation and comparison of the proposed techniques. The results show that Simulated Annealing is the best-performing of our solution methods because it produces excellent results on small instances and scales also very well to the size of our real-life instances and even larger. Furthermore, we show that the MIP model outperforms the CP model, which can solve well most of the realistic instances but not the largest ones.
- The metaheuristic local search methods, which were devised in this work, are already being successfully employed in the electronics industry.

1.3 Structure of the thesis

The remainder of the thesis is structured as follows:

Chapter 2 presents the problem statement and a more detailed review of related work in the literature.

Chapter 3 provides the analysis of theoretical complexity of the PLP, yielding the NP-hardness result. Additionally, tractable cases of the problem are analyzed, which leads to the introduction of the Fixed-Order PLP and a dynamic programming algorithm.

In Chapter 4 we turn towards solution approaches. Section 4.1 investigates exact modeling techniques, namely CP and MIP. In Section 4.2 we turn towards local search methods and describe algorithms and neighborhood operators.

Chapter 5 finally presents an experimental evaluation of all mentioned solution methods. We put special emphasis on investigating which instances can be solved using exact methods and where the strengths of the metaheurisic methods come into play. As we have no instances from the literature, we present also random instance generation techniques.

CHAPTER 2

Problem Statement and Related Work

The Production Leveling Problem is a new combinatorial optimization problem which we contribute to the scientific community. In this chapter we will present an introduction and formal definition. Afterwards we summarize related work which has been published in the literature and highlight the similarities and differences of the investigated problems to the PLP.

2.1 Problem Statement

We first want introduce by means of examples how the problem looks like and how the objectives and constraints work. Afterwards a formal definition is presented.

2.1.1 Informal problem statement

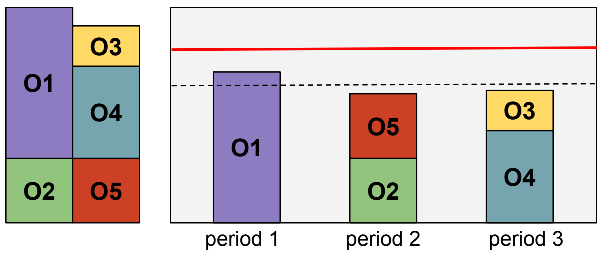
The input to the PLP are a list of orders, each of them having a demand value, priority and product type. Furthermore, we are given a set of periods and the maximum production capacity per period, both for all product types together and for each one separately. An optimal solution to the problem is an assignment of orders to periods, such that the production volume is balanced while trying to stick to the sequence implied by the order's priorities as good as possible. We can see the objective function of the PLP as the task of finding a good tradeoff between the following goals:

1. Minimize the deviation of the planned production volume to the average demand (i.e. the target value) for each period, ignoring the product types. This makes sure that the overall production per period is being leveled.

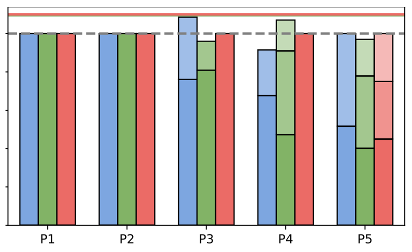
2. Minimize the sum of deviations of the production load of each product type to its respective mean (target) value, making sure that the production of each product type is being leveled.
3. Minimize the number of times a higher prioritized order is scheduled in a later period than a lower prioritized order, which we call a priority inversion. This objective makes sure that more important orders are scheduled in earlier periods.

Figure 2.1 visualizes the three optimization goals by example. The plots represent different views on the solution, each corresponding to one sub-objective:

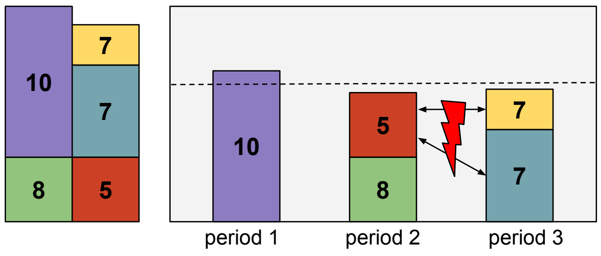
1. Figure 2.1a shows a tiny example instance with five orders, which are shown as boxes, where the box height corresponds to the order size. The orders should be assigned to the three periods such that the distances of the stacks of orders and the dashed target line is minimized and no stack crosses the red line wich signalizes the capacity limit. It is easy to see that this solution is optimal w.r.t. the leveling objective.
2. Figure 2.1b shows a slightly larger problem instance which has three product types (blue, green and red). It visualizes the solution from the perspective of the second objective, which works the same way as the first but discriminates by product types. That is, we seek to minimize the sum of deviations of each stack of orders for each product type and period from the dashed target line.
3. Figure 2.1c comes back to the first example, but views it from the perspective of the third objective. The numbers inside the orders signalize the priorities, where a larger number indicates a higher importance. That being said it is obvious that the red order should not be assigned to an earlier period than the yellow or the blue one. We say, there exist priority inversions between the two pairs of orders, and their number should be minimized. A better solution can be easily constructed for example by swapping the red order with the yellow one, because it would make both priority inversions disappear.



(a) Leveling objective for the total production amount. This is the optimal solution w.r.t. this objective.



(b) The leveling objective for each product type (blue, green, red).



(c) The same solution as above has two priority inversions. An optimal solution w.r.t. priorities would be to swap the red and the yellow order.

Figure 2.1: Example solutions visualizing the three optimization goals. The dashed line is the target value and the bold red line is the capacity limit.

We have seen in the examples, that an optimal solution w.r.t. one objective is not necessarily optimal w.r.t. another. As we want to combine the three objectives into one by a weighted sum, the location of the optima will clearly depend on the weights. These weights must be determined on the basis of a specific use case because there is no general way to decide without domain knowledge e.g. how important one priority inversion is compared to one unit more of imbalance. We worked out a sensible default weighting in cooperation with our industrial partner based on their real-life data and use it for all experiments throughout the paper. How the objective function can be formally stated and how the weighting works is described in more detail in the following section.

2.1.2 Mathematical Formulation

Now we turn towards a formal description of the the problem, consisting of parameters, variables, constraints and the objective function.

Input parameters

$K \subseteq \mathbb{Z}^+$	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 in each 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

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

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

- For each period the sum of all planned order's demands (helper variable):

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

- For each period and product the sum of all planned order's demands (helper variable):

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

Hard Constraints

- 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 following three objective functions represent the three targets to minimize:

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

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

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

Function f_1 represents the sum over all periods of deviations from the overall target production volume (i.e. all product types at once). Function f_2 states the sum over all product types of sums over all periods of the deviations from the target production volume for that product type, normalized by the respective target value. The normalization is done so that every product has the same influence onto the objective function regardless of whether its target is high or low. Function f_3 counts the number of priority inversions in the assignment, or in other words the number of order-pairs (i, j) for which i is planned after j even though i has a higher priority than j.

In order to combine these three objectives into a single objective function and achieve a weighting which does not change its behavior between instances with different number of orders, periods or product types, the cost components need to be normalized.

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

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

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

The normalization 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 $k \cdot (k-1)/2$.

The final objective function is then a weighted sum of the three normalized objective functions, where the weight a_i of an objective can be seen approximately as its relative importance.

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

Quadratic objective Function

For some instances of the PLP the above presented objective function based on absolute differences may not be well suited. Intuitively, one could argue that a solution containing n periods with a missing demand of 1 is better compared to an otherwise equal one containing 1 period with missing demand n. However, for the presented formulation the two scenarios are penalized in exactly the same way. Therefore, we introduce an alternative variant of the objective function which penalizes deviations from the target by taking squared differences. This implies that also the normalization factors need to be adapted.

$$\tilde{g}_1 = \frac{1}{n \cdot (d^*)^2} \cdot \sum_{i \in N} (d^* - w_i)^2 \tag{2.8}$$

$$\tilde{g}_2 = \frac{1}{n \cdot m} \cdot \sum_{t \in M} \left(\frac{1}{(d_t^*)^2} \cdot \sum_{i \in N} (d_t^* - w_{i,t})^2 \right) \tag{2.9}$$

$$\tilde{g}_3 = g_3 \tag{2.10}$$

The final objective function using squared differences is again a weighted sum of the three normalized objective functions:

$$\textbf{minimize } \tilde{g} = a_1 \cdot \tilde{g}_1 \ + \ a_2 \cdot \tilde{g}_2 \ + \ a_3 \cdot \tilde{g}_3 \tag{2.11}$$

To our experience the two variants of the objective function behave very similarly for the vast majority of our instances. Only in some rare cases, where no well-balanced solution is possible and there are large trade-offs to be made, we found that using the quadratic objective function produces results which intuitively look better than the ones produced by the absolute objective. However, there are also disadvantages to consider when using the alternative objective:

- It renders the otherwise linear problem a quadratic one, which makes it harder to solve using MIP.
- Due to the squares in the denominators of the normalization factors, delta costs are often very small which increases the risk of numerical instabilities when using exact solvers.

There is also evidence in the literature that there is no a priori reason to prefer one of the two objectives. Schaus et al. [Sch+07] investigated balancing objectives in a more general form and stated that a set of violation measures for the perfect balance is given by the L_p-norm of a vector of variables X minus its mean m for $p \geq 0$. For $p = 1$ that corresponds to $\sum_{X \in \mathcal{X}} |X - m|$, which is the same as our linear objective f_1. Similarly, the variant with $p = 2$ corresponds to the quadratic-difference based objective of the PLP. As a conclusion of the study of the different variants the authors state that neither criterion subsumes the others [Sch+07], which means that there is no reason to commit oneself to only one.

2.2 Related Work

The term *production leveling* is commonly associated with the Toyota Production System (TPS), where it is also called Heijunka. It is a concept which aims to increase efficiency and flexibility of mass-production by leveling the production in order to keep the stock size low and reduce waste. Ideally the result of applying Heijunka is zero fluctuation at the final assembly line. Heijunka can mean both the leveling of volume at the final assembly line and the leveling of the production of intermediary materials [OR98].

The PLP is clearly inspired by Heijunka in the sense that the usage of resources should be leveled in order to increase production efficiency but its concepts differ quite substantially from the classical implementation of Heijunka (in the TPS) in the following points:

- The PLP does not operate on the level of schedules but disregards the ordering which the items are produced within a period. In other words, it is concerned with planning and not scheduling, which is performed subsequently for each production period.

- Intermediate materials are not part of the PLP. While Heijunka aims to level also *their* production to keep stock sizes of intermediary products small, the PLP is currently only concerned with one level.

There exists a whole research area concerning scheduling problems inspired by ideas from the TPS and especially Heijunka. Under the umbrella term level scheduling there exist several problems such as the Output Variation Problem and the Product Rate Variation Problem [Kub93; BFS09]. They all have in common that they aim to find the best schedule for production at the final assembly line so that the demand for intermediary materials and their production is leveled which keeps the necessary stock sizes low. However, all of these problems are quite different from the PLP due to the same reasons presented above with respect to Heijunka.

However, under the term *Balancing Problems* several other problems are known in the literature, which are actually more closely related to the PLP:

- The BACP: This problem deals with assigning courses to semesters such that the student's load is balanced and prerequisites are fulfilled [Chi+12]. The balancing of the sum of course sizes assigned to a semester is equivalent to the balancing of production load which we are confronted with in the PLP. There exists also variant called the Generalized BACP which introduces so-called *Curricula* where each of them should be leveled [DS08]. The concept is similar to the product-types of the PLP except for that an order has only one product type while a course can be in multiple curricula.

  The big difference to our problem are additional constraints of the BACP, which enforce prerequisites between courses – a concept appearing frequently in the diverse balancing problems. While they may seem very similar to the priority objective of the PLP, the truth is that they work very differently: First of all, prerequisites are hard constraints while priorities are soft constraints. Furthermore, prerequisites require one course to be finished strictly before another starts while it does not seem sensible for the PLP to require a penalty in case two differently prioritized orders are scheduled to the same period. There we only want to penalize when the ordering implied by the priorities is inverted, hence the term *priority inversion*. This is a fundamental difference which makes it impossible to solve one problem by converting it to the other.

- Nurse scheduling problems are an active field of research since their introduction in the 70s [War76]. While most of the contributions do not consider workload balancing, a few of them, starting with Mullinax and Lawley in 2002 [ML02], do consider also a fair distribution of the nurses' workload. They propose an Integer Programming model for the Nurse to Patient Assignment Problem in neonatal intensive care, which is concerned with finding the optimal assignment of patients to a set of working nurses, so that the workload of the team is balanced an a number of restrictions are fulfilled. The main difficulty is the variability of the infant's condition which greatly influences the amount of work needed. The problem is often solved in two steps by first assigning nurses to zones of the nursery and then assigns infants to nurses. More recent work in this area is for example by a paper by Schaus et al. who investigated a CP approach using the `spread` constraint for balancing [SHR09]. Furthermore, stochastic programming based approaches with Bender's decomposition have been proposed [PRB13].

  The balancing objective of the Nurse to Patient Assignment Problem is again very similar to the objective function which we introduced for the PLP. However, we cannot directly compare to the results because the priorities of the PLP are have no equivalent in this problem and also vice-versa some side-constraints and the zone assignment cannot be expressed.

- Simple Assembly Line Balancing (SALB): An assembly line consists of identical work stations aligned along a conveyor belt. Workpieces move along the conveyor belt and at each station a set of (assembly) tasks is carried out, where each of them has a task time. The sum of all task times of the tasks assigned to one work station during one production cycle is called the station time. By the cycle time we denote the time after which workpieces are moved on to the next station. The goal is either to minimize the number of work stations needed given a fixed cycle time or to minimize the cycle time given a fixed number of work stations.

  The SALB problem is the simplest and most intensively studied variant of Assembly Line Balancing. A comprehensive overview over the different variants is provided by Boysen et al. [BFS07]. When comparing the SALB problem to the PLP, tasks map to orders, task times to order sizes and the fixed cycle time to the maximum capacity per production period. Hence, minimizing the cycle time is equivalent to minimizing the maximum load of a production period of the PLP, which would also be an admissible balancing objective. There is also recent work by Azizoğlu and İmat [Aİ18] where the sum of squared deviations of the workstation loads is minimized, which is equivalent to the second variant of the objective which we proposed. However, the difference between precedence relations on the one hand and priority inversion minimization on the other hand, disallows once again a direct comparison between the problems.

For a more extensive list of Balancing Problems please refer to the dissertation of Pierre Schaus which investigates CP modeling approaches for a very diverse set of Balancing and Bin-Packing Problems [Sch09].

CHAPTER 3

Complexity Analysis

As we are studying a new problem we are interested in analyzing its computational complexity. In this chapter we provide an NP-completeness proof of a decision variant of the PLP, followed by an argumentation of NP-hardness of the PLP optimization problem presented previously. Afterwards we present the Fixed-Order PLP, which is a variant of the problem with two further restrictions. We show that this variant is tractable by introducing a dynamic programming algorithm.

3.1 NP-hardness of the Production Leveling Problem

In order to prove NP-completeness, we consider the following decision variant of the problem where the objective function is dropped completely. Hence the task is solely to find a feasible assignment of orders to periods:

	Production leveling (Decision problem)
Instance:	A set of orders K, of products M and of periods N. For each order $j \in K$ its demand d_j with $d_j > 0$, priority p_j and product type t_j. The maximum production capacity per period c and for each product type $t \in M$ its associated maximum production capacity per period c_t.
Question:	Does there exist an assignment $\{y_j : j \in K\}$ of orders to periods such that the capacity limit c and the capacity limit for each product type $\{c_t : t \in M\}$ are not exceeded for any period?

Theorem 1. *The Production Leveling decision problem is NP-complete even on instances with $|M| = 1$, i.e., a single product type.*

Proof. In order to prove NP-hardness we give a polynomial time reduction from the NP-complete Bin Packing decision problem [Vaz03], which is defined as follows:

Bin Packing (Decision problem)	
Instance:	A set of n bins $S_1, S_2, \ldots, S_n$ of size V and a list of k items of respective sizes $a_1, a_2, \ldots, a_k$
Question:	Can the items be packed into the bins? I.e., is there an n-partition $S_1 \cup S_2 \cup \ldots \cup S_n$ of the set $\{1, 2, \ldots, k\}$ such that $\sum_{i \in S_j} a_i \leq V$ for all $j \in \{1, \ldots, n\}$?

The construction of the PLP instance is straightforward:

- $M = \{1\}$
- $N = \{1, 2, \ldots, n\}$
- $K = \{1, 2, \ldots, k\}$
- $d_j = a_j \quad \forall j \in K$
- $p_j = 1 \quad \forall j \in K$
- $t_j = 1 \quad \forall j \in K$
- $c = c_1 = V$

That is, bins are converted to periods, each item with size a_i to an order with demand d_i and the bin capacity V becomes the maximum capacity per period c. There is only one product type and order priorities can be defined as some arbitrary constant.

If there exists a feasible solution to this instance of the Production Leveling decision problem (i.e. an assignment of orders to periods such that the capacity limit is obeyed), it follows that there exists also a valid bin packing into n bins because each bin with size V corresponds exactly to a period with the same capacity. Analogously, if no feasible solution of the PLP exists we know that the corresponding instance of Bin Packing is infeasible as well. Hence any instance of Bin Packing can be solved by converting it to an instance of the Production Leveling decision problem and solving that one. As the conversion is possible in linear time, the Production Leveling decision problem must be at least as hard as Bin Packing. Consequently we have proven that it is NP-hard, even when considering only a single product type ($|M| = 1$).

In order to prove NP-membership, let us consider an assignment $\{y_j : j \in K\}$ of orders to periods. In order to verify whether this assignment is a valid solution, we need to check whether all capacity constraints are fulfilled:

- Is the overall capacity limit satisfied for each period?

$$\sum_{\substack{j \in K: \\ y_j = i}} d_j \overset{?}{\leq} c \quad \forall i \in N$$

- Are the capacity bounds per period and product type satisfied?

$$\sum_{\substack{j \in K: \\ y_j = i \wedge t_j = t}} d_j \overset{?}{\leq} c_t \quad \forall i \in N, t \in M$$

In total, the number of inequalities that need to be checked is: $|N| + |N| \cdot |M|$ which is clearly polynomial in the size of the instance.

As the Production Leveling decision problem with only one product type is both NP-hard and in NP it is NP-complete. $\square$

The Production Leveling optimization problem presented in Section 2.1.2 differs from the decision variant in that we do not only search a feasible assignment of orders to periods but the best possible one according to an objective function. Obviously, NP-hardness holds as well because the decision problem can be solved by the optimization problem. NP-membership of the optimization variant, however, is clearly not the case as there is no polynomial-time algorithm for deciding whether a given solution is the optimal one in the general case.

3.2 Fixed-Order Production Leveling

In this section we consider a variant of the Production Leveling optimization problem, where the priority values of all orders are unique and the correct ordering with respect to the priorities is enforced. We will show that these two restrictions render the problem solvable in polynomial time. The problem variant is based on ideas of Marie-Louise Lackner and was developed in collaboration with her. The work has not yet been officially published but a technical report is available for download [LVM19].

Fixed-Order Production leveling (Optimization problem)	
Instance:	Set of orders $K = \{i \in \mathbb{Z}^+ \mid 1 \leq i \leq k\}$ Set of products $M = \{i \in \mathbb{Z}^+ \mid 1 \leq i \leq m\}$ Set of periods $N = \{i \in \mathbb{Z}^+ \mid 1 \leq i \leq n\}$ For each order $j \in K$ its demand $d_j > 0$, priority p_j (unique among all orders) and product type t_j Maximum production capacity c Maximum production capacity c_t for each product type $t \in M$ Target production capacity: $d^* \in \mathbb{Z}^+$ Target production capacity for product type $t \in M$: $d^*_t \in \mathbb{Z}^+$
Objective:	Find a mapping of orders to periods $y : K \to N$ that minimizes $g = a_1 \cdot g_1 + a_2 \cdot g_2$ (like Equation (2.7) but without the priority objective g_3)
Constraints:	Respect the priorities: $y(i) \leq y(j)$ for orders $i, j \in K$ with $p_i > p_j$

The only difference to the original version of the PLP is that the priorities are treated as hard constraints instead of soft constraints. This new constraint together with the assumption of unique priority values allows us to view the construction process of the solution as partitioning the sequence of orders sorted by priority instead of assigning arbitrary subsets to each period, which also explains the name Fixed-Order Production Leveling Problem. The view as an assignment problem would be unnecessarily complex for this problem variant because the new hard constraint assures that no priority inversion can exist and the assumption of unique priority values removes the ambiguity when sorting the orders by priority.

The view as a partition problem reminds strongly of the list partition problem, as described for example in *The Algorithm Design Manual* [Ski98]. This problem is defined as follows: Given a sequence S of length k of non-negative numbers $s_1, s_2, \dots, s_k$ and an integer n, find a partition of S into n ranges, i.e. consecutive elements of the sequence, so as to minimize the maximum sum over all the ranges. This problem allows for efficient solutions using dynamic programming.

As our problem can be seen as an extension of the list partition problem with a different objective we investigate this solution approach further. The algorithm used for the list partition problem cannot be used directly in our setting, since it is not sufficient that the maximum planned capacity of all period is as small as possible and thus as close to d^* as possible; we require that orders are divided evenly on periods for all other periods as well. However, the algorithm described in the following is heavily inspired by the one presented for list partition in [Ski98].

The main idea of the algorithm is that an optimal partitioning of a sequence according to some objective[1] can be calculated step by step. In each step of the procedure, we extend some prior solution by one more partition until we reach the desired number. The new partition corresponds always to the last period, so it contains always the l last orders in the sorted order list, which are the ones with lowest priority. In order to decide how large the partition should be (i.e. how many orders from the end of the order list it should contain) the cost for all possibilities is calculated. All possibilities where we can have at least one order in each period are considered. For example, assume that we want to compute the optimal total cost of assigning the last l elements of the sequence to the new partition. Intuitively, it consists of the optimal total cost of assigning the first $k - l$ orders (where k is the total number orders) to the previous number of partitions plus the cost of assigning the last l elements to a new partition. The first part is already known from the previous step of the algorithm, while the second part can be easily calculated. After having computed the cost for all possibilities we define the optimal total cost for this subproblem as the minimum of the encountered cost values.

Theorem 2. *The Fixed-Order Production Leveling problem can be solved in polynomial time using a dynamic programming approach: If n denotes the number of periods and k the number of orders, it can be solved in $\mathcal{O}(nk^2)$ time.*

Proof. Without loss of generality, let us assume that the unique priority values p_j for $j \in K$ are elements of the set $\{1, \dots, k\}$. In a preprocessing step, sort the capacity demands of the orders in decreasing order of their priorities. That is, after sorting d_1

[1]The objective must have the property that the cost of each partition is independent from the cost of other partitions. This condition on the objective is clearly met in the case of the PLP, as each period (which corresponds to a partition) contributes independently to the total cost.

denotes the capacity demand of the order with priority k, d_2 is the capacity demand of the order with priority $k-1$, aso. until d_k which is the capacity demand of the order with priority 1. This sorting requires $\mathcal{O}(k \log k)$ time.

Furthermore, we assume that $k \geq n$ which is sensible as otherwise some periods would be forced to stay empty and we could simply reduce the number of periods. With this assumption we can safely exclude all cases from the algorithm, where some period stays empty because compared to planning two orders for one period and leaving one empty it is always at least as good to plan the two orders in different periods.

In the following, we describe how the Fixed-Order Production leveling optimization problem can be solved using a dynamic programming approach. For this purpose, let us denote by $O(j, l)$ the optimal value of the objective function under consideration, i.e. the minimum value of g, when assigning the first j orders $1, \ldots, j$ to l periods. Thereby it is important to point out that the normalization factors inside the objective function are always those of the original problem, regardless of the choice of j and l. The optimal objective value of the original problem is clearly given by $O(k, n)$ because k and n are the original number of orders and periods.

As indicated in the intuitive explanation of the algorithm above, $O(j, l)$ for $l \leq j \leq k$ and $1 \leq l \leq n$ can be calculated recursively by selecting the variant with minimum cost out of the following two extreme cases and all cases in between:

- assigning only one order to the new period and $j-1$ to the $l-1$ periods before
- assigning $j-l$ orders to the new period and $l-1$ to the $l-1$ periods before.

The table entries $O(j, l)$ with $j < l$ are not defined because we do not consider cases whose optimal solution involves empty periods. Formally, the recursion for calculating the values $O(j, l)$ is given by

$$O(j, l) = \min_{l \leq i \leq j} \left(O(i-1, l-1) + h_1(i, j) + h_2(i, j)\right) + \mathit{constr}(i, j) \tag{3.1}$$

The functions $h_1(i, j)$ and $h_2(i, j)$ denote the respective cost increase of f_1 and f_2, if we would assign the set of orders $\{i, \ldots, j\}$ to a new and empty period. Formally they are defined as follows for $1 \leq i \leq j+1$:

$$\begin{aligned} h_1(i, j) &= \frac{a_1}{n} \cdot \left| \frac{d^* - \sum_{s=i}^{j} d_s}{d^*} \right| \\ h_2(i, j) &= \frac{a_2}{n \cdot m} \cdot \sum_{t \in M} \left| \frac{d_t^* - \sum_{s=i \mid t_s = t}^{j} d_s}{d_t^*} \right| \end{aligned}$$

The penalty function $\mathit{constr}(i, j)$ checks whether the capacity restrictions c and c_t allow for assigning the set of orders $\{i, \ldots, j\}$ to the same period and returns 0 if so and ∞

otherwise. If the final value $O(k,n)$ happens to be ∞ we immediately know that the instance is infeasible. The reason why this approach works is that both constraints can be checked for each period separately.

The calculation of $O(j,l)$ results indeed in the optimal objective value of the subproblem where the first j orders are assigned to l periods because

- all open possibilities are enumerated (due to the fixed ordering and the requirement of having at least one order in each period their number is only $j-l$), and
- the previously computed values $O(i-1,l-1)$ stay meaningful also when adding another period because the associated costs of the periods are independent.

The base cases of the recursion are those, where between 1 and k orders are assigned to one period, which is trivially given by :

$$O(j,1) = h_1(1,j) + h_2(1,j) + \mathit{constr}(1,j) \qquad \text{for } 1 \leq j \leq k.$$

In order to calculate $O(k,n)$, we store the partial results $O(j,l)$ for $1 \leq j \leq k$ and $1 \leq l \leq n$ in a table of size (k,n). In order to compute one of these entries $O(j,l)$, we require the values $O(i-1,l-1)$ with $1 \leq i \leq j+1$, i.e., all elements in the column to the left of and not below $O(j,l)$. We thus fill in the table column by column from left to right and top to bottom within a column. Moreover, we require the values $h_1(i,j)$ and $h_2(i,j)$ for every $1 \leq i \leq j$. Since these are also required for further elements of the table, these values $h_1(i,j)$ and $h_2(i,j)$ are pre-computed for all i,j with $1 \leq i \leq j \leq k$, which requires $\mathcal{O}(k^2)$ time.

Given these pre-computations, the time needed to compute each entry $O(j,l)$ is in $\mathcal{O}(k)$ because the minimum of $j+1 \leq k$ values needs to be found, each of which requires only access to 3 previously computed values. As the table has the size $k \cdot n$, computing all elements of the table can be done in $\mathcal{O}(n \cdot k^2)$ time.

We are not merely interested in computing the the value of g for an optimal solution but also in describing this optimal solution. That is, we need to know which orders are assigned to which period. While we compute the values $O(j,l)$, we thus also store the value of i for which this minimum was achieved in Equation (3.1)[2]. This is stored in the array $M = M(j,l)_{1\leq j\leq k, 1\leq l\leq n}$ with

$$M(j,l) = i \Longleftrightarrow O(j,l) = O(i-1,l-1) + f_{i,j} + h_{i,j}.$$

Computing $M(j,l)$ in addition to $O(j,l)$ adds only a constant amount to the computational complexity, so that the asymptotic behaviour does not change.

Once all values for $O(j,l)$ and $M(j,l)$ have been computed, the assignment of orders to periods can be reconstructed as follows, starting with the last period and ending with the first one:

[2]If this value of i is not unique, we pick the smallest such i.

- The orders $o_{M(k,n)}, \ldots, o_k$ are assigned to the last period
- Given that the first order assigned to period l with $l > 2$ is $o_i = o_{M(j,l)}$ for some j, the orders assigned to period $l-1$ are: $o_{i'}, \ldots, o_{i-1}$ with $i' = M(i-1, l-1)$.
- The remaining orders are assigned to the first period.

Reconstructing the solution requires a linear amount of time in the number of periods n and the number of orders k. When we assess the asymptotic complexity of the whole dynamic programming algorithm, the computation of the $O(j,l)$ values with complexity $\mathcal{O}(n \cdot k^2)$ clearly outweighs all other parts. Therefore, the total complexity is also $\mathcal{O}(n \cdot k^2)$. □

Example 1. *As an example, consider the following instance with $k = 20$ orders:*

type	*gray*	*red*	*blue*	*green*
order = (priority, demand)	*(20,10) (15,10) (10,10) (7,15) (4,5)*	*(18,5) (14,20) (9,10) (8,5)*	*(19,5) (13,10) (6,15) (3,5)*	*(17,5) (16,20) (12,25) (11,20) (5,30) (2,5) (1,5)*

These orders need to be scheduled to $n = 5$ periods and we have $d^* = \frac{1}{n}\sum_{s=1}^{k} o_s = 47$, $d^*_{gray} = 10$, $d^*_{red} = 8$, $d^*_{blue} = 7$, $d^*_{green} = 22$. We choose the weights of objectives as follows: $a_1 = 1$ and $a_2 = 1$. The sorted list of orders is given as follows:

$$(o_1, \ldots, o_{20}) = (10, 5, 5, 5, 20, 10, 20, 10, 25, 20, 10, 10, 5, 15, 15, 30, 5, 5, 5, 5).$$

The values for $O(j,l)$ and $M(j,l)$ with $0 \le j \le 20$ and $1 \le l \le 5$ are given in Table 3.1.

The optimal objective value is given by $O(20,5) \approx 0.86$. The entries showing a '–' are those where $j < l$, which are unnecessary to compute, as stated above. In all cases where $O(j,l) = \infty$ there does not exist a feasible solution to the subproblem.

The $M(j,l)$ values printed in bold face visualize the path through the table which allows us to reconstruct the optimal solution. The reconstruction works as follows:

- $M(20,5) = 17$, *thus the orders assigned to period 5 are:* $o_{17}, \ldots, o_{20}$.
- $M(16,4) = 14$, *thus the orders assigned to period 4 are:* $o_{14}, \ldots, o_{16}$.
- $M(13,3) = 10$, *thus the orders assigned to period 3 are:* $o_{10}, \ldots, o_{13}$.
- $M(9,2) = 7$, *thus the orders assigned to period 2 are:* $o_7, \ldots, o_9$.
- $M(7,1) = 1$, *thus the orders assigned to period 1 are:* $o_1, \ldots, o_6$.

Table 3.1: Dynamic Programming matrices for the Fixed-Order PLP example instance. On the left-hand-side the values $O(j,l)$, stating the optimal objective value of each sub-problem and on the right-hand-side the values $M(j,l)$ which keep track of the optimal partition boundaries.

j \ l	1	2	3	4	5
1	0.31	–	–	–	–
2	0.25	0.65	–	–	–
3	0.20	0.60	1.00	–	–
4	0.17	0.57	0.97	1.37	–
5	0.05	0.43	0.83	1.23	1.63
6	0.12	0.34	0.74	1.14	1.54
7	∞	0.30	0.68	1.08	1.48
8	∞	0.22	0.61	1.01	1.41
9	∞	0.31	0.48	0.86	1.26
10	∞	∞	0.58	0.74	1.13
11	∞	∞	0.49	0.65	1.04
12	∞	∞	0.41	0.57	0.96
13	∞	∞	0.42	0.58	0.92
14	∞	∞	∞	0.67	0.83
15	∞	∞	∞	0.61	0.77
16	∞	∞	∞	0.62	0.79
17	∞	∞	∞	∞	0.80
18	∞	∞	∞	∞	0.75
19	∞	∞	∞	∞	0.89
20	∞	∞	∞	∞	**0.86**

j \ l	1	2	3	4	5
1	1	–	–	–	–
2	1	2	–	–	–
3	1	2	3	–	–
4	1	2	3	4	–
5	1	5	5	5	5
6	**1**	5	5	5	6
7	∞	6	6	7	6
8	∞	6	6	7	8
9	∞	**7**	8	8	9
10	∞	∞	10	10	10
11	∞	∞	10	10	10
12	∞	∞	10	10	10
13	∞	∞	**10**	10	13
14	∞	∞	∞	13	13
15	∞	∞	∞	13	13
16	∞	∞	∞	**14**	14
17	∞	∞	∞	∞	16
18	∞	∞	∞	∞	16
19	∞	∞	∞	∞	17
20	∞	∞	∞	∞	**17**

Figure 3.1 visualizes the obtained solution in two different ways. The view on the left shows the overall production volume assigned to each period. The sum of absolute differences between the bar and the dashed line corresponds to f_1. The center figure shows the production volume per period, and the sum of absolute differences corresponds to f_2. The bold colored lines are the maximum capacities c and c_t which need to be respected. The right figure shows the priority values of the orders planned in each period, which are obviously perfectly sorted.

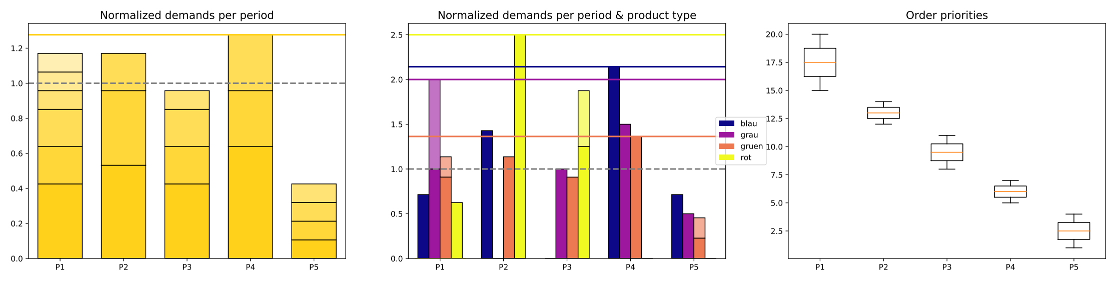


Figure 3.1: Optimal solution of the presented example instance. From left to right, visualizations of the objectives f_1, f_2 and f_3.

CHAPTER 4

Solution Approaches

The PLP is a weakly constrained optimization problem as the only existing constraints are upper bounds on the planned production volume per period and product. There are no constraints involved in the prioritization and the objective function is a trade-off which means for example that pruning solutions with a bad priority objective is not immediately possible as long as there is enough room for improvement in the balancing objectives. In other words, the feasible solution space is very large.

First we want to develop models for the PLP that allow us to obtain proven optimal solutions by means of complete methods. In the field of constrained combinatorial optimization problems, two of the most frequently applied options are CP and MIP. We propose models for both paradigms which are all based on the formal problem definition of Chapter 2 but each of them adapted for the respective technology. In both cases, we include dominance breaking constraints which help to reduce the tremendous size of the search space.

However, as we will see in Chapter 5 during the presentation of the experiments, large instances cannot be solved well by any of the exact techniques. This is not very surprising, as the problem is NP-hard, i.e. it belongs to a class of problems for which no polynomial-time algorithms have been found so far and it is even unclear whether such algorithms exist. It follows that there exist instances for which the required running time of any known exact algorithm is exponential. For this reason we investigate also metaheuristic local search methods which are generally faster in solving large instances, but do not provide guarantees about the solution quality. We will present our local search approaches in the second part of this chapter.

4.1 Exact modeling techniques

In this section we introduce models of the PLP which are based on Constraint Programming (CP) and Integer Programming. Both of them are declarative programming paradigms which allow to formally specify combinatorial optimization problems in a way that a general purpose solver software can compute the optimal solution to arbitrary problem instances.

A CP model consists of variables and constraints, which are relations between the variables. A feasible solution is an instantiation of all variables such that all relations between the variables hold. Usually a CP-solver works towards a feasible solution by performing alternately some kind of search over the respective domains of the variables and an inference step, where the domains of the variables are restricted according to the specified constraints. The propagation step is very important to reduce the exponentially sized search space but would not be sufficient to solve arbitrary problems on its own. If we want to minimize an objective function, the solver adds for each found solution i with cost c_i a constraint on the cost variable z which states that $z < c_i$ – i.e. the search is continued only on improving solutions. These additional constraints on the objective value can be again propagated, as outlined above. When the search tree has been processed completely, it is guaranteed that the optimal solution has been found [RBW06].

Integer programming is a paradigm which deals with maximizing or minimizing a linear function over a set of integer variables which are subject to linear constraints. When a model has both integer and continuous variables, we instead call it Mixed Integer Programming (MIP)[1]. The restriction to linear constraints comes due to the way how MIP solvers work. The solving process relies on the fact that linear programs (with continuous variables only) can be solved efficiently by the Simplex algorithm. The solver exploits that by first solving a relaxation of the problem, where it treats all variables as continuous and afterwards adds integrality restrictions again step by step. Any non-linear constraint to be included must be rewritten by introducing one or more helper variables and only linear constraints which link them together. For example, to find the maximum of a set of variables S, an additional variable x and a set of constraints must be introduced stating for each $s \in S$, that $x \geq s$ [WN14].

When comparing the two paradigms, CP allows for a more high-level modeling because built-in global constraints can be used to abstract the complexity. Furthermore, solvers can provide dedicated constraint propagation algorithms for each global constraint which make the search more efficient. Compared to that, MIP solvers work on a very restricted language but on the other hand they are highly optimized for that. They are especially efficient for formulations whose linear relaxation is tight.

[1]Subsequently we continue to use exclusively the term Mixed Integer Program because the model which we will present uses both integral and continuous variables. However, most of the times it would make no difference whichever term we used.

4.1.1 A Constraint Programming model

The CP model is an extension of the mathematical problem formulation presented in Section 2.1.2. The problem input parameters, variables and hard constraints stated there are also part of this model, which is why we do not repeat them. The extension takes place in the following two areas:

- Express parts of the problem structure by global constraints.
- Find redundant constraints which improve the results or save time.

We express the model in the solver-independent general-purpose modeling language MiniZinc [Net+07]. The advantage is that we can apply the same model to various solvers without any changes.

Global Constraints

One part of the problem which can be captured by means of a global constraint is the packing of orders into periods. As we have seen in Section 3.1, solving the PLP also involves solving a Bin Packing problem. Therefore, the part of the PLP which is concerned with packing orders into periods can be expressed using a global constraint for Bin Packing as introduced by Shaw [Sha04]. The constraint catalog of the used modeling language MiniZinc [Net+07] defines the constraint as follows:

Definition 1. *The constraint* `bin_packing_load` (l, b, w) *requires that each item* $i \in I$ *with weight* w_i *be put into bin* b_i *and the sum of the weights of the items in each bin* $j \in B$ *is equal to* l_j*. Formally speaking, the constraint holds if and only if*

$$l_j = \sum_{\substack{i \in I: \\ b_i = j}} w_i \quad \forall j \in B$$

In our CP model, we can thus link the decision variables for order assignment y_j to the variables w_i for the load in each period by adding the constraint `bin_packing_load` (w, y, d). Similarly, the variables $w_{i,j}$ can be linked with one constraint per product-type.

Another part of the problem which can be captured using global constraints is the calculation of the balancing objective, where absolute deviations from a mean value are to be calculated. That can be achieved using the `deviation` constraint for balancing a set of variables, which was introduced by Schaus et al. [Sch+07]. This paper provides the following definition:

Definition 2. *The constraint* `deviation`(X, m, D) *states that the collection of values taken by the variables of* X *exhibits an arithmetic mean* m *and a sum of deviations to* m *of* D*. More formally,* `deviation`(X, m, D) *holds if and only if*

$$n \cdot m = \sum_{i=1}^{n} X_i \quad \textit{and} \quad D = \sum_{i=1}^{n} |X_i - m|$$

This constraint is ideally suited for calculating the first two components of our objective function, f_1 and f_2 (as stated in the Equation (2.1) and (2.2)). For example, assuming the parameters and variables of the formal definition, we only need the constraint $\texttt{deviation}(w, d^*, f_1)$ in order to force f_1 to the correct objective value. The problem is, however, that this constraint is not implemented in any state-of-the art CP-solver to the best of our knowledge. Therefore, we cannot use this constraint in our model and instead express f_1 and f_2 with their defining formula (2.1) and (2.2), respectively, going without fast propagation of the balancing constraints.

Redundant constraints

The other goal when creating the model was to find redundant constraints which make the solving process more efficient. A popular and frequently applied variant of redundant constraints is symmetry breaking, where equivalent solutions are removed from the search space and at least one of them is kept. For example, in the PLP it is easy to see that if there was no priority objective, there would exist for every pair of orders, where both have the same demand value and product type, two equivalent (symmetric) solutions. The only difference would lie in the period assignments but due to the orders being equal, the balancing objectives would not change. However, as we are also confronted with a priority objective, this kind of symmetry only holds between orders which additionally have equal priority values, which is a too rigorous restriction to be useful.

There is also the less widely applied concept of dominance relations which can help to exploit the idea of repeated orders. Dominance relations are a generalization of symmetry, that can reduce the search space by a similar or even greater amount [CS15]. They can be seen as a kind of unidirectional symmetry which prevents the exploration of subtrees of the search space where the optimal solution cannot reside because we can prove that the solutions there are dominated (i.e. provably worse than some other solution). In the area of balancing problems, Monette et al. showed that dominance rules can be successfully applied to the BACP. They introduce a dominance relation between equally large courses that leads to a considerable reduction of the search space[Mon+07].

We define now a similar rule for the PLP which exploits the idea of repeated orders. As in the BACP, it is expressed by less or equal constraints between the assignment variables. The following theorem presents the conditions for the dominance rule to hold:

Theorem 3. *Posting a constraint $y_i \leq y_j$ between two orders i and j preserves at least one optimal solution of the PLP if the following conditions are met:*

1. $d_i = d_j$*: The order's demand values are equal*
2. $t_i = t_j$*: The order's product types are equal*
3. $p_i \geq p_j$*: Order i has a higher or equal priority than order j*

Proof. Assume towards a contradiction that the constraint $y_i \leq y_j$ would cut off all optimal solutions, while all three conditions hold. Let z be an arbitrary optimal solution which is cut off by the introduced constraint – i.e. all conditions hold and the dominance constraint is violated (i.e. $y_i > y_j$). Now let us consider the solution z' which is derived from z by swapping the period assignment of orders i and j. Please note that all three conditions continue to hold for z' and the constraint $y_i \leq y_j$ is now fulfilled, i.e. the solution is not cut off. As we assumed that all optimal solutions are cut off by the constraint, the objective value $f(z')$ must be strictly larger than $f(z)$.

Conditions one and two ensure that the balancing objectives g_1 and g_2 of z and z' are equal because the orders do not differ in any value which is used to compute the objectives[2]. Hence, in order for $f(z') > f(z)$ to hold, $g_3(z') > g_3(z)$ and consequently $f_3(z') > f_3(z)$ must hold because a_3 is a constant. Equation (2.3) defines $f_3 = |\{(l,k) \in K^2 : y_l > y_k \text{ and } p_l > p_k\}|$, which counts the number of order pairs where the one with the higher priority is planned in a strictly larger period. We can also describe it as the cardinality of the priority inversion set. As the solutions z and z' differ only in the assignments of y_i and y_j we only need to look for differences in the set $\{(i,j),(j,i)\}$ instead of K^2. For the solution z', where both $y_i \leq y_j$ and $p_i \geq p_j$ hold, neither of the two order pairs can satisfy the two conditions of a priority inversion at the same time, so the priority inversion set is empty. As a set cannot contain fewer elements than 0, whatever number of priority inversions the solution z may have, the solution z' has fewer or at most equally many. Consequently, $f_3(z') > f_3(z)$ cannot hold, which is the desired contradiction to our assumption. $\square$

We have just shown that we can add constraints of the form $y_i < y_j$ between repeated orders with $p_i \geq p_j$ without cutting off all optimal solutions, which means that the objective value does not change. As we normally do not see large numbers of repeated orders in realistic problem instances, we add all such constraints to the model.

4.1.2 A Mixed Integer Program

Now we want to investigate the second exact modeling approach, which is MIP. As the previously presented CP model is written in the solver-independent language MiniZinc, we can also execute it using state-of-the-art MIP solvers. However, in this case we do not have any control over the compilation process and how the linearization works. In order to find out whether performing the necessary linearizations on our own would improve the results we devised a dedicated MIP model which is implemented for the solver Gurobi [Gur19]. The model is stated in the following, where the input parameters are again the same as stated in Section 2.1.2.

[2]Recall from Equation (2.7), that the objective function is a linear combination of three objectives with all components from $\mathcal{R}^+$: $g = a_1 \cdot g_1 + a_2 \cdot g_2 + a_3 \cdot g_3$.

Variables

$x_{ij} \in \{0,1\}$	for each $i \in N$, $j \in K$ stating if order j is planned in period i
$y_j \in N$	for each $j \in K$, whose value is the assigned period of order j
$z_{ij} \in \{0,1\}$	for orders $i, j \in K$ where $p_i > p_j$, existence of a priority inversion between i and j
$s_i^+ \in \mathbb{R}^+$	for each $i \in N$ the surplus demand for period i
$s_i^- \in \mathbb{R}^+$	for each $i \in N$ the missing demand for period i
$s_{it}^+ \in \mathbb{R}^+$	for each $i \in N$, $t \in M$ the surplus demand for period i and product t
$s_{it}^- \in \mathbb{R}^+$	for each $i \in N$, $t \in M$ the missing demand for period i and product t

Formulation

$$\min \quad a_1 g_1 + a_2 g_2 + a_3 g_3 \tag{4.1}$$

$$\text{s.t.} \quad \sum_{i \in N} x_{ij} = 1 \qquad j \in K \tag{4.2}$$

$$\sum_{i \in N} i \cdot x_{ij} = y_j \qquad j \in K \tag{4.3}$$

$$y_i - y_j \leq (n-1) z_{ij} \qquad i, j \in K \mid p_i > p_j \tag{4.4}$$

$$\sum_{i \in K} d_j x_{ij} + s_i^+ - s_i^- = d^* \qquad i \in N \tag{4.5}$$

$$\sum_{j \in K \mid t_j = p} d_j x_{ij} + s_{it}^+ - s_{it}^- = d_t^* \qquad i \in N, \; t \in M \tag{4.6}$$

$$d^* + s_i^+ \leq c \qquad i \in N \tag{4.7}$$

$$d_p^* + s_{it}^+ \leq c_p \qquad i \in N, \; t \in M \tag{4.8}$$

$$y_i \leq y_j \qquad i, j \in S, S \subseteq K \mid p_i \geq p_j, d_i = d_j, t_i = t_j \tag{4.9}$$

$$\sum_{p \in M} (s_{it}^- - s_{it}^+) = s_i^- - s_i^+ \qquad i \in N \tag{4.10}$$

Constraints (4.2) to (4.6) are the model's required helper constraints. Constraint (4.2) makes sure, that there is exactly one period to which an order is assigned. Constraint (4.3) links the x_{ij} to the y_i variables. Constraint (4.4) links the y_i to the $z_{i,j}$ variables. It makes sure that for every pair of orders i, j where i has a higher priority than j, z_{ij} is 1 (representing an inversion) if i is planned later than j. Constraint (4.5) states for each period that the total demand planned plus the surplus minus the slack equals d^*. As both variables have positive domains and they are subject to minimization, at most one of them will be non-zero in any optimal solution. Constraint (4.6) repeats this relationship over the variables s_{it}^+ and s_{it}^- for each product type t.

Constraints (4.7) and (4.8) ensure that the capacity bound per period and the bound per period and product type, respectively, are satisfied. This is elegantly achieved by stating that the target demand plus the surplus variable stays below the threshold.

Finally, there are two redundant constraints for strengthening the formulation: Constraint (4.9) enforces the same dominance relation that was introduced in the CP model for all pairs of orders which have the same product type and demand value. Constraint (4.10) links the $s_i^{\{+,-\}}$ and $s_{it}^{\{+,-\}}$ variables together, which also leads to improvements in the average runtime.

4.1.3 Absolute-difference-based objective

The following objective function is equivalent to the one presented in Section 2.1.2 but here it is stated on the variable set of the MIP formulation. It is not hard to see that in function g_1 $s_i^+ + s_i^-$ is equivalent to the absolute difference between planned an target demand $|d^* - w_i|$, because at most one of them will be different from 0. The same holds true for the analogous variables in g_2.

$$g_1 = \frac{1}{n \cdot d^*} \cdot \sum_{i \in N} (s_i^+ + s_i^-) \tag{4.11}$$

$$g_2 = \frac{1}{n \cdot m} \cdot \sum_{t \in M} \left(\frac{1}{d_t^*} \cdot \sum_{i \in N} (s_{it}^+ + s_{it}^-) \right) \tag{4.12}$$

$$g_3 = \frac{2}{k \cdot (k-1)} \cdot \sum_{i,j \in K} z_{i,j} \tag{4.13}$$

4.1.4 Squared-difference-based objective

The second variant of the objective function which uses squared differences can also be easily expressed:

$$\tilde{g}_1 = \frac{1}{n \cdot (d^*)^2} \cdot \sum_{i \in N} (s_i^+)^2 + (s_i^-)^2 \tag{4.14}$$

$$\tilde{g}_2 = \frac{1}{n \cdot m} \cdot \sum_{t \in M} \left(\frac{1}{(d_t^*)^2} \cdot \sum_{i \in N} (s_{it}^+)^2 + (s_{it}^-)^2 \right) \tag{4.15}$$

$$\tilde{g}_3 = g_3 \tag{4.16}$$

The main difference to the first version is that the surplus and missing demand appears squared in the objective function. Furthermore, the normalization factors have been adapted.

Obviously, this formulation is no longer a linear program. However, as many state-of-the art solvers also support quadratic optimization we consider this variant worth reporting.

4.2 Metaheuristic local search

In this section we present metaheuristic local search techniques to solve the PLP. To obtain initial solutions we present two different approaches. Afterwards two neighborhood structures for the PLP are described and finally we explain the local search algorithms.

4.2.1 Construction of initial solutions

We developed a greedy construction heuristic which is capable of constructing good initial solutions in a very small amount of time. The parameters of the algorithm are a list of orders, the number of periods n and the random selection size r. The first step of the algorithm is sorting the orders by priority decreasingly which is already the approximate handling of objective 3. Then we loop over all periods i from 1 to n, performing the following steps:

1. Examine sequentially the orders from the head of the sorted order list: For each of them, if it still fits into this period obeying the capacity limits, calculate the delta cost for g_1 and g_2 (as defined in (2.4) and (2.5) in Section 2.1.2) which the inclusion of this order into the period would bring with it. If the delta cost is smaller than zero (i.e. including the order improves the objectives), it is added to a list of suitable orders. The orders from the head of the sorted list are processed in this way until the suitable order list has size $\frac{k}{n}$ (i.e. the average number of orders for each period) or there are no orders left.
2. Afterwards, if the list is not empty, select randomly one of the r best suitable orders, plan it for period i, remove it from the sorted order list and go back to 1.
3. Otherwise (if there was no suitable order) repeat with $i := i + 1$.

Finally, we check whether there are any orders left which could not be assigned due to the capacity limits. If that is the case, they get assigned one by one to the period where the remaining capacity is maximal. This way especially those periods which are not filled well get assigned the remaining orders and the probability of a hard constraint violation is minimized. However, violating the maximum capacity constraint is allowed in this step because a complete assignment of orders to periods is required for the subsequent local search.

The parameter r controls the random selection size of step 2. If we set it to 1, the algorithm is deterministic. When using values greater than 1, the construction heuristic is randomized, which can be useful for some local search techniques (e.g. GRASP).

4.2.2 Neighborhood relations

We devised two types of moves for generating different neighborhoods of a solution which will be introduced in the following subsections. Furthermore, we briefly describe the delta evaluation approach and the methods of neighborhood exploration.

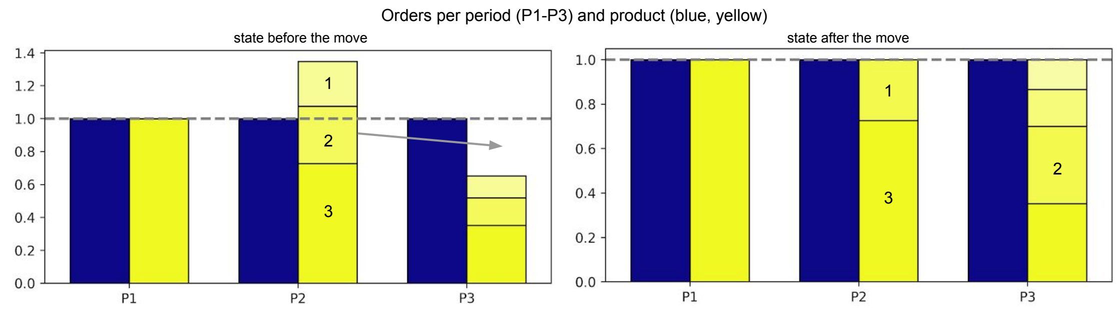


Figure 4.1: Example of a move-order move: solutions before (left) and after (right)

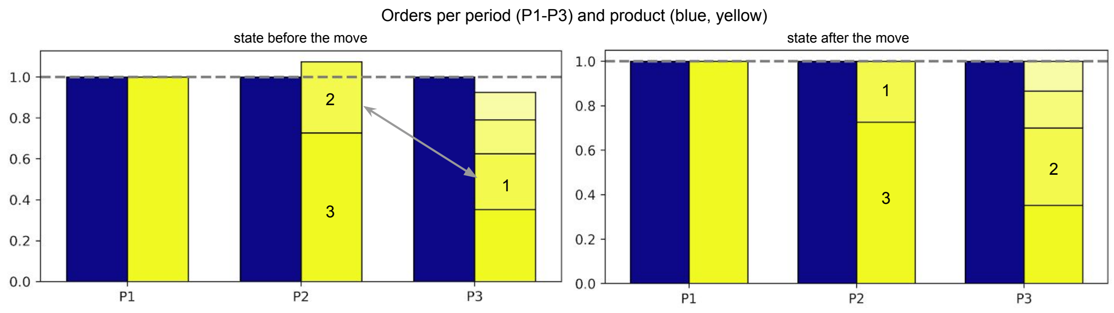


Figure 4.2: Example of a swap-orders move: solutions before (left) and after (right)

Move-order neighborhood

The move-order neighborhood (or simply move neighborhood) of a solution s consists of all solutions whose only difference to s is that one order has been moved to a different period. Figure 4.1 visualizes such a move. The figure on the left shows the leveling objective per product type before the move and on the right side we can see the result of applying the move. Order 2 is moved from $P2$ to $P3$ which yields in this case a better solution.

Enumerating the move neighborhood involves iterating over k orders for each of $n-1$ possible target periods, i.e. the neighborhood size is exactly $k \cdot (n-1)$.

Swap orders neighborhood

The swap-orders neighborhood (or simply swap neighborhood) of a solution s consists of all solutions s' whose only difference to s is that two orders not assigned to the same period in s appear with swapped period assignments in s'. Figure 4.2 visualizes such a move. Order 1 is swapped with order 2 which in this case again yields a better solution.

Enumerating the swap neighborhood involves iterating over all pairs of orders not assigned to the same period. Hence the neighborhood size is in $O(k^2)$.

Neighborhood exploration

We investigated three types of neighborhood exploration:

- **First Improvement**: Generate and evaluate moves until the point where the first move is found who would improve the current solution. In order to prevent a bias towards the start of our neighborhood (e.g. the first orders in our input) the neighborhood traversal is performed in a cyclic way. That is, instead of starting every time at the same point we start right after the position where we found the first improving move the last time and search until either an improving move is found or we arrive again at the point where we started.

- **Best Improvement**: Generate and evaluate the complete neighborhood of a solution and select the move which leads to the biggest improvement.

- **Random Neighbor**: Generate and evaluate a random neighbor of the given solution.

Move evaluation

In order to explore a neighborhood systematically, we need to be able to compare moves with respect to their quality. Given two moves a and b, the first criterion to check is the number of hard constraint violations which each of them introduces or resolves and if a introduces fewer or resolves more of them we say that a is better than b. If the hard constraint violations are equal, we compare by selecting the one which has the lower move cost, which is defined as the change of the current solution's objective value if we would perform this move.

To avoid costly complete evaluations of whole solutions we propose a delta evaluation that efficiently evaluates how much the objective value changes for a given move. The delta evaluation implementations for the two move types both use the same primitive for evaluating the cost of moving one order to a different period. When performing swaps, we calculate the cost of moving order one to the period of order two, the cost for moving order two to the period of order one and compensate the error which results from assuming in both calculations that the respective other order remains unchanged. The delta cost of moving an order is calculated for the three objective function components separately:

1. For the leveling objective we only need to keep track of the planned production volume for each period, so that we can calculate the effect the move on the difference to the target value.

2. For the per-product leveling objective we can do the same thing, given that we keep track of the planned production volume for each period and product.

3. The priority objective is the hardest and most time-consuming part of delta evaluation because moving an order from period i to j can introduce or resolve inversions between the moved order and every order assigned to a period between i and j. When the number of orders is very large it is little efficient to iterate over all such orders and perform comparisons because we need to do that for every candidate move. Our idea for optimizing this evaluation is based on the insight that the only thing we care about when moving an order past a period is the number of orders in that period which have smaller and larger priorities, respectively, not the actual priority values. Therefore, we maintain the priority values of all orders assigned to a certain period in a sorted list (one for each period), so that we can efficiently retrieve via binary search how many orders have smaller / larger priorities than the order which we currently want to move.

The delta cost of the three objective function components is aggregated to a single value by the usual formula for the objective value (2.7).

4.2.3 Algorithms

In this section we present details of the metaheuristic local search methods which we investigated for solving the PLP, namely the simple and deterministic VND as well as Simulated Annealing.

Variable Neighborhood Descent

VND is a deterministic local search technique which can be seen as an extension of hill climbing to multiple neighborhoods. The general idea is to go on to the next neighborhood if the current one gets stuck in a local optimum and return to the first one as soon as a further improvement is found. The selection of an improving move in the neighborhood is usually done by using a deterministic exploration technique, i.e. either first or best improvement. Algorithm 4.1 shows the details using pseudo code, as it is given in the Handbook of Metaheuristics [Han+10].

The idea of using multiple neighborhoods is based on the following insights [Han+10]:

- A local optimum w.r.t. one neighborhood structure is not necessarily a local optimum w.r.t. another.
- A global optimum is a local optimum w.r.t. all possible neighborhood structures.

That implies it is beneficial to use several complementary neighborhoods and try to escape local optima of one neighborhood by switching to another.

Algorithm 4.1: Variable Neighborhood Descent

```
Data: $initialSolution$, neighborhoods $\mathcal{N}_1, \dots, \mathcal{N}_k$, $timeLimit$, $iterationLimit$
Result: a solution at least as good as $initialSolution$
1  $currentSolution \leftarrow initialSolution$;
2  $iterationCount \leftarrow 1$;
3  $j \leftarrow 1$;
4  while $j \leq k$ and $\neg$ out of time and $iterationCount \leq iterationLimit$ do
5      $bestMove \leftarrow$ select a neighbor of $currentSolution$ w.r.t. $\mathcal{N}_j$;
6      if $bestMove$ is an improvement then
7          $currentSolution \leftarrow$ doMove($bestMove$);
8          $j \leftarrow 1$;
9      else
10         $j \leftarrow j + 1$;
11     end
12     $iterationCount \leftarrow iterationCount + 1$;
13 end
14 return $currentSolution$;
```

Simulated Annealing

Simulated Annealing is a metaheuristic optimization method introduced by Kirkpatrick et al. in 1983 [KGV83]. It resembles the physical process of annealing in metallurgy insofar as both methods use a cooling schedule in order to control the amount of random movements in the process, which in theory allows for convergence to the optimal state. Even though convergence to the optimal solution is usually not achieved in practical settings, Simulated Annealing is still one of the most widely used methods metaheuristic optimization methods.

Given an initial solution, a set of neighborhoods $\mathcal{N}_i$ with associated probabilities p_i, the starting temperature t_{max}, minimum temperature t_{min}, number of iterations per temperature w, time limit and iteration limit the version of Simulated Annealing we propose works as shown in Algorithm 4.2.

The pseudo code makes use of two functions **Accept**, standing for the acceptance criterion, and **Cool-Off**, defining the cooling schedule, which we discuss in the following:

- **Acceptance Criterion:** We use the metropolis criterion as acceptance function, which was introduced in the original paper by Kirkpatrick in 1983 [KGV83]. The probability of acceptance $P(i \Rightarrow j)$ of a move from solution i to solution j (for the case of minimization), with $f(x)$ standing for the objective value of solution x, can be defined as follows:

$$P(i \Rightarrow j) = \begin{cases} 1, & \text{if } f(j) \leq f(i). \\ exp\left(\frac{f(i)-f(j)}{t}\right), & \text{otherwise.} \end{cases} \tag{4.17}$$

Algorithm 4.2: Simulated Annealing

```
Data: initialSolution, neighbohoods $\mathcal{N}_i$ with probabilities $p_i$, $t_{max}$, $t_{min}$,
      iterations per temperature $w$, timeLimit, iterationLimit
Result: a solution at least as good as initialSolution
1  currentSolution ← initialSolution;
2  bestSolution ← currentSolution;
3  $t \leftarrow t_{max}$;
4  while $t \geq t_{min}$ and ¬ time limit reached and ¬ iteration limit reached do
5      foreach $i \in 1, \ldots, w$ do
6          $\mathcal{N} \leftarrow$ choose one of neighborhoods $\mathcal{N}_i$ according to probabilities $p_i$;
7          $m \leftarrow$ select a random move out of $\mathcal{N}$(currentSolution);
8          if Accept(m, t) then
9              currentSolution ← Apply($m$, currentSolution);
10             if currentSolution is better than bestSolution then
11                 bestSolution ← currentSolution;
12             end
13         end
14     end
15     $t \leftarrow$ Cool-Down($t$);
16 end
17 return bestSolution;
```

If the candidate solution j is at least as good as the current solution i, it is accepted unconditionally. Otherwise it is accepted with a probability which is decreasing exponentially as a function of the negative delta cost divided by the current temperature. That means, if a candidate solution is much worse than the current one it will be accepted with a lower probability than a solution which is just a little bit worse.

- **Cooling schedule:** The temperature is decreased during the search process by means of a cooling schedule which is usually a geometric row. In our case it depends on the cooling rate α and the iterations per temperature level w. The function **Cool-Down()** reduces the temperature after every w iterations by the following formula:

$$t_i = \alpha \cdot t_{i-1} \tag{4.18}$$

We want to stress now briefly how α and w interact. Assuming we are given an iteration limit l, the initial temperature t_{max} and the final temperature t_{min} there exist many different options to reach t_{min} after l iterations, namely all combinations of α and w such that $t_{min} = \alpha^n \cdot t_{max}$ where the number of temperature steps $n = \left\lfloor \frac{l}{w} \right\rfloor$. Two examples of schedules following that formula with $l = 30000$, $t_{max} = 1$ and $t_{min} = 0.001$ are depicted in Figure 4.3. Please observe that for both

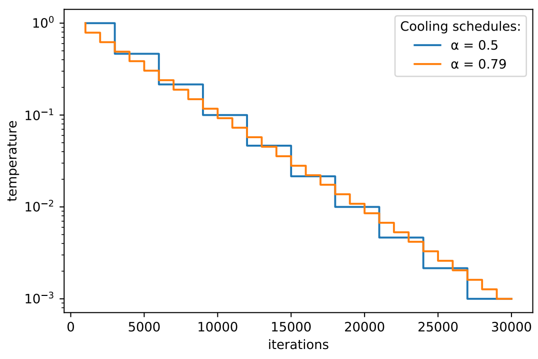


Figure 4.3: Two cooling schedules with different cooling rates and iterations per temperature but identical start and end points

options depicted in the figure the temperature at each time is approximately the same as the different step sizes and widths compensate each other. Therefore, it is sufficient to fix the cooling rate α when tuning the parameters of Simulated Annealing and let the cooling schedule be determined only by the variation of w.

If we do not know the number l, we can also derive a formula which relates two cooling schedules (α_1, w_1) and (α_2, w_2) that have the same slope:

$$\frac{w_1}{w_2} = \frac{\log \alpha_1}{\log \alpha_2} \tag{4.19}$$

Using this relationship one can construct alternative cooling schedules which decrease equally fast on average.

CHAPTER 5

Experimental Evaluation

In this chapter we evaluate the practical contributions of our work and provide answers to the questions which have been raised. As the PLP is a new problem we initially elaborate on the problem instances and propose two instance generation procedures. Next we describe properties of the test set, define parameters and describe the processing environment. After that we turn towards the actual evaluation and look at the computational results of the exact methods in detail. Ultimately the metaheuristic approaches are extensively evaluated.

5.1 Problem Instances

The problem we describe emerges from a real-life use case of our industrial partner which also provided us some data from a production system. In total we obtained 27 PLP instances which all have 20 periods, 4 to 8 product types and 79 to 1585 orders. This set of instances will be called from now on R_1. As these instances do not suffice for a thorough evaluation and we do not want to restrict ourselves to their size, we designed also two random instance generation procedures which are described in the following.

5.1.1 Perfectly solvable instances

We devised a method of generating instances which allow for a perfectly balanced solution with zero cost, that we know from the construction process. That is, of course, a restriction of generality, but it is extremely useful as a means of evaluating the optimality gap for large instances which would otherwise be impossible as we have currently no way of solving them exactly with usual compute resources. Despite the existence of a perfectly balanced solution with no priority inversions the instances are still not easy to solve to optimality, at least not as long as you don't provide the information of perfect realizability to the solvers.

The instance generation process relies on the subroutine for random integer partitioning shown in Algorithm 5.1. It takes as arguments the integer to partition, the number of partitions and a minimum value for each partition. The main idea is to represent the number n as an array of $n - k \cdot minV$ zeros and then inserting $k - 1$ ones at random positions. In the resulting array an integer partition of the number $n - k \cdot minV$ into k parts can be found by looking at the number of zeros between every two neighboring ones. Finally we add $minV$ to every element of the result array to obtain the requested partition with minimum value.

Algorithm 5.1: Integer partitioning algorithm

```
Data: $n$, $k$, $minV$
Result: An array with k integers whose sum equals n, each of which being
        $\geq minV$
1 let $array \leftarrow$ an array consisting of $n - (k \cdot minV)$ zeros;
2 Insert $k - 1$ ones into $array$ at random positions;
3 let $spaces \leftarrow$ number of zeros between the ones in $array$;
4 add $minV$ to every element of $spaces$;
5 return $spaces$;
```

Using this partitioning algorithm, Algorithm 5.2 defines the procedure for generating random instances with a fixed number of orders, periods and products. First the total number of orders is partitioned into one part for each period where each part has to have at least as many orders as we have products. This is important because every product needs to meet its target in every period in order to achieve an objective value of 0. The same thing is done for each period to decide upon the number of orders for each product type.

Next we draw the overall target value for the production volume (which is the same for each period) by taking the desired *avgDemandPerOrder* and multiplying with the average number of orders per period plus a random deviation of at most 10%. Then we partition that value into one part for each product, which is the demand for each product per period.

Finally we need to partition the demand for each product, which we decided upon in line 4, into the number of orders for each period and product which we calculated in line 2. The priorities must be chosen such that no inversion can exist, which is achieved by assigning each period a range of priority values decreasingly such that the ranges do not overlap, and choosing for each order randomly one of the allowed values. From this data the order and product type list can be built, which completes the instance. The optimal solution is known as well from the construction process.

Using Algorithm 5.2 we generated 1000 instances, sampling the parameters for each one independently as follows: The number of orders k is chosen from $100 \ldots 4000$, the number of periods n from $2 \ldots 80$, the number of products m from $1 \ldots 20$ and

Algorithm 5.2: Procedure for the creation of perfectly solvable instance

```
Data: $m$, $n$, $k$, $avgDemandPerOrder$
Result: A realizable instance with $m$ products, $n$ periods, $k$ orders and the
        optimal solution
1  let $ordersPerPeriod \leftarrow$ partition$(k, n, m)$;
2  let $ordersPerPeriodAndProduct \leftarrow$ partition($ordersPerPeriod[o]$, $m$, 1) for every
     order $o$;
3  let $plannedDemand \leftarrow \frac{k \cdot avgDemandPerOrder}{n} \pm 10\%$;
4  let $plannedDemandPerProduct \leftarrow$ partition($plannedDemand$, $m$,
     max($ordersPerPeriodAndProduct$));
5  let $orderDemands \leftarrow$ partition($plannedDemandPerProduct[p]$,
     $ordersPerPeriodAndProduct[t, p]$, 1) for every period $t$ and product $p$;
6  let $allowedPriorities \leftarrow$ for each period $n$ a distinct set of priorities s.t. they
     decrease with increasing $n$;
7  let $orderPriorities \leftarrow$ choose for each order one of the priorities which are
     allowed according to the period of the order;
8  build the list of orders and products and shuffle them;
9  assign random product names;
10 return a new solution from the list of orders and products and the optimal
     solution;
```

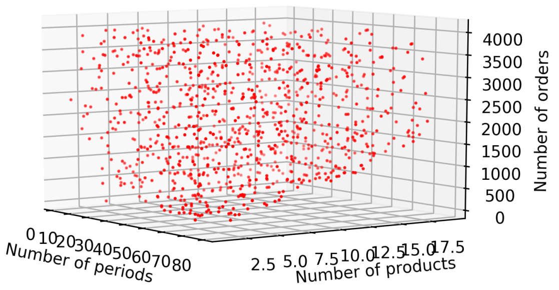


Figure 5.1: 3D Scatter plot of the parameters of the randomly perfect instance set R_2

avgDemandPerOrder from $5 \ldots 500$. The resulting set of instances is subsequently called R_2.

Figure 5.1 visualizes the randomly chosen parameters of the 1000 generated instances. One can see clearly that the combination of a high number of periods and products but a low number of orders is not possible, because we need to have at least one order per period and product to achieve a perfect assignment. However, apart from that the distribution of parameters looks uniform.

5.1.2 Random instances

We also devised a second instance generation procedure where the optimal solutions are not known by design and we can't even guarantee that there exists a feasible one, which is surely a more practice-oriented approach. The instances are designed to share some properties of the 27 realistic instances:

- There exist only a limited number $l \ll k$ of different order demand values. This means we frequently see repeated orders which may have different priorities though.
- Orders of different products draw their demand data from different distributions. Whereas product a may have demand values between 0 and 1000, product b may have it between 0 and 5000.
- Sometimes there exist product types whose number of orders is smaller than the number of periods which implies that the demand for some periods will exceed the target while for others it must be zero.

The actual generation process is very simple. Given a number of orders k, periods n and product types m the algorithm works as follows:

1. Partition the number of orders k into m parts $c_1 \dots c_m$.
2. Choose the maximum priority of all orders $p_{max} \in [1; 3n]$
3. Choose $1 - 50$ allowed demand values $d \in [1; random(1000 - 5000)]$ for each product p, named D_p.
4. For each product $p \in [1; m]$, generate c_p orders, choosing the demand from the set D_p and the priority from $[0; p_{max}[$.

Using this procedure, we generated the instance set R_3 consisting of 1000 instances by sampling the parameters randomly as it has been done above with the other procedure. The number of orders k is chosen from $100 \dots 4000$, the number of periods n from $2 \dots 80$ and the number of products m from $1 \dots 20$. Furthermore, we generated a set of 10 small instances, named R_4, where the number of orders k is chosen from $30 \dots 100$, the number of periods n from $5 \dots 20$ and the number of products m from $1 \dots 5$.

5.2 Experimental Setting

The instances which are described above are split into training and test set so that the parameter tuning is not executed on the same instances as the validation. The test set consists of the whole set of realistic instances R_1, 50 instances of R_2, 50 instances of R_3 and all 10 instances in R_4. Table 5.1 provides an overview of the instance sets and the way they were split.

Table 5.1: Overview of the different instance sets and the split into training and test set

	Name	Count	Training set	Test set
R_1	`realistic_instance`	27	-	01-27
R_2	`randomly_perfect`	1000	0001-0950	0951-1000
R_3	`randomly_generated`	1000	0001-0950	0951-1000
R_4	`randomly_generated_small`	10	-	01-10

Table 5.2: Minimum, maximum, mean and standard deviation of number of orders k, number of product types m and number of periods n for every part of the test set

		min	max	mean	std
Parameter	**Instance Set**				
k	R_1	79	1585	307.19	412.56
	R_2	105	3896	1595.86	954.09
	R_3	112	3991	2076.76	1207.02
	R_4	34	98	61.20	19.70
m	R_1	4	8	6.93	1.24
	R_2	1	19	8.82	5.50
	R_3	1	19	9.04	5.48
	R_4	1	4	2.80	1.03
n	R_1	20	20	20.00	0.00
	R_2	4	78	39.50	22.48
	R_3	4	77	39.26	22.04
	R_4	7	18	10.90	4.04

We chose to build the test set out of four different instance types because we wanted to make sure that our algorithms can cope with different characteristics and sizes. The size distribution is shown by Table 5.2 which states for each instance parameter — k (number of orders), m (number of product types), and n (number of periods) — the minimum, maximum and mean value on each part of the test set. The smallest instances are R_4, followed by the realistic instances R_1. The instances coming from R_2 and R_3 are much larger on average as we want to evaluate also the scalability of our algorithms. The set of test instances is publicly available on the following website, where also information about the instance format as well as validation scripts are published: https://dbai.tuwien.ac.at/staff/jvass/production-leveling.

We use the absolute-difference-based objective function (2.7) to produce all the subsequent results. The reason is that in our setting the advantages over the objective with squares outweigh the disadvantages, especially because it enables us to solve much more instances exactly. The evaluation of the metaheuristics could just as well be done using the quadratic objective function (2.11) but as we want to compare to the exact results we use formula (2.7) as well. Hard constraint violations are not part of the objective function but undergo a special treatment where possible by reporting the number of violated constraints as a separate number or separate plot. In some cases, e.g. statistical significance tests, we handle objective and constraint violations at once by adding them up. Due to the small magnitude of the objective a penalization factor for hard constraint violations is not necessary. Furthermore, we devised default values for the the objective function weights a_1, a_2 and a_3 in cooperation with our industrial partner, namely 1, 1 and $1/3$, respectively. All experiments of the evaluation are using this weighting.

Wherever nothing different is stated the algorithm parameters are defined as follows:

- The greedy heuristic has only one parameter, r, for controlling the amount of randomness, which we set to 1 (i.e. deterministic) for all the experiments. The parameter is only necessary for some local search techniques like GRASP which would require a randomized construction heuristic.
- For VND the move neighborhood is used first because it can be enumerated very quickly. Only when no improving move can be found any more the larger swap neighborhood gets employed. This ordering leads to a much quicker termination because the first neighborhood is searched much more often than the second. As the neighborhood exploration strategy we use Next Improvement with restart at the last position (as defined in Section 4.2.2), which showed at least equal performance to Best Improvement in preliminary experiments.
- The parameters of Simulated Annealing are tuned automatically. The concrete process and the results get introduced later on.
- The MIP model is executed using Gurobi Optimizer 8.1.1 [Gur19] using the default settings.
- The CP model is compiled using Minizinc 2.3.1 [Net+07] for Gecode 6.1.1 and Gurobi Optimizer 8.1.1.
- Each run was limited to a single thread and a maximum of 15 GB memory usage.

All experiments were conducted on a computing cluster with with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 252 GB of memory, running Ubuntu 16.04.1 LTS. The metaheuristic algorithms are implemented in C# and executed using Mono 4.2.1.

5.3 Evaluation of exact modeling techniques

In this section we examine the CP model and the MIP formulation presented in Section 4.1 with respect to their practical performance. First we present benchmark results on the test set. We aim to determine, which of the two produces better results, and explore how the instance size correlates to the quality of found solutions, if at all. After that we take a deeper look into the results of the native MIP model. Finally we evaluate the effect of the dominance constraints. The concrete results of all exact methods can be found in tabular form in Appendix A.

5.3.1 Comparison of exact modeling techniques

In the following we present a comparison of the two exact models on different solvers. While the native MIP model is implemented specifically for Gurobi, the CP model can theoretically be executed on any solver that supports MiniZinc models. Unfortunately, most of the CP solvers do not support continuous variables, which we need for the normalization of the objective function even though the actual decision variables are binary. Therefore, we ended up with only one CP solver, namely Gecode, which we compare against native Gurobi and through MiniZinc. We executed each of the variants for each instance of the test set on a single core with a time limit of one hour. If the time or memory limit was exceeded, the execution was aborted and the latest found solution counted.

The results are shown by Figure 5.2, which depicts the relative optimality gap [1] of the best solution found by each of the three variants for all instances where at least one of the variants found a solution. The dedicated MIP model for Gurobi is shown by blue 'X'-signs, Gurobi through the CP-model by orange stars and Gecode through the CP-model by red plus signs. Marks are missing if a variant could not find any solution within the limits.

We can see clearly that the number of instances which have been solved by the dedicated MIP model for Gurobi is the highest, followed by Gurobi through MiniZinc. This difference presumably stems from the necessary compilation step of the latter variant, which itself exceeded exceeded the memory limit many times and used an increasingly large percentage of the execution time the larger the instance was. Where both variants found solutions the native version found in 15 cases the better solution, the MiniZinc version in 9 and in 7 cases the solutions were equal. However, in all cases the differences were quite small. Between the MIP variants we can, therefore, conclude that both can solve the smaller instances similarly well. However, when pushing the solvers to the limit, the benefits of a dedicated MIP model become very apparent.

Furthermore, the figure shows that the CP solver Gecode could not find any solution for most of the instances at all. While the two Gurobi approaches solved 41 and 31 out of 137 instances, respectively, Gecode could only find a feasible solution in three cases. In addition, these solutions were in all cases worse compared to the Gurobi solutions. We cannot conclude with certainty, however, that CP is generally not suited for solving the PLP because the `deviation` constraint is not implemented in any suitable solver and there are very few which currently support continuous variables. If these circumstances happen to change at a later point in time, it would definitely be interesting to repeat this experiment one more time.

[1] The relative optimality gap is a common measure of how near a solution is to a global optimum. It is calculated as the percentage corresponding to one minus the ratio between the cost value of the best lower bound and the cost at the incumbent solution.

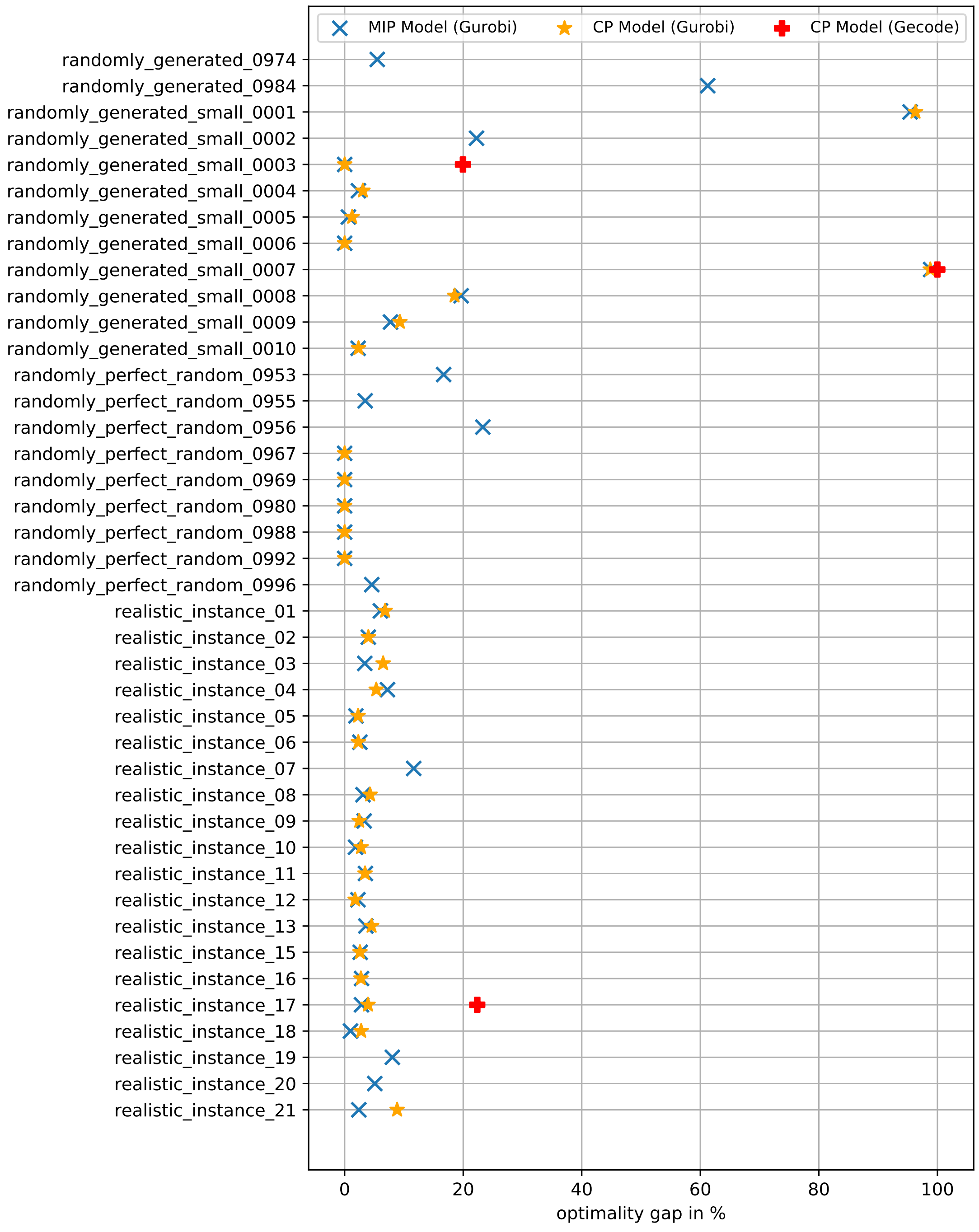


Figure 5.2: Comparison of the solutions found by our three exact approaches. The optimality gap of all instances, where at least one of the approaches found any solution, is compared. The optimality gap of solutions in R_2 cannot be calculated due to the best bound being 0, so we report instead $objectiveValue \cdot 100$ for this group.

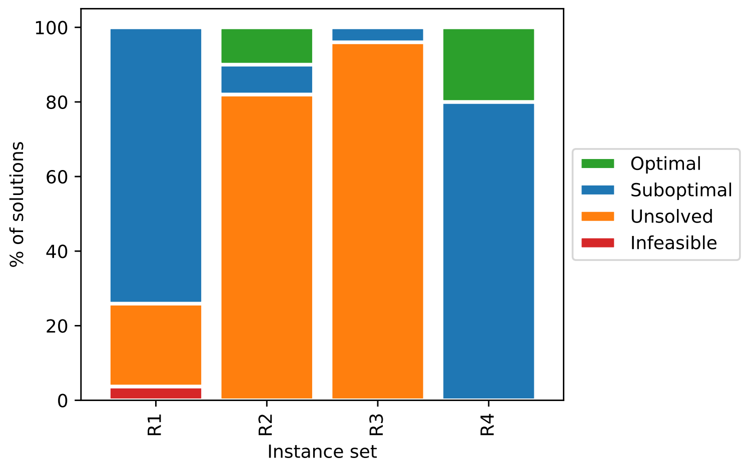


Figure 5.3: Share of solution statuses of MIP for each subset of the test set. Optimal means proven optimal. Suboptimal implies that an integer solution has been found but it has not been proven that it is optimal. Unsolved means that within the time/memory limit no integer solution has been found and it is thus unclear whether there exists a feasible solution at all. Infeasible means that the solver proved that no feasible solution exists.

5.3.2 Evaluation of the dedicated MIP model

The comparison of exact modeling techniques showed, that the dedicated MIP model works best in our problem setting. Therefore, we now take a deeper look into the results of this approach. We first break down the results by the different instance sets of which the test set is composed. Afterwards we will investigate how the instance size affects the solution quality.

We want to find out how well each part of the test set can be solved using MIP. For a description of the different parts please refer to Table 5.2. Figure 5.3 visualizes the shares of optimally solved, feasibly but not optimally solved, infeasible and unsolved instances per group R_1 to R_4. The most noticeable difference between the sets is that in R_2 and R_3 the vast majority of the instances are unsolved while for the other two most of the instances are solved (but still not proven optimal). Presumably, the reason for that is that most of the instances in these sets are very large. An interesting fact is, though, that about 10 % of the instances in R_2 could be solved to proven optimality but not a single one in R_3 even though the instance sizes of the two sets have been sampled from the same distribution. One potential reason for that could be that the instances in R_2 are designed to have optimal solutions with objective value 0. That should make optimality proofs easy for the solver once the optimal solution has been found because no part of the objective function can by negative.

Table 5.3: Optimality gap of MIP for suboptimal instances in R_1 and R_4

	min	max	mean	std
R_1	0.99%	11.65%	3.96%	2.55%
R_4	0.63%	98.85%	31.14%	41.52%

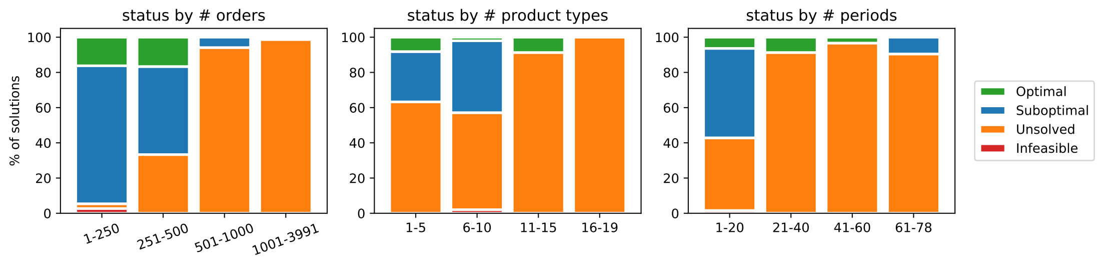


Figure 5.4: Solution statuses of MIP on the test set, grouped by value ranges of the instance features k (number of orders), m (number of products) and n (number of periods)

For the instance sets R_1 and R_4 over 70% of the instances end up with some solution, which is not proven optimal. We want to investigate how good these solutions are and use for that purpose once more the relative optimality gap with respect to the best lower bound. Table 5.3 shows the minimum, maximum and mean optimality gap as well as the standard deviation for all suboptimal solutions. With an average gap of only 3.96% the realistic instance set R_1 is solved really well, so that the MIP model might be usable in practice when the instances are not too large and a runtime of one hour is not an issue. On the other hand, R_4 has a low minimum and a high maximum gap as well as a large standard deviation. That means that the randomly generated instances are quite difficult to solve using MIP, even though the ones in R_4 are mostly smaller than the realistic ones.

Finally, we investigate in more detail how the instance size correlates with the results of the MIP model. Figure 5.4 visualizes the solution statuses of all instances in the test set, grouped by the number of orders k, the number of products m and the number of periods n, from left to right. The most apparent relationship is a correlation between the number of orders k and the percentage of unsolved instances. While below 250 orders almost every instance has either been proven feasible or infeasible, the share of unsolved solutions increases drastically when increasing k. When looking at the middle and right-hand-side figure, we can see that the share of unsolved instances is also increasing with increasing number of periods and product types but it starts already quite high in the smallest bin. We can conclude that instances with 250 orders or less can be solved with a high probability by the MIP model, but there is no such bound which we could state on the number of periods or product types. While increasing n and m clearly complicates the problem, making them small does not automatically make the problem easy to solve.

Table 5.4: Results for the native Gurobi MIP model with/without dominance constraints. The left column shows the number of instances in the test set which have been solved and the right column the average objective value of those instances, which have been solved by both methods.

	# solved instances	mean objective
MIP with dominance	41	0.627583
MIP without dominance	106	0.636446

5.3.3 Dominance Constraints

All the experiments presented so far were conducted with the dominance constraints (see Section 4.1.1) included. In the following we investigate the benefits of this inclusion. To that end we excluded the dominance constraints from the dedicated MIP model and evaluated it again on the test set with the same time and memory limit.

Table 5.4 shows the results on the test set for the dedicated MIP model with and without dominance constraints in comparison. At first sight it surprises that the variant with dominance constraints included solves not even half the number of instances compared to the variant without dominance constraints. However, it turns out that the additional solutions are completely useless because even our Greedy heuristic produces better results in 63 out of the 65 cases – and that in almost no time. Compared to VND and Simulated Annealing, not a single one of the additional solutions is better. When looking at the right column, which shows the average objective value of all instances solved by both variants, we can clearly see that the dominance constraints do bring a small advantage. The improvement corresponds on average roughly to a 0.8% lower optimality gap.

To sum up, it is advisable to include dominance constraints into the exact models because the solution quality increases. Large instances may not be solvable any more when including the constraints but that is not a problem because the solutions which are cut off have a low-quality, which is easily exceeded (e.g. by our metaheuristic approaches).

5.4 Evaluation of metaheuristic local search

As we have seen, neither of the investigated exact methods is suitable for reliably solving large instances, which is why we developed local search methods for the PLP as well. In this section we will first deal with the automatic parameter tuning for Simulated Annealing and validate the claim that it is sound to fix the cooling rate. Then we analyze benefits and shortcomings of our two approaches VND and Simulated Annealing in detail on the basis of results on our test set, comparing also against the greedy heuristic. Thereafter we examine the sensitivity of Simulated Annealing to variations in the weighting of the neighborhoods. Finally we examine how close the metaheuristic solutions get to the global optima by using dual bounds.

Table 5.5: Configuration space of Simulated Annealing. The upper section are parameters which we tuned while the ones in the lower section have been set to fixed values.

Parameter	**Type**	**Min**	**Max**	**Default**	**Tuned**
Iterations Per Temperature	integer	10^3	10^6	10^3	$2.52 \cdot 10^5$
Move Neighborhood Probability (%)	integer	0	100	50	40
Initial Temperature	real	0.1	10.0	5.0	0.22
Minimum Temperature	fixed real	0	0	0	0
Cooling Rate	fixed real	0.95	0.95	0.95	0.95

5.4.1 Algorithm Configuration

As described in Section 4.2.3, Simulated Annealing depends on parameters whose setting has a huge influence on the algorithm's efficiency and effectiveness. We deal with their configuration by means of Sequential Model-based Algorithm Configuration (SMAC), an automatic algorithm configuration tool written in python. It relies on Bayesian Optimization in combination with an aggressive racing mechanism in order to efficiently search through huge configuration spaces [Lin+19].

We applied SMAC to tune the parameters of Simulated Annealing as it was presented above. The set of instances which was use for the tuning can be found in the column *Training Set Selection* of Table 5.1. The parameter optimization was executed for 24 hours on 24 cores in parallel. We used a time limit of five minutes per run and no iteration limit. The cooling rate was not tuned but set to a value of 0.95, which is not a restriction of generality as long as the number of iterations per temperature can still be adjusted (see Figure 4.3). This claim will be verified in a separate experiment.

We tuned the initial temperature t_{max}, the number of iterations per temperature w and the probability p that the move neighborhood is used to generate the next random move (hence $1 - p$ is the probability of the swap neighborhood). Tuning the minimum temperature t_{min} is not necessary because the results cannot get worse when Simulated Annealing is run until the time limit instead of aborting when the minimal temperature is reached. Indeed, preliminary results showed that setting the minimum temperature to zero instead of using the tuning results of SMAC improves results to a small but significant extent. The configuration space with minimum and maximum values as well as the defaults and the tuning result is shown in Table 5.5.

5.4.2 Experiments about fixing the cooling rate

We claimed in Section 14 that the cooling rate α could be set to a constant value because it was redundant as long as the number of iterations per temperature w can be freely configured. During algorithm configuration we did exactly that and set $\alpha \leftarrow 0.95$. Now we want to verify by means of an experiment that this is sound. We derive four more cooling schedules from the one defined by the result of parameter tuning whose temperature

Table 5.6: Five equivalent cooling schedules which have the same slope on average. The value for w in the line with $\alpha = 0.95$ comes from parameter tuning and the rest has been derived so that the slope is does not change.

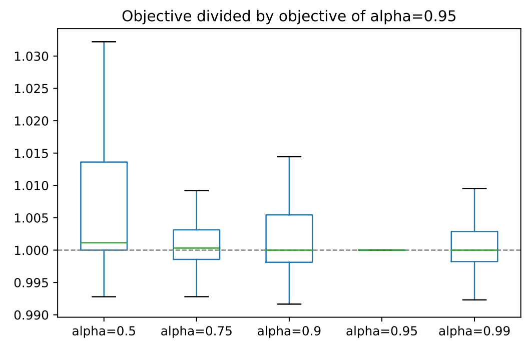
Cooling Rate α	**Iterations per temperature w**
0.50	3412581
0.75	1416349
0.90	518723
0.95	*252533*
0.99	49481

profile follows the same slope. Then we benchmark each configuration on the whole test set ten times with different random seeds and take the median of the objective values and number of constraint violations for each instance.

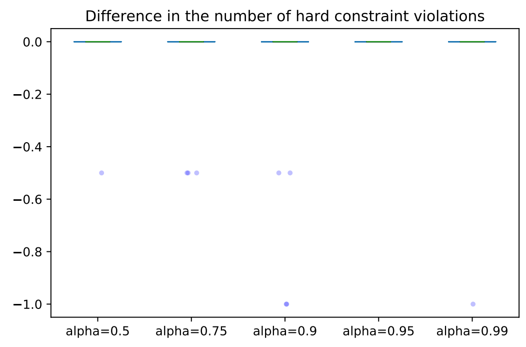
In Section 14 we already introduced an equation which allows to derive cooling schedules with equal average slopes. We selected the alternative cooling rates 0.5, 0.75, 0.9 and 0.99 and computed the associated values of w. A summary of the resulting cooling schedules is shown in Table 5.6.

Figure 5.5a shows a box plot for each of the schedules, each of them plotting the median objective value resulting from the derived schedule divided by the median objective value resulting from the original schedule, per instance. That means, values below one show an improvement over the original schedule and values above one a degradation. For $0.75 \leq \alpha \leq 0.99$ the whole inter-quartile range is within $\pm 0.5\%$ of the original schedule's objective value and nearly all the rest in $\pm 1.5\%$ (except for a couple of outliers outside the plotting range). The median of the schedules $\alpha = 0.9$ and $\alpha = 0.99$ is almost exactly at 1 whereas the other two variants have median differences slightly higher than one, although the differences are still very small. Figure 5.5b visualizes the same for the number of hard constraint violations, except that we do not divide but take the difference between the alternative schedules and the original one because the interpretation is more intuitive in this case. The whole box plot, except for some outliers, is zero which means that for most of the instances there is no change in the number of hard constraint violations. However, there are a few outliers going down to -1 which means that for these instances the alternative schedules have one violation less. Compared to the 137 instances of the test set the number of outliers is very small though.

In order to clarify whether the differences which we see are statistically significant or not we conducted a Wilcoxon signed-rank test between the original schedule and each of the derived ones. In order to take also the hard constraint violations into account their number was added to the objective value as a penalty. The null hypothesis was that the median of the differences is zero and the alternative that it is different from zero. We want to reject the null hypothesis in case we find a p-value which is smaller than 0.05. The result of the tests yielded a p-value of $2.6 \cdot 10^{-8}$ for $\alpha = 0.5$, 0.041 for $\alpha = 0.75$, 0.966



(a) Relative objective value of alternative schedule divided by objective value of the original schedule.



(b) Number of hard constraint violations of alternative schedule minus value for the original schedule.

Figure 5.5: Results of the experiment with different cooling schedules. All the results are comparisons per instance against the schedule with $\alpha = 0.95$.

for $\alpha = 0.9$ and 0.26 for $\alpha = 0.99$. That implies that we must reject the null hypothesis for the schedules with $\alpha \leq 0.75$. For the other two schedules the statistical test does not let us reject the null hypothesis that the differences which we see are the result of chance.

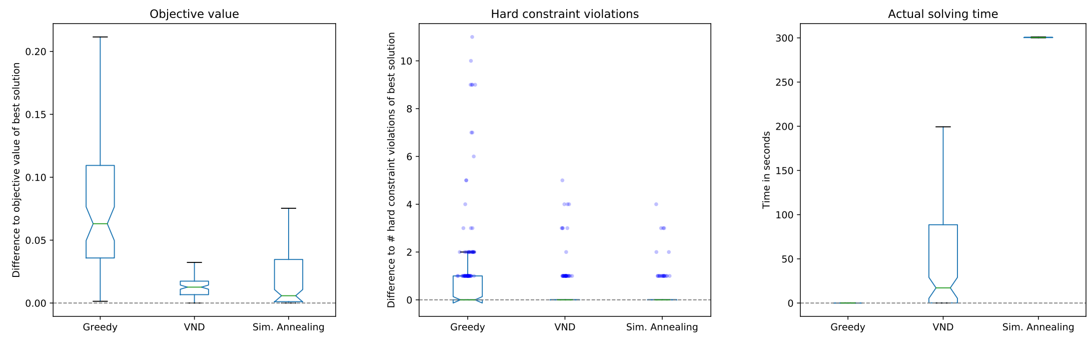
To sum up, the claim that we can change alpha without changing the result is valid in our setting as long as α is set high enough (i.e. in our experiment at least 0.9). That implies that we were on the safe side when fixing it to 0.95 during parameter tuning. For values $\alpha \leq 0.75$ there exists a tiny but statistically significant difference which corresponds to a median objective value which is about 1 ‰ above the one of our tuned configuration.

5.4.3 Comparison of metaheuristic techniques

We want to compare now the presented heuristic and metaheuristic techniques for solving the PLP, namely the greedy algorithm presented in Section 4.2.1, and VND and Simulated Annealing which have been presented in Section 4.2.3. Therefore, these three algorithms were benchmarked on the test set with the usual settings. For Simulated Annealing we conducted 10 runs to account for randomness in the search process and aggregated the runs by taking the median value of each measure. In order to be able to compare objective values and hard constraint violations visually, we report for each instance the difference to the best solution we ever obtained using any method and time limit [2].

The results are shown by Figure 5.6: To the left one can see the objective values of the three approaches. The median difference between the greedy heuristic's objective values and the best known ones is about 0.07. Furthermore, we can see in the center figure

[2]This is a sound approach because the objective function is already normalized so that the instance size does not have an influence on the magnitude of the objective value. Using a ratio instead of the difference, like in the previous experiment, is not possible here because the best known solution for the instances in R_2 have objective value 0 which would lead to a division by 0.



(a) Objective value of metaheuristic solution methods minus best known objective value.

(b) Number of hard constraint violations of metaheuristic solution methods minus value of the best known solution.

(c) Solving time of solution methods in seconds.

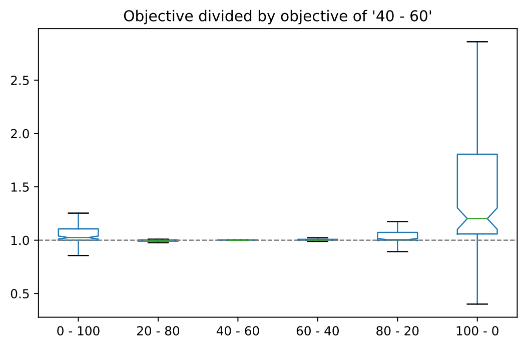
Figure 5.6: Comparison of objective values, hard constraint violations and solving time of the Greedy heuristic, VND and Simulated Annealing on the test set

that a considerable number of instances could not be solved without hard constraint violations by the Greedy heuristic. Expressed in numbers, that's the case for 63 out of 137 instances or some 46%. Compared to the Greedy, VND delivers solutions with much better objective values and fewer constraint violations, but there are still 33 solutions or 24% where at least one capacity constraint is violated. Simulated Annealing achieves the best median objective value of all methods and furthermore the fewest instances with constraint violations (22 out of 137, 16%). The non-overlapping notches of the box plot in the left figure indicate that the difference to VND is significant. The rightmost plot shows the solving time of the different methods in seconds. The greedy heuristic needs always less than a second of time. VND is also mostly fast, because the search continues only until a local optimum w.r.t. all neighborhood structures is found, which takes long only for the very largest of our instances. Simulated Annealing uses always the complete available time because we don't stop at a minimum temperature in order to maximize the solution quality.

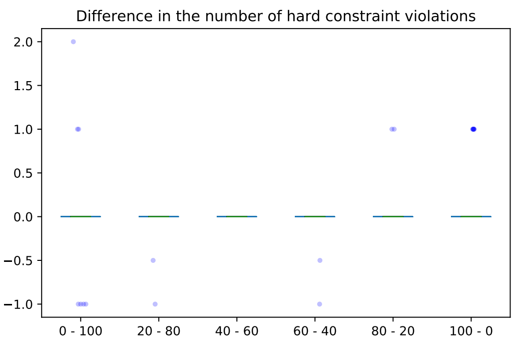
Appendix B presents the objective value and the number of hard constraint violations for each instance of the test set for all reported algorithms.

5.4.4 Sensitivity to neighborhood weightings in Simulated Annealing

Before selecting the next random move in the Simulated Annealing algorithm the neighborhood is chosen randomly according to some weighting. This weighting has been tuned by SMAC, resulting in a probability of 0.4 for the move neighborhood and 0.6 for the swap neighborhood. The tuning progress of SMAC revealed quite large fluctuations in this weighting. Therefore, we conduct a sensitivity analysis in order to find out what impact different weightings have on the results.



(a) Relative objective value of alternative schedule divided by objective value of the original schedule.



(b) Number of hard constraint violations of alternative schedule minus value for the original schedule.

Figure 5.7: Simulated Annealing results with different neighborhood weightings, compared per-instance with the schedule '40 - 60'. The first number of each label is the weight of the move neighborhood and the second the weight of the swap neighborhood.

We evaluate 6 different weightings, one of which is the result of SMAC. The probability p for the move neighborhood in the 6 scenarios ranges from 0 to 1 in steps of 0.2 and the probability of the swap neighborhood is the complementary probability $1 - p$. Each configuration is executed on the test set 10 times with the usual time limit of five minutes. The runs are again aggregated using the median.

Figure 5.7a shows a box plot for each of the alternative weightings, each of them plotting the associated objective value divided by the objective value of the original weighting. The labels 'x - y' mean that the move neighborhood has weight x and the swap neighborhood has weight y. The objective value gets worse in the extreme cases which can be seen because the leftmost and the two rightmost boxes lie completely above the dashed line. The other cases are practically equal which means that the objective value does not change compared to the reference weighting.

Figure 5.7b shows the difference of the number of hard constraint violations to the respective number of the reference weighting '40 - 60'. All boxes are completely flat contained in $\{0\}$ which means that the inter-quartile range is equal for all weightings. The outliers show that a few instances of the extreme configurations have one or two more hard constraint violations more than the reference configuration, while the weightings '20 - 80' and '60 - 40' have tendentially fewer. However, compared to the number of 137 instances contained in the test set the amount of outliers is very small.

To sum up, the tested neighborhood weightings between '20 - 80' and '60 - 40' are equally good and therefore it is logical to assume that the untested weightings in between are good as well. Using one of the more extreme weightings leads to worse results, especially in the case where only the move neighborhood is used.

5.4.5 Optimality gap of metaheuristic solutions

An interesting question during the evaluation of metaheuristic techniques is by how far the solutions deviate from the optimal solution. As stated previously, MIP solvers measure this property in terms of the optimality gap, which is calculated by taking one minus a lower bound divided by the objective value. In the following we assess how large the optimality gap of the metaheuristic solutions is which can be done by using the lower bounds obtained though MIP. However, as only a small fraction of the instances could be solved well enough that this kind of evaluation makes sense, we present afterwards also an analysis based on the randomly generated instances with known optimal solutions.

1. **Optimality gap for small instances**: We want to analyze the optimality gap of the solutions produced by our metaheuristic approaches. Therefore, we use the best dual bound found by the MIP solver. In order to get even better bounds, we executed the solver again with a time limit of 10 hours and used for each instance the best available bound. We restrict this evaluation to the instances in R_1 and R_4 because they are the only sets where the instances are small enough so that we could obtain mostly good bounds. Furthermore, we select the subset of instances whose optimality gap of the best MIP solution is below 10% because we can only assume safely that the dual bound is good if the gap is small. This step eliminates 8 instances of R_1 (30%) and 4 of R_4 (40%), which means that 25 instances remain for our evaluation. By using the best bound the optimality gap for each of the metaheuristic approaches is calculated on the selected instances.

   Figure 5.8 shows the optimality gap for each instance in the reduced set for the greedy heuristic, VND, Simulated Annealing and MIP. For solutions, which are not valid because of constraint violations, no mark is shown. The figure conveys, that the solutions found by the greedy construction heuristic have gaps between 10% and 25% and a considerable number of instances has is not solved to feasibility at all. VND already achieves drastic improvements by solving all instances but four with a maximum gap of 10% and mostly around 5%. Simulated Annealing is clearly the best metaheuristic for this restricted set of small instances, as it produced always valid solutions which have a similar optimality gap as the MIP solutions and in several cases even better. The gap is most almost always below 5% percent and with an average of about 3%.

   The resulting numbers state how large the relative difference between metaheuristic and optimal solutions is *at most* [3]. This result is interesting because it proves that our Simulated Annealing approach solves the majority of the small instances (which includes most of the realistic instances) extremely well. On average the solutions are at most 3% above the optimal one.

[3]The gap is calculated using the best lower bound which was proven by the MIP solver. As most of the solutions were not proven optimal, the bounds are most probably smaller than the optimal objective value and thus the calculated gap an upper bound of the actual gap.

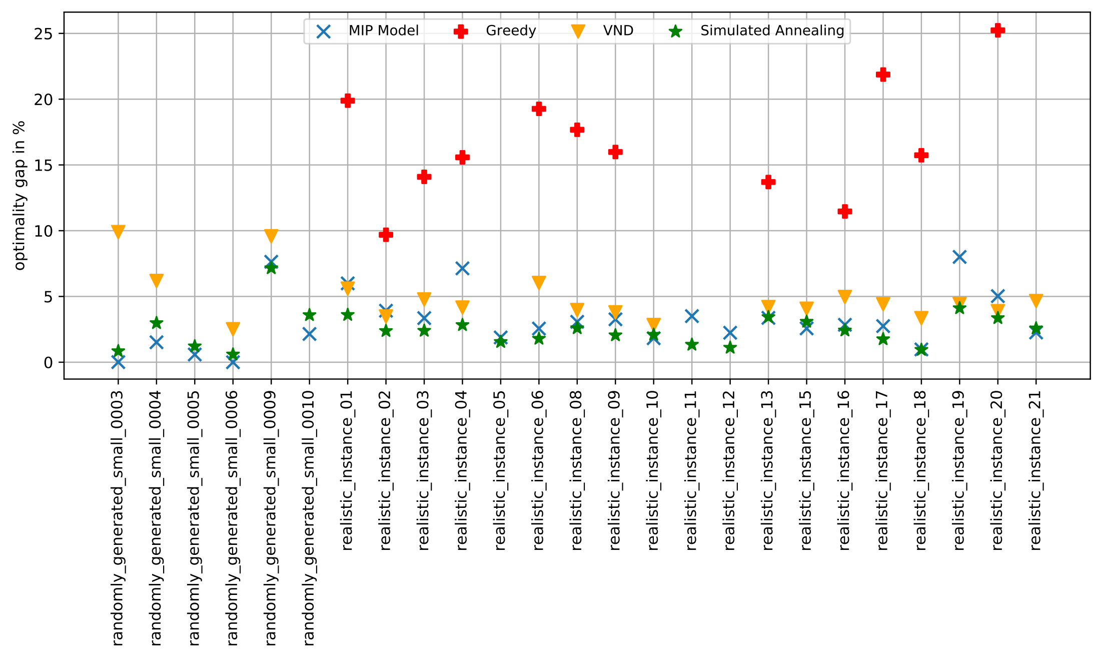


Figure 5.8: Comparison of the optimality gap of valid solutions obtained though Greedy, VND, Simulated Annealing and MIP. Marks are missing for solutions that violate constraints. The gap is computed using the dual bounds of MIP. The evaluation contains all instances of R_1 and R_4 which could be solved with an optimality gap of 10% or lower using MIP.

2. **Comparison on large randomly perfect instances $\mathbf{R_2}$**: The instances in R_2 have been constructed in a way that the optimal solutions are known and have the objective value 0. This enables us draw some conclusions about the optimality gap also for larger instances than those for which we can obtain bounds using MIP. However, the optimality gap as defined above cannot be computed for the instances in R_2 because the best known bound is zero which would lead to a division by zero. Instead, we can say something similar by looking at the objective function. The first objective function component g_1, defined in Equation (2.4) states, informally speaking, the gap to a hypothetical perfectly leveled solution averaged over all periods. In case of the instances R_2 this hypothetical solution actually exists and the value g_1 can be interpreted as the average percentage by which planned demand for each period exceeds or falls short of the target. The same argument holds for g_2, defined in Equation (2.5), which states average percentage by which planned demand for each period exceeds or falls short of the target, averaged over the different products. The objective component g_3, defined in Equation (2.6), can be interpreted as the percentage of actual priority inversions measured against the theoretical maximum number of inversions $\frac{k \cdot (k-1)}{2}$.

Table 5.7: The values of the objective function components g_1, g_2 and g_3 multiplied by 100 so that they can be interpreted as percentages for each each algorithm for each instance of R_2 where a valid solution has been reached. The rightmost column states for how many percent of the solutions each algorithm reached a valid solution.

	g_1(%)	g_2(%)	g_3(%)	valid (%)
Greedy	1.78	2.25	1.78	78.00
VND	0.04	0.34	2.62	96.00
Simulated Annealing (median)	0.32	0.47	7.95	92.00
Optimum	0.00	0.00	0.00	100.00

Table 5.7 shows the values of g_1, g_2 and g_3 multiplied by 100, so that they can be interpreted as percentages, as explained above for each each algorithm and for each instance of R_2 where a valid solution has been reached. The fourth column reports the percentage of valid solutions. We can see that Simulated Annealing and VND reach negligible mean deviations for the first two objectives. However, also the greedy heuristic produces levelings which deviate from the target by only 2% on average, so we can assume that this part of the task is not very hard for this instance set. With respect to the priority objective the results leave some more room for improvements. The greedy heuristic reaches the best value here (at the cost of fewer valid solutions and a worse leveling), followed by VND and Simulated Annealing. The best total results are clearly reached by VND for this instance set. It is not entirely clear why the results on R_2 differ so much from the results which are obtained on the other parts of the test set but we suspect that it might have something to do with that the greedy heuristic finds very good initial solutions on this instance set. That might help VND more than Simulated Annealing because that latter starts with a random search which destroys some of the good structure which was already there.

The above analysis of the optimality gap on small, realistic instances and the larger perfectly solvable ones revealed that our metaheuristic methods produce solutions which are only few percentage points away from the optimum. When taking also the comparison of the different metaheuristics on the whole test set into account, which was presented in the previous section, we can conclude that Simulated Annealing is the overall best of our algorithms for the PLP because it can both solve small instances in an excellent way and scale to the largest instances which we have.

CHAPTER 6
Conclusion

We introduced a new combinatorial optimization problem in the area of production planning, which concerns the assignment of orders to production periods. Thereby a number of production capacity constraints needs to be fulfilled and a work balancing objective as well as the prioritization of the orders must be optimized.

We started with giving a precise mathematical formulation and reviewing the literature for related problems. Thereafter a proof of NP-hardness was given and the Fixed-Order Production Leveling problem was introduced and shown to be tractable. Then we turned towards solution methods for the original problem and introduced both a CP model and a MIP formulation. Finally we investigated local search methods and defined two neighborhood structures for the problem, which we evaluated using VND and Simulated Annealing.

The main results of this work are:

- The PLP is NP-hard, which was shown by an NP-completeness proof of the associated decision problem via a reduction from Bin Packing.
- The Fixed-Order Production Leveling problem, a variant where the correct ordering by priorities is enforced as a hard constraint, can be optimally solved in $\mathcal{O}(n \cdot k^2)$ time using our dynamic programming algorithm for all instances where all priority values are unique.
- The CP model is able to solve most of the small instances when being used with the MIP solver Gurobi, but does not produce satisfactory results with the CP solver Gecode. When using the native MIP model instead of the CP model, Gurobi performs slightly better, especially for large instances. The complexity of solving the MIP model grows with every dimension of the problem, but most notably with the number of orders k. For instances with less than 250 orders we can expect

either a feasible solution or the proof of infeasibility within one hour of time, while we cannot count on finding any solution for instances with $k \geq 300$.

- The introduced dominance constraints cut off parts of the search space which do not contain optimal solutions. In experiments with the native MIP model that led to fewer instances being solved, but those who were solved having a better objective value.
- With Simulated Annealing very good solutions can be obtained within five minutes. The real-life instances can all be solved well and for most of them we can show though the use of dual bounds that the solutions are within 5% of optimality. Experiments based on the set of instances with perfectly leveled solutions indicate that Simulated Annealing is capable of providing really good solutions also for much larger instances.
- Another metaheuristic local search method studied in this work is VND. While the objective value and the number of hard constraint violations is a bit higher on average compared to Simulated Annealing, it still finds good solutions most of the time. The instances with perfectly leveled solutions R_2 are even solved better by VND than Simulated Annealing.
- An experiment regarding the weighting of the two neighborhoods in Simulated Annealing showed that it is clearly advantageous to use both neighborhoods instead of either of them alone. The best weighting between the move and the swap neighborhood is between '20 - 80' and '60 - 40'.

A lesson learned during the practical phase of the thesis is the great importance of efficient delta evaluation procedures during local search. For example, the quadratic number of potential priority inversions renders a naive implementation of delta evaluation for the priority objective too inefficient so as to solve large instances of the problems well in a short amount of time, as we do. Out of this experience, we want to stress that an efficient implementation of delta evaluation can be very important to achieve good results.

Potential future work on Production Leveling includes improvements of the MIP model, e.g. by decomposition based techniques, so that more instances can be solved optimally or better bounds are obtained. That would open up ways to assess the quality of metaheuristic solutions of large instances. Another promising idea is to study extensions of the problem, like for example secondary resource usage, that could make the problem even more relevant in practice.

APPENDIX A

Computational results of exact modeling techniques

Table A.1: Objective value for all reported exact methods and the best lower bound proven by MIP for each instance of the test set.

	CP (Gecode)	CP (Gurobi)	MIP (Gurobi)	MIP bound	MIP without dominance
randomly_generated_0951	-	-	-	-	0.7510
randomly_generated_0952	-	-	-	-	-
randomly_generated_0953	-	-	-	-	0.5913
randomly_generated_0954	-	-	-	-	0.4922
randomly_generated_0955	-	-	-	-	-
randomly_generated_0956	-	-	-	-	0.5083
randomly_generated_0957	-	-	-	-	-
randomly_generated_0958	-	-	-	-	0.3441
randomly_generated_0959	-	-	-	-	0.4612
randomly_generated_0960	-	-	-	-	1.1012
randomly_generated_0961	-	-	-	-	-
randomly_generated_0962	-	-	-	-	0.5255
randomly_generated_0963	-	-	-	-	1.1472
randomly_generated_0964	-	-	-	-	0.4722
randomly_generated_0965	-	-	-	-	0.4806
randomly_generated_0966	-	-	-	-	0.5668
randomly_generated_0967	-	-	-	-	-
randomly_generated_0968	-	-	-	-	0.6821
randomly_generated_0969	-	-	-	-	-
randomly_generated_0970	-	-	-	-	0.1509
randomly_generated_0971	-	-	-	-	-
randomly_generated_0972	-	-	-	-	-
randomly_generated_0973	-	-	-	-	0.4981
randomly_generated_0974	-	-	0.8627	0.8153	0.8643
randomly_generated_0975	-	-	-	-	0.1470
randomly_generated_0976	-	-	-	-	-
randomly_generated_0977	-	-	-	-	-
randomly_generated_0978	-	-	-	-	0.7919
randomly_generated_0979	-	-	-	-	-
randomly_generated_0980	-	-	-	-	-
randomly_generated_0981	-	-	-	-	-
randomly_generated_0982	-	-	-	-	0.4966

Continued on next page

	CP (Gecode)	CP (Gurobi)	MIP (Gurobi)	MIP bound	MIP without dominance
randomly_generated_0983	-	-	-	-	-
randomly_generated_0984	-	-	0.2655	0.1029	0.2512
randomly_generated_0985	-	-	-	-	0.4897
randomly_generated_0986	-	-	-	-	-
randomly_generated_0987	-	-	-	-	0.3039
randomly_generated_0988	-	-	-	-	0.4941
randomly_generated_0989	-	-	-	-	-
randomly_generated_0990	-	-	-	-	0.1680
randomly_generated_0991	-	-	-	-	-
randomly_generated_0992	-	-	-	-	-
randomly_generated_0993	-	-	-	-	-
randomly_generated_0994	-	-	-	-	0.7256
randomly_generated_0995	-	-	-	-	-
randomly_generated_0996	-	-	-	-	-
randomly_generated_0997	-	-	-	-	-
randomly_generated_0998	-	-	-	-	-
randomly_generated_0999	-	-	-	-	-
randomly_generated_1000	-	-	-	-	0.4595
randomly_generated_small_0001	-	0.0198	0.0159	0.0007	0.0155
randomly_generated_small_0002	-	-	0.5448	0.4236	0.5228
randomly_generated_small_0003	0.3364	0.2693	0.2693	0.2693	0.2693
randomly_generated_small_0004	-	0.3516	0.3489	0.3407	0.3483
randomly_generated_small_0005	-	0.7901	0.7850	0.7800	0.7889
randomly_generated_small_0006	-	0.6820	0.6820	0.6820	0.6978
randomly_generated_small_0007	0.1080	0.0016	0.0017	0.0000	0.0017
randomly_generated_small_0008	-	0.2133	0.2163	0.1738	0.2109
randomly_generated_small_0009	-	0.8044	0.7906	0.7294	0.7988
randomly_generated_small_0010	-	0.7165	0.7160	0.6997	0.7160
randomly_perfect_random_0951	-	-	-	-	0.4343
randomly_perfect_random_0952	-	-	-	-	0.2468
randomly_perfect_random_0953	-	-	0.1670	0.0000	0.2768
randomly_perfect_random_0954	-	-	-	-	-
randomly_perfect_random_0955	-	-	0.0346	0.0017	0.1028
randomly_perfect_random_0956	-	-	0.2332	0.0000	0.2459
randomly_perfect_random_0957	-	-	-	-	0.4665
randomly_perfect_random_0958	-	-	-	-	-
randomly_perfect_random_0959	-	-	-	-	0.4659
randomly_perfect_random_0960	-	-	-	-	-
randomly_perfect_random_0961	-	-	-	-	0.4361
randomly_perfect_random_0962	-	-	-	-	0.4318
randomly_perfect_random_0963	-	-	-	-	0.4155
randomly_perfect_random_0964	-	-	-	-	0.5037
randomly_perfect_random_0965	-	-	-	-	0.4641
randomly_perfect_random_0966	-	-	-	-	0.4076
randomly_perfect_random_0967	-	0.0000	0.0000	0.0000	0.0222
randomly_perfect_random_0968	-	-	-	-	0.4858
randomly_perfect_random_0969	-	0.0000	0.0000	0.0000	0.0000
randomly_perfect_random_0970	-	-	-	-	0.4242
randomly_perfect_random_0971	-	-	-	-	0.3972
randomly_perfect_random_0972	-	-	-	-	0.5076
randomly_perfect_random_0973	-	-	-	-	0.4632
randomly_perfect_random_0974	-	-	-	-	0.2860
randomly_perfect_random_0975	-	-	-	-	0.4050
randomly_perfect_random_0976	-	-	-	-	0.5041
randomly_perfect_random_0977	-	-	-	-	0.5203
randomly_perfect_random_0978	-	-	-	-	0.5181
randomly_perfect_random_0979	-	-	-	-	0.5170
randomly_perfect_random_0980	-	0.0000	0.0000	0.0000	0.0000
randomly_perfect_random_0981	-	-	-	-	0.2639
randomly_perfect_random_0982	-	-	-	-	0.4659
randomly_perfect_random_0983	-	-	-	-	0.4899
randomly_perfect_random_0984	-	-	-	-	0.4775
randomly_perfect_random_0985	-	-	-	-	0.4235
randomly_perfect_random_0986	-	-	-	-	0.3703
randomly_perfect_random_0987	-	-	-	-	-
randomly_perfect_random_0988	-	0.0000	0.0000	0.0000	0.0339
randomly_perfect_random_0989	-	-	-	-	0.5209
randomly_perfect_random_0990	-	-	-	-	0.4483

Continued on next page

	CP (Gecode)	CP (Gurobi)	MIP (Gurobi)	MIP bound	MIP without dominance
randomly_perfect_random_0991	-	-	-	-	0.4774
randomly_perfect_random_0992	-	0.0000	0.0000	0.0000	0.0000
randomly_perfect_random_0993	-	-	-	-	0.5171
randomly_perfect_random_0994	-	-	-	-	0.4556
randomly_perfect_random_0995	-	-	-	-	0.5029
randomly_perfect_random_0996	-	-	0.0458	0.0000	0.1448
randomly_perfect_random_0997	-	-	-	-	0.4787
randomly_perfect_random_0998	-	-	-	-	0.4705
randomly_perfect_random_0999	-	-	-	-	0.3756
randomly_perfect_random_1000	-	-	-	-	0.4964
realistic_instance_01	-	1.0552	1.0463	0.9831	1.0496
realistic_instance_02	-	1.1923	1.1922	1.1446	1.1784
realistic_instance_03	-	1.0323	0.9992	0.9653	1.0023
realistic_instance_04	-	1.0386	1.0598	0.9832	1.0317
realistic_instance_05	-	1.1658	1.1614	1.1393	1.1753
realistic_instance_06	-	1.0879	1.0909	1.0628	1.0931
realistic_instance_07	-	-	1.2802	1.1311	1.2514
realistic_instance_08	-	1.1349	1.1208	1.0861	1.1259
realistic_instance_09	-	1.1129	1.1225	1.0855	1.1079
realistic_instance_10	-	1.1317	1.1210	1.1004	1.1430
realistic_instance_11	-	1.0605	1.0613	1.0240	1.0520
realistic_instance_12	-	1.1139	1.1192	1.0941	1.1259
realistic_instance_13	-	1.0086	0.9985	0.9627	1.0133
realistic_instance_14	-	-	-	-	-
realistic_instance_15	-	0.9442	0.9446	0.9198	0.9671
realistic_instance_16	-	1.1386	1.1399	1.1073	1.1362
realistic_instance_17	1.2479	1.0085	0.9972	0.9688	0.9950
realistic_instance_18	-	1.0663	1.0469	1.0365	1.0550
realistic_instance_19	-	-	0.5877	0.5404	0.5763
realistic_instance_20	-	-	0.6296	0.5976	0.6313
realistic_instance_21	-	0.7336	0.6851	0.6686	0.6863
realistic_instance_22	-	-	-	-	1.0735
realistic_instance_23	-	-	-	-	1.0513
realistic_instance_24	-	-	-	-	-
realistic_instance_25	-	-	-	-	-
realistic_instance_26	-	-	-	-	0.7337
realistic_instance_27	-	-	-	-	0.5667

APPENDIX B

Computational results of metaheuristic local search

Table B.1: Objective value and number of hard constraint violations for Greedy, VND, and Simulated Annealing for each instance of the test set.

	Objective					**Constraint Violations**		
	Greedy	VND	SA (median / mean / std)			Greedy	VND	SA (median)
randomly_generated_0951	0.4723	0.3780	0.3754	0.3755	0.0006	1	1	0
randomly_generated_0952	1.0752	0.9216	0.9065	0.9068	0.0010	1	1	1
randomly_generated_0953	0.3029	0.2255	0.2212	0.2209	0.0005	1	0	0
randomly_generated_0954	0.2059	0.1268	0.1270	0.1269	0.0006	2	0	0
randomly_generated_0955	0.1095	0.0412	0.0322	0.0325	0.0009	1	1	0
randomly_generated_0956	0.0905	0.0077	0.0215	0.0216	0.0007	0	0	0
randomly_generated_0957	0.6956	0.5784	0.5609	0.5612	0.0010	4	1	0
randomly_generated_0958	0.0943	0.0566	0.0356	0.0358	0.0010	1	0	0
randomly_generated_0959	0.1347	0.0067	0.0082	0.0081	0.0003	0	0	0
randomly_generated_0960	0.7608	0.6704	0.6657	0.6659	0.0004	9	0	0
randomly_generated_0961	0.4053	0.3576	0.3471	0.3477	0.0028	9	4	3
randomly_generated_0962	0.0178	0.0034	0.0227	0.0227	0.0000	0	0	0
randomly_generated_0963	1.0480	0.9419	0.9245	0.9245	0.0008	0	0	0
randomly_generated_0964	0.0763	0.0126	0.0821	0.0821	0.0000	0	0	0
randomly_generated_0965	0.0383	0.0154	0.0090	0.0090	0.0004	0	0	0
randomly_generated_0966	0.2501	0.1610	0.1605	0.1605	0.0010	0	0	0
randomly_generated_0967	0.2984	0.1794	0.1714	0.1712	0.0006	6	3	3
randomly_generated_0968	0.3573	0.2646	0.3004	0.3011	0.0021	0	0	0
randomly_generated_0969	0.3415	0.2370	0.2421	0.2419	0.0010	2	1	1
randomly_generated_0970	0.1261	0.0943	0.0895	0.0895	0.0012	0	0	0
randomly_generated_0971	0.3999	0.3881	0.4633	0.4632	0.0026	9	2	1
randomly_generated_0972	0.1641	0.0787	0.0817	0.0817	0.0003	1	1	0
randomly_generated_0973	0.1416	0.0829	0.0795	0.0796	0.0006	2	0	0
randomly_generated_0974	0.9347	0.8627	0.8381	0.8376	0.0018	0	0	0
randomly_generated_0975	0.0150	0.0133	0.0328	0.0328	0.0017	0	0	0
randomly_generated_0976	0.2035	0.1424	0.1340	0.1337	0.0009	5	1	0
randomly_generated_0977	0.3216	0.2385	0.2308	0.2306	0.0008	5	3	0
randomly_generated_0978	0.5532	0.4485	0.4460	0.4459	0.0007	1	1	0
randomly_generated_0979	0.6387	0.5409	0.5192	0.5193	0.0010	10	4	1
randomly_generated_0980	0.1583	0.0846	0.0827	0.0825	0.0012	1	1	1
randomly_generated_0981	0.4051	0.3148	0.3648	0.3655	0.0017	1	1	1
randomly_generated_0982	0.0504	0.0148	0.0063	0.0063	0.0009	0	0	0
randomly_generated_0983	0.4268	0.3222	0.3166	0.3162	0.0016	7	3	2

Continued on next page

	Objective					**Constraint Violations**		
	Greedy	VND	SA (median / mean / std)			Greedy	VND	SA (median)
randomly_generated_0984	0.1916	0.1435	0.1217	0.1220	0.0014	1	0	0
randomly_generated_0985	0.0426	0.0087	0.0104	0.0107	0.0008	0	0	0
randomly_generated_0986	0.4988	0.4230	0.4106	0.4106	0.0013	9	5	4
randomly_generated_0987	0.0794	0.0096	0.0803	0.0804	0.0016	2	0	0
randomly_generated_0988	0.1268	0.0365	0.0445	0.0446	0.0006	1	0	0
randomly_generated_0989	0.9005	0.7609	0.7536	0.7538	0.0011	0	0	0
randomly_generated_0990	0.0146	0.0094	0.0027	0.0028	0.0003	0	0	0
randomly_generated_0991	0.4081	0.3189	0.3117	0.3115	0.0016	1	0	0
randomly_generated_0992	0.2769	0.1438	0.2180	0.2172	0.0024	2	1	1
randomly_generated_0993	0.8421	0.7384	0.7111	0.7110	0.0010	7	3	3
randomly_generated_0994	0.3612	0.2701	0.2718	0.2719	0.0006	0	0	0
randomly_generated_0995	0.2057	0.1190	0.1140	0.1142	0.0007	2	1	1
randomly_generated_0996	0.8163	0.6418	0.6505	0.6509	0.0013	2	1	1
randomly_generated_0997	1.0120	0.8782	0.8601	0.8600	0.0009	11	4	2
randomly_generated_0998	0.2151	0.0949	0.0917	0.0916	0.0009	1	1	1
randomly_generated_0999	0.2127	0.1018	0.0938	0.0939	0.0007	2	1	1
randomly_generated_1000	0.0750	0.0144	0.0153	0.0155	0.0011	0	0	0
randomly_generated_small_0001	0.1288	0.0566	0.0303	0.0294	0.0028	0	0	0
randomly_generated_small_0002	0.5228	0.4957	0.4499	0.4499	0.0019	1	0	0
randomly_generated_small_0003	0.2956	0.2988	0.2715	0.2715	0.0021	1	0	0
randomly_generated_small_0004	0.3838	0.3662	0.3541	0.3542	0.0021	1	0	0
randomly_generated_small_0005	0.6023	0.7912	0.7900	0.7903	0.0009	1	1	0
randomly_generated_small_0006	0.7852	0.6995	0.6860	0.6864	0.0010	1	0	0
randomly_generated_small_0007	0.0338	0.0200	0.0081	0.0082	0.0011	0	0	0
randomly_generated_small_0008	0.2849	0.2248	0.2166	0.2152	0.0052	1	0	0
randomly_generated_small_0009	0.8680	0.8075	0.7865	0.7860	0.0017	1	0	0
randomly_generated_small_0010	0.7400	0.7280	0.7267	0.7266	0.0055	1	1	0
randomly_perfect_random_0951	0.0877	0.0227	0.0467	0.0464	0.0026	0	0	0
randomly_perfect_random_0952	0.0218	0.0104	0.0391	0.0397	0.0020	1	0	0
randomly_perfect_random_0953	0.0983	0.0119	0.0227	0.0229	0.0042	0	0	0
randomly_perfect_random_0954	0.0169	0.0076	0.0278	0.0278	0.0000	0	0	0
randomly_perfect_random_0955	0.0023	0.0020	0.0050	0.0050	0.0000	0	0	0
randomly_perfect_random_0956	0.0569	0.0145	0.0178	0.0179	0.0021	0	0	0
randomly_perfect_random_0957	0.0656	0.0153	0.0406	0.0411	0.0013	0	0	0
randomly_perfect_random_0958	0.0141	0.0103	0.0265	0.0265	0.0000	0	0	0
randomly_perfect_random_0959	0.0485	0.0155	0.0490	0.0488	0.0035	2	0	1
randomly_perfect_random_0960	0.0091	0.0105	0.0212	0.0212	0.0000	0	0	0
randomly_perfect_random_0961	0.0729	0.0166	0.0367	0.0375	0.0023	1	0	0
randomly_perfect_random_0962	0.0564	0.0182	0.0454	0.0453	0.0017	1	0	0
randomly_perfect_random_0963	0.0465	0.0158	0.0593	0.0593	0.0000	0	0	0
randomly_perfect_random_0964	0.0836	0.0120	0.0194	0.0283	0.0232	0	0	0
randomly_perfect_random_0965	0.0557	0.0151	0.0450	0.0456	0.0020	0	0	0
randomly_perfect_random_0966	0.0589	0.0129	0.0611	0.0627	0.0162	2	1	1
randomly_perfect_random_0967	0.0014	0.0042	0.0036	0.0029	0.0016	0	0	0
randomly_perfect_random_0968	0.0750	0.0107	0.0283	0.0288	0.0017	0	0	0
randomly_perfect_random_0969	0.0035	0.0104	0.0104	0.0104	0.0000	0	0	0
randomly_perfect_random_0970	0.0435	0.0140	0.0347	0.0348	0.0027	0	0	0
randomly_perfect_random_0971	0.0798	0.0063	0.0667	0.0635	0.0443	0	0	0
randomly_perfect_random_0972	0.0862	0.0125	0.0639	0.0636	0.0325	0	0	0
randomly_perfect_random_0973	0.0592	0.0185	0.0408	0.0405	0.0019	0	0	0
randomly_perfect_random_0974	0.0269	0.0120	0.0356	0.0354	0.0003	0	0	0
randomly_perfect_random_0975	0.0634	0.0162	0.0402	0.0437	0.0117	1	0	0
randomly_perfect_random_0976	0.0344	0.0025	0.0422	0.0422	0.0000	0	0	0
randomly_perfect_random_0977	0.0545	0.0076	0.0602	0.0602	0.0000	0	0	0
randomly_perfect_random_0978	0.0675	0.0073	0.0269	0.0267	0.0012	0	0	0
randomly_perfect_random_0979	0.0486	0.0032	0.0159	0.0160	0.0006	0	0	0
randomly_perfect_random_0980	0.0391	0.0258	0.0475	0.0472	0.0094	0	0	0
randomly_perfect_random_0981	0.0356	0.0131	0.0301	0.0300	0.0015	0	0	0
randomly_perfect_random_0982	0.0730	0.0111	0.0344	0.0520	0.0292	0	0	0
randomly_perfect_random_0983	0.0602	0.0138	0.0346	0.0341	0.0017	0	0	0
randomly_perfect_random_0984	0.0654	0.0170	0.0542	0.0552	0.0094	1	0	1
randomly_perfect_random_0985	0.0493	0.0203	0.0518	0.0520	0.0015	1	0	0
randomly_perfect_random_0986	0.0339	0.0128	0.0386	0.0388	0.0012	0	0	0
randomly_perfect_random_0987	0.0135	0.0084	0.0325	0.0328	0.0016	1	1	0
randomly_perfect_random_0988	0.0049	0.0147	0.0147	0.0137	0.0018	0	0	0
randomly_perfect_random_0989	0.0689	0.0086	0.0753	0.0753	0.0000	0	0	0
randomly_perfect_random_0990	0.0553	0.0161	0.0420	0.0425	0.0022	1	0	0
randomly_perfect_random_0991	0.0541	0.0152	0.0301	0.0303	0.0019	0	0	0

Continued on next page

	Objective					**Constraint Violations**		
	Greedy	VND	SA (median / mean / std)			Greedy	VND	SA (median)
randomly_perfect_random_0992	0.0051	0.0108	0.0148	0.0112	0.0061	0	0	0
randomly_perfect_random_0993	0.0431	0.0042	0.0094	0.0097	0.0008	0	0	0
randomly_perfect_random_0994	0.0419	0.0128	0.0428	0.0427	0.0016	0	0	0
randomly_perfect_random_0995	0.0724	0.0142	0.0366	0.0451	0.0187	0	0	0
randomly_perfect_random_0996	0.0035	0.0031	0.0076	0.0076	0.0000	0	0	0
randomly_perfect_random_0997	0.0532	0.0148	0.0540	0.0545	0.0024	0	0	0
randomly_perfect_random_0998	0.0846	0.0160	0.0590	0.0521	0.0120	2	0	1
randomly_perfect_random_0999	0.0410	0.0150	0.0356	0.0360	0.0016	0	0	0
randomly_perfect_random_1000	0.0551	0.0171	0.0364	0.0355	0.0023	0	0	0
realistic_instance_01	1.2277	1.0418	1.0203	1.0204	0.0038	0	0	0
realistic_instance_02	1.2684	1.1867	1.1734	1.1738	0.0015	0	0	0
realistic_instance_03	1.1241	1.0140	0.9894	0.9894	0.0029	0	0	0
realistic_instance_04	1.1657	1.0268	1.0128	1.0126	0.0016	0	0	0
realistic_instance_05	1.3426	1.2144	1.1575	1.1571	0.0020	3	1	0
realistic_instance_06	1.3166	1.1311	1.0823	1.0821	0.0008	0	0	0
realistic_instance_07	1.3468	1.2154	1.1723	1.1728	0.0027	3	1	0
realistic_instance_08	1.3197	1.1313	1.1153	1.1150	0.0023	0	0	0
realistic_instance_09	1.2925	1.1287	1.1087	1.1088	0.0026	0	0	0
realistic_instance_10	1.2907	1.1325	1.1240	1.1236	0.0025	2	0	0
realistic_instance_11	1.2241	1.0926	1.0380	1.0383	0.0023	3	1	0
realistic_instance_12	1.2952	1.1245	1.1064	1.1065	0.0012	1	1	0
realistic_instance_13	1.1180	1.0071	0.9991	0.9996	0.0022	0	0	0
realistic_instance_14	1.2777	1.1615	1.1488	1.1482	0.0034	2	1	1
realistic_instance_15	1.0607	0.9594	0.9496	0.9504	0.0025	1	0	0
realistic_instance_16	1.2509	1.1652	1.1349	1.1347	0.0024	0	0	0
realistic_instance_17	1.2413	1.0145	0.9871	0.9867	0.0017	0	0	0
realistic_instance_18	1.2303	1.0726	1.0466	1.0466	0.0011	0	0	0
realistic_instance_19	0.6811	0.5657	0.5638	0.5650	0.0050	1	0	0
realistic_instance_20	0.7998	0.6221	0.6187	0.6191	0.0043	0	0	0
realistic_instance_21	0.8092	0.7022	0.6873	0.6891	0.0043	1	0	0
realistic_instance_22	0.5624	0.4453	0.4379	0.4380	0.0012	0	0	0
realistic_instance_23	0.6113	0.4684	0.4676	0.4673	0.0013	0	0	0
realistic_instance_24	0.6569	0.5045	0.4944	0.4948	0.0011	0	0	0
realistic_instance_25	0.4971	0.3701	0.3565	0.3566	0.0012	0	0	0
realistic_instance_26	0.8475	0.6311	0.6187	0.6192	0.0015	2	0	0
realistic_instance_27	0.8199	0.5276	0.5169	0.5168	0.0012	2	0	0

List of Figures

2.1 Example solutions visualizing the three optimization goals 7

3.1 Optimal solution of the Fixed-Order PLP example instance 20

4.1 Example of a move-order move: solutions before (left) and after (right) . 29
4.2 Example of a swap-orders move: solutions before (left) and after (right) . 29
4.3 Two cooling schedules with different cooling rates 34

5.1 3D Scatter plot of the parameters of the randomly perfect instance set R_2 37
5.2 Comparison of exact solution approaches 42
5.3 Share of solution statuses of MIP for each subset 43
5.4 Solution statuses of MIP grouped by instance features 44
5.5 Results of the experiment with different cooling schedules 48
5.6 Comparison of metaheuristic techniques on the test set 49
5.7 Simulated Annealing results with different neighborhood weightings . . . 50
5.8 Comparison of the optimality gap of metaheuristic solutions 52

List of Tables

3.1 Dynamic Programming matrices $O(j, l)$ and $M(j, l)$ 20

5.1 Overview of the different instance sets and the split into training and test set 39
5.2 Overview over the instance features for each instance set 39
5.3 Optimality gap of MIP for suboptimal instances in R_1 and R_4 44
5.4 Results for the native MIP model with/without dominance constraints . . 45
5.5 Configuration space of Simulated Annealing 46
5.6 Five equivalent cooling schedules 47
5.7 Results for metaheuristic methods on R_2 53

A.1 Computational results for exact methods 57

B.1 Computational results for metaheuristic methods 61

List of Algorithms

4.1 Variable Neighborhood Descent 32
4.2 Simulated Annealing for the PLP 33

5.1 Integer Partitioning 36
5.2 Generate perfectly solvable instance 37

Acronyms

BACP Balanced Academic Curriculum Problem. 2, 11, 24

CP Constraint Programming. 2, 3, 12, 21–26, 40, 41, 55

MIP Mixed Integer Programming. 2, 3, 10, 21, 22, 25, 40, 41, 43–45, 51, 52, 55–57, 65

PLP Production Leveling Problem. 1–3, 5, 9–16, 21–24, 28, 31, 35, 41, 45, 48, 53, 55, 69

SALB Simple Assembly Line Balancing. 12

SMAC Sequential Model-based Algorithm Configuration. 46, 49, 50

TPS Toyota Production System. 10, 11

VND Variable Neighborhood Descent. 2, 31, 40, 45, 48, 49, 51–53, 55, 56, 61

Bibliography

[Aİ18] Meral Azizoğlu and Sadullah İmat. „Workload smoothing in simple assembly line balancing“. In: *Computers & Operations Research* 89 (2018), pp. 51–57. ISSN: 0305-0548. DOI: 10.1016/j.cor.2017.08.006. (Visited on 2019-09-26).

[BFS07] Nils Boysen, Malte Fliedner, and Armin Scholl. „A classification of assembly line balancing problems“. In: *European Journal of Operational Research* 183.2 (2007), pp. 674–693. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2006.10.010.

[BFS09] Nils Boysen, Malte Fliedner, and Armin Scholl. „The product rate variation problem and its relevance in real world mixed-model assembly lines“. In: *European Journal of Operational Research* 197.2 (2009), pp. 818–824. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2008.06.038. (Visited on 2019-07-12).

[Chi+12] Marco Chiarandini, Luca Di Gaspero, Stefano Gualandi, and Andrea Schaerf. „The balanced academic curriculum problem revisited“. en. In: *Journal of Heuristics* 18.1 (2012), pp. 119–148. ISSN: 1572-9397. DOI: 10.1007/s10732-011-9158-2. (Visited on 2019-06-27).

[CS15] Geoffrey Chu and Peter J. Stuckey. „Dominance breaking constraints“. en. In: *Constraints* 20.2 (2015), pp. 155–182. ISSN: 1572-9354. DOI: 10.1007/s10601-014-9173-7. (Visited on 2019-09-22).

[DS08] Luca Di Gaspero and Andrea Schaerf. „Hybrid Local Search Techniques for the Generalized Balanced Academic Curriculum Problem“. In: *Hybrid Metaheuristics, 5th International Workshop, HM 2008, Málaga, Spain, October 8-9, 2008. Proceedings.* Ed. by Maria J. Blesa, Christian Blum, Carlos Cotta, et al. Vol. 5296. Lecture Notes in Computer Science. Springer, 2008, pp. 146–157. ISBN: 978-3-540-88438-5. DOI: 10.1007/978-3-540-88439-2_11.

[Gur19] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2019. URL: http://www.gurobi.com.

[Han+10] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. „Variable Neighborhood Search“. en. In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2010, pp. 61–86. ISBN: 978-1-4419-1665-5. URL: https://doi.org/10.1007/978-1-4419-1665-5_3 (visited on 2019-07-03).

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. „Optimization by Simulated Annealing“. en. In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.220.4598.671. (Visited on 2019-07-04).

[Kub93] Wieslaw Kubiak. „Minimizing variation of production rates in just-in-time systems: A survey“. In: *European Journal of Operational Research* 66.3 (1993), pp. 259–271. ISSN: 0377-2217. DOI: 10.1016/0377-2217(93)90215-9. (Visited on 2019-07-26).

[Lin+19] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, et al. *SMAC v3: Algorithm Configuration in Python.* 2019. URL: https://github.com/automl/SMAC3 (visited on 2019-10-01).

[LVM19] Marie-Louise Lackner, Johannes Vass, and Nysret Musliu. *Extended Complexity Results for the Production Leveling Problem.* en. Technical Report CD-TR 2019/2. Vienna: TU Wien, 2019, p. 8. URL: https://dbai.tuwien.ac.at/staff/jvass/publications/cd-tr-2019-2.pdf (visited on 2019-10-01).

[ML02] C. Mullinax and M. Lawley. „Assigning patients to nurses in neonatal intensive care“. en. In: *Journal of the Operational Research Society* 53.1 (2002), pp. 25–35. ISSN: 1476-9360. DOI: 10.1057/palgrave.jors.2601265.

[Mon+07] Jean-Noël Monette, Pierre Schaus, Stéphane Zampelli, Yves Deville, Pierre Dupont, et al. „A CP Approach to the Balanced Academic Curriculum Problem“. In: *Seventh International Workshop on Symmetry and Constraint Satisfaction Problems.* Vol. 7. Citeseer, 2007. URL: https://www.info.ucl.ac.be/~pschaus/assets/publi/symcon2007_bacp.pdf (visited on 2019-10-01).

[Net+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, et al. „MiniZinc: Towards a Standard CP Modelling Language“. In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings.* Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 529–543. ISBN: 978-3-540-74969-1. DOI: 10.1007/978-3-540-74970-7_38.

[OR98] Taiichi Ohno and C. B. Rosen. *Toyota production system: beyond large-scale production.* eng. Portland, OR: Productivity Press, 1998. ISBN: 978-0-915299-14-0.

[PRB13] Prattana Punnakitikashem, Jay M. Rosenberber, and Deborah F. Buckley-Behan. „A stochastic programming approach for integrated nurse staffing and assignment“. In: *IIE Transactions* 45.10 (2013), pp. 1059–1076. ISSN: 0740-817X. DOI: `10.1080/0740817X.2012.763002`. (Visited on 2019-09-26).

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming.* en. Google-Books-ID: Kjap9ZWcKOoC. Elsevier, 2006. ISBN: 978-0-08-046380-3.

[Sch+07] Pierre Schaus, Yves Deville, Pierre Dupont, and Jean-Charles Régin. „The Deviation Constraint“. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings.* Ed. by Pascal Van Hentenryck and Laurence A. Wolsey. Vol. 4510. Lecture Notes in Computer Science. Springer, 2007, pp. 260–274. ISBN: 978-3-540-72396-7. DOI: `10.1007/978-3-540-72397-4_19`.

[Sch09] Pierre Schaus. „Solving balancing and bin-packing problems with constraint programming“. en. PhD thesis. UCL - Université Catholique de Louvain, 2009. URL: `https://dial.uclouvain.be/pr/boreal/en/object/boreal%3A23871` (visited on 2019-09-09).

[Sha04] Paul Shaw. „A Constraint for Bin Packing“. In: *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings.* Ed. by Mark Wallace. Vol. 3258. Lecture Notes in Computer Science. Springer, 2004, pp. 648–662. ISBN: 978-3-540-23241-4. DOI: `10.1007/978-3-540-30201-8_47`.

[SHR09] Pierre Schaus, Pascal Van Hentenryck, and Jean-Charles Régin. „Scalable Load Balancing in Nurse to Patient Assignment Problems“. In: *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.* Ed. by Willem Jan van Hoeve and John N. Hooker. Vol. 5547. Lecture Notes in Computer Science. Springer, 2009, pp. 248–262. ISBN: 978-3-642-01928-9. DOI: `10.1007/978-3-642-01929-6_19`.

[Ski98] Steven S. Skiena. *The Algorithm Design Manual.* en. Springer Science & Business Media, 1998. ISBN: 978-0-387-94860-7. URL: `https://books.google.at/books?id=TrXd-gxPhVYC` (visited on 2019-10-01).

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms*. en. Springer-Verlag Berlin Heidelberg, 2003. ISBN: 978-3-540-65367-7. URL: `https://books.google.at/books?id=bJmqCAAAQBAJ` (visited on 2019-03-11).

[War76] D. Michael Warner. „Scheduling Nursing Personnel According to Nursing Preference: A Mathematical Programming Approach". In: *Operations Research* 24.5 (1976), pp. 842–856. ISSN: 0030-364X. DOI: `10.1287/opre.24.5.842`. (Visited on 2019-09-26).

[WN14] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. de. John Wiley & Sons, 2014. ISBN: 978-1-118-62686-3. URL: `https://onlinelibrary.wiley.com/doi/book/10.1002/9781118627372` (visited on 2019-10-01).