UNIVERSITEIT TWENTE.

ORTEC

Master's Thesis

Tuning the Parameters of a Loading Algorithm

Author:
Tim Gerard van Dijk

Supervisors:
Dr. ir. M.R.K. Mes
Dr. ir. J.M.J. Schutten
Dr. J.A. dos Santos Gromicho

February 28, 2014

Author
T.G. (Tim) van Dijk
University
University of Twente
Master programme
Industrial Engineering & Management
Specialization
Production Logistics & Management
Graduation Date
21 March 2014

Graduation Committee
Dr. ir. M.R.K. Mes
University of Twente

Dr. ir. J.M.J. Schutten
University of Twente

Dr. J.A. dos Santos Gromicho
ORTEC BV

UNIVERSITEIT TWENTE.

University of Twente
Drienerlolaan 5
7522 NB Enschede
The Netherlands

ORTEC

ORTEC
Houtsingel 5
2719 EA Zoetermeer
The Netherlands

Management Summary

The newly developed software solution ORTEC Pallet and Load Building (OPLB) is the intended successor of LoadDesigner. It contains a heavily parameterized algorithmic framework of which the performance on customer-specific problem instances depends on the quality of its parameter configuration. These parameters form the subject of this report. The objective of this research is as follows.

> **Develop a tuning algorithm that quickly finds good parameter settings for the existing algorithmic framework such that the performance of OPLB is optimized.**

OPLB consists of a deterministic, greedy construction phase and a stochastic Simulated Annealing (SA)-driven local search procedure. Because of the nature of the SA procedure (each neighborhood switch contains a reconstruction phase with the same greedy construction), the quality of a single-construction solution translates well to the quality of the final solution. Therefore, we restrict any tuning effort to the greedy construction phase, as this allows for faster, noise-free measurements, hence benefits the learning process of our tuning algorithm.

The running time at the customer site must be kept to a minimum. Moreover, our parameter space analysis reveals a high robustness of parameter configurations among similar problem instances. Therefore, we choose for an offline, problem family tuning method. That is, we tune the parameters on a training set of instances that are representative for the customer's typical problem instance, and the resulting parameter configuration can be readily applied to newly occurring instances without any further tuning.

Method

The tuning algorithm we develop, uRace, is a unified Race algorithm. It performs batch measurements of multiple configurations on the same training instances and focuses on rapid elimination of candidate configurations based on statistical inferiority. Moreover, it enforces an increasingly localized search around the most promising areas in the solution space, and it allows for a tunable tradeoff between exploration of new configurations and exploitation of known configurations to obtain more reliable estimates on their performance that naturally tends towards exploitation as the tuning process proceeds. Finally, by determining the tuning budget (i.e., total time spent on tuning) and the number of parameter configurations that are tuned (and subsequently applied to the actual problem at the customer site), the user is free to make its own offline and online running time-performance tradeoff.

Results

The parameter configurations tuned with uRace outperform alternative configurations that represent known constructive heuristics in the literature, both on academic problem instances, in which the objective is to maximize volume utility, and on a set of practical problem instances from a large multinational,

	Academic Problems	Practical Problems	
Parameter Configuration	Volume Utility (%)	Containers	Pallets
Wall Building (George and Robinson, 1980)	80.59	2.0	47.2
Horizontal Layer (Bischoff et al., 1995)	82.70	2.0	47.2
Configuration Tuned with uRace	83.99	1.7	44.2

Table 1: Performance of a single iteration (i.e., one greedy construction) of OPLB with three alternative parameter configurations. The alternative configurations represent constructive heuristics from the literature that are known to perform well. The academic problems are 700 weakly heterogeneous BR instances (see Section 2.1.3). The practical problems are ten large, complex instances from a well-known multinational. The running times of the three configurations are roughly the same.

Author	Volume Utility	Running Time (sec.)
Juraitis et al. (2006)	89.26	-
Bortfeldt and Gehring (2001)	90.06	316
Zhang et al. (2012)	95.35	135
OPLB Single - 1 Configuration	83.99	0.06
OPLB Single - 5 Configurations	85.99	0.24
OPLB with SA - 5 Configurations	90.24	74

Table 2: Performance of state-of-the-art algorithms in the literature and OPLB with the greedy parameters as tuned with uRace on weakly heterogeneous BR instances.

in which the objective is to minimize the usage of loading equipment (see Table 1).

Moreover, by application of multiple parameter configurations on the online problem and inclusion of the SA procedure, OPLB outperforms some of the state-of-the-art, highly specialized loading algorithms on academic problems with competitive running time (see Table 2).

Recommendations

The practical relevance of this research emerges as customers start to employ OPLB to facilitate their loading processes. During the implementation phase at a new customer, uRace may be used to find a good parameter configuration either based on a set of historical instances or on a customer classification, thereby customizing ORTEC's software solution to the specific customer. Moreover, uRace may be used to periodically, whenever computational resources are amply available, update the parameter configuration based on recent developments at the customer site, resulting in an automatically adjusting loading algorithm to provide an efficient, customized loading solution for ORTEC's customers.

Finally, uRace is a generic tuning algorithm, which means its potential use goes beyond OPLB. Other ORTEC software solutions, provided they are similarly parameterized as OPLB, can also be efficiently configured with uRace.

Preface

I am proud to present you my MSc. thesis, which marks the end of my studentship at the University of Twente, my internship at ORTEC, *and, at least for a while, my membership to endless* Delphi, `C++`, VBA, *and* LaTeX *fora. This report contains the result of a six-month battle against a pile of boxes that refuse to be optimally placed in a container. Surely, global optima are still an illusion, and the war is far from over, but this tiny little battle was won.*

This victory would have never been achieved without the support of many people battling shoulder to shoulder with me. Martijn and Marco, thank you sincerely for all your substantive feedback and suggestions, for all the time you spent proof-reading my drafts, and for all the ink you spent correcting every grammatical error, poor sentence structure, and misplaced comma. It is very motivating to know that someone takes the time to read your work with such care. Joaquim, thank you for giving me the opportunity to undertake this engaging project at ORTEC, *and for introducing me to a wonderfully complex world of development platforms, data structures, and software systems. I owe great gratitude to my colleagues at* ORTEC *for helping me out on the many occasions that this world quickly went from wonderfully complex to excessively aggravating, and to my friends for providing the necessary distraction on the few occasions that a bit of this aggravation had managed to follow me home.*

Finally, I am deeply grateful for the unconditional interest, support, and love from my family and girlfriend. Indeed, although this thesis marks the end of many -ships, as long as the friendships and relationships remain, I am sure it is actually merely the beginning.

Tim

List of Abbreviations

OPLB	ORTEC Pallet and Load Building	Name of ORTEC's newly developed software solution for loading problems, the successor of LoadDesigner
BR	Bischoff and Ratcliff	BR instances are regularly used benchmark loading problems
C&P	Cutting & Packing	Loading problems fall under the umbrella of C&P problems
CLP	Container Loading Problem	Problem of loading items into a container
PLP	Pallet Loading Problem	Problem of loading items onto a pallet
GA	Genetic Algorithm	Optimization technique based on population evolution
SA	Simulated Annealing	Optimization technique based on local optimization
KPI	Key Performance Indicator	In loading problems, the KPI is typically volume utility or container usage
R&S	Ranking and Selection	Type of learning problem, in which the objective is to find the alternative with the highest value from a finite, and typically small, set of alternatives
UCB	Upper Confidence Bound	UCB policy is a simple sequential measurement technique in which the alternative with the highest UCB is selected for measurement
BSD	Biased Sampling Distributions	Sequential measurement technique that randomly samples new parameters from sampling distributions that are biased towards the values taken by known, well-performing configurations
LHD	Latin Hypercube Design	Sampling design that ensures uniform coverage of the parameter space
LHS	Latin Hypercube Sample	Random sample (of parameter configurations) that results from a LHD
OR	Operations Research	Discipline that deals with the application of mathematical methods to solve complex decision-making problems

Table of Contents

1 Introduction

This research study finds its origin at the software development department of ORTEC, one of the largest global providers of planning and optimization solutions. Generally speaking, ORTEC applies methods from Operations Research (OR) to help their customers optimize their decision making. In particular, they develop stand-alone or SAP-embedded software solutions covering a wide range of OR challenges, which are employed by customers worldwide. One of these software solutions is the subject of this research: ORTEC Pallet and Load Building (OPLB), an optimization tool that facilitates efficient pallet and container loading. Like all ORTEC's software solutions, their loading tool is subject to a continuous improvement effort in order to keep offering cutting edge technologies to their customers, and therefore remain a competitive player in the industry. OPLB is the projected successor of LoadDesigner, the tool that is presently employed by ORTEC's customers to facilitate their loading. The discrete transition from LoadDesigner to OPLB is mainly necessitated by the desire to have one generic optimizer, an algorithmic framework that can easily be customized towards any specific customer, without the need to alter the source code. Currently, there is approximately one hard-coded version of LoadDesigner for every customer, whereas OPLB is intended as a generic framework that can be adjusted to any customer's wishes by altering some external parameter configuration. This research study, which forms a graduation assignment as part of the Master program Industrial Engineering & Management at the University of Twente, aims to aid in the development of OPLB.

This chapter further introduces this research. In Section 1.1, we analyze the context in which our research resides. That is, we discuss the specific purposes that LoadDesigner currently fulfills for ORTEC's customers, and we further describe the intentions behind the transition to OPLB and the progress achieved so far in its development. This discussion leads to the problem identification in Section 1.2, in which we formally state the problem that is subject to this research. Afterwards, Section 1.3 bounds the scope of the research, and Section 1.4 formally describes our research goal. Finally, Section 1.5 outlines the succession of research questions that need to be answered to satisfy this goal.

1.1 Context Analysis

As can be deduced from the introduction, this research is intrinsically an improvement study, aiming to improve on the current situation. Indeed, ORTEC already has a fully functional loading tool in place, and any contributions we intend to make in the development of OPLB ultimately aim to outperform LoadDesigner. Therefore, we first elaborate on this incumbent loading tool in Section 1.1.1, by discussing the specific purposes that LoadDesigner currently fulfills for ORTEC's customers. Moreover, ORTEC has already made a substantial step in the development of the new tool OPLB. The supporting data structures are largely built and the groundwork to implement the algorithms is laid. Section 1.1.2 reveals the development so far and explains the technology behind OPLB.

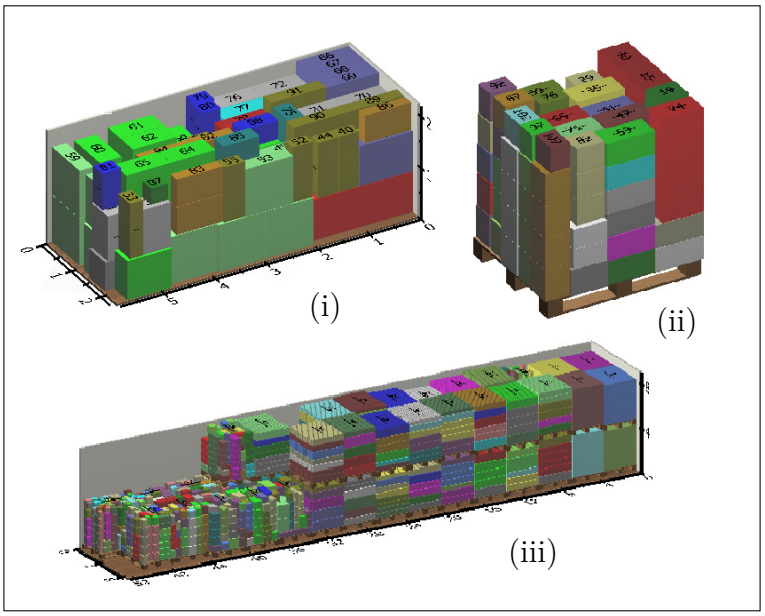

Figure 1.1: Example of a visualization of (i) a load of boxes in a container, (ii) a load of boxes on a pallet, and (iii) a load of (pick)pallets in a container

1.1.1 LoadDesigner for the Customer

The typical user of LoadDesigner is a commercial company that handles some logistic operation regarding pallet and load building. The list of companies that employ LoadDesigner is substantial, and includes well-known multinationals such as Coca Cola, Audi, and Procter & Gamble, as well as many small companies. Generally speaking, these companies use the LoadDesigner software to facilitate efficient loading while meeting their specific requirements. We briefly sketch two situations that exemplify some of the typical applications, objectives, and restrictions faced by ORTEC's customers. SCA, a manufacturer of personal care products, uses LoadDesigner for their inter-facility stock and transport management. They often transport containers filled with pick-pallets: pallets loaded with individual boxes or items. Optimized pallets and loads, which support the actual product demand downstream, are designed to ship against minimal transportation costs. Coca Cola uses LoadDesigner in conjunction with ORTEC Routing to simultaneously optimize their loading and routing operations. They use the software to determine which fleet composition and transportation modes to use in light of their palletizing, loading, and timing constraints. Typically, they are concerned with loading stock pallets into containers, which introduces stackability constraints. Moreover, the relatively high density of their products introduces restrictions regarding axle weight. In both scenarios, the planner employs LoadDesigner, and receives direct feedback in the form of Key Performance Indicators (KPIs) such as volume utility and weight (distribution), along with a three-dimensional visualization of the load. Figure 1.1 shows examples of such visualizations for three different loading applications.

Besides illustrating the scope of applications, these two examples mean to demonstrate the broad range of constraints and objectives the tool should incorporate. A larger customer, such as Coca Cola, typically has an in-house logistics department with sufficient knowledge to implement and apply the algorithms in LoadDesigner themselves. Many smaller companies, however, prefer an off-the-shelf ready-to-be-used tool that simply gives a good solution according to their optimization criteria and constraints. To meet this variety in requirements, there currently is approximately one hard-coded version of LoadDesigner for every customer. This leaves the source code increasingly long and complicated, and denies any user without expert logistic and programming knowledge the opportunity to adapt the

software according to daily variability in input and objectives. Moreover, the parallel existence of all these distinct versions complicated further development, and induces substantial maintenance costs. To overcome all these issues, OPLB intends to capture all, previously distinct, versions of LoadDesigner into one integrated framework that can easily be directed and customized with some external parameter configuration.

1.1.2 OPLB behind the Scenes

As said, OPLB is the new and improved version of LoadDesigner, so it should at least perform equally well from the customer point of view. Now that we have briefly discussed the range of customer-specific purposes that need to be fulfilled, and mentioned LoadDesigner's primary drawback, we continue to explore the technical details of the approach through which OPLB aims to outperform its predecessor. We confine ourselves to a technical description of the newly developed OPLB. The algorithm behind OPLB combines a greedy construction phase with an atypical local search procedure driven by Simulated Annealing (SA) (see Section 2.2.2). We first discuss the greedy construction and subsequently the local search procedure that surrounds it.

The greedy construction incrementally constructs a solution. In each iteration, it decides *which* item to place *where* in the container and *how*. For the which-part, it selects an item from the items that have not been loaded yet. For the where-part, it picks one of the available containers, a subspace from the available empty space in the container, and a position within that space. Finally, for the how-part, it chooses an orientation of the item and an arrangement of multiple of the same items to which the item pertains. These three decisions are captured into six consecutive sorting steps called filters, where every filter tackles one specific decision: one filter for *which*, two for *how*, and three for *where*. In this context, a filter is essentially a sorting mechanism that processes the possible choices and outputs a list of options ordered by preference, a preference list, according to the filter's parameter settings. This list, or rather the highest untried option on the list, is input for the second filter. This filter produces a preference list regarding its own decision *conditional on* the first filter's preferred choice. This process continues until the sixth filter outputs the six preferred options that collectively describe an item placement. If the placement is feasible, the item is loaded accordingly, the filters are updated with the new problem state, and a new iteration starts. If the preferred placement is infeasible (according to whichever restrictions the customer may impose), the algorithm starts sacrificing the filter's preferences bottom-up. That is, it will continue to try the preferred option of the first five filters with the second-best option of the sixth filter, and so on. Figure 1.2 illustrates this iterative process in a reduced example with three filters. The process repeats until either all remaining optional item placements are infeasible or all items are loaded. At this point we have constructed our initial, single-construction solution.

The parameter settings in each filter, along with the order of the filters, determine the construction process and therefore the quality of the resulting single-construction solution. Chapter 3 contains an exhaustive discussion of the filters and their parameters. Any effort that succeeds the greedy construction means to improve on this single-construction solution. Note that the greedy procedure is completely deterministic, which means that any repetition of the greedy construction is superfluous. However, an option that seems second-best in the myopic view of the greedy construction may ultimately lead to a better solution. To exploit this, a degree of randomness is introduced in the construction process and subsequently multiple solutions are constructed by, on occasion, deliberately selecting a less preferred option at random. This idea is enacted by an atypical SA procedure, where the neighborhood structure is to remove the last few placed items from the solution, and then reconstruct. In other words, from the final solution we go back a random number of iterations, randomly select a less-preferred option in that iteration, and reconstruct. Reconstruction occurs with the same parameter settings in the greedy part.

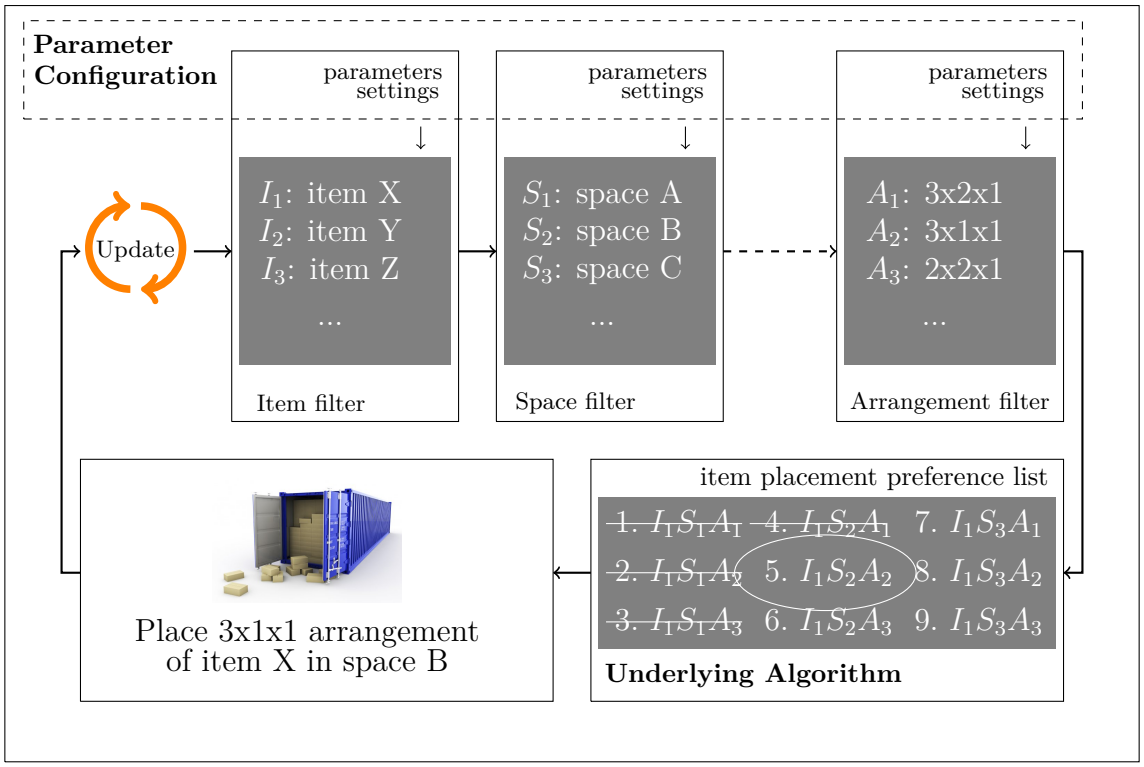

Figure 1.2: Visualization of the iterative, greedy construction process within OPLB. The parameter configuration determines the preference lists in each individual filter (e.g., item X is the most preferred item), and the order of the filters determines their priority (e.g., the second most preferred option is $I_1S_1A_2$, which sacrifices on the (last) arrangement filter). The underlying algorithm governs the feasibility of the item placements. In this example, the first four preferred item placements are infeasible, and the fourth option (i.e., $I_1S_2A_2$) is carried out. Afterwards, the current problem state is updated, and the process repeats.

The new solution is accepted according to the change in objective value, with a decreasing acceptance ratio. The entire process is repeated a number of times, and ultimately the best found solution is accepted as the best solution. It is important to realize that this is indeed an atypical SA-approach it the sense that it cannot be seen separate from the construction heuristic. Ordinarily, the explorative nature of SA at high temperatures would advocate against putting extensive effort into a sophisticated construction heuristic. However, in our SA procedure every neighborhood switch contains a reconstruction phase according to the same construction heuristic, which makes the construction heuristic, and therefore the parameter settings in the greedy part of OPLB, especially important.

Collectively, the greedy construction and the SA-driven random sampling procedure form a working algorithm, of which the greedy part in particular can be customized with much degree of freedom. By sheer tuning of its parameter configuration, the algorithmic framework of OPLB can be customized to a broad range of customer-specific loading problems.

1.2 Problem Identification

The technological description of the newly developed OPLB reveals the importance of the parameters in the greedy part of the algorithm. In fact, as we show in Chapter 3, the parameter settings in the greedy construction have a great influence on the quality of a single-construction solution. Moreover, as the integrated SA procedure carries out the same greedy construction in every neighborhood switch, they have a great influence on the quality of the final solution as well. This advocates the need for good parameter settings in the greedy part of OPLB, where the definition of "good" parameter settings is different per customer. Every customer has their own specific loading problem instances, and the same algorithm, described by the same parameter settings, likely performs differently on different instances. This poses the question in which context we want to seek optimal parameter settings. We distinguish two possibilities.

- Problem instance tuning. First, we can opt for problem instance tuning. In this case we tune the parameters each time a customer has to load a container. The parameter settings are chosen such that they optimize the solution to the specific problem instance it faces.
- Problem family tuning. Second, we can opt for problem family tuning. In this case we tune the parameters for the loading problem the customer *typically* faces. The significance of the word typical is substantial as it means that the algorithm's configuration needs to generalize to unseen problem instances with minimal loss of quality, so that the algorithm performs well on the complete, possibly infinite, set of similar unseen problem instances. We base our optimization on a set of problem instances, a training set, that is representative for the customer's typical problem. Generally, the higher the similarity between the training set and the actual problem, the better the resulting parameter settings. We are effectively simulating reality (the unseen problem instance) with representative situations (other instances in the same problem family) and optimize our parameter settings based on the simulations.

In essence, the first option an extreme of the second option in which the training set coincides with the actual problem instance. Therefore, all else being equal, the parameter settings found through problem instance tuning are likely to outperform those found through problem family tuning. However, for problem instance tuning, the computationally expensive tuning process occurs every time a customer wants to load a container, which imposes severe limitations on the running time of the tuning process. We return to this discussion in Section 1.3.

Whichever of these two options we wish to pursue, acquiring well-performing parameter settings is not trivial. Indeed, complete enumeration of all possible parameter settings is, as Chapter 3 reveals, computationally prohibitive because of our *enormous* search space. A relatively small problem instance already results in billions of possible parameter configurations. Moreover, note that if we pursue problem family tuning, measuring the performance of the set of parameter configurations on one problem instance is not sufficient. In fact, to draw any sort of statistically significant conclusions from the observations, we need to apply the same configurations to an increasing number of different problem instances, depending on the reliability and accuracy we require. Given that testing one configuration on one very small problem instance takes roughly three seconds on an average computer, we would need thousands of years of computation time to conduct a full factorial design on one problem instance and many more to come to satisfactory conclusions for problem family tuning. Clearly, we need to find a faster way to identify good parameter settings. This leads to the following problem statement:

> **The newly developed OPLB software tool requires a tuning algorithm that is able to quickly identify good parameter settings for the algorithmic framework**

This research intends to find a satisfactory solution to this problem.

1.3 Research Scope

In this section, we specify some boundaries for the tuning algorithm we seek to develop. As stated before, ORTEC has already developed an algorithmic framework for OPLB and we do not attempt to replace this in any way. We merely aim to find the best way to deploy this framework by tuning its parameters. Another limitation is formed by the desire to keep running time at the end-customer to a minimum. The choice between problem-instance and problem family tuning determines the severity of this limitation. In the latter option, the learning occurs whenever computational resources are amply available, by optimizing the parameter settings based on a simulated reality. In this case, the output of our algorithm forms the input for the customer, which may subsequently be applied with no further tuning. That means that, for problem family tuning, the running time of the problem family tuning algorithm is not prioritized, and the word "quickly" in the problem statement should be seen relative to complete enumeration. For problem instance tuning however, the learning occurs at the site of ORTEC's customer, every time they want to perform a loading operation. In this case, the running time is severely limited, and the word "quickly" should be seen relative to the running time of LoadDesigner. Indeed, LoadDesigner provides solutions in a matter of seconds, at most minutes, and since OPLB is commercialized as LoadDesigner's new and improved successor, its running time should definitely not be of a larger order of magnitude.

Considering our problem space, we focus on two sets of loading problem instances that represent the range of problems typically faced by ORTEC's customers. On the one hand, we consider simple, academic loading problems, which deal with loading a weakly heterogeneous set of boxes into a container, and exclude restrictions on, e.g., weight distribution and stackability. These problems are regularly used in the literature to compare the performance of loading methodologies. On the other hand, we consider a set of complex, practical loading problems from a large multinational, which deal with loading boxes onto pallets and, subsequently, (pick)pallets into containers, and include many customer-specific constraints.

1.4 Research Goal

In Section 1.1.2, we identified the algorithmic framework of OPLB as highly customizable by tuning its parameters, and Section 1.2 argued the need for a tuning algorithm to quickly identify good parameter settings. Broadly speaking, our research goal is to satisfy this need. We formulate our main research goal as:

> **Develop a tuning algorithm that quickly finds good parameter settings for the existing algorithmic framework such that the performance of OPLB is optimized.**

We focus on the deterministic, greedy part of OPLB. That is, we develop a tuning algorithm that quickly identifies parameter settings that optimize the quality of a single-construction solution. As stated in Section 1.1.2, the quality of a single-construction solution translates well to the quality of the final solution, and through the focus on the greedy part of OPLB, we radically speed up our measurements, while removing any stochastic noise from our observations, which greatly benefits the learning process of our tuning algorithm.

1.5 Research Questions

In order to find a satisfactory resolution to the posed problem, we have to answer a succession of research questions, the collection of which covers the entire scope of the project. First, we study the literature to obtain knowledge that is of value to our research. We want to discover the approaches that are typically taken to solve loading problems as well as the techniques that are most successful in tuning the parameters of algorithms.

- Q1 What is currently known in the literature in relation to loading methodologies and parameter tuning?
  - Q1a What are conventional optimization methods for loading problems?
  - Q1b What are suitable practices for tuning parameterized algorithms?

Second, we want to conduct an analysis of our parameter space. Before diving into sophisticated tuning procedures, we devote some time to explore the space in which we seek to optimize the parameters. Strengthened by the fact that our parameter space consists mainly of categorial variables whose interrelations are not clear a priori, we conduct experiments in order to find existing correlations, interdependencies, and dominance relations within our parameter space.

- Q2 What are the main characteristics of our parameter space?
  - Q2a What are the tunable parameters and what values can they take?
  - Q2b What does the performance landscape of our parameter space look like?
  - Q2c What correlations, interdependencies, and dominance relations exist in the parameter space?

Third, we want to combine the literature review and the parameter space exploration to develop a tuning algorithm that is most promising with regard to our problem.

- Q3 What tuning algorithm is best capable to quickly identify good parameter settings for ORTEC's existing algorithmic framework?

Finally, we want to analyze the results of our research. This analysis is two-fold. First, we consider the performance of our tuning algorithm in relation to alternative parameter tuning methods. Second, we consider the output of our algorithm, analyzing the performance of OPLB with tuned parameters and the value of our tuning algorithm for ORTEC and its customers.

Q4 How well does our tuning algorithm perform?

- Q4a How well does our proposed tuning algorithm perform relative to alternative tuning methods?
- Q4b How well does OPLB perform with the best parameter configuration as found with our tuning algorithm, compared with alternative, a priori sensible configurations?

The answers to these four research questions cover the entire scope of the project, and collectively meet our research goal. The remainder of this thesis is structured such that each of the four subsequent chapters answers one of the research questions. Finally, Chapter 6 provides our conclusions and recommendations.

2 | Literature Review

This chapter reveals what is currently known in the literature in relation to loading methods and parameter tuning. In Section 2.1, we commence with a discussion on the optimization methods that are typically used to solve loading problems. Afterwards, in Section 2.2, we turn ourselves to the scope of parameter tuning techniques known in the literature.

2.1 Loading Problems

This section discusses the methods that are typically used in the literature to solve loading problems. We first cover a typology of loading problems in Section 2.1.1. Subsequently, Section 2.1.2 outlines the optimization methods that are used throughout the literature to tackle the loading problem types most common to ORTEC's customers, and Section 2.1.3 discusses the performance of these methods on benchmark problem instances.

2.1.1 Typology of Cutting and Packing Problems

According to a generally accepted typology by Wäscher et al. (2007), loading problems fall under the umbrella of Cutting and Packing (C&P) problems. There are many types of C&P problems, which collectively represent a wide range of practical applications. The typology introduced by Dyckhoff (1990) initially provided excellent guidelines for the organization and categorization of C&P literature. The paper gives a comprehensive typology, integrating the various kinds of problems as they appear in the literature. Examples of problem designations that appear loosely in the literature and are captured by the typology include cutting stock, bin packing, knapsack, and container loading problems. Although Dyckhoff's work exhaustively covered the existing literature at the time, the considerable increase in the number of publications and the developments in the area of C&P problems have revealed some flaws and insufficiencies of the typology. Wäscher et al. (2007) propose an improved version that captures recent trends in C&P research, which we briefly discuss here.

All C&P problems share an identical structure, which can be summarized as follows. We have two sets of elements, a set of large objects and a set of small items, which are defined in one, two, or three geometric dimensions (and may possess other characteristics such as weight and value). The C&P problem is solved by selecting small items, grouping them into one or more subsets, and assigning each of them to one of the large objects such that a given objective function is optimized and the following geometric restrictions are met: (a) all small items of the subset lie entirely within the large object and (b) the small items do not overlap. Other problem-specific constraints may apply on top of these (e.g., fixed orientations and support, axle weight, and stackability constraints). Many C&P problems possess several assumptions that make them single-objective, single-period (i.e., the decision is made at one point in time rather than in a rolling time horizon), deterministic problems. However, the literature covers several problem *variants*, where some of these assumptions are abandoned. Examples of such problem

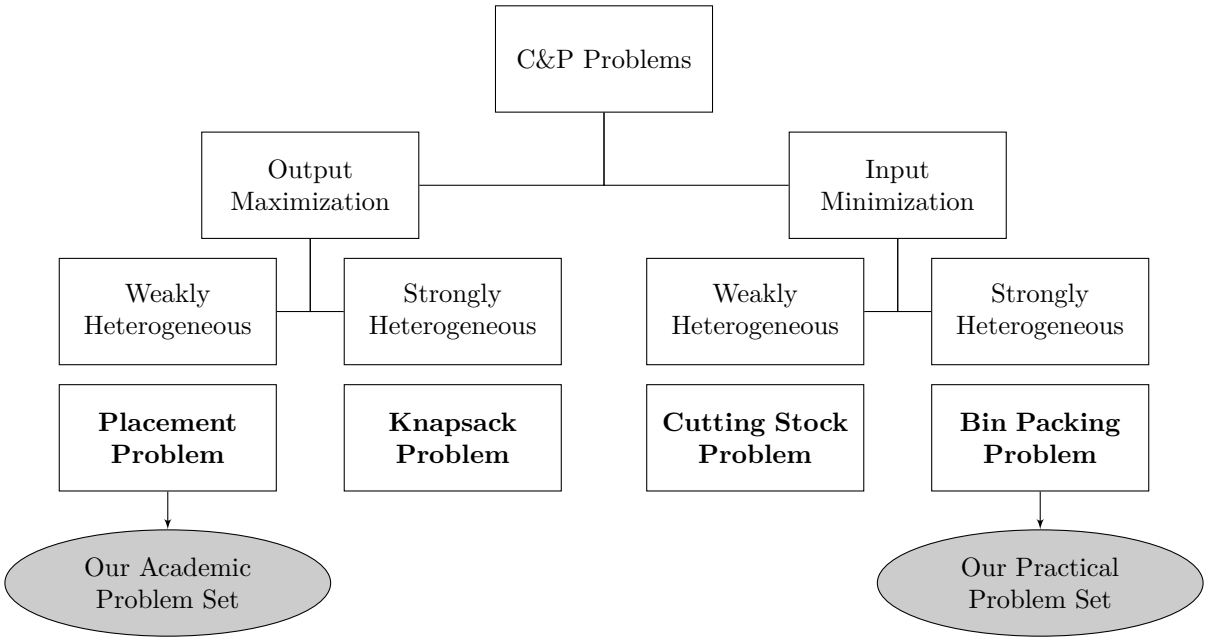

Figure 2.1: Basic types of loading problems. Adapted from Wäscher et al. (2007).

variants are stochastic problems, in which the sizes of the items are random variables (Das and Ghosh, 2003), and online problems where packing decisions have to be made in a rolling horizon without knowing the complete set of small items (Epstein and van Stee, 2007).

Wäscher et al. (2007) consider five criteria for the distinction of problem types:

1. Dimensionality. They distinguish between one, two, and three-dimensional problems.
2. Kind of assignment. They distinguish between two types of assignment: (a) output maximization, in which the full set of large objects is utilized but insufficient to accommodate all small items and the objective is to choose the best subset of small items, and (b) input minimization, in which the full set of small items needs to be accommodated whilst minimizing the selection of large objects utilized.
3. Assortment of small items. They distinguish between homogeneous, weakly heterogeneous, and strongly heterogeneous assortments of small items.
4. Assortment of large objects. They distinguish between problems with one large object and those with multiple large objects, where the sizes of the large objects may be either identical or different.
5. Shape of small items. They distinguish between regular (cuboids, circles, etc.) and non-regular shapes in the case of two and three-dimensional problems.

By electing the kind of assignment and the assortment of small items, several basic problem types are obtained, which exist in all three degrees of dimensionality. Figure 2.1 indicates the proposed nomenclature of these basic C&P problems. Any subsequent election of the remaining criteria further refines the problem. In Section 2.1.2, we use this typology to pinpoint the C&P problems of interest and discuss the techniques that are typically used to solve them.

2.1.2 Conventional Loading Methodologies

Recall from Section 1.3 that the two sets of loading problems we use to represent the loading problems typically faced by ORTEC's customers are a set of weakly heterogeneous academic benchmark instances

and a set of complex pick-pallet instances from a large multinational. They are weakly heterogeneous output maximization and strongly heterogeneous input minimization problems respectively, and both contain assortments of regularly shaped, three-dimensional items and one or more large objects. According to the characterization in Figure 2.1, the academic problems fall under the basic problem type Placement Problem, and the practical problems under the basic problem type Bin Packing Problem. In any further reference, if we use the general designation "loading problem" in the context of academic or practical problem instances, we refer to a Placement Problem or a Bin Packing Problem respectively.

Despite all efforts to create a widely accepted nomenclature, problems of these types are still known under many different names in the literature. One application of a problem with weakly heterogeneous assortments of small (often cuboid) items is most commonly known under the name Container Loading Problem (CLP), in which a consignment of boxes is loaded into a given set of containers. The typical objective function in these problems is the maximization of volume utility. Furthermore, the case where cargo has to be packed onto a pallet is most commonly referred to as the (Distributor's) Pallet Loading Problem (PLP). In fact, the PLP can be regarded as a special case of the CLP with a variable height of the container.

In the remainder of this section, we discuss some of the methodologies used throughout the literature to tackle loading problems. Roughly speaking, the existing solution methods take one or more of the following heuristic approaches (Pisinger, 2002):

1. Wall building approach, in which the container is filled by vertical cuboid layers, "walls", along the longest side of the container.

2. Stack building approach, in which the items are packed in stacks, which are themselves arranged on the floor of the container.

3. Horizontal layer building approach, in which the container is filled by horizontal cuboid layers that each cover the largest possible fraction of the surface underneath.

4. Cuboid arrangement approach, in which identical items are packed into cuboid arrangements in order to fill the largest possible subspace of the container without internal volume utility losses.

5. Guillotine cutting approach, in which the container has a slicing tree representation, where each slicing tree corresponds to a successive guillotine partitioning of the container into smaller pieces, and the small items represent the leaf nodes.

Over the years, loading methodologies have become increasingly successful at applying these heuristic approaches. In the early 21st century, numerous authors reported substantially improved algorithms. Juraitis et al. (2006) propose a randomized heuristic based on the wall building approach in which the focus is on finding the best mixture of constructive heuristics. Moura and Oliviera (2005) take a similar wall building approach and try to improve the single-construction through a GRASP meta-heuristic, while Bortfeldt and Gehring (2001) present a hybrid Genetic Algorithm (GA) to improve the single-construction. Standing on the shoulders of giants, authors have recently developed increasingly effective algorithms. In a recent review by Bortfeldt et al. (2013), a selection of articles that cover the current state-of-the-art in loading methodologies is discussed. From this review, appended with some other recent notable work, we briefly discuss the most successful methods here. Fanslau and Bortfeldt (2010) propose a tree search method that combines a generalized cuboid arrangement approach with a special form of tree search. Their method is based on building composite arrangements (i.e., existing of various different items) and continuously selecting the most appropriate arrangement for a given residual space in the container. Zhang et al. (2012) build on their work by adapting the composite arrangement selection

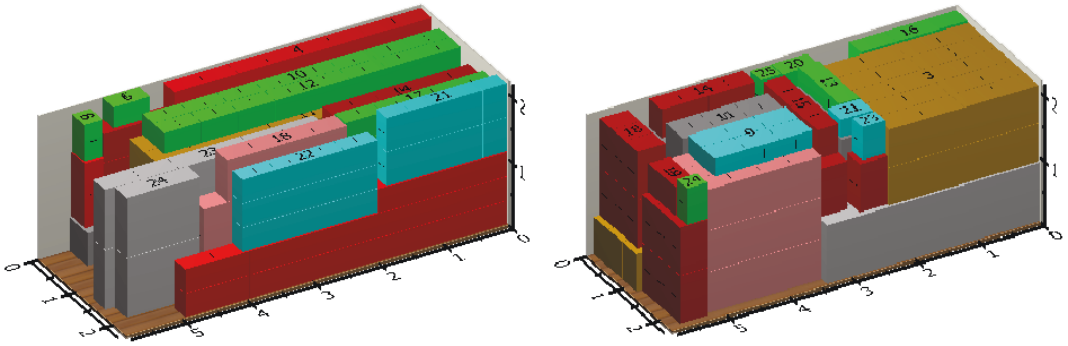

Figure 2.2: On the left a wall building approach, on the right a cuboid arrangement approach.

policy such that a holistically optimal rather than a myopically optimal choice is made. Gonçalves and Resende (2012) propose a multi-population GA combined with a novel procedure to join free spaces to acquire full support from below. Finally, Terno et al. (2000) use a horizontal layer building approach where optimal two-dimensional loading patterns are obtained in a branch and bound framework.

Although these state-of-the-art approaches have proven to be particularly strong at solving loading problems, they all go beyond the degree of freedom obtainable by solely configuring the parameters of the existing algorithmic framework, and are therefore not readily applicable to our problem. However, the algorithmic framework in OPLB is designed such that, by sheer tuning of its parameters, we can mimic the 5 general construction approaches that form the basis of all the state-of-the-art techniques that we just discussed. Figure 2.2 shows, for example, the resulting loads for a wall building and a cuboid arrangement approach, both realized with OPLB.

2.1.3 State-of-the-art Performance

To determine the best method among the state-of-the-art techniques from Section 2.1.2, in fact to claim that one loading algorithm is better than another at all, we must evaluate their performance, indicated by the quality of the solutions they produce. Evidently, to ensure fair comparison, the various algorithms must be tested on the same problem instances. Moreover, to ensure statistically significant comparison, the algorithms must be tested on multiple problem instances. Finally, to ensure practically meaningful comparison, the algorithms must be tested on a set of instances that closely resemble the practical instances they intend to solve. The problem instances that are regularly used in an academic setting to compare various loading algorithms are Bischoff and Ratcliff (BR) benchmark problem instances (Bischoff and Ratcliff, 1995). These instances are randomly generated and replicable, which allows the production of large sets of replicable problem instances. The input parameters for the generation of problem instances include the heterogeneity of the input set (i.e., the number of different item types), limits on the item dimensions, the volume of the container, and a random seed number. Given these input parameters, the set of possible and replicable problem instances is infinite.

The state-of-the-art loading methodologies mentioned before report highly competitive results on BR benchmark instances. The complete set of BR instances is segregated in homogeneous (i.e., BR0 with 1 type of item), weakly heterogeneous (i.e., BR1-BR7 with 3-20 types of items), and strongly heterogeneous (i.e., BR8-BR15 with 30-100 types of items) problem instances. We use the weakly heterogeneous sets of BR instances as our academic problem space, as they best reflect the problem instances faced by ORTEC's customers. Table 2.1 indicates the average volume utility, typically the most important performance indicator in loading problems, of several of the algorithms mentioned on weakly heterogeneous (i.e., BR1-BR7) and strongly heterogeneous (i.e., BR8-BR15) sets of instances.

Author	Weakly Heterogeneous (BR1-BR7)		Strongly Heterogeneous (BR8-BR15)	
	Volume Utility (%)	Computation Time (sec.)	Volume Utility (%)	Computation Time (sec.)
Juraitis et al. (2006)	89.26	-	-	-
Moura and Oliviera (2005)	92.65	-	87.69	-
Bortfeldt and Gehring (2001)	90.06	316	87.34	316
Fanslau and Bortfeldt (2010)	95.01	319	93.82	320
Zhang et al. (2012)	95.35	135	93.82	1001

Table 2.1: Performance of algorithms by different authors on weakly and strongly heterogeneous BR instances. Adapted from Zhang et al. (2012).

Note that these algorithms are highly specialized, and have computation times that go beyond the limitations set in our research. Indeed, a running time that exceeds five minutes on these relatively simple BR instances translates to a running time that is considerably longer than the running time of LoadDesigner on more complex problems (see Section 1.3). Moreover, our problem space is actually the space of possible parameter settings that govern the algorithmic framework, rather than the space of possible loading patterns. That is, we are limited to the loading patterns that can be achieved with the available parameters, and it is not guaranteed that the global optimum in our problem space (i.e., the best possible parameter configuration) corresponds to the optimal loading solution. For these reasons, we do not expect to realize similar performance, but we seek to approximate these performances as closely as possible under the given research constraints.

2.1.4 Conclusion

This section served three main purposes. First, we formally described Cutting & Packing problems, the branch of OR challenges to which the loading problems at ORTEC's customers belong. Second, we covered a selection of state-of-the-art optimization methods specific to the types of C&P-problems of concern, and indicated how the algorithmic framework of OPLB is able to mimic such techniques by sheer parameter tuning. Finally, we introduced our academic problem space, the weakly heterogeneous BR instances, and provided a basis for comparison of the performance analysis of OPLB on this problem space.

2.2 Meta-Optimization

This section discusses the field of meta-optimization, the use of one optimization method to optimize another optimization method. The roots of meta-optimization go back as far as 1978 when Mercer and Sampson (1978) proposed a method to automatically tune the parameters of a GA. Its motivation lies in the fact that configuring an optimization method by hand is a laborious task that is susceptible to human misconceptions of what behavior makes the optimizer perform well. This section includes two subsections that each approach the task of meta-optimization from a different angle. In Section 2.2.2 we confine ourselves to the automatic tuning of parameters and discuss the applicable parameter tuning literature. We start, however, with a wider perspective by exploring the world of hyper-heuristics in Section 2.2.1, in which automation of the process of selecting or generating several simpler optimizers is pursued. Finally, Section 2.2.3 discusses Race algorithms, a recent concept that enables efficient application of parameter tuning techniques to the specific task of tuning parameterized algorithms in a large search space.

2.2.1 Hyper-Heuristics

A hyper-heuristic is a search method or learning mechanism that aims to automate the design and adaptation of heuristic methods in order to solve hard computational search problems (Burke et al., 2010). The motivation behind this relatively new research field is twofold: (i) to pursue the idea that optimizing on the search space of other, simpler optimizers is more efficient than optimizing on the solution space directly, and (ii) to raise the level of generality at which optimization technologies can operate. The term hyper-heuristics was first used in 2000 to denote "heuristics that choose heuristics" in the context of combinatorial optimization (Cowling et al., 2000). In this light, a hyper-heuristic is a high-level search approach that can select and apply appropriate low-level heuristics from a given search space.

We use a classification by Burke et al. (2010) as a basis for our discussion. They distinguish the scope of hyper-heuristics in three dimensions, which we use here to categorically discuss some noteworthy work in the field.

Heuristic Search Space: Construction or Perturbation

First of all, Burke et al. (2010) distinguish two types of heuristics that the low-level heuristic search space may comprise.

- Constructive heuristics: Start with an empty solution and incrementally construct a solution.
- Perturbative heuristics: Start with a feasible solution and attempt to improve on it through a local search procedure.

The hyper-heuristic follows the behavior of its underlying low-level heuristics. For example, if the low-level heuristic search space consists of constructive heuristics, the challenge is to build a solution incrementally, by continuously selecting the heuristic that is most suitable for the given problem instance in the current problem state. The hyper-heuristic should therefore learn to associate specific problem instances and partial solution stages with adequate low-level construction heuristics.

Numerous approaches have been proposed to facilitate efficient collaboration between existing constructive or perturbative heuristics. In the field of exam timetabling, for example, Bilgin et al. (2006) conducted an experimental study on the performance of several perturbative heuristic selection techniques and corresponding move acceptance strategies. For loading problems in particular, efforts have been made to learn a solution process by using a learning classifier system to relate the problem state to the most suitable of a small set of constructive loading heuristics, where the problem state is defined in a binary system using the percentage of small items that still need to be packed and the distribution of those items over several sizes (Ross et al., 2002).

Heuristic Employment: Selection or Generation

Besides the nature of the low-level heuristics, hyper-heuristics differ in how they employ these heuristics. Burke et al. (2010) distinguish two ways to employ the low-level heuristics.

- Heuristic selection: Develop fruitful combinations of pre-existing heuristics.
- Heuristic generation: Produce new heuristics from the basic components (building blocks) of pre-existing heuristics.

In case of heuristic selection, the challenge is to know *when* to select *which* heuristic. All previous examples fall in this category. In case of heuristic generation, the challenge is to automatically generate

new heuristics that outperform the pre-existing heuristics that were decomposed to obtain the building blocks for the new heuristic. Many such efforts take a genetic programming approach (Burke et al., 2009). For the famous traveling salesman problem, for example, hyper-heuristic based on generation have been developed that routinely produce highly competitive tours (Keller and Poli, 2007). For loading problems in particular, genetic programming has successfully been applied to generate good quality heuristics from very basic building blocks: arithmetic operators and geometric parameters such as item volume and the coordinates of the remaining empty spaces (Burke et al., 2007).

Learning Mechanism: Online or Offline

Finally, the nature of learning is a factor that divides the spectrum of hyper-heuristics. We can distinguish between two types of learning that are used by the high-level search method: online and offline learning. The difference between these two is best understood by an example. Suppose we want to solve a particular loading problem instance x_n, which belongs to a family of problem instances $\mathcal{X} = \{x_1, x_2, .., x_k\}$. Then there are two manners in which the search method may learn.

- Online learning: The learning takes place while solving problem x_n. That is, the hyper-heuristic is tailor-made to solve problem x_n. During the solution process, it receives feedback on the most suitable solution approach for this specific problem instance and adapts accordingly.
- Offline learning: The learning takes place while solving a set of training problem instances that are similar, but not necessarily identical to x_n, the problem family $\mathcal{X}$. These similar problem instances are used as training instances to develop a (meta)heuristic that performs well on the entire problem family. Subsequently, the resulting (meta)heuristic can be used to solve both the unseen instance x_n and any other instance in the problem family. In this case the solution process for x_n alone is typically substantially faster, since the computationally intensive learning process has already occurred offline.

Note that this distinction is the same as our previously made distinction between problem instance and problem family tuning (see Section 1.2). In case of offline learning (problem family tuning), we essentially simulate reality with a number of problem instances similar to the actual problem instance. Most previous examples fall in this category. In case of online learning (problem instance tuning), the challenge is to tune the parameters for one specific problem instance. For the vehicle routing problem, this has been successfully done using roulette wheel selection of low-level neighborhood structures with an adaptive layer that stochastically controls the selection process according to their past performance and a Simulated Annealing based acceptance strategy (Pisinger and Ropke, 2007). For loading problems in particular, a similar method has been successfully proposed, where instead of an adaptive layer with reinforcement learning, a tabu search is used as the high-level heuristic to traverse the perturbative heuristic search space (Dowsland et al., 2007).

Conclusion

In short, hyper-heuristics mean to generate efficient hybridization of existing heuristics. The greedy part in the algorithmic framework of OPLB is essentially a search space of low-level heuristics, as each complete parameter configuration corresponds to a constructive heuristic. Moreover, with the division into online and offline learning, the hyper-heuristic literature covers both problem family and problem instance tuning. Note that our research scope limits our freedom to the heuristics that can be described by the available parameter settings. That is, we cannot widen our search space, but only select heuristics from the available search space, which makes heuristic generation unsuitable. Still, we essentially want

to *select* the best alternative from a search space of *constructive heuristics*, making the hyper-heuristic literature seem particularly appropriate to our problem. However, our search space of heuristics is formed by all possible parameter configurations, and is, as Chapter 3 reveals, *much* larger than the hyper-heuristic literature ordinarily deals with. More importantly, our research scope limits our degree of freedom to such extent, that we cannot change our heuristic selection during the solution process. This forbids typical hyper-heuristic effort to select the most appropriate heuristic dependent on the problem state, essentially leaving the hyper-heuristic literature redundant. Our degree of freedom is limited to the set of parameters that we can tune within predefined boundaries, directing our view to the field of parameter tuning.

2.2.2 Parameter Tuning

In the field of parameter tuning, we are interested in "a staggeringly difficult yet beautifully concise problem" (Lizotte, 2008):

$$\arg\max_{y \in \mathcal{Y}} f(y) \tag{2.1}$$

For example, the function f can be the deterministic fertility of the soil or the stochastic solution quality of an algorithm, where the corresponding sets $\mathcal{Y}$ would be the space of possible locations we can start a fruit plantation or the space of parameter settings that can be given as input to the algorithm. The aim in solving Equation 2.1 is to quickly identify, from a set of alternatives $\mathcal{Y}$, the one alternative that gives the highest return. The function f may be either deterministic or stochastic. In the latter case, any measurement $f(y)$ contains noise, the return is a random variable, and the objective is usually to maximize its expectation value. The matter of stochasticity is an interesting one in our research and we return to this in Section 4.1.1. For now, we include approaches that deal with stochastic as well as deterministic functions in our review. In light of its application to parameter tuning, we restrict ourselves to situations where $y \in \mathcal{Y}$ is a multi-dimensional vector of numerical, ordinal, and categorical attributes and the function $f : \mathcal{Y} \to \mathbb{R}$ is an algorithm that translates the input vector into a single value, indicating the solution quality.

The literature covers many techniques that attempt to solve this problem. In the remainder of this section, we cover three distinct branches in the scope of parameter tuning techniques.

Local Optimization-Based Techniques

An acceptable solution to Equation 2.1 is an $y \in \mathcal{Y}$ that results in a near-optimal point in the complete solution space. That is, we attempt to find the global optimum. Whereas global optimization is often very difficult, *local* optimization in a smaller subspace proves to be much easier. To verify that a solution y^+ is a local optimum, we only need to examine points in a neighborhood of y^+. If the function f is differentiable, this can be achieved by examining its first and second derivatives and for multi-dimensional, non-differentiable functions too, efficient local optimization techniques exist.

The branch of global optimization techniques that is based on local optimization aims to utilize this efficiency of local optimizers to find a global optimum. The simplest example of such a technique is random restarts, in which several points are randomly drawn from the set $\mathcal{Y}$ and subsequently local optima in the vicinity of each of these points are found through some efficient local optimizer (e.g., hill climbing). To avoid repeated convergence to the same local optimum, the density clustering method rejects any randomly drawn restarts in the regions around the local optima found so far (Horst and Pardalos, 1995). Another popular global optimization technique based on local optimization and random sampling is Simulated Annealing (SA), first introduced by Kirkpatrick et al. (1983). The technique is inspired by a

physical analogy with cooling crystal structures that spontaneously "converge" to stable configurations. In this approach, a random walk over the domain of the function is taken. The walk is defined by a proposal distribution, typically a predefined neighborhood structure of y^+, and an acceptance probability function. We initially accept almost all movements regardless of their effect on $f(y)$ and gradually start rejecting inferior points with increasing probability. Provided several conditions are met (e.g., the use of a neighborhood structure that allows traversal of the entire domain, and a sufficiently gradually decreasing temperature) SA converges to the global optimum in the limit of infinite iterations (Horst and Pardalos, 1995), although this is often too time-consuming to be practical.

Population-Based Techniques

Another branch of parameter tuning techniques that find their inspiration in nature are techniques based on the evolution of populations. Genetic Algorithms (GAs) in particular attempt to directly apply the biological process of evolution and natural selection to the problem of parameter tuning. In such methods, a population of individuals evolves over time according to their fitness using biologically inspired cross-over, where two individuals are combined, and mutation, where individuals are randomly perturbed.

One of the first attempts to apply a GA to tune a parameterized algorithm is Meta-GA (Greffenstette, 1986). Essentially every GA can be used as a parameter tuner as long as it can cope with the vectorial representation of the tunable parameters (Eiben and Smit, 2011), but the noise in the measurements and the computationally expensive evaluations form insoluble issues for most GAs. To overcome these drawbacks, special GAs have been developed specifically for tuning purposes.

Relevance Estimation and Value CAlibration of parameters (REVAC) (Nannen and Eiben, 2006), for example, is a specific type of meta-GA where the population approximates the probability density function of the most promising areas of the performance landscape. That is, the population mimics the evidence collected on parameter values and their performances such that parameter values that have proven to result in good performance are more abundant in the population. This method is able to incorporate measurement noise and runs faster than Meta-GA. However, the function that REVAC uses to sample its population is decomposed by parameter, hence blind for parameter interactions, which may impose difficulties in applications where parameters have a high degree of synergy.

Sequential Measurement Techniques

Finally, there is a branch of parameter tuning techniques that focus on sequential measurements. The local optimization-based techniques we discussed all rely on random sampling and random walks through the domain in the hope to eventually converge to a near-optimal solution. In the field of Optimal Learning, the focus is on "the walk". Instead of a random walk, Optimal Learning techniques sequentially and deterministically guide the walk towards the next point depending on previous observations (Powell, 2010). An important factor in this is that the number of measurements is kept to a minimum, because they are costly or time-consuming. Therefore, as much information as possible is absorbed from the outcome of previous measurements before deciding which alternative to measure next.

Every Optimal Learning problem has three fundamental components: a measurement decision, obtained information, and an implementation decision. The Optimal Learning literature is divided into two parts through their respective focus on online or offline problems, whose definition is different from the one we discussed in Section 2.2.1 for hyper-heuristics. Offline problems are solely concerned with the value of the implementation decision. Online problems, also denoted by multi-armed bandits (after the exemplifying application on slot machines: one-armed bandits), seek to maximize the cumulative

value of all the measurement (and implementation) decisions. In our case, the measurements, although time-consuming, are not reflected in actual implementations in which the intermediate performances are of importance, so we deal with an offline problem. Some relatively simple heuristics are applicable to both online and offline problems (Frazier et al., 2008). Examples of such heuristics include Boltzmann exploration and Upper Confidence Bound (UCB) policies, in which the measurement policy is to consistently measure the alternative with the highest UCB resulting in continuously shrinking performance distributions (Chang et al., 2007). Besides such exceptions, the literature of online and offline learning problems is traditionally segregated. What they have in common, however, is their focus on reducing the number of measurements, and their revolvement around numerical or ordinal parameters. Many techniques, including Boltzmann exploration and UCB, require each alternative to be measured at least once (Frazier et al., 2008). Such techniques are typically applied to Ranking & Selection (R&S) problems, in which the objective is to find the alternative with the highest value from a finite, and typically small, set of alternatives (Bechhofer, 1954).

More sophisticated techniques are able to tackle larger search spaces. In such approaches, a priori knowledge on interrelations in the parameter space is used to design correlated beliefs about the effect of the measured performance of some choice y_n on the expected performance of some other choice y_k, such that we develop vague ideas on the performance of choices we have not measured. Such extrapolation of the measurement outcomes of one alternative to the beliefs on other alternatives is called generalization. Most sequential measurement policies that are applicable to large search spaces rely heavily on generalization techniques. For example, the knowledge gradient policy, by Frazier et al. (2008), myopically maximizes the expected increment in the value of information by continuously directing the walk towards the point that will reveal the most information on other unseen points. While the updated knowledge-gradient policy by the same authors assumes a priori knowledge on correlations in the belief structure of $\mathcal{Y}$, the hierarchical knowledge gradient (Mes et al., 2011) solely requires some a priori known aggregation structure of the parameter space, which is generally easier to obtain. A related field of Bayesian global optimization of continuous functions places a prior belief on the function f and relies on the laws of probability to update these beliefs to posterior distributions based on accumulated observed data. The response-surface methodology (Law and Kelton, 2000), for example, takes a Bayesian approach to fit a second order response surface to a series of experiments, conducted using a simple experimental design, and subsequently finds the optimal point on this surface. Most sequential measurement policies are designed for stochastic experiments with noisy measurements, but some can be applied to deterministic cases, and others are designed specifically for noise-free measurements. The efficient global optimization (EGO) method (Jones et al., 1998), for example, assumes deterministic measurements and attempts to fit a smooth, differentiable function through previous measurements while continuously selecting points based on expected improvement of the objective function.

The problem with generalization techniques such as the knowledge gradient and EGO, in light of our research, is that they assume a spatial relationship between the parameter space on the one hand, and a supposedly smooth performance landscape on the other hand. Indeed, whether the tuning technique looks for linear relations, smooth function fits, or response surfaces, they all assume such a spatial relationship, which they attempt to exploit to guide the tuning process. In many practical applications with numerical variables, such an assumption proves to be a reasonable one. However, for categorical variables, which only have a symbolic meaning and do not have a natural way to be encoded, the assumption of a regular or smooth performance landscape is not likely to hold. Indeed, it is not trivial to quantify the distance between symbolic parameter values (e.g., sorting items by volume or by priority), let alone to assume that this unquantifiable distance corresponds with similar distances in the performance landscape. Figure 2.3 illustrates how a hill-climbing method, an example of a simple sequential measurement policy, exploits

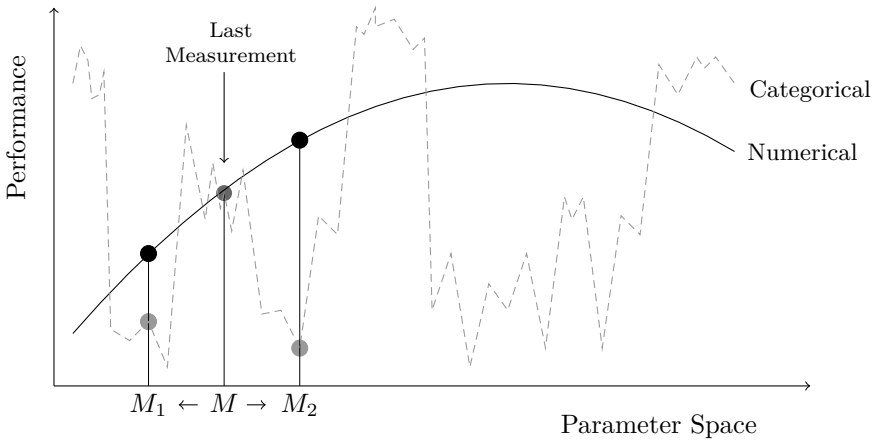

Figure 2.3: An example of the tuning process conducted by a hill-climbing method on a typical numerical and categorical parameter space. The last tested configuration is M. The hill-climbing method enforces M_1 and M_2 as the next measurements. On the numerical parameter space, the search continues at M_2, and ultimately ends at the top of the smooth performance landscape. On the categorical parameter space, it is non-trivial to even determine M_1 and M_2, and the search is stuck in a local optimum.

the assumption of a smooth performance landscape, and how this assumption backfires on a categorical parameter space.

There are some simpler generalization techniques that *can* be applied to categorical parameters. These methods are typically non-parametric in the sense that they do not assume any characteristic space structure, but determine the structure from the data. The k-nearest neighbor (kNN) method (Buttrey, 1998), for example, puts beliefs on unseen points in the domain based on the k closest previous measurements. How we define "closest" depends on the application. For numerical parameters, we have the one-dimensional distances, but we may need to prioritize among the parameters. For categorical parameters, we need to rely on a priori knowledge on the parameter space to define even one-dimensional distances.

A suitable sequential measurement technique, in light of our research, is the use of Biased Sampling Distributions (BSD) (Birattari et al., 2002). This technique does not require any distance metric, and randomly samples parameter settings for new measurements, while continuously adapting the sampling distributions based on previous measurements. For categorical parameters, new values are sampled from discrete sampling distributions with a bias towards the parameter settings from known alternatives that have shown good performance.

In summary, the applicability of sequential measurement techniques is limited by the size of the search space, eliminating any technique that requires a full factorial design. In general, their focus on reduction of the number of measurements makes the techniques particularly suitable for problems where measurements are costly. In our case, this would be particularly true if we choose for problem instance tuning, in which case running time is especially valuable. If we choose for problem family tuning, the tuning would occur whenever computational resources are amply available, and the pressure to reduce measurements is relatively low. Finally, the inclusion of categorical parameters in our parameter space limits the applicability of sophisticated generalization techniques, as they attempt to model the performance landscape, exploiting the assumption of a spatial relationship between the parameter space and a supposedly smooth performance landscape, which ordinarily does not hold for categorical parameters. Local optimization-based techniques and population-based techniques often present similar problems, be it less explicitly. In a GA, for example, each member is represented by an encoded description which evolves through mutation and cross-over, where there is a certain mutation probability proportional to

spatial relation between parameters. Again, this assumes some distance metric in the parameter space, which is meaningless for categorical variables. In brief, the use of Biased Sampling Distributions is the most suitable sequential measurement technique in light of our research, as it does not require the existence of a smooth and predictable performance landscape nor a full factorial design, while still allowing directed search based on previous observations.

2.2.3 Race Algorithms

A Race algorithm is essentially a tool to efficiently apply sequential measurement techniques such as BSD to one specific type of application: problem family tuning of parameterized algorithms. They are intended as "an automatic hands-off procedure for finding a good configuration through statistically guided experimental evaluations, while minimizing the number of experiments" (Birattari et al., 2002), and were originally proposed to efficiently solve the model selection problem in the field of machine learning (Maron and Moore, 1997). They take inspiration of sequential measurement and population-based techniques and modify them to be successfully applied to the tuning of parameterized algorithms (Birattari et al., 2002). Race algorithms are based on the statistical comparison of the performance of different algorithms, which are regarded as black boxes. Therefore the requirement of an appropriate encoding scheme or distance metric, and therefore the impediment for application to categorical parameters, is removed. The objective of a Race algorithm is to find good settings for the parameters of an algorithm in a limited timespan in order to achieve its full potential. The tedious and time-consuming task of tuning the parameters is often done manually and some authors state that up to 90% of the total time needed to develop an algorithm is devoted to tuning its parameters (Adenso-Diaz and Laguna, 2006). An effective automatic tuning procedure would radically speed up the process and improve the algorithm's performance. Indeed, the tuning of parameters that drive algorithms is itself a scientific endeavor and deserves more attention than it has received in the OR literature (Barr et al., 1995).

Race algorithms are intended for problem family tuning, and therefore aim to optimize the algorithm's performance on a typical, yet unknown problem instance. An algorithm's performance on a typical problem can be indicated by its average performance μ on a possibly infinite training set of representative problem instances, the problem family $\mathcal{X}$. To determine the true μ, we should evaluate the algorithm's performance on the complete set $\mathcal{X}$. In cases where $\mathcal{X}$ is large, even infinite, this becomes computationally prohibitive. The basic idea behind Race algorithms is that we do not determine μ precisely, but rather estimate it with a sample mean m based on a subset of $\mathcal{X}$. To determine with 100% accuracy that algorithm A outperforms algorithm B, we have to establish that $\mu_A > \mu_B$, but since a near-optimal solution would suffice, it is sufficient to conclude, with some statistical significance, that $\mu_A - \mu_B > 0$ based on $m_A - m_B$. With the right statistical tests, this distinction substantially decreases the required computation time. Figure 2.4 shows an example of the effect of such statistical tests.

One of the first successful efforts to develop a Race algorithm was done by Birattari et al. (2002), the resulting algorithm of which they named "F-Race". This procedure measures batches of configurations on the same problem instance and uses Friedman's (hence the name F-Race) non-parametric two-way analysis of variance by ranks (Conover, 1999), where no assumptions have to be made about the performance distribution, to test for statistically inferior configurations. The Friedman analysis first tests the null hypothesis that none of the configurations is inferior to at least one other configuration. If rejected, it justifies the imposition of pairwise Student t-test comparisons between individual candidates. The suitability of statistical test may differ per application. For example, dependent on whether batch measurements are used, we may conduct either paired or unpaired Student t-tests.

The original F-Race procedure starts with a set Θ of configurations from which it tries to exclude statistically inferior members as soon as possible while solving a set of problem instances, the training

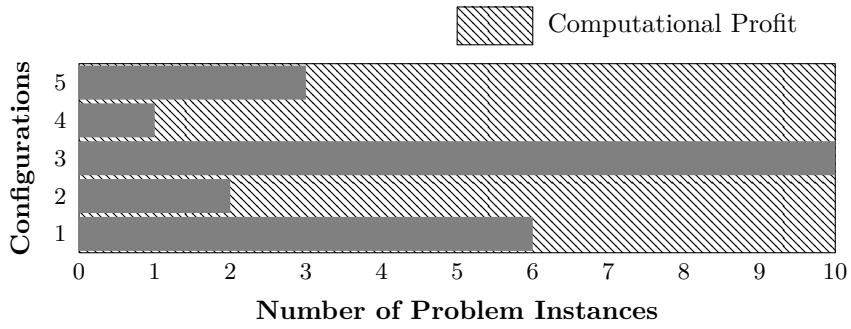

Figure 2.4: Example of the effect of statistical inferiority tests in a Race algorithm with five configurations and a training set of ten problem instances. The configurations are represented by the horizontal bars, which end at the instance after which the configuration is deemed statistically inferior to at least one other known configuration. This frees up computation time for more reliable evaluation of the most promising configurations. In this case, we conclude that configuration 3 is the best, and the overall computational profit of imposing the statistical tests is the hatched area.

problems, until only one configuration remains: the winner of the race. Statistical evidence is collected by applying the algorithm to different problem instances, and, since it is typically applied to stochastic algorithms, statistical evidence is strengthened through repeated application to the same problem instances. An obvious drawback of this procedure is that the initial set Θ need be obtained through a full factorial design which becomes impractical and computationally prohibitive for algorithms with many tunable parameters. Balaprakas et al. (2007) propose a modification of F-Race that tackles this issue: the Iterated F-Race (I/F-Race), which consists of multiple, iterative applications of F-Races. For each race, or iteration, a small set of possible configurations is selected using a probabilistic model defined on the parameter space. In the first race, a multi-variate uniform distribution is used as initialization. In each subsequent race, the sampling distribution is biased towards the most promising parameter values. That is, a multi-variate distribution is fit on the surviving parameter vectors, using BSD (see Section 2.2.2), which is then used as a probability density function to sample points for a new population (Eiben and Smit, 2011). This way we create a population that approximates the probability density function of the underlying performance landscape much like REVAC in Section 2.2.2. For categorical parameters, a multi-variate discrete distribution is used as a probabilistic model.

Finally, in an iterated Race algorithm, the question arises which configurations to include in the first race. The aim of the initial sampling should be to obtain a good representation of the entire parameter space. The I/F-Race procedure uses random sampling to initiate the first race.

In summary, a Race algorithm is a problem family tuning method that efficiently finds good parameter settings in a large search space, and it can readily be applied to categorically parameterized algorithms. An iterated Race algorithm consists of the following three main components:

1. Initial Sampling: Select a set of configurations that are used for the first race.
2. Statistical test: Select a method to test for statistical inferiority of known configurations.
3. Resampling: Determine a resampling policy to generate new configurations based on previous observations.

Along with these components, the Race algorithm needs several parameters such as the number of parallel configurations and the computational budget.

2.2.4 Conclusion

This section covered the scope of meta-optimization techniques that are potentially applicable to our research. We first took a wide perspective, discussing the field of hyper-heuristics, a branch of algorithms that aim for efficient hybridization of simpler heuristics. Although the suitability of hyper-heuristic techniques for our research seems plausible, our limited degree of freedom leaves them redundant. Afterwards, we confined ourselves to parameter tuning techniques. We discussed three branches of global optimization techniques for parameter tuning. Many of such techniques are inapplicable due to the size of our search space and the categorical nature of the parameters. Moreover, whether we choose for problem instance or problem family tuning, and therefore how valuable running time is, determines the applicability of techniques with a strong focus on reducing the number of measurements. Finally, we discussed Race algorithms, a tool to efficiently apply parameter tuning techniques to the task of problem family tuning of parameterized algorithms. Given their ability to handle enormous search spaces of categorical parameters, Race algorithms, with BSD as the sequential measurement technique, form a good basis for our method in case we opt for problem family tuning. In case we opt for problem instance tuning, repeated measurements of the same configurations on multiple problem instances are superfluous (i.e., we are only concerned with the performance on the specific problem instance at the customer site), hence our tuning process contains no stochastic elements anymore. In this case, the statistical inferiority tests of Race algorithms are redundant, and we solely need the Biased Sampling Distributions to guide the walk through our parameter space.

3 | Parameter Space Analysis

This chapter provides an analysis of our parameter space: the parameters that govern the greedy construction phase within OPLB. Section 3.1 reports an exhaustive list of all the tunable parameters and their possible settings. Furthermore, Section 3.2 discusses the size of the search space, justifying our choice to disregard complete enumeration. Afterwards, in Section 3.3 we conduct some experiments on our parameter space, of which the results aid in the development of an efficient parameter tuning algorithm.

3.1 Tunable Parameters

As pointed out in Section 1.1.2, the greedy part of the algorithm consists of six consecutive sorting steps, so called filters, that each govern a subdecision of an item placement. Three of them, the item, orientation, and arrangement filters, determine which (arrangement of) item(s) needs to be placed in what orientation. The other three, the space, position, and load equipment filters, address the position in which it should be placed. Each filter is a sorting mechanism that sorts the possibilities regarding its subdecision according to its parameter settings. Therefore most parameters in each filter are sorting criteria, which can take several categorical values, and corresponding (binary) sorting directions, which can be set to either "up" or "down", sorting the possibilities in ascending or descending order respectively. The first sorting criterion (categorical) and the first direction of sorting (binary) in each filter provide an ordered preference list. The subsequent sorting criteria and directions *within the same filter* break any ties in previous sorting. Some filters have additional parameters. All parameters are presented in one integrated XML script, which forms the recipe for the underlying algorithm. Throughout this section, we provide several examples of loading solutions on relatively simple BR2 instances (see Section 2.1.3), and provide the corresponding XML snippets. We define the dimensions X, Y, and Z and the origin, or coordinate (0,0,0), of the container as in Figure 3.1.

We discuss the six filters individually in Section 3.1.1 through Section 3.1.6, and the order of the filters in Section 3.1.7.

3.1.1 Item Filter

The item filter sorts all items that still need to be loaded. All items have a predefined length, width, and height, which are of importance when certain orientations are not allowed (e.g., "this side up" packages). Table 3.1 reports the different categorical values that can be taken by the first, and any subsequent, sorting parameter in the item filter. The settings "maxHeight" through "minLength" consider the current problem state, and are conditional on the available space in the container and preferred orientation. Figure 3.2 shows an example of the effect of parameter settings in the item filter.

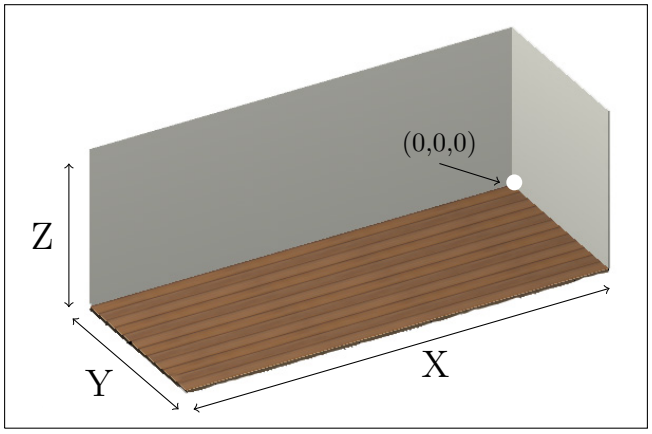

Figure 3.1: Defined dimensions and origin of the container.

Value	Description
minDimension	the length of the item in its shortest dimension
maxDimension	the length of the item in its longest dimension
volume	the volume of the item
priority	the priority of the item
length	the predefined length of the item
height	the predefined height of the item
width	the predefined width of the item
maxHeight	the maximum height at which the item can be placed
minHeight	the minimum height at which the item can be placed
maxWidth	the maximum width at which the item can be placed
minWidth	the minimum width at which the item can be placed
maxLength	the maximum length at which the item can be placed
minLength	the minimum length at which the item can be placed

Table 3.1: Possible parameter settings for the sorting parameters in the item filter.

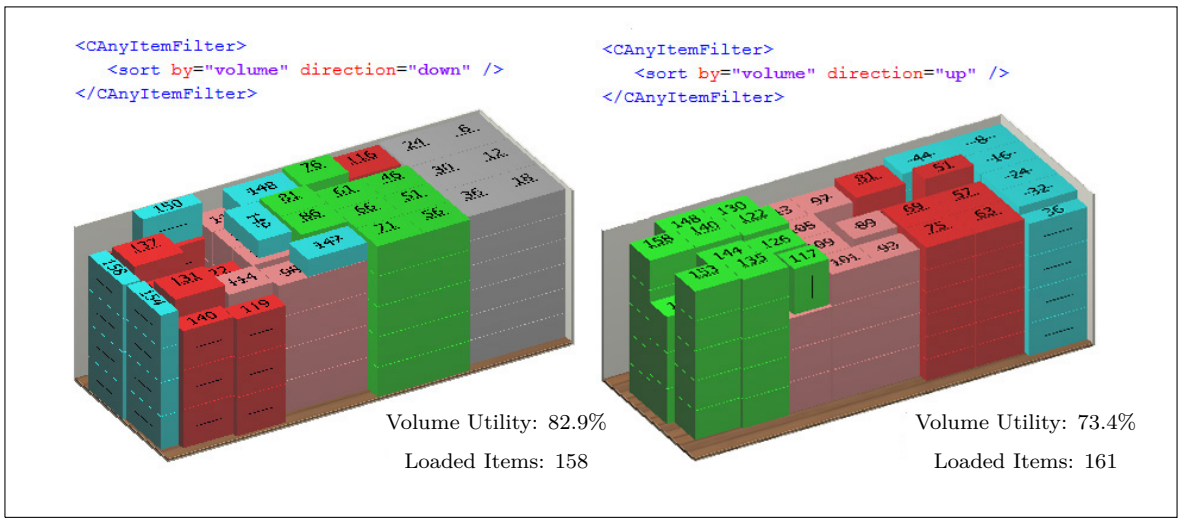

Figure 3.2: Example of the effect of parameter settings in the item filter. Note that none of the large gray items are loaded in case we start with the smallest items (i.e., picture on the right). Note, moreover, that more items are loaded in the picture on the right, yet the volume utility, typically the most important KPI, is lower.

Value	Description
length	the length of the container
width	the width of the container
height	the height of the container

Table 3.2: Possible parameter settings for the sorting parameters in the orientation filter.

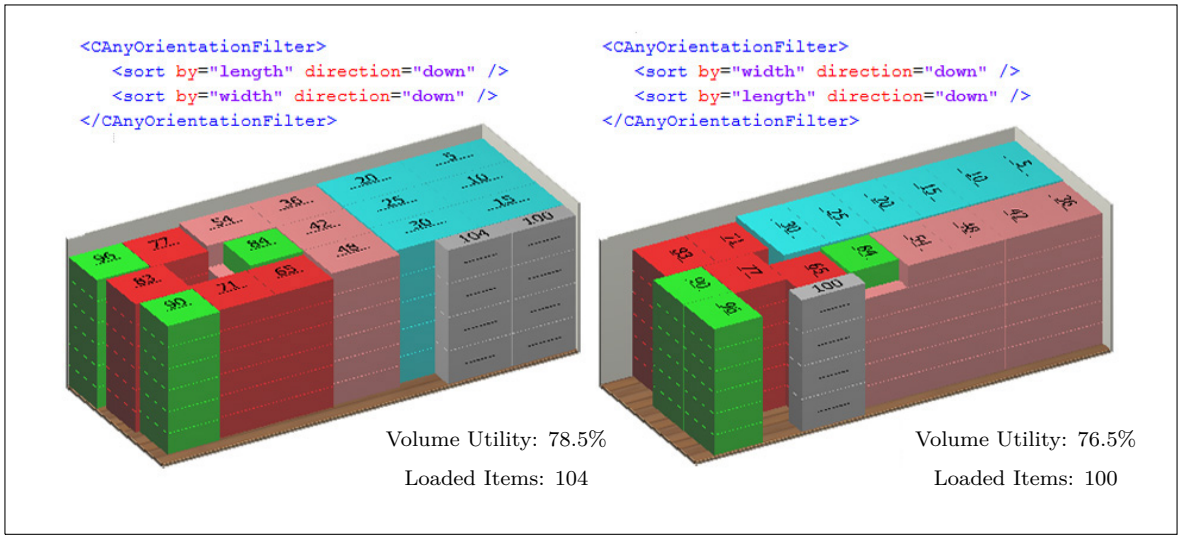

Figure 3.3: Example of effect of parameter settings in the orientation filter. Note that not all items are placed in their preferred orientation (e.g., the gray items in both pictures). The reason for this is that the preferred orientation is infeasible for the selected space, so the underlying algorithm selects a less preferred orientation.

3.1.2 Orientation Filter

The orientation filter considers the orientation of the item in the allocated space. That is, it aligns the dimensions of the item with the dimensions of the container. Table 3.2 reports the different categorical values that can be taken by the first, and any subsequent, sorting parameter in the orientation filter. These parameter values indicate the dimensions of the container. For example, sorting by "length" in direction "down" means that we prefer an orientation in which the item's longest dimension lies along the length of the container. Figure 3.3 shows an example of the effect of parameter settings in the orientation filter.

3.1.3 Load Equipment Filter

The load equipment filter considers which of the available larger containers to place the item in. Once a container turns out to have no sufficient space available for the current item, it guides the solution process to a new container. We can only set the parameter "ConsiderOnce" here, with the possible values "true" and "false". A value of false means that once a container is found insufficient, it is never reconsidered for any future items. A similar parameter, "PersistentForNewOption", can be employed in other filters. This parameter is purely meant to reduce the running time by avoiding unnecessary feasibility checks.

3.1.4 Space Filter

The space filter contains two steps that are both influenced by parameter settings. First, the empty space in the container is divided into multiple smaller spaces. Second, the obtained subspaces are ordered on

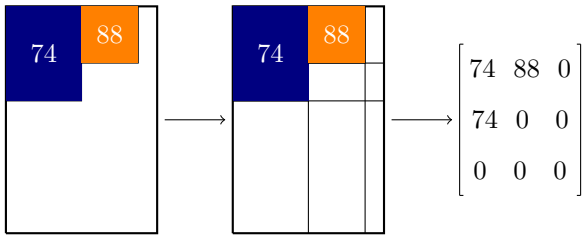

Figure 3.4: Top view of a container with two item placements with its tabulated version and its NGOI representation.

preference according to the settings of the sorting and direction parameters.

To divide the empty space in the container into smaller subspaces, an NGOI matrix is used (Ngoi et al., 1994). Figure 3.4 shows the top view of a container in which two items are placed with a height of 74 and 88 respectively and the tabulated version of that same top view. The abstract NGOI representation of this container is a simple 3×3 matrix. Additional coordinate information is saved to map the NGOI matrix to the actual container.

The NGOI matrix is used to define a list of spaces in which the next item could be placed. First, a list of surfaces is produced according to one of three strategies, governed by the parameter "spaceDividingStrategy" and its three possible values:

- maximumSpaces: for each cell in the NGOI matrix, make maximum cell arrangements such that they cannot be extended in either direction without either exceeding the boundaries of the matrix or including a cell with a strictly higher number. In the example of Figure 3.4, this strategy leads to six unique cell arrangements, as given below in blue. The corresponding surfaces are obtained by mapping these cell arrangements to surfaces in the container at a height of the largest number in the arrangement.

$$\begin{bmatrix} 74 & 88 & \color{blue}{0} \\ 74 & 0 & \color{blue}{0} \\ 0 & 0 & \color{blue}{0} \end{bmatrix} \begin{bmatrix} 74 & 88 & 0 \\ 74 & \color{blue}{0} & \color{blue}{0} \\ 0 & \color{blue}{0} & \color{blue}{0} \end{bmatrix} \begin{bmatrix} 74 & 88 & 0 \\ 74 & 0 & 0 \\ \color{blue}{0} & \color{blue}{0} & \color{blue}{0} \end{bmatrix} \begin{bmatrix} \color{blue}{74} & 88 & 0 \\ \color{blue}{74} & 0 & 0 \\ \color{blue}{0} & 0 & 0 \end{bmatrix} \begin{bmatrix} 74 & 88 & 0 \\ \color{blue}{74} & \color{blue}{0} & 0 \\ \color{blue}{0} & \color{blue}{0} & 0 \end{bmatrix} \begin{bmatrix} \color{blue}{74} & \color{blue}{88} & \color{blue}{0} \\ \color{blue}{74} & \color{blue}{0} & \color{blue}{0} \\ \color{blue}{0} & \color{blue}{0} & \color{blue}{0} \end{bmatrix}$$

- supportedMaximumSpaces: for each cell in the NGOI matrix, make maximum cell arrangements such that they cannot be extended in either direction without either exceeding the boundaries of the matrix or including a cell with a strictly higher or a strictly lower number. This strategy leads to five, unique cell arrangements, as given below in blue. Note that all of these surfaces are fully supported from below.

$$\begin{bmatrix} 74 & 88 & \color{blue}{0} \\ 74 & 0 & \color{blue}{0} \\ 0 & 0 & \color{blue}{0} \end{bmatrix} \begin{bmatrix} 74 & 88 & 0 \\ 74 & \color{blue}{0} & \color{blue}{0} \\ 0 & \color{blue}{0} & \color{blue}{0} \end{bmatrix} \begin{bmatrix} 74 & 88 & 0 \\ 74 & 0 & 0 \\ \color{blue}{0} & \color{blue}{0} & \color{blue}{0} \end{bmatrix} \begin{bmatrix} \color{blue}{74} & 88 & 0 \\ \color{blue}{74} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 74 & \color{blue}{88} & 0 \\ 74 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- topSpace: return one arrangement with the highest number in the matrix including all the cells. This strategy leads to this single cell arrangement:

$$\begin{bmatrix} \color{blue}{74} & \color{blue}{88} & \color{blue}{0} \\ \color{blue}{74} & \color{blue}{0} & \color{blue}{0} \\ \color{blue}{0} & \color{blue}{0} & \color{blue}{0} \end{bmatrix}$$

Afterwards, the obtained surfaces are transformed into three-dimensional, cuboid spaces. The spaces are

Value	Description
minX	Smallest X-coordinate in the space
maxX	Largest X-coordinate in the space
minY	Smallest Y-coordinate in the space
maxY	Largest Y-coordinate in the space
minZ	Smallest Z-coordinate in the space
maxZ	Largest Z-coordinate in the space
volume	Volume of the space
length	Size of the space in the length-dimension of the container
width	Size of the space in the width-dimension of the container
height	Size of the space in the height-dimension of the container
XYarea	XY-surface of the space
XZarea	XZ-surface of the space
YZarea	YZ-surface of the space

Table 3.3: Possible parameter settings for the sorting parameters in the space filter

obtained according to one of two strategies, governed by the parameter "spaceAddingStrategy" and its possible values:

- max: extend all surfaces to the height of the container, creating one cuboid space for every surface.
- all: extend all surfaces to the height of the container, *and* to the height of the surrounding, already loaded items, potentially creating multiple cuboid spaces for every surface.

Subsequently, the filter sorts the obtained spaces according to its sorting and direction parameters. Table 3.3 reports the different categorical values that can be taken by the first, and any subsequent, sorting parameter in the space filter. The first six values consider the position of the space within the container, the others describe the shape of the space. Figure 3.5 shows an example of the effect of parameter settings in the space filter.

3.1.5 Position Filter

In the position filter, there are no tunable parameters. By default, the items are placed as closely as possible to the defined origin of the container, given the selected subspace.

3.1.6 Arrangement Filter

The arrangement filter combines the items into blocks of multiple items. Subsequently the greedy algorithm treats the arrangement as an individual item, and attempt to place the entire arrangement at once. In this filter, there are three numerical parameters, the restricting parameters, that limit the maximum number of items an arrangement can contain along any dimension of the container: "maximumInLength", "maximumInWidth", and "maximumInHeight". These parameters are integers and are bounded by the size of the problem instance (i.e., the number of identical items in the problem instance, and the number of identical items that fit within the dimensions of the container).

Second, several sorting criteria are used to establish a preference list of possible arrangements. The idea of arrangements is that it allows placing a collection of items at once without any internal volume utility losses. We always attempt to maximize the arrangement under the given upper bounds. The sorting procedure merely guides the direction in which we attempt to maximize first. Table 3.4 reports the different categorical values that can be taken by the first, and any subsequent, sorting parameter in the arrangement filter, and Figure 3.6 shows an example of the effect of parameter settings in the arrangement filter.

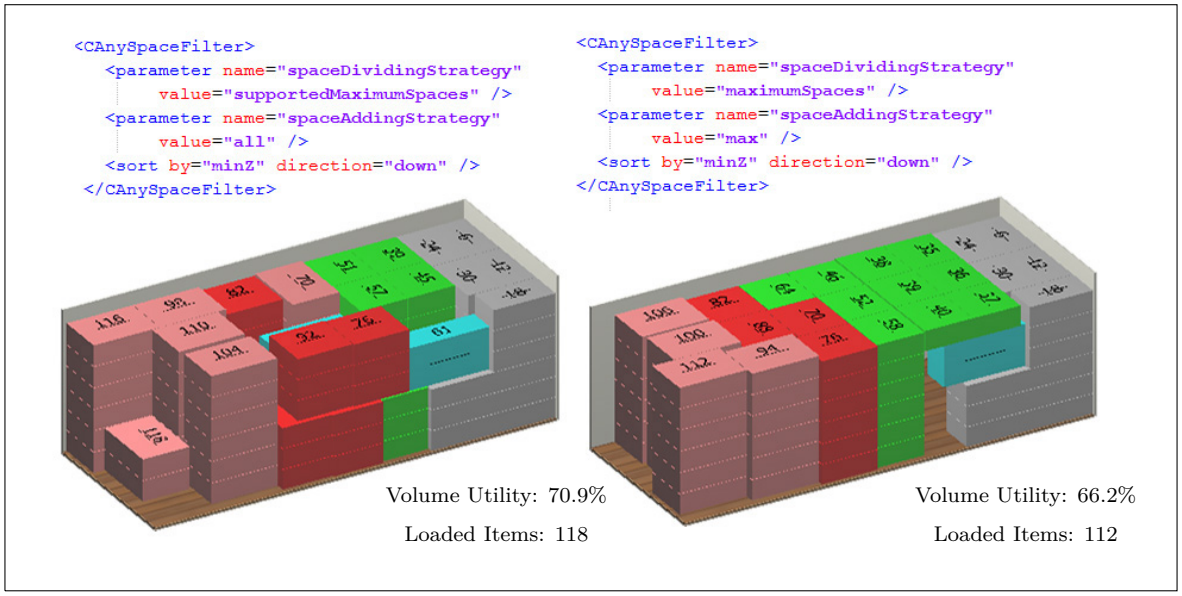

Figure 3.5: Example of effect of parameter settings in the space filter. Note that in the picture on the left, all items are fully supported from below, which is ensured by the parameter setting "supportedMaximumSpaces". In the picture on the right, full support is not required. This is essentially a less restricted case, but, as shown in this example, does not necessarily lead to a better solution. Moreover, note that the use of one sorting parameter "minZ" is insufficient to produce fully deterministic preference lists (i.e., multiple, non-identical spaces can have the same minZ-coordinate). The underlying algorithm selects an option at random in case preferences are not fully specified.

Value	Description
lengthCount	Number of items along length of the container
widthCount	Number of items along width of the container
heightCount	Number of items along height of the container
lengthxwidthCount	Number of items in the XY-surface
lengthxheightCount	Number of items in the XZ-surface
widthxheightCount	Number of items in the YZ-surface
count	Total number of items in the arrangement

Table 3.4: Possible parameter settings for the sorting parameters in the arrangement filter

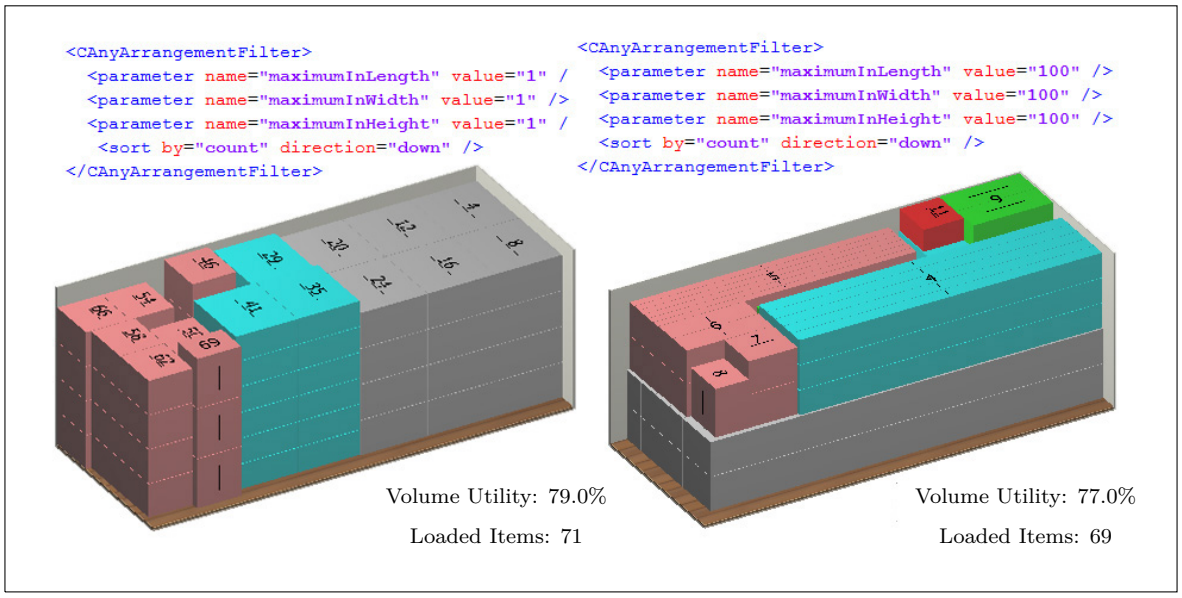

Figure 3.6: Example of effect of parameter settings in the arrangement filter. Note that in the picture on the right, in which the composition of arrangements is not restricted in any dimension, items are placed in large arrangements without any internal volume utility losses. Still, as shown in this example, this does not necessarily lead to better solutions.

Value	Description					
1	Equipment	Item	Orientation	Space	Position	Arrangement
2	Equipment	Item	Space	Orientation	Position	Arrangement
3	Equipment	Space	Item	Orientation	Position	Arrangement
4	Item	Equipment	Orientation	Space	Position	Arrangement
5	Item	Equipment	Space	Orientation	Position	Arrangement
6	Item	Orientation	Equipment	Space	Position	Arrangement

Table 3.5: Possible filter orders.

3.1.7 Order of Filters

Recall that the order in which the filters are presented to the algorithm influences the order in which specific item placements will be explored in case the most preferred item placement is infeasible, which frequently occurs. That is, in case the most preferred item placement is infeasible, the algorithm starts exploring item placements with less preferred subdecisions per filter, starting at the last filter. Therefore the order of the filters influences the solution process. In total there are six filter orders that are allowed in the current algorithmic framework. In effect, the possible orders can be represented by one additional parameter than can take any of the 6 categorial values, describing the possible orders, as shown in Table 3.5. Ultimately, the preferences of the first filter are prioritized, but, as Section 3.3 reveals, it is the collaborative alignment of the parameter settings in all filters that results in good performance.

3.2 Size of Search Space

Now that we have discussed all possible parameter settings in the individual filters, it is useful to determine the size of our parameter search space. The main objective of this section is not to determine the size of our parameter space exactly, but to get an idea about the approximate size, as this may justify the choice to disregard complete enumeration, and aid to develop the most suitable tuning algorithm.

	Possible Settings	Unique Settings (LB)
Item Filter	$5{\times}10^{13}$	26
Orientation Filter	48	6
Load Equipment Filter	2	2
Space Filter	$3{\times}10^{14}$	156
Position Filter	1	1
Arrangement Filter	$6{\times}10^{11}$	112
Total	$6{\times}10^{39}$	$3{\times}10^{7}$

Table 3.6: Total search space of parameter settings in the greedy part. For example, in the space filter, we have $13 \times 2 = 26$ for the its first sorting parameter, which can be freely combined with $3 \times 2 = 6$ different settings for the non-sorting parameters, giving a total of $26 \times 6 = 156$ unique settings.

For each individual filter, we indicated all possible parameter settings. Note that there is no theoretical limit on the amount of sorting parameters we can include in every filter. In practice, however, it is only useful to include additional sorting parameters if there is a possibility that they break any ties in previous sorting. For this reason, including more than one sorting parameter with the same value is always superfluous, which imposes a practical finite limit to our search space. Allowing the inclusion of as many sorting parameters as there are possible settings results in the numbers of possible settings in Table 3.6. For the item filter, for example, we have 13 possible values for every sorting parameter, and the sets of possible sorters can be presented in any order (13!) and in any direction (2^{13}). This gives $13! \times 2^{13} \approx 5 \times 10^{13}$ possible combinations. Since all the settings in the individual filters can be freely combined with one another, and on top of that there are 6 possible orders of the filters, we obtain the total number of possibilities by taking the product of the possibilities within the filters and 6. We obtain a total of $6{\times}10^{39}$ possible parameter settings. Given that a measurement of one configuration on one instance takes roughly three seconds, complete enumeration of all possible configurations would take billions of years.

However, many of these settings are identical to one another in practice. For example, in the item filter all settings that start with the sorting parameters {length, height, width, priority} are identical to one another for every problem instance. That is, if two items are different from one another, they have to be different in at least one of these characteristics, which means they cannot be tied in preference under those sorting parameters, and any subsequent tie-break sorting procedure is redundant. Moreover, for instances in which all items are uniquely dimensioned, any tie-break sorting parameter are redundant and we only have $13 \times 2 = 26$ unique settings. This suggests we may be able to radically reduce the search space by a priori disregarding any non-unique configurations. Determining exactly how many unique settings there are is not trivial. Moreover, the exact number depends on the problem instance. We rather provide a rough lower bound of the number of unique settings, as proving a lower bound is too high would be sufficient to rule out complete enumeration or any tuning algorithm that requires a full factorial design. For each filter, we construct our lower bound by only allowing the tuning of the first sorting parameter and corresponding direction. This gives preference lists that are not completely specified, in which case the greedy algorithm will break ties at random. Table 3.6 gives the corresponding possibilities. Note that this lower bound is very loose: the actual number of unique settings is *much* higher. Still, we are left with $3{\times}10^{7}$ possibilities which accounts to years of computation time to test all configurations on just a single problem instance. Evidently, this gives us sufficient basis to justify our choice to disregard complete enumeration as well as any tuning algorithm that requires a full factorial design.

$$\text{LHD} = \begin{array}{c} \begin{matrix} Y_1 & Y_2 & Y_3 & Y_4 \end{matrix} \\ \begin{bmatrix} 1.6 & 7.9 & 4.8 & 1.4 \\ 4.2 & 2.1 & 3.3 & 2.9 \\ 3.9 & 0.3 & 1.3 & 0.3 \\ 7.3 & 9.7 & 0.1 & 3.9 \\ 9.3 & 5.5 & 2.7 & 4.1 \end{bmatrix} \end{array}$$

Figure 3.7: Example of a matrix representation of an LHD with sample size 5 and a parameter space of 4 continuous parameters: $Y_1, Y_2 \in [0, 10]$ and $Y_3, Y_4 \in [0, 5]$

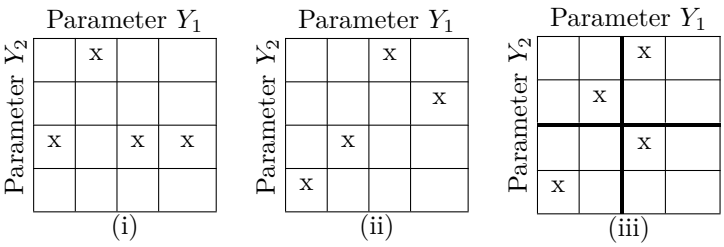

Figure 3.8: Example of drawing a sample of size 4 from a two-dimensional parameter space resulting from (i) random sampling, (ii) LHD, and (iii) orthogonal LHD. In an LHD, all columns and rows are equally represented in the sample. In an orthogonal LHD, all multi-dimensional subspaces (in this case the four two-dimensional quadrants) are also equally represented in the sample.

3.3 Performance Landscape

In this section, we conduct experiments on our parameter space in order to reveal the corresponding performance landscape, and to draw conclusions that help to develop an applicable tuning algorithm. We do not perform any tuning yet, but solely draw one sample of parameter configurations (which is representative for the entire parameter space), and subsequently analyze their performance. Section 3.3.1 introduces the method we employ to generate such a sample, and Section 3.3.2 provides an analysis of the performance of this sample, discussing the main effects of our parameters, their interdependencies, and the robustness of parameter configurations over individual problem instances.

3.3.1 Latin Hypercube Design

As said, we need to draw a limited, yet representative sample, uniformly covering the entire parameter space, as a basis for our experiments. Latin Hypercube Design (LHD), first proposed by McKay et al. (1979), ensures such a uniform coverage. LHD selects n values from q parameters by dividing the range of each parameter into n non-overlapping, equally spaced intervals. Subsequently, n random values are drawn for each parameter, one from every interval, and randomly combined with the values of other parameters to obtain n k-dimensional input vectors. The LHD with these n samples can be represented by an $n \times k$ matrix in which the n^{th} row contains the values for all k parameters to be used in the n^{th} sample. The matrix in Figure 3.7 shows an example of such an LHD. This procedure ensures that samples are drawn evenly in each parameter's single dimension. Orthogonal LHD adds the requirement that every (non-overlapping) multi-dimensional subspace is sampled evenly (Ai et al., 2012). Figure 3.8 shows an example of samples resulting from random sampling, LHD, and orthogonal LHD respectively in case we have two parameters.

Note that LHD is intended for numerical, preferably continuous parameters. Indeed, dividing a range

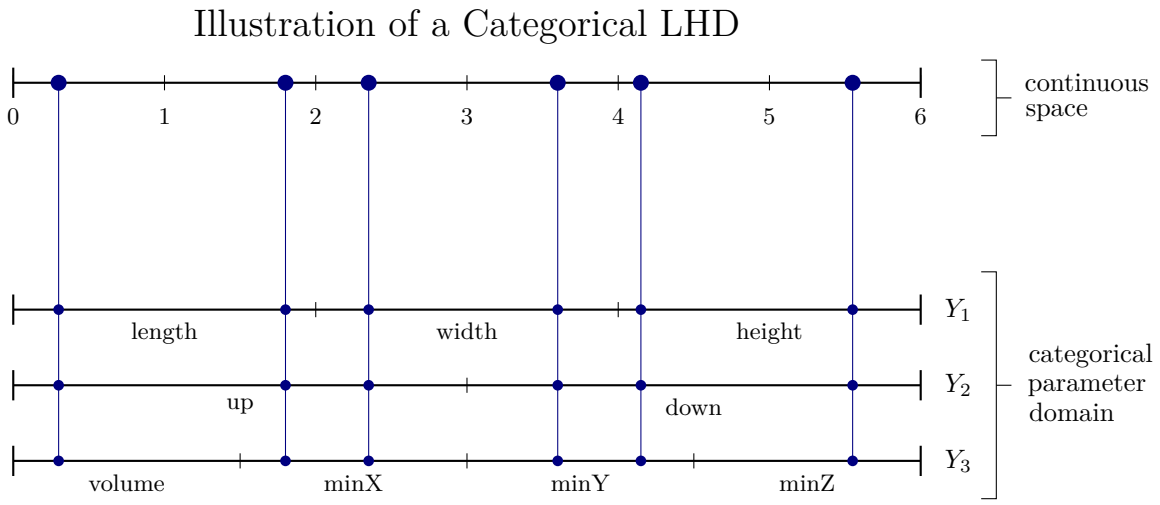

Figure 3.9: Illustration of the construction of a categorical LHD. To sample n configurations, we use a fictional continuous space of length n. Subsequently, we draw a random value from each integer interval on the continuous space, and map it to each of our categorical parameter spaces Y_1, Y_2, and Y_3.

with categorical values in equally spaced intervals is not trivial. Moreover, LHD requires every parameter range to be divided into the same number of intervals, which is problematic in case categorical parameters have different amounts of possible values they can take. We propose an adaptation of the traditional LHD to allow application to categorical parameter spaces.

Categorical LHD

To adapt the traditional LHD to be applied to categorical parameters, we propose a straightforward mapping structure. That is, instead of uniformly drawing a sample of size n directly from the k categorical parameter spaces, we use a single, continuous, one-dimensional space $[0, n]$. From this space, we randomly draw n values, one from each integer interval. Subsequently, we map the drawn values to each of the k categorical parameter spaces. Figure 3.9 illustrates this process for a situation where we a sample of $n = 6$ configurations from a parameter space with $k = 3$ categorical parameters (Y_1, Y_2, and Y_3). Note that, in this example, parameter Y_3 is not uniformly represented in the sample (the values "minX" and "minZ" occur twice). If we require uniform representation of every parameter, we are somewhat limited in the size of our sample (in this example, we would have to draw a multiple of 12 (smallest common denominator) configurations).

Having drawn n (categorical) values for all k parameters, we combine the individual parameter values to form n k-tuplets, the sample of n parameter configurations. We can represent this process with the use of a $n \times k$ permutation matrix A, as in Figure 3.10, where each column contains a random permutation of the integers $\{1, 2, ..., n\}$, and the element A_{nk} indicates that in sample n we use the value of the categorical parameter that originated in the integer interval $[A_{nk} - 1, A_{nk}]$ of the continuous space. To obtain an orthogonal LHD, we may adjust the permutation matrix accordingly. By using this LHD, we obtain a representative sample with uniform coverage of our entire, categorical parameter space: a Latin Hypercube Sample (LHS).

3.3.2 Performance Analysis

This section discusses some preliminary results on the parameter space exploration we conduct using our categorical adaptation of the LHD. We use an LHD to draw a sample of 500 parameter configurations

$$A = \begin{bmatrix} 2 & 5 & 4 \\ 4 & 3 & \mathbf{6} \\ 3 & 2 & 1 \\ 6 & 4 & 5 \\ 1 & 6 & 2 \\ 5 & 1 & 3 \end{bmatrix} \quad \rightarrow \quad \text{LHD} = \begin{bmatrix} Y_1 & Y_2 & Y_3 \\ \text{length} & \text{down} & \text{minY} \\ \text{width} & \text{up} & \mathbf{\text{minZ}} \\ \text{width} & \text{up} & \text{volume} \\ \text{height} & \text{down} & \text{minZ} \\ \text{length} & \text{down} & \text{minX} \\ \text{height} & \text{up} & \text{minX} \end{bmatrix}$$

Figure 3.10: Example of a random permutation matrix A and the resulting LHD. For example, $A_{23} = 6$ indicates that the third parameter (i.e., parameter Y_3) in the 2^{th} sample takes the value "minZ", because the randomly drawn value on the interval $[5, 6]$ (i.e., approximately 5.6) maps to this value on the 3^{rd} parameter space.

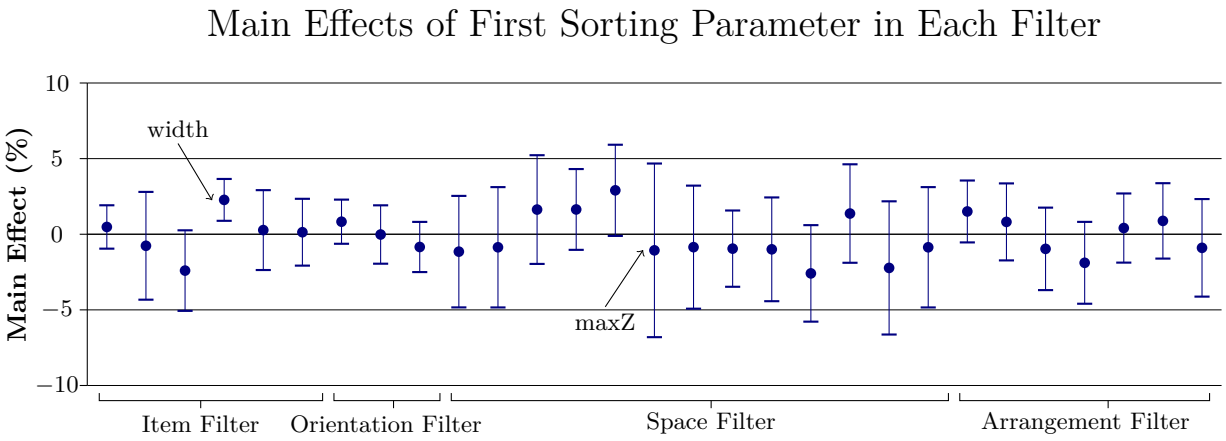

Figure 3.11: Main effects of the first sorting parameter in each filter with a 95% confidence interval. For example, all configurations with the value "maxZ" for its first sorting parameter in the space filter perform slightly worse than all other configurations on average. Aside from the value "width" in the item filter, all main effects are insignificantly different from zero under $\alpha = 0.05$.

from our parameter space, which is just a tiny, yet representative fraction of our parameter space, and measure their performance on a set of 80 BR instances (resulting in $500 \times 80 = 40.000$ measurements): the first 10 instances from the sets BR0-BR7 (see Section 2.1.3). Moreover, we draw an additional sample of 50 configurations, and apply them to a set of 8 practical pick-pallet instances from a large multinational.

Based on the performance of the configurations in these samples, we draw some preliminary conclusions on the performance landscape of our parameter space that will aid to develop an efficient tuning algorithm. Unless explicitly mentioned, the results below hold for both the academic and the practical problem set.

Main Effects of Parameters

First, we inquire the main effects of individual parameter settings, as this may cause a priori exclusion of, or bias towards, certain values. We define the main effect of a parameter setting as the percentual difference between the average performance (in this case volume utility) of all configurations with that particular setting compared with all other configurations. Figure 3.11 and Figure 3.12 show the resulting main effects of the most important parameters, along with their 95% confidence intervals. If the interval of a certain parameter setting includes zero, its main effect is insignificant under $\alpha = 0.05$.

The analysis shows little proof for dominance within our parameter domain. Indeed, almost all settings in our parameter space have main effects that are insignificantly different from zero on the

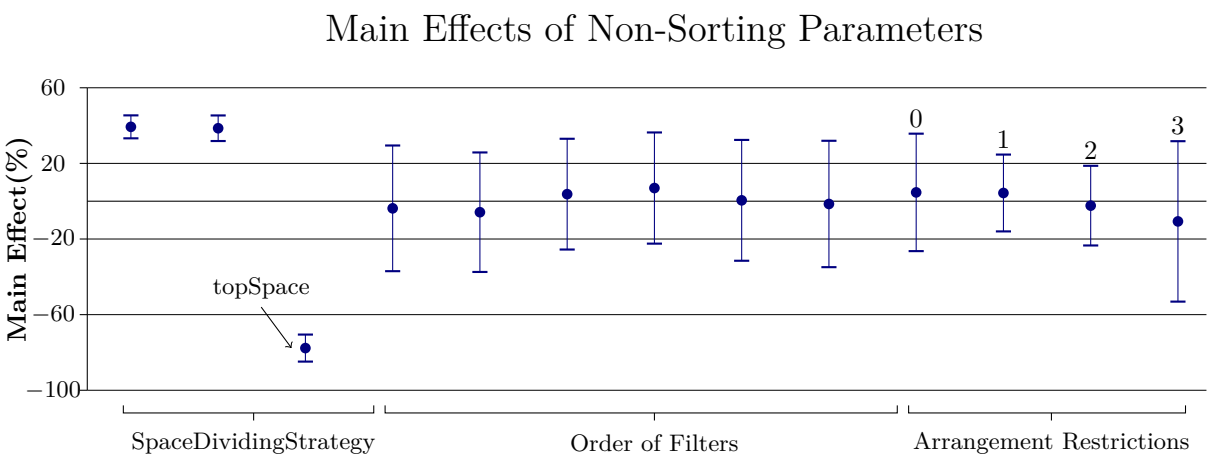

Figure 3.12: Main effects of the parameter "spaceDividingStrategy", the order of the filters, and the number of restrictions in the arrangement filter with a 95% confidence interval. The arrangement restrictions indicate how many of the parameters "maximumInLength", "maximumInWidth", and "maximumInHeight" take the value "1" (rather than "100", which represents unrestricted arrangements), preventing the construction of arrangements in that direction.

training set of 80 instances. Moreover, an analysis of the best configurations in our LHS reveals that all parameter settings can potentially lead to good performance, if they reside in a complementary configuration. The one notable exception is the parameter "spaceDividingStrategy", which suggests that the value "topSpace" is dominated by its two alternatives. Indeed, configurations with "topSpace" are almost exclusively worse than those with alternative settings, and none of the best 20% configurations in our LHS take the "topSpace" setting. In other words, we feel safe to exclude the value "topSpace" from any further tuning effort. Other than "topSpace", however, we cannot a priori eliminate any parameter values.

Performance Distribution: Academic Problem Set

Although the main effects of the parameters individually prove to be insignificant, Figure 3.13 shows that the effect of the entire parameter configuration on the performance of the algorithmic framework is enormous. Indeed, although the main effects of individual parameter settings are insignificant, it is the collaborative alignment of all parameters that results in very diverse performance.

Typically, the most important KPI in a BR instance is volume utility, the percentage of occupied volume of the container, which we aim to maximize. Figure 3.13 shows the distribution of the average performances of our sample on the 80 BR instances. Evidently, the figure indicates that, judging from the widely ranged performance distribution, the selected parameter configuration strongly impacts the performance of the underlying algorithm, justifying the relevance of our research. Moreover, Figure 3.13 suggests that the average performance is not normally distributed (nor are the performance distributions of individual configurations over the problem space), which is an important finding with regard to any statistical tests we may want to include in our tuning algorithm.

Performance Distribution: Practical Problem Set

Similarly as for the BR instances, the selection of the parameter configuration proves to be of great influence to the performance of the algorithmic framework on our practical problem set. The practical pick-pallet instances we include in our research are input minimization rather than output maximization problems. In other words, all of the items must be loaded, and the objective is to minimize some

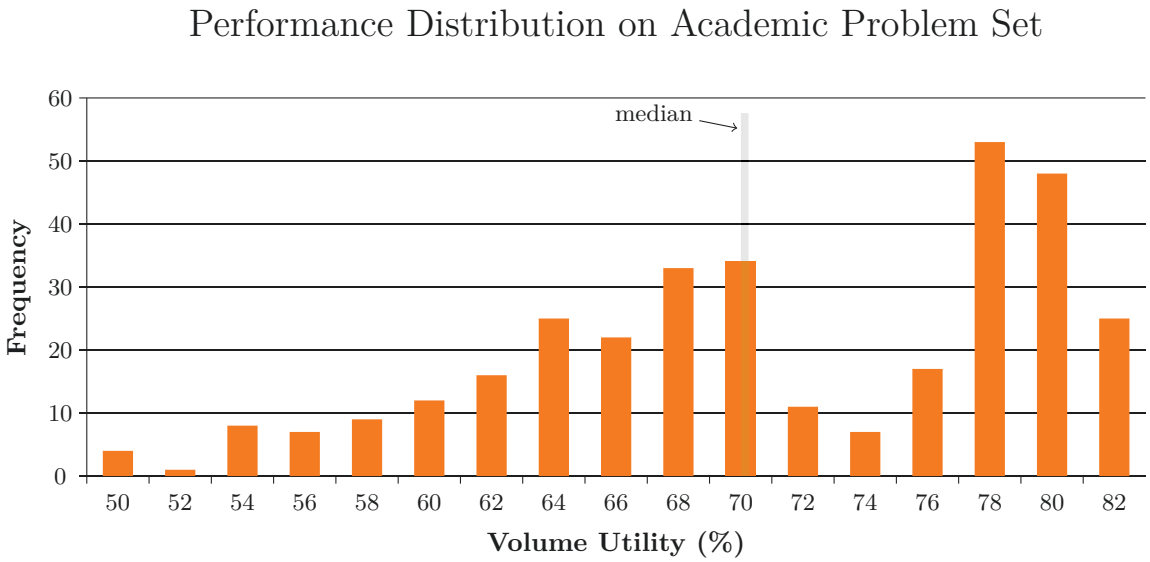

Figure 3.13: Performance distribution of our LHS ("topSpace" excluded). The performances are given as the average volume utility over all 80 problem instances and the frequency indicates how many configurations in our LHS of size 334 have that performance (because of our LHD, one third of the original 500 configurations takes the value "topSpace", and is therefore excluded).

indicator of the required space. In these instances, the most important KPI is the number of pallets needed to palletize all items. Figure 3.14 shows a similar, yet mirrored (as we have a minimization problem now), performance distribution as for the BR instances. In this case, in our LHS of size 50, the setting "topSpace" is a priori excluded.

Robustness

Aside from average performances, we consider the variance of the performances over the different problem instances, the robustness of the parameter configuration. The reason for this is two-fold. First, it is crucial for the credibility of ORTEC's software solutions that negative outliers are prevented. Second, the degree of robustness of the configurations largely determines the suitability of either problem instance or problem family tuning. Indeed, if the parameter configurations turn out to be highly robust over the problem space, potential quality losses that result from problem family tuning are likely to be small. Figure 3.15 shows the relation between the average performance of the configurations in our LHS on the BR instances and their robustness, which is expressed as the variance of the performance over the problem instances, where a low variance indicates a high robustness. The graph shows a strong negative correlation (Pearson coefficient of -0.87 p-value = 0.000) between the absolute value of the variance and the average performance, indicating that those parameter configurations that perform well on average also generalize better to unseen instances. In other words, while optimizing the average performance of a parameter configuration, we simultaneously optimize its robustness. This is a crucial result for our research, and it advocates the use of problem family tuning.

To further strengthen the notion that problem family tuning is suitable for our research, Table 3.7 shows the Pearson Correlation Coefficients of the performances of parameter configurations over the BR instances. We see strong positive correlations throughout our entire problem set, which indicates a linear dependance of the performances of the configurations. In other words, when a configuration performs well on one particular type of instance, it generally performs well on other instances too, again suggesting a high degree of robustness of parameter configurations over our problem space. Note that the

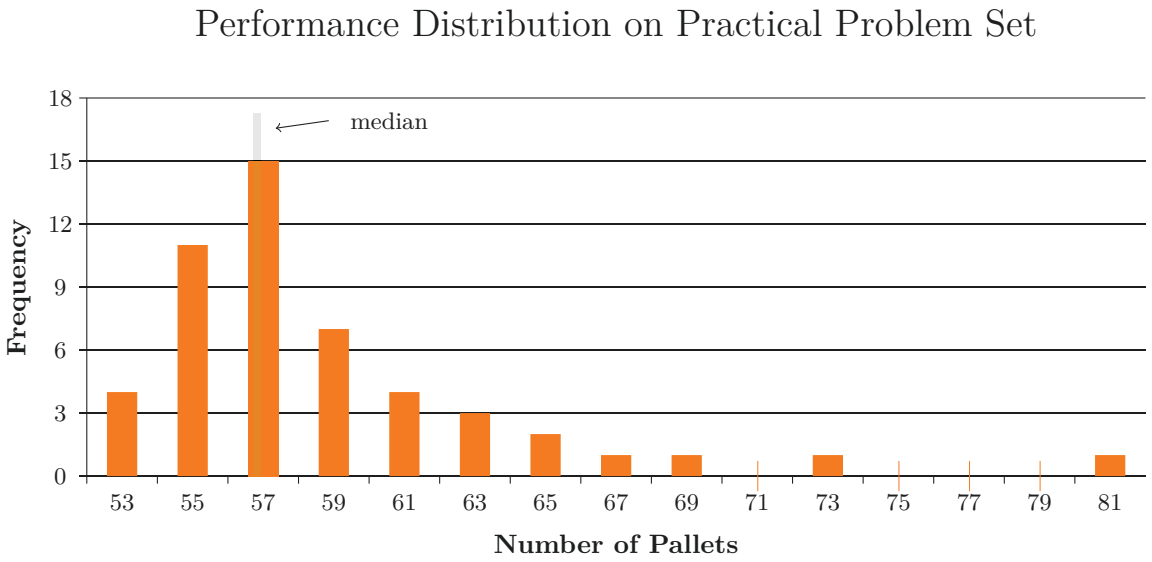

Figure 3.14: Performance distribution of our LHS of 50 configurations on 8 practical pick-pallet instances. The performances are given as the average number of pallets utilized over all 8 problem instances and the frequency indicates how many configurations in our sample of 50 have that performance.

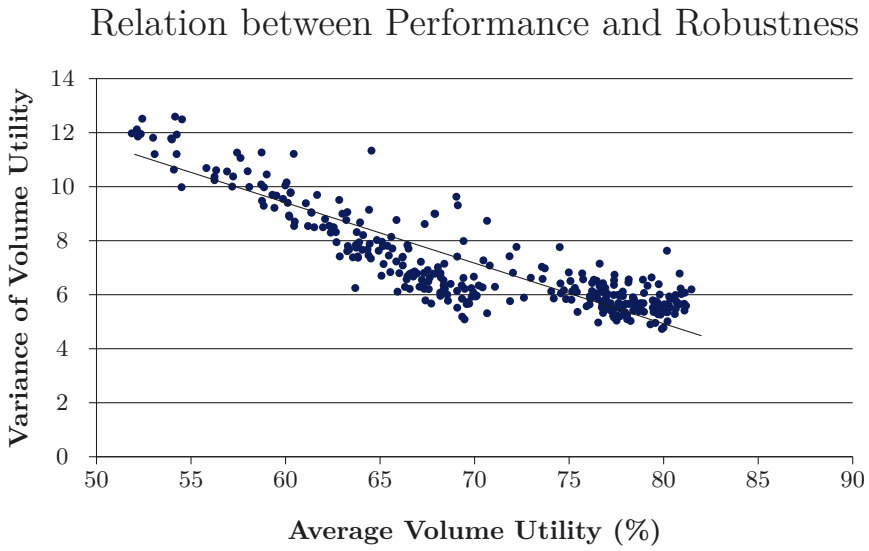

Figure 3.15: Relation between the average performance of parameter configurations and their robustness across the parameter space with a linear trendline.

	BR0	BR1	BR2	BR3	BR4	BR5	BR6	BR7
BR0	1.00							
BR1	0.92	1.00						
BR2	0.88	0.99	1.00					
BR3	0.86	0.98	0.99	1.00				
BR4	0.85	0.98	0.99	1.00	1.00			
BR5	0.84	0.97	0.99	1.00	1.00	1.00		
BR6	0.83	0.97	0.99	1.00	1.00	1.00	1.00	
BR7	0.82	0.96	0.98	0.99	0.99	1.00	1.00	1.00

Table 3.7: Pearson Correlation Coefficients between the average performance on various BR problem families. All corresponding p-values are 0.001 or smaller.

	BR0	BR1	BR2	BR3	BR4	BR5	BR6	BR7
Best Configuration on Average	0.717	0.819	0.831	0.842	0.837	0.834	0.816	0.822
Best Configuration on Specific Set	0.754	0.833	0.850	0.842	0.842	0.834	0.821	0.822
Loss of Quality	-4.9%	-1.7%	-2.2%	0%	-0.6%	0%	-0.6%	0%

Table 3.8: Performance of the best configuration on average over the entire training set versus the highest performances on every individual instance set, and the relatively small, percentual quality losses (i.e., difference in performance between the best configuration on average and the best configurations on the subsets) that result from tuning on a larger training set.

correlations are stronger for closely related problem families, suggesting the intuitively sensible notion that the more representative the training set is, the better the generalization of your configuration to unseen target instances becomes. Indeed, BR6 and BR7 instances share more commonalities with each other than BR0 and BR7 instances, which is represented by a stronger correlation between these families.

The result of such high robustness is that a good average performance corresponds to good performances on individual problem instances. Indeed, Table 3.8 shows that the best configuration on average over the entire problem set performs admirably on every individual set of BR instances (i.e., compared to the best configuration on that specific set).

Statistical Inferiority

The fact that the parameter configurations show such high robustness over the problem instances introduces the idea that reliable conclusions can be drawn about the true average performance of configurations on the complete training set based on just a few observations. As discussed in Section 2.2.3, this notion is exploited by Race algorithms, and has the potential to severely speed up the tuning process, as statistically inferior configurations can be eliminated at an early stage, wasting no more computation time. Previous findings regarding the configuration's performance distributions have revealed that we cannot rely on the usual assumptions of normal distributions with equal variances. The signed-rank test, proposed by Wilcoxon (1945), is a non-parametric alternative for the pairwise Student t-test to test for statistical inferiority in paired data without the need for such assumptions. Figure 3.16 shows (in orange) the effect of applying this test on our initial LHS, which is just a tiny fraction of our parameter space, for different levels of significance α. Similarly as in Figure 2.4, the horizontal bars represent the candidate configurations that are still in the race. In other words, the bars represent the number of instances that each configuration has gone through before being deemed statistically inferior under the given α. The profit of such statistical removal with regard to exhaustive evaluation (i.e., $\alpha = 0$) is the combined surface of the horizontal bars, relative to the complete rectangular surface of the graph. Table 3.9 shows, for each α, the fraction of the problem set that would actually be evaluated. Clearly, larger values of α result in more rapid removal and therefore more benefits regarding computation time at the expense of

	Fraction of Computation Time		Best Remaining Performance	
	Paired	Unpaired	Paired	Unpaired
$\alpha = 0.00$	1.000	+0%	0.815	+0%
$\alpha = 0.01$	0.254	+7%	0.815	+0%
$\alpha = 0.05$	0.195	+33%	0.815	+0%
$\alpha = 0.10$	0.169	+54%	0.815	+0%
$\alpha = 0.30$	0.143	+18%	0.812	-1.2%

Table 3.9: Results of retrospective statistical tests on the performance of our sample. The measures for the unpaired tests are expressed relative to the paired tests. For example, for $\alpha = 0.30$, the best remaining performance for the unpaired tests (0.802) is 1.2% worse than for the paired tests (0.812), while requiring 18% more computation time.

a higher probability of excluding the true best configuration. Since in this case, we *have* exhaustively evaluated all configurations on the complete problem set, we can indicate, retrospectively, the effect that the severity of the statistical test has on the best remaining configuration. The average performance on the complete problem set of the best remaining configuration relative to the true best configuration is also tabulated in Table 3.9. Note that for $\alpha \leq 0.10$, we require substantially less computation time than for exhaustive evaluation and still find the true best configuration. Using an α of 0.30 reduces the required computation time further, and still gives a configuration that performs comparably well with the true best configuration.

If we do not have observations on the same problem instances for every configuration, we cannot perform pairwise comparisons, and we have to employ unpaired tests instead. Figure 3.16 also shows (in blue) the results of the unpaired variant of the Wilcoxon signed-rank test, the sum-rank test (Wilcoxon, 1945), on the same, yet randomly ordered, sample data. The corresponding results regarding computation time and best remaining configuration are tabulated in Table 3.9. Note that this test is considerably less powerful than the paired test, as it needs more measurements to come to the same conclusions. On average, the unpaired test needs 28% more measurements (on top of the minimum of 10), and therefore 28% more computation time, to find the same (or worse) configurations.

3.4 Conclusion

This chapter served four purposes. First, we exhaustively described the possible parameter settings and their general effect on the solution process, giving practical meaning to the symbolic values. Second, we gave an approximate idea about the size of our search space, justifying our choice to disregard complete enumeration, and excluding any tuning algorithm that requires a full factorial design. Third, our performance analysis showed a high degree of synergy in our parameter space, in the sense that it is the collaborative alignment of all the parameters that results in good performance. Finally, we showed that the parameter configurations possess a high robustness over the instances in our problem space, such that the quality of a configuration with a high mean performance on a training set of problem instances generalizes well to individual, similar problem instances. This induced our choice for a problem family tuning method, as this substantially reduces the computation time at the customer site without substantial losses in solution quality.

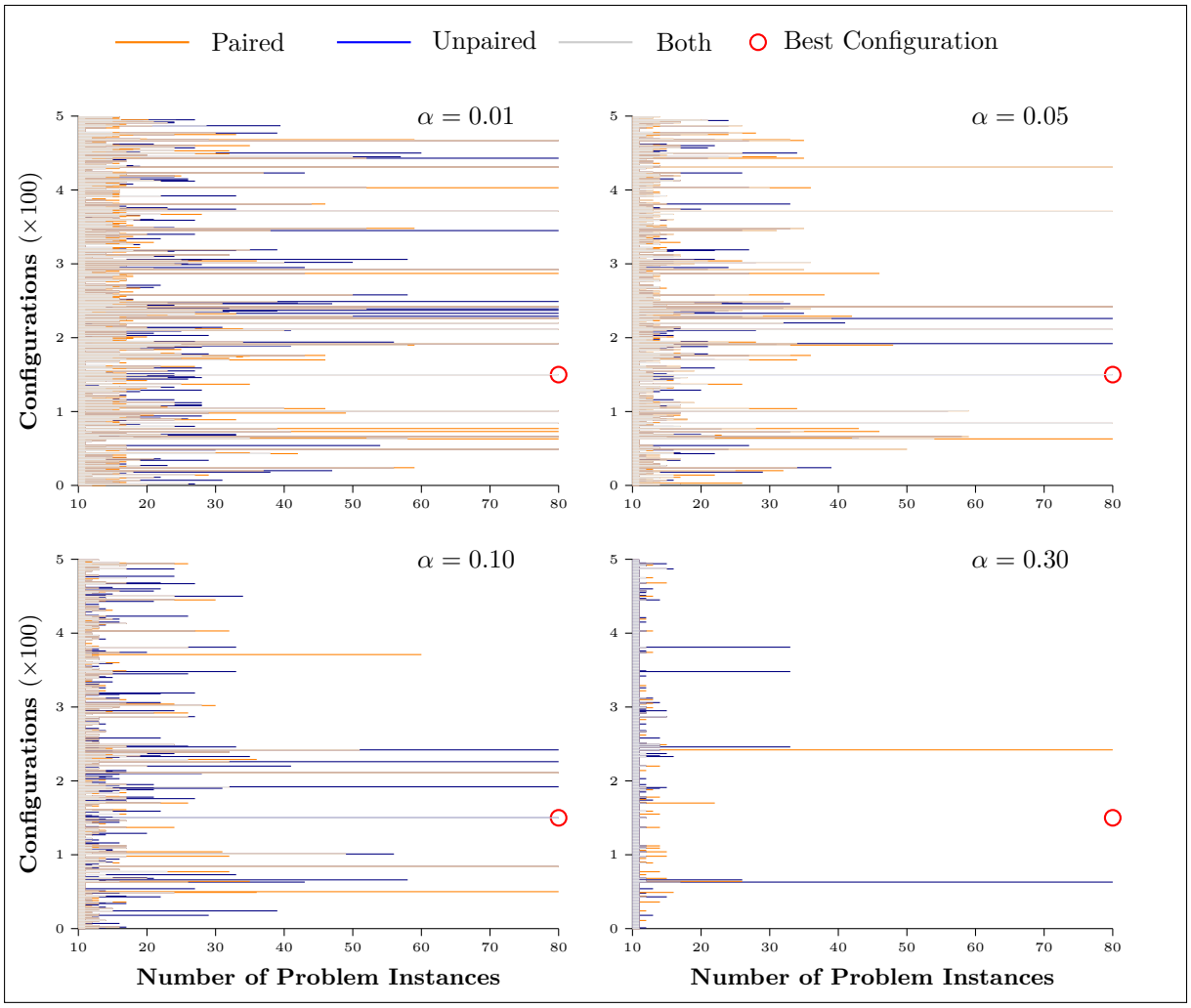

Figure 3.16: Paired and unpaired statistical inferiority tests on our LHS. The 500 parameter configurations are on the y-axes. The bars represent the number of instances that each configuration has gone through before being deemed statistically inferior under the given level of significance α. For example, for $\alpha = 0.30$, we are left with only two parameter configurations after 80 problem instances; all other configurations are eliminated. For larger values of α, less evidence is required to eliminate configurations. Note that for $\alpha = 0.30$, we eliminate the true best configuration based on relatively poor performance on the first few instances. The configuration that remains for the paired and unpaired tests is the second-best and the third-best configuration of the entire LHS respectively. For unpaired tests, the bars collectively cover a larger surface, which means more computation time is required (see Table 3.9). For example, for $\alpha = 0.10$, we are left with 7 parameter configurations after 80 problem instances, compared with the 4 configurations for the paired test.

4 | Design of a Tuning Algorithm

In this chapter, we propose a solution method. We rely on the literature review in Chapter 2, and exploit the knowledge about our parameter space that we obtained in Chapter 3 to develop a tuning algorithm that is most suitable for the task of tuning the parameters in OPLB's algorithmic framework. In Section 4.1, we designate a general solution approach, and in Section 4.2 we develop our tuning algorithm explicitly.

4.1 General Solution Approach

In this section, we cover two topics that influence the choice for a general solution approach. Section 4.1.1 tackles the interesting role of stochasticity in our research, and Section 4.1.2 covers the implications of our parameter space analysis on the suitability of tuning methods. We conclude in Section 4.1.3.

4.1.1 Stochasticity

The results of our parameter space analysis induced our choice for problem family tuning. This introduces an interesting stochastic element to our research regarding problem instance selection. The algorithm, whose parameters we want to tune, is entirely deterministic: the outcome of applying one greedy construction within OPLB to a loading problem is fully deterministic and reproducible. Therefore, we know the true performance of the algorithm on a specific problem instance after one run, and repeated application of the same parameter configuration on the same instance is superfluous. However, we do not want to optimize the parameter configuration for a single problem instance, but we want to find a "jack-of-all-trades" that performs well on all problem instances of one problem family. The fact that our output needs to perform well on the entire, infinite set of seen and unseen problem instances introduces a degree of stochasticity very similar to the case where we tune the parameters of a stochastic algorithm for one specific problem instance. Let us illustrate this analogy by means of an example from the Optimal Learning literature.

The multi-armed bandit problem (Powell, 2010) is an example of an elementary (online) learning problem. The multi-armed bandit problem is, by the optimal learning definition, an online problem in the sense that the objective is to maximize the cumulative value of all measurements, whereas, as mentioned in Section 2.2.2, we have an offline problem, since we are only concerned with the terminal value of our implementation decision. However, the problem illustrates the analogy nicely. In the multi-armed bandit problem, we have a choice to play on any of M slot machines (often called one-armed bandits). The return on each slot machine is stochastic, which means that we do not know beforehand how much we will win, even when we have played on that particular slot machine before. The question is how to select the most profitable slot machine. Or rather, how to choose an optimal measurement policy to maximize our (cumulative) earnings.

The analogy with our problem may be unapparent at first, because in our case the individual experiments, the slot machines, are not stochastic. As said, however, our stochasticity lies in the fact that we do not know to what specific problem instance the greedy heuristic is applied, once the choice of parameters is fixed. All the problem instances in itself are deterministic slot machines, but our choice of alternatives is not on which slot machine, but in which of M casinos to play. Each casino (a parameter configuration) has the same floorplan with the same infinite amount of deterministic slot machines (the problem instances). However, in each casino, the slot machines are calibrated differently (say, the behavior of the slot machines is influenced by the different magnetic forces in each casino caused by their location on the earth's surface). The challenge is to select the casino that gives the highest expected earnings if we play on one random slot machine. We obtain information about the casinos by taking measurements: repeatedly playing on one of the slot machines in one of the casinos.

In effect, we simulate the customer's actual loading problem (the earnings on a random slot machine, or expected earnings), with a training set of similar problem instances (other slot machines) and attain statistical significance through multiple measurements. If we were to construct our meta-optimizer such that it continuously selects random problem instances for each parameter configuration and disregards any information related to the problem instance used, our problem is identical to the optimization of a stochastic algorithm on one problem instance. However, in our problem, it *is* possible to use this information. Indeed, if we measure the earnings in different casinos on the same slot machines (i.e., different parameter configurations on the same problem instance), we obtain more reliable comparisons. We create a situation in which the differences in earnings are solely caused by differences in the casinos and not by any unpredictable random selection of slot machine. This makes statistical comparisons considerably more powerful. For tuning stochastic algorithms or simulations, such fair comparisons are typically pursued by imposing the same random number streams (Common Random Numbers) for every configuration. Although this may serve its purpose at the start of the tuning process, it often introduces complications with non-synchronous streams (Law and Kelton, 2000). For problem family tuning of deterministic algorithms, however, we have the unique ability to ensure perfectly fair comparisons throughout the entire tuning algorithm by simply measuring the performances of parameter configurations on the same problem instances.

This discussion advocates the intuitive appeal of a batch measurement policy, in which batches of configurations are sequentially tested on the same problem instances, such that we can exploit the enhanced statistical power of fair comparisons. Moreover, this intuitive appeal is supported by the quantitative results in Figure 3.16. A Race algorithm, as discussed in Section 2.2.3, enables the use of a batch measurement, hence the exploitation of enhanced statistical power.

4.1.2 Parameter Space

As indicated in Section 3.2, our parameter search space is enormous. For this reason, we have already dismissed complete enumeration and any sequential measurement based method that requires a full factorial design. Those sequential measurement techniques that *are* capable of dealing with large search spaces rely on generalization and aggregation techniques. We already excluded most generalization techniques for their inapplicability to categorical parameters. Moreover, the high degree of synergy in our parameter domain, as indicated by our parameter space exploration, disregards most aggregation techniques for their blindness towards parameter interaction. Instead we need a method that does not rely on the assumption of a smooth, predictable performance landscape, but is still able to tackle large search spaces. Chapter 2 suggests Biased Sampling Distributions as the most suitable technique to determine which configurations to measure next.

4.1.3 Conclusion

In conclusion, a Race algorithm in its generic form is a good basis for our tuning algorithm. First and foremost, for its ability to exploit the power of fair comparisons and focus on rapid statistical elimination. The parameter space analysis in Chapter 3 further supports the choice for a Race algorithm, as its results induce our choice for problem family tuning, for which Race algorithms are intended, and indicate the potential to substantially limit computation time through statistical tests. Indeed, the retrospective analysis of statistical tests in our experimental sample in Section 3.3.2 indicates that such statistical tests are able to reduce computation time substantially (over 80% reduction), while still finding the configuration with the true best performance. Since a full factorial design over our search space is computationally prohibitive, we need an iterative Race algorithm, where we employ Biased Sampling Distributions to generate new configurations. By evolving, in parallel, a population of configurations, continuously performing fair statistical comparisons, an iterative Race algorithms allows combining the best attributes of population evolution and sequential measurements based methods, and forms a great basis for a tuning algorithm with the purpose of problem family tuning of categorically parameterized, deterministic algorithms.

4.2 Our Tuning Algorithm: uRace

As deduced in Section 4.1, iterative Race algorithms form a good basis for our method. We are left with the freedom to guide the behavior of the iterative Race algorithm in large part by selecting its three main components discussed in Section 2.2.3: initial sampling, statistical testing, and resampling. In this section, we explicitly develop our proposed tuning algorithm. First, we give a brief formal description of the offline algorithm configuration problem (Birattari, 2009), the problem that we intend to solve with our method.

4.2.1 Offline Algorithm Configuration Problem

We have a parameterized algorithm with k parameters, and each parameter $Y_d,\ d = 1, ..., k$ can take any categorical (or numerical) value y_d from a predefined finite set of possibilities: $y_d \in \mathcal{Y}_d,\ \forall d$. A configuration $\theta = \{y_1, y_2, .., y_k\}$ is a unique assignment of values to parameters, and $\Theta \supset \theta$ is the finite set of possible parameter settings (see Section 3.2). We have a training set of problem instances $\mathcal{X}$ that is representative for the problem instances at the customer site, and a weighted probability measure $P_{\mathcal{X}}$ over the set of instances, where $P_{x(i)}$ gives the probability that instance x_i occurs. The function $A(\theta|x_i) : \mathcal{X} \to \mathbb{R}$ is the greedy construction phase of the algorithmic framework within OPLB that gives the solution quality of configuration θ on problem instance x_i, where the fitness function is formulated such that we obtain a maximization problem. The criterion we want to optimize is the expected value of $A(\theta)$ on $\mathcal{X}$. In other words, we want to find the configuration θ^* such that

$$\theta^* = \arg\max_{\theta \in \Theta} \mathbb{E}\left[A(\theta)\right] = \arg\max_{\theta \in \Theta} \sum_{i=1}^{|\mathcal{X}|} P_{x(i)} A(\theta|x_i) \tag{4.1}$$

Note the similarity between the offline configuration problem in Equation 4.1 and the more generically formulated meta-optimization problem in Equation 2.1. In the remainder of this section, we discuss the tuning algorithm through which we aim to solve Equation 4.1.

4.2.2 Structure of Our Tuning Algorithm

The basic structure of our tuning algorithm, is presented in Algorithm 1 and visualized in Figure 4.1. Each iteration j starts with a set of "racing" configurations Θ_r, where the size of Θ_r depends on the (remaining) computation budget and the size of our parameter space $\mathcal{Y}$. First, all configurations in Θ_r are tested for statistical inferiority based on previous observations. If no statistical inferiority is detected, all configurations are measured on a new problem instance x_j. This instance is drawn randomly from the set $\mathcal{X}$ according to the probability measure $P_{\mathcal{X}}$. Any inferior configuration is eliminated, and replaced by a newly generated child configuration θ^{child}. As said, this newly generated configuration is measured on the same problem instances as its parent, as long as it is not found to be statistically inferior. Once we have found sufficient non-inferior child configurations such that the set of racing configurations Θ_r is back to its designated size, we continue with a new iteration.

The tuning algorithm requires as input a set of training instances $\mathcal{X}$, the parameter space $\mathcal{Y}$, and the underlying algorithmic framework $A(\theta|x_i)$. The user is free to set the tuning budget B, enabling easy regulation of the computation time spent on the tuning procedure. Moreover, we have two tunable parameters α and β, which can be used to affect the tradeoff between exploitation and exploration and the degree of localized search respectively. We return to these tunable parameters in Section 4.2.4 and Section 4.2.5.

input
$\mathcal{X}$: training set
$\mathcal{Y}$: parameter space
$A(\theta, x_i) : \mathcal{Y} \to \mathbb{R}$: algorithmic framework

tunable parameters
B : tuning budget
α : level of significance
β : degree of bias

initialization
$\Theta_r \leftarrow \text{LHD}(\mathcal{Y})$
$j \leftarrow 1$

while *tuning budget left* **do**
  while *inferior configuration in* Θ_r *under* α **do**
    $\Theta_r \leftarrow \Theta_r \setminus \theta^{\text{inferior}}$
    $\theta^{\text{child}} \sim \text{Resample}\left(\mathcal{Y}, \Theta_r, \beta\right)$
    for *every instance i solved by parent* **do**
      $A\left(\theta^{\text{child}}, x_i\right)$
      if $\theta^{\text{child}} = \text{inferior}$ **then** break
    end
    $\Theta_r \leftarrow \Theta_r \cup \theta^{\text{child}}$
  end
  for each $\theta \in \Theta_r$ **do** $A\left(\theta, x_j\right)$
  $j \leftarrow j + 1$
end

output: θ^*

Algorithm 1: Basic Structure of uRace

Our tuning algorithm is largely inspired by I/F-Race, an algorithm that is developed specifically to solve the offline configuration problem of Equation 4.1 (see Section 2.2.3). However, we propose an important structural change to more efficiently use the obtained information. Following the discussion in Section 4.1.1, we exploit the opportunity to enforce fair comparisons between configurations, by letting

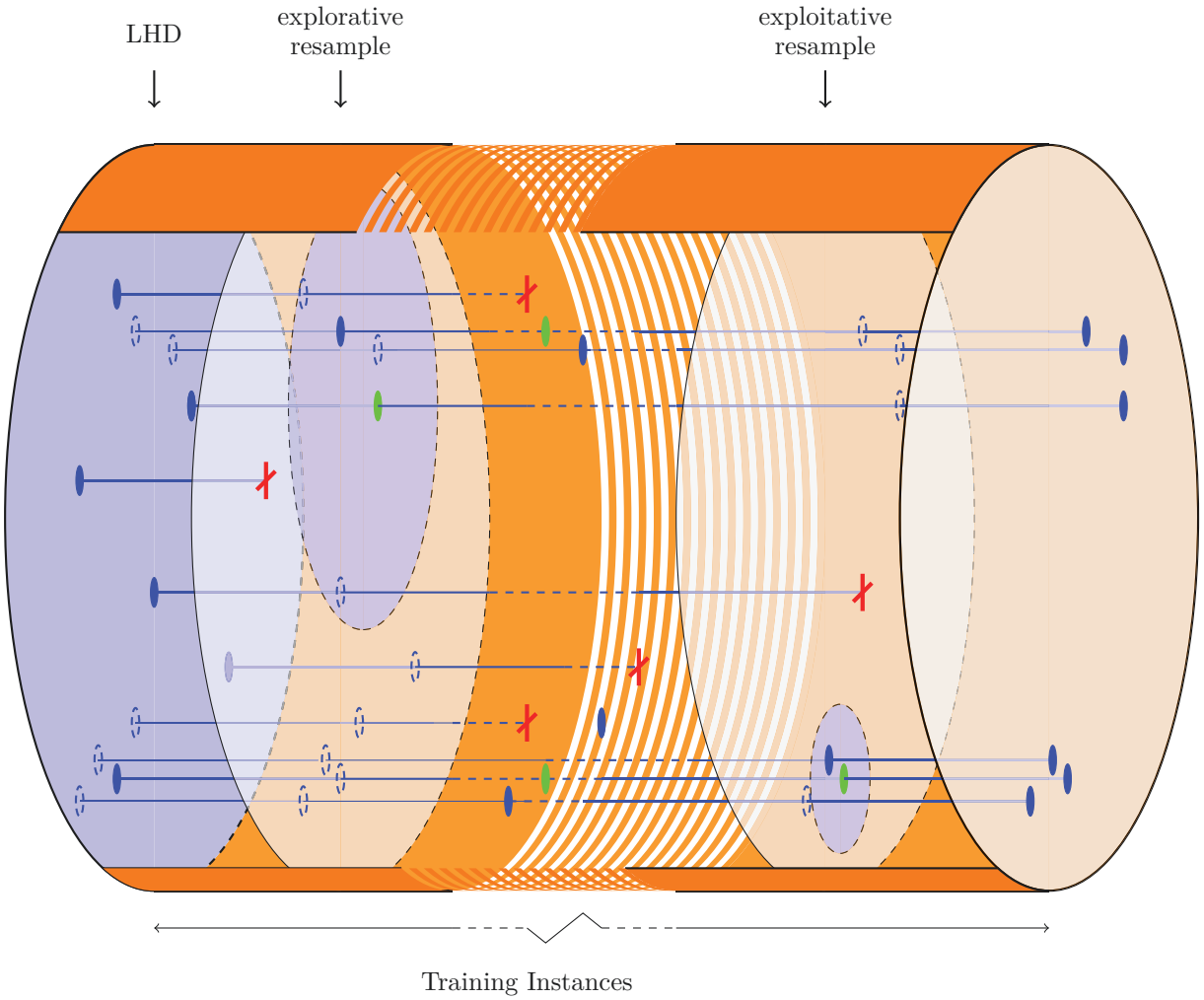

Figure 4.1: Visualization of uRace, showing the parallel evolution of parameter configurations and an increasingly localized search. The blue ellipse on the left represents our multidimensional parameter space, from which we generate our initial sample of configurations (i.e., the first six filled blue dots that uniformly cover the ellipse) using LHD. Next, we start conducting batch measurements on our sample, represented by the blue lines that grow horizontally. After every problem instance, we perform a statistical test, where we eliminate any configuration that is inferior to at least one other known configuration (represented by a red cross). For every eliminated configuration, we select a known parent configuration (i.e., the green dots) and sample a new child configuration (i.e., the filled blue dots halfway the training instances) from its neighborhood (i.e. the smaller blue ellipses). Note that our search becomes more localized (i.e., smaller blue ellipse on the right) as the tuning process proceeds, such that we perform a more exploitative resampling procedure. Note that a new child configuration first solves the instances previously solved by its parent (i.e., blue lines go left from the child to first instance) and it may occur that the child is eliminated based on those "old" instances. Ultimately, uRace evolves several local optima in parallel.

all considered configurations solve the same problem instances, without wasting any valuable information. I/F-Race enforces fair comparisons, by letting each iteration be an independent F-Race, where information from previous iterations is solely used to generate a set of configurations to commence the next F-Race. The exact results of individual measurements are disregarded. To enable the use of this valuable information, while still ensuring fair comparisons, we prefer to let every newly generated configuration first solve the same problem instances as those previously solved by its parent configuration. This gives us more information about the new configurations than measurements on new problem instances would, and allows more efficient use of previously obtained information.

The result of this structural change is that we create one unified Race algorithm, which we denote by uRace, rather than a sequence of autonomous F-Race algorithms. We consider each iteration to be a batch measurement of all candidate racers on *a single* problem instance, and we generate new configurations whenever statistically inferior configurations are eliminated in our sample. This allows for more sensible allocation of computation budget throughout the tuning process, and more natural allocation of computation time to either exploration (i.e., exploring new points in the solution space) or exploitation (i.e., acquiring more reliable information on known solution points). Indeed, at the start of the tuning process, following our preliminary analysis in Section 3.3.2, we expect many inferior configurations, naturally allocating computation time to exploration. As the tuning process proceeds, and we approach the best configuration θ^*, we expect the candidates to be closer to one another in terms of performance. Therefore, by only exploring new options in case of statistical inferiority, computation time allocation naturally tends towards exploitation.

The remainder of this section describes uRace in detail, essentially tracing into the generic pseudocode in Algorithm 1 at several lines. The general concept of uRace is inspired by Simulated Annealing in the sense that we start with a wide exploration of the solution space and, as the tuning process proceeds, increasingly localize our search around the most promising areas (as reflected in Figure 4.1). However, our tuning algorithm is different from Simulated Annealing in the way it accomplishes this. In SA, generally only one solution point is carried (and an additional one saved in memory), and the next solution point is typically sampled at random from a neighborhood structure that is a subspace of the entire solution space. By rejecting worse solutions with increasing probability, the algorithm converges toward the best neighborhoods, and therefore the search is localized around the most promising areas. In our tuning algorithm, the probability of going from one solution point to any other point in the solution space is always non-zero (which guarantees convergence to the global optimum as $B \to \infty$). In other words, our neighborhood structure is essentially the entire solution space, from which we sample a new solution point at random, assigning weighted sampling probabilities according to expected performance. Moreover, the stochastic nature of our problem enforces uRace to carry multiple solution points, which are evolved in parallel.

4.2.3 Initial Sampling

As mentioned in Section 2.2.3, the initial sampling is meant to cover the parameter space as evenly as possible. Indeed, the initial sample forms the roots of the entire subsequent genealogy, so, intuitively, it is desirable that it uniformly covers the entire parameter space. For this purpose, we use the adapted categorical LHD, as proposed in Section 3.3.

As said, we let the size of the set of racing configurations depend on the size of our parameter space, our computation budget, and our objective. That is, we set $|\Theta_r| \equiv N_r = k$ (i.e., the number of parameters in $\mathcal{Y}$) at the start of uRace, and let it gradually (with τ^β, see Section 4.2.5) decrease to 2 (i.e., $s+1$, see Equation 4.4) as the algorithm proceeds. If, at some point during the tuning process, the current size of Θ_r exceeds its designated size while no statistically inferior racers are detected, the worse configurations

are eliminated.

4.2.4 Statistical Inferiority Test

As soon as sufficient statistical evidence is collected that a configuration is inferior to at least one other configuration in Θ_r, we waste no more computation time and eliminate the inferior configuration. As said, we enact a batch measurement policy, where we let all racers solve the same sequence of randomly drawn problem instances. We exploit the pairwise observations resulting from the batch measurements, by using a pairwise statistical test. As indicated in Section 3.3.2, we cannot assume normally distributed performances, nor equal variances among the configurations, leaving Wilcoxon's paired signed-rank test as the most suitable non-parametric statistical test for our purpose. Section 3.3.2 employs the normal approximation of the test statistic, which is only valid after at least 10 paired observations. The results show that many configurations can already be eliminated after the first test, suggesting that is it may be worthwhile to start testing for statistical inferiority earlier, in which case we rely on the actual test-statistic (Sani and Todman, 2006). We perform one-tailed tests, and compare the best configuration (highest mean performance) with all others.

The level of significance α determines the severity of the test. The higher we set α, the less evidence we require to exclude configurations. This speeds up the procedure, but also increases the probability of falsely rejecting the null hypothesis or, in other words, eliminating non-inferior configurations. The results in Section 3.3.2 suggest that the robustness of the configurations is so high that even for relatively high levels of significance (i.e., $0.1 \leq \alpha \leq 0.3$), unlawful rejection of the null hypothesis hardly occurs. However, because of the structure of uRace, the value of α also has an important effect on the aforementioned exploration-exploitation tradeoff. Recall that the structure of uRace enforces that the tradeoff naturally tends more towards exploitation as the tuning process proceeds, but the value of α determines the severity of this transition. For example, in the limit that $\alpha \to 0$, we can only obtain sufficient statistical evidence by determining the configuration's true mean performance. That means that configurations are never eliminated, new configurations are never evolved, and our algorithm converges to exhaustive evaluation. In other words, all computation time is undesirably spent on exploitation of the initial sample. On the other hand, as $\alpha \to 1$, our one-tailed tests ensure that all configurations (except the very best one) are continuously eliminated. In other words, all computation time is spent on exploration, and we undesirably end up making a implementation decision based on a single measurement. We find $\alpha = 0.20$ to result in a good exploration-exploitation tradeoff for our problem space (see Section 5.1.4).

4.2.5 Resampling

Every time a statistically inferior configuration is eliminated, a newly generated child configuration, which resembles some known, well-performing parent configuration, takes its place. This resampling procedure consists of two parts. First, we select, from the set of remaining configurations Θ_r, one parent configuration θ^{parent}. Second, we evolve, from this parent, one child configuration θ^{child}.

Parent Selection

We select a parent configuration θ^{parent} by randomly sampling one configuration from Θ_r according to some probabilistic model. We generally prefer to assign a higher probability to more promising parents. The probabilistic model employed in I/F-Race provides such a situation in which the best parent in a set of size N_r has an N_r times higher sampling probability than the worst parent. However, to enable the parallel evolution of multiple solution points, we prefer a probabilistic model that initially values all

parent configurations equally, and evolves as the tuning process proceeds. That is, we commence with equal probabilities over all potential parents and assign increasingly higher probability to the best parent configurations, gradually converging to the probabilistic model of I/F-Race.

The probability $P(\theta_n = \theta^{\text{parent}})$ that configuration $\theta_n \in \Theta_r$, with $|\Theta_r| = N_r$ is selected as a parent is given by

$$P(\theta_n = \theta^{\text{parent}}) = \frac{1}{N_r} \times \left(1 - \tau^{\beta}\right) + \Delta P_n \tag{4.2}$$

where $0 < \beta$ is the degree of bias we impose, and

$$\Delta P_n = \frac{2\,(N_r - r_n + 1)}{N_r(N_r + 1)} \times \tau^{\beta} \tag{4.3}$$

where $\tau = {}^{B_{\text{used}}}/_{B}$ is the fraction of the tuning budget that is already spent, r_n is the rank of θ_n, such that $r_n = 1$ is assigned to the best configuration (highest mean performance) and $r_n = N_r$ to the worst configuration in Θ_r.

Note that the left part of Equation 4.3, and therefore also $P(\theta_n = \theta^{\text{parent}})$, is inversely proportional with r_n, such that the probability of selecting a configuration increases with its performance. Moreover, as the tuning algorithm proceeds, the fractional spent budget τ, and therefore ΔP_n increases, such that the bias towards the most promising parent configurations increases. By adapting the degree of bias β, differently shaped functions can be acquired, all gradually converging from equal probabilities to the probabilistic model of I/F-Race. Note that for $\beta = 0$, the functions reduce to those employed by I/F-Race. We find the S-shaped function acquired by letting $\beta(\tau) = 4(1 - \tau)^2$ to give the best results in our problem space (see Section 5.1.4).

Child Generation

Having selected a parent configuration, we evolve from it a child configuration. Considering the categorical nature of our parameter space and the lack of a logical aggregation structure therein, we propose to use the Biased Sampling Distribution method (cf. Section 2.2.2 and Section 2.2.3). In this light, it is important to consider the high degree of synergy in our parameter domain, as suggested by the results in Section 3.3.2, which means that the performance of a particular individual parameter setting in one configuration does not reflect on its performance within a completely different configuration. This suggests we should not employ (and bias) one general sampling distribution for all configurations. Such a method, for example employed in REVAC (see Section 2.2.2), is blind for the parameter interactions that are hugely abundant in our parameter space. Instead, we propose to assign individual sampling distributions to every configuration, which we bias based on the historic performances of that configuration and its ancestors only. In other words, each configuration carries a sampling distribution, with a bias towards the values taken by itself and its ancestors only, which is used to generate a child configuration if the given parent configuration is selected. The child configuration is randomly sampled using the sampling distribution over $\mathcal{Y}$ that is carried by the selected parent configuration.

The strength of the bias determines how locally we search around known configurations. To facilitate the concept of starting with random exploration and increasingly localizing the search around the most promising areas, we let the bias increase as the tuning algorithm proceeds. We start with a k-variate uniform distribution over each parameter space, and for each configuration we evolve a discrete k-variate sampling distribution biased towards the settings taken by itself and its ancestors. Let $P\left(Y_d = y_d\right)$ be the probability that we sample the value y_d for parameter Y_d, which we initialize as:

$$P_0(Y_d = y_d) = \frac{1}{|\mathcal{Y}_d|} \; \forall d$$

We adjust the sampling distributions such that a child configuration is most likely to inherit half of the parameters from its parent at the start of the tuning process, and all but one of the parameters towards the end of the tuning process (i.e., increasingly localized search). Suppose that M is a random variable indicating how many parameters of the child configuration inherit their value from the parent configuration, and p is the probability of sampling the same value on each one-dimensional parameter space. Then M is binomially distributed as $M \sim B(k, p)$, and we set the sampling probabilities at the start and the end of the race as follows.

$$p_{\text{start}} = \arg\max_p P\left(M = \frac{1}{2}k\right) \quad p_{\text{end}} = \arg\max_p P\left(M = k - 1\right)$$

During the tuning process, the biased sampling probabilities (that is, the probability of inheriting the parameter value on a one-dimensional parameter subspace) gradually increase from p_{start} to p_{end}. After every iteration j, the sampling distributions from the remaining configurations are updated as

$$P_j\left(Y_d = y_d\right) = \max\left\{p_{\text{start}}, \min\left\{p_{\text{end}}, P_{j-1}\left(Y_d = y_d\right) \times \left(1 - \tau^\beta\right) + \Delta P\right\}\right\} \quad \forall d$$

where $0 < \beta$ is the degree of bias we impose, and

$$\Delta P = \begin{cases} \tau^\beta & \text{if } Y_d = y_d \text{ for } \theta^{\text{parent}} \\ 0 & \text{otherwise} \end{cases}$$

Note that, as the tuning algorithm proceeds, the fractional spent budget τ increases, such that the bias becomes stronger, and we gradually search more locally around the most promising parameter settings. Similarly as with the parent selection, the degree of bias β that we impose determines the shape of this gradual process. Linear and exponential functions are acquired by letting $\beta = 1$ and $\beta > 1$ respectively. Alternatively, logarithmic functions can be attained by letting $0 < \beta < 1$. Again, we employ the S-shaped function acquired by letting $\beta(\tau) = 4(1 - \tau)^2$ (see Section 5.1.4).

Naturally, towards the end of the tuning process, the increased bias induces a substantial probability that a child inherits *all* parameter settings from their parent configuration, in which case the sampling process is repeated until we find a unique child configuration. The child configuration inherits the current sampling distribution of its parent configuration.

4.2.6 Multiple Implementation

In Section 4.2.1, we introduced the offline configuration problem in Equation 4.1, in which the objective is to find a single parameter configuration with the highest average performance on the training set. Recall that ultimately, the objective is not to seek optimal performance on the training set, but rather on the (online) problem at the customer site. As said, our parameter space analysis in Chapter 3 suggests that the performance of θ^* generalizes well to similar, unseen problems that belong to the same problem family, but naturally there is no guarantee that the best configuration on average on the training set, is the best configuration on every individual problem instance at the customer site. To counter this issue, and to avoid any poor outliers, which harm ORTEC's credibility, we introduce an extension to the offline configuration problem as proposed by Birattari (2009). Instead of looking for a single configuration θ^*, we seek for a set of configurations Θ_s^* of size s that maximizes the expected performance of the best member in the set. In a practical application, that means that we include all s configurations in the solution

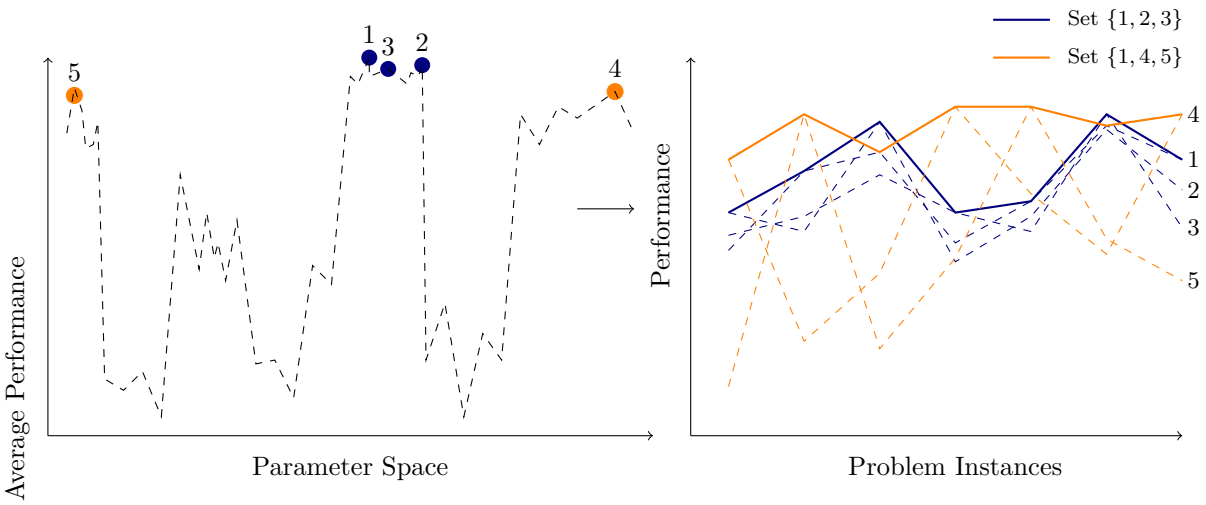

Figure 4.2: Example of alternative configuration selection and resulting performances on the test set. On the left, our categorical parameter space with 5 known configurations and their average performance on the training set. On the right, we see the fluctuating performances of these 5 configurations on individual instances in the test set. By sacrificing on average performance in favor of more diverse configurations, we are able to increase the online performance of the set.

process at the customer site, and subsequently choose the best found performance for implementation. This balances any poor outliers, improving both performance and robustness, but it also adds to the valuable running time at the customer site.

We refer to this alternative objective as the best-few objective, in which we want to find the subset of configurations $\Theta_s^* \subset \Theta$ of size s such that

$$\Theta_s^* = \arg\max_{\Theta_s \subset \Theta} \mathbb{E}\left[\arg\max_{\theta \in \Theta_s} A(\theta)\right] \tag{4.4}$$

Note that for $s = 1$, Equation 4.4 reduces to Equation 4.1. However, the problem in Equation 4.4 is different from finding the best s configurations on average on the training set. Instead, we maximize the expected maximum performance of the selected configurations on every individual instance, or the expected performance of the best member of the set Θ_s^* on each instance. In the extreme case that the performances of the configurations in Θ_s^* are identical on every instance, application of more than one configuration is superfluous and a waste of valuable running time at the customer site. Instead, the performance at the customer site benefits from a diverse selection of configurations Θ_s, so that poor performances on individual instances are balanced by other configurations. In other words, we may want to include a configuration, because it shows very diversely fluctuating performances over the problem instances, even if its average performance is worse than another known configuration.

We illustrate this point with an example. Figure 4.2 shows the same exemplar categorical performance landscape as in Section 2.2.2. The configurations $\{1, 2, 3\}$ are the best known configurations, but they are also very similar (i.e., close to one another in the parameter space), which results in similar performance on individual problem instances. The thick blue line on the right represents the performance of the set $\{1, 2, 3\}$ at the customer site. Alternatively, we can include the configurations 4 and 5 in our selection. That means we sacrifice on the average performance on the test set (e.g., 4 is worse than 3), in favor of a more diverse selection of configurations. Indeed, configurations 4 and 5 originate in distinct parts of our parameter space, which results in more diversely fluctuating behavior on the test set, and ultimately, represented by the thick orange line on the right, a higher performance at the customer site.

Particularly for the pursuit of diverse parameter configurations for multiple implementation, the

Component	Method
Initial Sampling	Latin Hypercube Design
Number of Parallel Configurations	Start with k configurations and gradually decrease to $s+1$ as $k - \tau^{\beta}(k-(s+1))$
Measurement Policy	Batch Measurements
Statistical Inferiority Test	Wilcoxon Paired Signed-Rank Test with $\alpha = 0.20$
Parent Selection	Rank-Based Sampling with Increasing Bias
Child Generation	Biased Sampling Distributions for Individual Configurations with Increasing Bias
Degree of Bias	$\beta(\tau) = 4(1-\tau)^2$

Table 4.1: The components of uRace, the unified Race algorithm we propose to develop.

parallel development of multiple local optima proves to be a valuable asset of uRace. Moreover, it suggests the possibility for sequential applications of uRace. That is, we may succeed our tuning effort with additional tuning efforts on subsets of our original training set containing problem instances on which the best found configuration(s) showed relatively poor performance.

4.3 Conclusion

In this chapter, we developed our tuning algorithm: uRace. First, we discussed the interesting role of stochasticity in our research. Through an analogy with the multi-armed bandit problem, we identified problem family tuning of deterministic algorithms to be very similar to problem instance tuning of stochastic algorithms. In both scenarios, it is important to compare the performance of configurations in the same external circumstances. Together with the results of our previous parameter space analysis, this induced our choice for an iterative Race algorithm as a suitable solution direction, for its parallel treatment of multiple parameter configurations and its ability to deal with enormous categorical parameter spaces.

Next, we formulated the offline configuration problem that we intend to solve, and we developed our tuning algorithm uRace explicitly. We proposed an important structural change compared to the most commonly used Race algorithms in the literature, which unifies the entire tuning process and allows for efficient usage of historic observations while enforcing fair comparisons. The general structure of uRace is given in Algorithm 1, but the exact behavior of uRace can be freely guided by adjusting its main components: initial sampling, statistical testing, and resampling. We developed the components such that they can be easily adapted by tuning the degree of bias β, the level of significance α, and the invested tuning time B. Finally, we introduced an extension to the traditional offline configuration problem in which we search for a mutually complementary set of good configurations in order to increase online performance and robustness.

In summary, we propose uRace, a unified Race algorithm of which the components and their properties are given in Table 4.1.

5 | Results

This chapter contains the results of our research. The analysis of the results of our efforts is two-fold. First, we analyze the behavior of uRace, and compare it to alternative tuning methods. Next, we consider the practical relevance of our research for ORTEC and its customers. That is, we analyze the performance of OPLB with the parameter settings as tuned with uRace. In this light, we perform two comparisons. First, we compare the performance with algorithms in the literature on academic benchmark problems. Second, we compare the performance with OPLB with alternative, a priori sensible parameter configurations on a complex practical problem set of a large multinational.

5.1 Performance of uRace

This section contains an analysis of the performance of uRace. First, in Section 5.1.1, we discuss the effect of the allocated tuning budget on the performance of uRace, proving the beneficial effects of a longer tuning effort. In Section 5.1.2, we discuss the alternative best-few objective (see Section 4.2.6) and how the parallel evolution of several local optima within uRace aids towards this objective. Section 5.1.3 discusses the importance of a representative training set, and Section 5.1.4 provides a sensitivity analysis on the tunable parameters within uRace. Finally, in Section 5.1.5, we compare the performance of uRace with alternative tuning methods.

5.1.1 Tuning Budget

We commence with an analysis of the invested tuning time versus the best found parameter configuration. Following the objectives in Equation 4.1 (Equation 4.4), the performance of uRace can be expressed as the expected volume utility of the selected configuration (the best configuration from the selection of configurations on every individual instance), on the complete training set $\mathcal{X}$. Note that uRace does not necessarily evaluate all problem instances in the training set during the tuning process (e.g., when the allocated tuning time is too short or further exploration is preferred over exploitation), but its performance is measured according to the expected volume utility over the complete training set. As a training set, we use the same set of weakly heterogeneous BR instances as in our parameter space analysis in Chapter 3. Figure 5.1 shows the relation between the performance of uRace and the invested tuning time for both the single-best and the best-few objective. As mentioned in Section 4.2.6, in the best-few scenario we take several, in this case five, parameter configurations from our tuning effort, apply all of them to the individual problem instance, and subsequently select the best performance on that specific instance. In other words, the performance is improved, but the (online) running time of the best-few scenario at the customer site would be, in this case, five times longer.

The performance of uRace increases logarithmically with the invested tuning time. Moreover, confidence intervals of the performance, caused by the stochastic nature of uRace, decrease with the invested

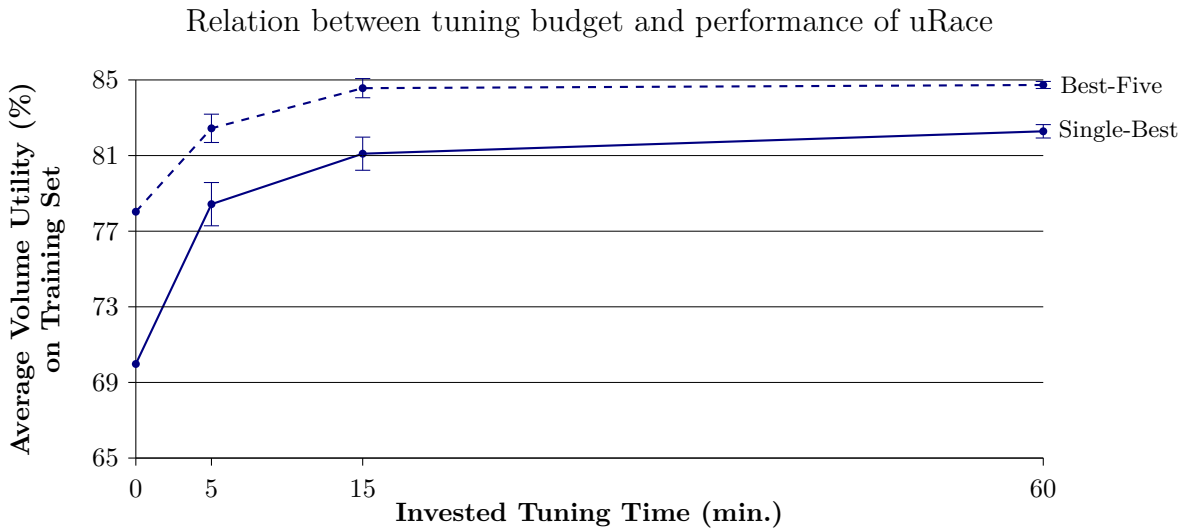

Figure 5.1: Relation between invested tuning time and the performance of uRace, with corresponding 95% confidence intervals, for a single-best and a best-few scenario.

tuning time, such that, similarly as in Section 3.3.2, we simultaneously optimize performance and robustness by allocating more tuning time. Finally, through the best-five application, we increase performance with several percent points while increasing robustness (smaller confidence intervals).

5.1.2 Multiple Implementation

Recall that the beneficial effect of a best-few application is greater when the performances of the selected configurations on individual instances are more diverse (see Section 4.2.6). In this light, under the assumption that more diverse parameter configurations lead to more diverse performances on individual instances, the parallel evolution of configurations in distinct parts of our parameter space proves to be a valuable asset of uRace.

Figure 5.2 shows an example of the evolution of three of the best parameter configurations resulting from a ten-hour tuning run of uRace. The performances are given as the average volume utility of that parameter configurations on the evaluated training instances. Recall that deteriorations in performance from parent to child configurations are accepted as long as no statistical inferiority is detected, which, as Figure 5.2 indicates, may ultimately lead to a better solution (e.g., configuration 928 to 1085). Two of the configurations (i.e., configuration 1756 and 1774) share the same ancestors, whereas the other configuration (i.e., configuration 1625) is evolved completely in parallel. This is reflected in the respective parameter configurations in the sense that configurations with the same ancestors share more commonalities (where a later segregation results in more commonalities), than configurations that are evolved in parallel. Table 5.1 shows the performance of the three remaining configurations in Figure 5.2 on the complete training set. Note that the single-best configuration on the evaluated instances (i.e., configuration 1774) is slightly worse than its nearest competitor (i.e., configuration 1756) on the complete set of instances. Moreover, note that, even though configuration 1625 is worse than 1756 on average, it contributes more to the best-few objective. The reason for this can be found in the evolution of the configurations, which causes 1774 and 1756 to be more similar in terms of parameter settings, which is reflected by a more similar performance on individual problem instances, as Figure 5.2 also shows. The worst of the three configurations on the training set (i.e., configuration 1625) shows more diversely fluctuating performance, therefore proving its value for the best-few objective even though its average performance is worse.

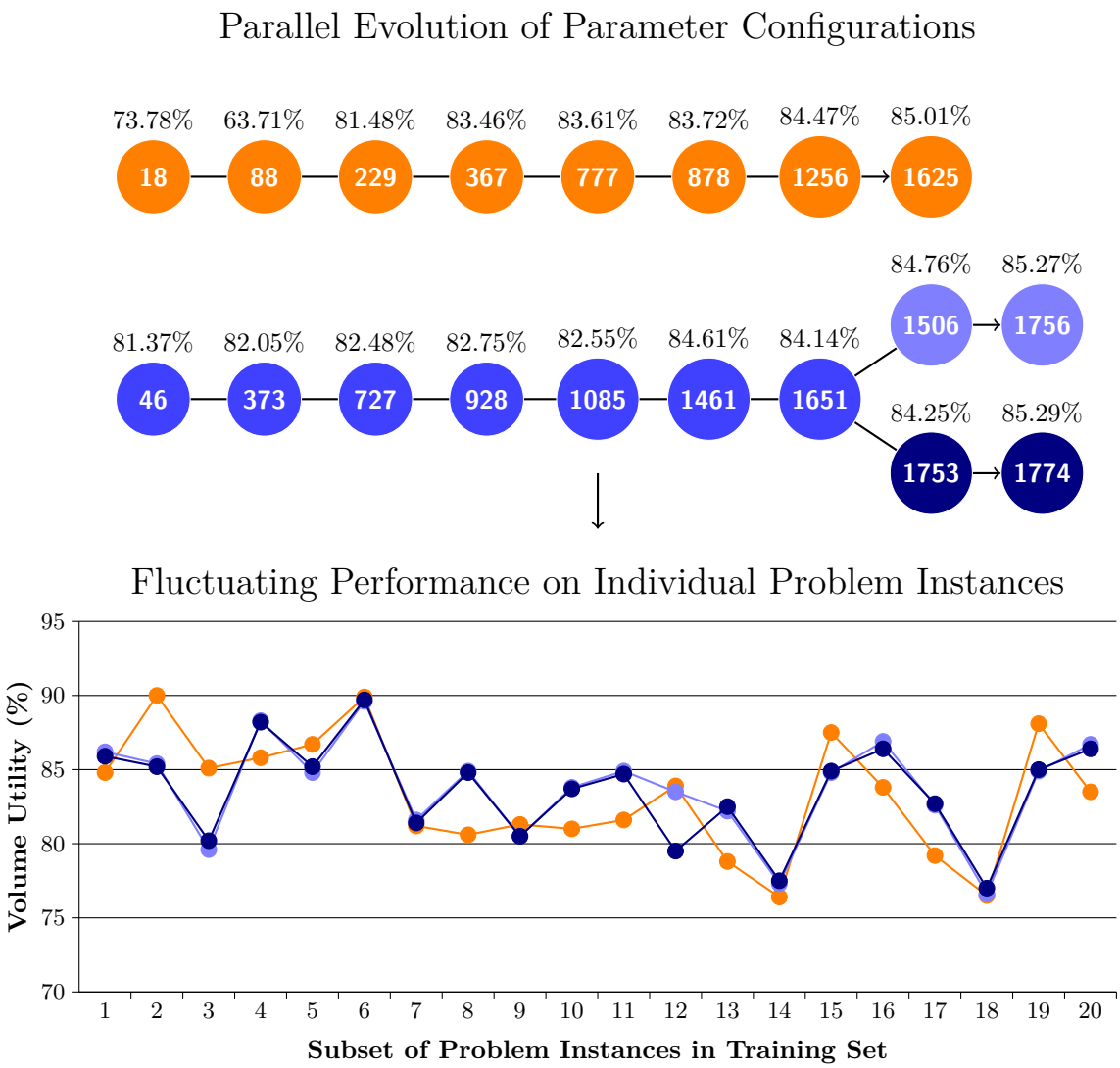

Figure 5.2: Example of the evolution of three of the best parameter configurations. Note that deteriorations in performance from parent to child configuration (e.g., configuration 928 to 1085) ultimately lead to better solutions. Configurations 1756 and 1774 share the same ancestors, whereas configuration 1625 is evolved entirely in parallel. The performance of the three configurations on part of the individual problem instances in the training set show that configurations that are evolved entirely in parallel, hence share less commonalities in terms of parameter settings, and therefore show more diversely fluctuating performance on individual instances.

	1774	1756	1625	Single-Best	Best-Three
Volume Utility	83.11%	83.13%	83.02%	83.11%	85.16%
Improvements (out of 70)	-	20	23		43
Contribution to Performance	-	+0.90%	+1.15%		+2.05%

Table 5.1: Performance of the three evolved parameter configurations in Figure 5.2 on the complete set of instances. The improvements indicate on how many instances the given parameter configuration provides an improvement on the single-best configuration (i.e., configuration 1774). The contribution to performance indicates the part of performance improvement of the best-few application that the given configuration contributes (i.e., $0.90 + 1.15 = 2.05 = 85.16 - 83.11$).

	Training Set		
	BR1	BR1-BR7	BR7
Volume Utility (%)	83.75	83.22	82.57

Table 5.2: Performance on a test set of BR1 instances of a single construction with the parameters tuned with uRace based on three alternative training sets.

5.1.3 Representative Training Set

When we introduced problem family tuning in Section 1.2, we already mentioned the desire for a "representative" training set. We now give support to the intuitive notion that the representativeness of the training set is proportional to the performance of the tuning algorithm. That is, as the problem instances in the training set share more commonalities with the actual online problem instance at the customer site, the performance of the tuned parameter configuration improves. Note that our academic problem set consists of relatively similar, weakly heterogeneous problem instances, but we may still distinguish between instances based on their degree of heterogeneity. Suppose the online problem instance that we want to solve is a BR1 instance (see Section 2.1.3), which we represent with a test set of 50 BR1 instances. We consider three alternative training sets: a set of BR1 instances, the complete set of heterogeneous BR instances (i.e., BR1-7), and a set of BR7 instances. Table 5.2 compares the performance of the best found parameter configuration on the same test set of BR1 instances, where the parameters were tuned with uRace based on the three different training sets with the same tuning time. Note that a more representative training set leads to a better performance. Still, the parameters tuned on the distinctly different BR7 instances still perform admirably on the test set, as BR7 and BR1 instances are ultimately both weakly heterogeneous BR instances and therefore relatively similar loading problems.

5.1.4 Sensitivity Analysis

In this section, we provide a sensitivity analysis on the tunable parameters within uRace. In Section 5.1.1, we showed the effect on the allocated tuning budget on the performance of uRace. In this section, we consider the parameters α and β.

Level of Significance α

As discussed in Section 4.2.4, the level of significance α determines the severity of our statistical inferiority tests. That is, the higher we set α, the less evidence we require to eliminate configurations. Moreover, recall from Section 4.2.4 that the value of α also plays an important role in the exploration-exploitation tradeoff. Indeed, higher values of α induce two effects. First, more configurations are eliminated, which means that, as we generate a new child configuration for every elimination, more alternative configurations are explored. Second, because more tuning time is spent on exploration, less is spent on exploitation, hence less problem instances are evaluated. Figure 5.3 illustrates these effects for a tuning time of one hour and five different values for α. For all values of α, the total number of measurements is roughly identical, but they are allocated differently, resulting in either more configurations or more problem instances. Clearly, to maximize either of these two measures, we simply set α to the corresponding boundary value (e.g., to maximize the number of evaluated instances, we set $\alpha = 0$).

Ultimately, we are concerned with neither the number of generated configurations nor the number of evaluated instances, but solely with solely with the average performance on the complete training set. Considering this objective, the optimal value of α is not on either of its boundaries, but somewhere in between. In Figure 5.3, this optimum is at $\alpha = 0.20$, for which we find a configuration with the highest average performance on the complete training set. This maximum coincides with a minimum of the

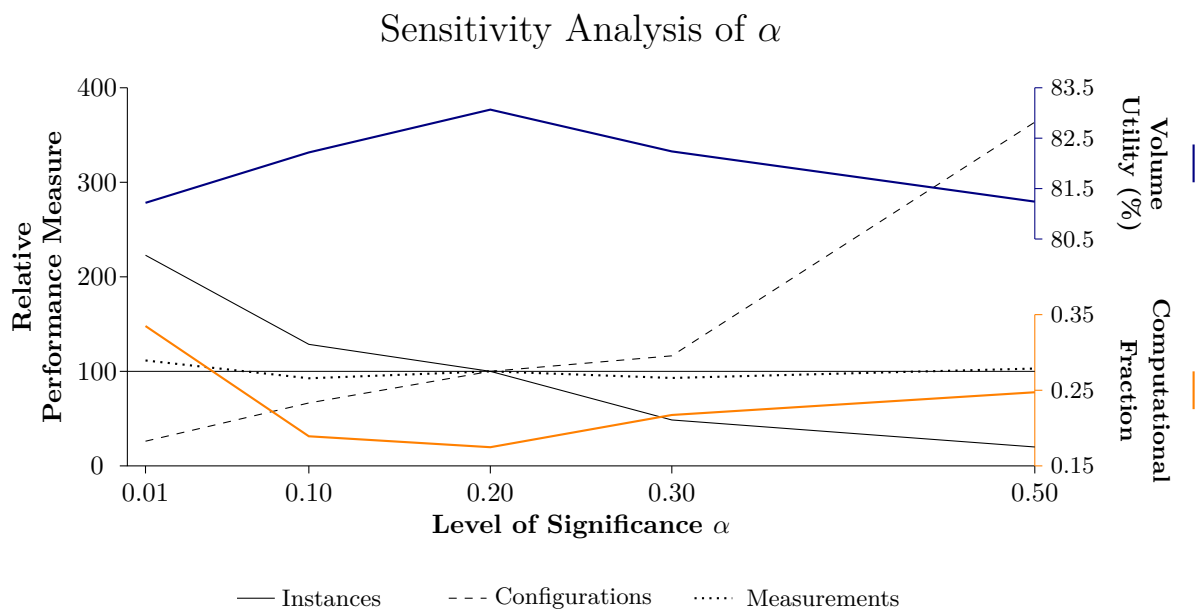

Figure 5.3: Illustration of the effect of the level of significance α on five performance indicators for a tuning time of one hour. The number of generated configurations, number of evaluated instances (for the best configuration), and total number of measurements are expressed relative to the index performance for $\alpha = 0.20$. For the performance (volume utility (%)) of the best configuration on the complete training set (in blue) and the computational fraction (orange), the absolute values are plotted on their own scales. The computational fraction is the fraction of computation time spent relative to exhaustive evaluation of all generated configurations on all evaluated instances (i.e., the number of measurements relative to the product of the number of configurations and the number of instances), similar to the results in Table 3.9.

computational fraction, or a maximum in the computational benefits resulting from our statistical tests (see Section 3.3.2). This fraction decreases with α for low values as less statistical evidence is required for elimination, but starts to increase again for high values of α (until a value of 1 for $\alpha = 1.00$) as the number of instances evaluated reduces and therefore the number of measurements required to perform the first statistical tests becomes relatively more substantial.

Degree of Bias β

As discussed in Section 4.2.5, the degree of bias β determines the locality of our search. That is, through three different angles, it affects how locally we search around known, relatively well-performing configurations: (i) through the gradual reduction of the number of parallel racers (see Section 4.2.3), (ii) through the bias towards well-performing configurations in the parent sampling, and (iii) through the bias towards the parameter settings taken by the selected parent in the child generation. The value of τ^β governs all three processes, where a higher value of τ^β induces a more localized search. As τ is the fraction of tuning time spent, we have $\tau^\beta = 0$ at the start, and $\tau^\beta = 1$ at the end of the tuning process. The value for β determines what happens in between. Figure 5.4 shows the gradual increase of τ^β for different values for β.

The effect of β on the performance of uRace is more subtle than that of α and not easily captured in performance measures. Generally, we want to avoid searching too locally, and therefore getting trapped in a local optimum early on, while spending sufficient time in the neighborhood of good known configurations towards the end of the tuning process. Which value of β enforces this behavior depends on the parameter space. The performance of uRace of one hour with the given values for β (and $\alpha = 0.20$) in our case is given in Figure 5.4, inducing our choice for the aforementioned S-shape (see Section 4.2.5).

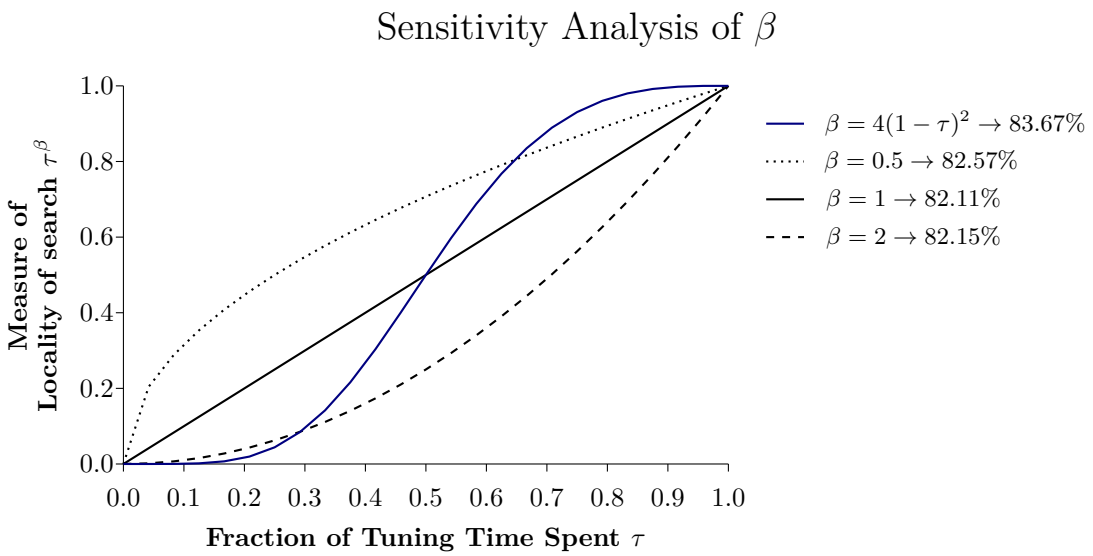

Figure 5.4: Illustration of the effect of the degree of bias β on the locality of the search throughout uRace. The higher the value for τ^β, the more localized our search around the most promising areas in our parameter space. On the right the performances of the best found configuration in a one-hour run of uRace are given for different values of β.

	Tuning Time (min.)			
Tuning Method	5	15	60	600
uRace	80.95	81.70	82.88	84.11
Random Sampling with Elimination	78.49	79.89	80.64	81.65
Random Sampling	74.38	78.66	80.18	81.29

Table 5.3: Performance (average volume utility (%) on training set) of alternative tuning methods.

5.1.5 Comparison with Alternative Tuning Methods

In this section, we provide the results of our proposed tuning algorithm in comparison the most apparent alternative method: random sampling, with and without statistical elimination. For the alternative methods, we employ the same training set of BR instances as for uRace, and, evidently, every tuning algorithm is allocated the same amount of computation time. Table 5.3 compares the performance of uRace with the two alternative tuning methods, showing that uRace consistently gives better results than the alternative methods.

5.2 Performance of OPLB with uRace

In this section we take a practical perspective, and analyze the performance of ORTEC's software tool, OPLB, with its parameters tuned with uRace. We consider both our academic and the practical problem set (see Section 2.1.1), and compare the performance of OPLB with uRace with loading algorithms in the literature and a priori sensible parameter configurations respectively.

5.2.1 Performance on Academic Problem Set

First, we analyze the performance of OPLB on the academic problem set. We compare OPLB with its parameters tuned with uRace with two alternative parameter configurations that are selected to mimic known construction algorithms in the literature. Both construction algorithms take one of the basic

Parameter Configuration	Volume Utility (%)	# Best
Wall Building approach (George and Robinson, 1980)	80.59	96
Horizontal Layer (Bischoff et al., 1995)	82.70	249
Configuration Tuned with uRace	83.99	363

Table 5.4: Performance of the parameter configuration tuned with uRace, compared with a priori sensible configurations based on known constructive heuristics on a set of 700 weakly heterogeneous BR instances. The last column indicates the number of problem instances on which the given parameter configuration gives the best performance (they add to 708 because of a few draws).

building approaches as discussed in Section 2.1.2. The first is developed by George and Robinson (1980), forming the basis for the algorithm by Moura and Oliviera (2005) (see Section 2.1.3), and takes a wall-building approach along the width of the container. The second is based on a construction heuristic by Bischoff et al. (1995), and takes a horizontal layer approach. These two construction heuristics can be "translated" into a parameter configuration for OPLB, and mean to represent parameter configurations that seem sensible a priori. That is, these construction heuristics are developed especially to optimize performance on academic benchmark instances, and are known to perform well. Table 5.4 shows the results of these two construction heuristics on a set of 700 weakly heterogeneous BR instances (the first 100 of BR1-BR7), along with the performance of OPLB with its parameters tuned with uRace, showing that OPLB outperforms both.

Now that we have shown to successfully outperform other parameter configurations based on simple constructive heuristics in the literature, we turn ourselves to a comparison with the state-of-the-art optimization methods discussed in Section 2.1.3. Table 5.5 tabulates the performances of the same algorithms as in Table 2.1, on a set of weakly heterogeneous BR instances, and compares it with the performance of OPLB with its parameters tuned with uRace. The parameters are tuned on a set of similar, yet distinct BR instances, such that the tuning time does not count towards the online running time. Aside from the single-best single construction algorithm, which is identical to the one in Table 5.4, we also report the performance of a best-few application (see Section 4.2.6) with selections of 5 and 20 parameter configurations. Moreover, we report the performance of OPLB as a whole, including the SA procedure (with 500 iterations) as described in Section 1.1.2. The SA Single-Best performance, for example, is obtained by applying SA to the best performing greedy construction configuration (out of the twenty tested) on every test problem instance. Hence the online running time for evaluating all greedy constructions (i.e., 1.8 seconds for 20 configurations) is incorporated in the SA running time.

Note that the loading algorithms in Table 5.5 are highly specialized on relatively unrestricted, academic BR problem instances, hence they are likely to perform less well on more complex, practical problems, whereas OPLB is a generic framework that is ultimately intended for practical problems. Moreover, the loading algorithms in Table 5.5 completely neglect gravity, whereas OPLB enforces at least some support from below. Still, we are able to obtain rather competitive results, and even outperform some of the state-of-the-art techniques with competitive running time.

5.2.2 Performance on Practical Problem Set

In this section, we analyze the performance of OPLB on the practical problem set: a set of large, complex pick-pallet instances from a well-known multinational. Recall that these problem instances are input minimization problems (see Section 2.1.1). Essentially, these problem instances consist of multiple, sequential loading problems. First, the items are loaded onto pick-pallets (i.e., pallets with multiple individual (layer) items in any composition) or stock-pallets (i.e., pallets with a fixed load). Second, the pallets are loaded into the containers. We first confine ourselves to pallet loading, and

Author	BR1-BR7	
	Volume Utility (%)	Computation Time (sec.)
Juraitis et al. (2006)	89.26	-
Moura and Oliviera (2005)	92.65	-
Bortfeldt and Gehring (2001)	90.06	316
Fanslau and Bortfeldt (2010)	95.01	319
Zhang et al. (2012)	95.35	135
OPLB Single Construction Single-Best	83.99	0.06
OPLB Single Construction Best-Five	85.99	0.24
OPLB Single Construction Best-Twenty	87.00	1.8
OPLB with SA Single-Best	88.59	15
OPLB with SA Best-Five	90.24	74

Table 5.5: Performance of known algorithms in the literature and OPLB with its parameters tuned with uRace on weakly heterogeneous BR instances.

	Average Performance		
Parameter Configuration	Pallets	Time (sec.)	# Best
Wall Building approach (George and Robinson, 1980)	47.4	28	0
Horizontal Layer (Bischoff et al., 1995)	47.2	261	0
Configuration Tuned with uRace	44.1	20	10

Table 5.6: Performance of the best parameter configuration, as found with uRace, compared with a priori sensible configurations based on known constructive heuristics on the pallet loading phase on a set of 10 complex, practical problem instances. The last column indicates the number of problem instances on which the given parameter configuration gives the best performance.

subsequently consider the multi-phase loading. We evaluate 10 problem instances and compare the performance of OPLB with its parameters tuned with uRace on the one hand, and OPLB with a priori sensible parameter configurations from the literature on the other hand (the same as in Table 5.4). In this case, we tune the parameter configuration on a training set of similar problem instances from the same multinational, hence we find a completely different parameter configuration than before on the training set of BR instances. That is, by simply imposing a different training set, we are able to customize OPLB to the new types of instances.

The training instances are similar, yet distinct to the instances in the training set. In other words, the tuning time does not count towards the (online) running time of the algorithm. The difference in running time between the configurations is mainly caused by the parameters "SpaceDividingStrategy" and "SpaceAddingStrategy" (see Section 3.1.4), where more spaces lead to a longer running time.

Pallet Loading

In this scenario, the parameter configuration is tuned to minimize the number of pallets that are required to load all items. Table 5.6 tabulates the performances of OPLB with the three alternative parameter configurations. Note that the configuration found with uRace is consistently better on all problem instances in the test set, and results in nearly 10% less pallet usage on average.

Multi-Phase Loading

In this scenario, the parameter configuration is tuned to minimize the number of containers that are required to load all items. As secondary and tertiary objectives (i.e., in case of equal container usage), the tuning process prioritizes solutions with lower pallet usage and higher stability (i.e., the ratio of the top surface versus the bottom surface of the pick-pallets) respectively. To this end, we adjust our fitness

Parameter Configuration	Average Performance			# Best
	Containers	Pallets	Time (sec.)	
Wall Building approach (George and Robinson, 1980)	2.0	47.4	28	0
Horizontal Layer (Bischoff et al., 1995)	2.0	47.2	261	0
Configuration Tuned with uRace	1.7	44.2	22	10

Table 5.7: Performance of the best parameter configuration, as found with uRace, compared with a priori sensible configurations based on known constructive heuristics on a set of 10 complex, practical multi-phase loading problem instances. The last column indicates the number of problem instances on which the given parameter configuration gives the best performance.

function (i.e., optimization criterion of the tuning process), keep the parameter settings that govern the containerization (i.e., the loading of pallets into the container) fixed, and again tune the parameter settings that govern the pallet loading. Table 5.7 tabulates the performances of OPLB with the three alternative parameter configurations. Note that in this case the configuration found with uRace results in slightly higher pallet usage (i.e., 44.2 versus 44.1 in Table 5.6, which comes down to 1 pallet more in one of the ten instances) than the configuration that was tuned explicitly to minimize the pallet usage. In other words, the pallet loading and subsequent container loading cannot be seen as separate, independent processes, and configurations that result in suboptimal pallet loading may ultimately lead to better container usage.

Again, the configuration found with uRace outperforms the other two configurations on all instances. On three out of ten problem instances, we are able to spare a container with uRace, while outperforming the alternative configurations on pallet usage on all problem instances.

5.3 Conclusion

This chapter provided the results of our research. First, we analyzed the behavior of uRace. We showed the relation between invested tuning time and performance gains, showed the merits of the parallel evolution of parameter configuration for a best-few application, and quantitatively supported the intuitive appeal of a representative training set. Moreover, we proved the efficiency of uRace compared to alternative tuning methods. Second, we analyzed the performance of OPLB, with its parameters tuned by uRace. On our academic problem set, we showed that we outperform known construction heuristics in the literature. Moreover, by including the SA procedure and employing a best-few application, we are able to compete with state-of-the-art solution methods with competitive online running time. Finally, on our practical problem set, consisting of large, complex, multi-phase loading problems of a well-known multinational, the parameter configuration tuned by uRace consistently outperforms construction heuristics in the literature, improving average performance with up to 10%.

6 | Conclusions and Recommendations

In Section 1.5, we stated the four research questions to which the answers collectively satisfy our research goal. Each of the subsequent chapters answered one of the research questions. In Chapter 2, we covered what is currently known in the literature in relation to loading methodologies and parameter tuning. In Chapter 3, we analyzed the parameter space of OPLB. We designed our tuning algorithm in Chapter 4, and, finally, we analyzed its performance in Chapter 5. For a quick recap on any of the research questions, we refer to the subconclusion in the corresponding chapter.

In this chapter, we discuss the theoretical and practical implications of our research. We divide our conclusions and recommendations into two parts. First, we take a theoretical perspective, and discuss the theoretical relevance of uRace as an innovative tuning method. Second, we consider the practical relevance of our research, and discuss how it may benefit ORTEC and its customers.

6.1 Theoretical Conclusion

We developed the tuning algorithm uRace, an innovative approach to solve the offline configuration problem, for application to OPLB. However, uRace is more widely applicable than just to our specific research. In this section, we discuss the practicality of uRace in light of other situations. We consider its suitability to other types of tuning and other parameter spaces respectively. Finally, we discuss some ideas to further develop uRace.

6.1.1 Types of Tuning

We developed uRace for the purpose of problem family tuning of a parameterized, deterministic algorithm. However, the applicability of uRace is not necessarily limited to this specific type of tuning. Table 6.1 shows the suitability of uRace for four types of tuning on a Likert scale (Likert, 1932). For problem instance tuning of deterministic algorithms, the performances of parameter configurations are completely deterministic. For this reason, repeated application of the same configuration (be it with a different random number stream or on a different problem instance), and subsequent statistical tests, are redundant, and uRace is not suitable. For the other two types of tuning, however, uRace is a suitable tuning method. For example, as Section 4.1.1 reveals, problem instance tuning of stochastic algorithms is in essence a very similar type of tuning. In uRace, fair comparisons between configurations are enforced by simply applying the configurations to the same problem instances. For problem instance tuning of stochastic algorithms, performance measurement on multiple instances is redundant, since the only concern is the performance of the algorithm on a single problem instance. However, we still have a stochastic process that influences the algorithm's performance, and we may enforce similarly fair comparisons between configurations through the use of Common Random Numbers in the stochastic algorithm by imposing identical, synchronous, random number streams.

	Type of Parameterized Algorithm	
	Deterministic	Stochastic
Problem Family Tuning	++	+-
Problem Instance Tuning	–	+

Table 6.1: Suitability of uRace for different types of tuning on a Likert scale.

For problem family tuning of stochastic algorithms, there are two simultaneous stochastic processes that influence the tuning. Dependent on the application, one may keep one of the two processes fixed (reducing the type of tuning to either problem instance tuning of stochastic algorithms or problem family tuning of deterministic algorithms) or vary both stochastic processes. In the latter case, uRace should be extended with repeated application of the same parameter configurations to the same problem instance.

Besides algorithms, other parameterized stochastic processes can be tuned similarly with uRace. To tune parameterized stochastic processes (e.g., the number of parallel machines used in a production line with stochastic product demand and failures), they are often modeled in a simulation environment, which is in essence an indefinitely long stochastic algorithm that translates a parameter configuration into an estimate value of some objective function. By viewing a measurement of the performance of a stochastic algorithm on one problem instance as one additional time unit of simulation running time, such simulation environments can potentially be tuned very efficiently with uRace, eliminating and generating parameter configurations mid-run, as soon as sufficient statistical evidence is acquired.

6.1.2 Parameter Space

We developed uRace for the purpose of tuning an enormous search space of categorical parameters that show a high degree of synergy, in the sense that it is the collaborative alignment of the entire configuration that results in good performance, rather than the decomposed, optimal settings of every individual parameter. However, the suitability of uRace is not limited to this specific type of parameter space. For parameter spaces of similar (or larger) size as that of OPLB, but with numerical parameters, the generic structure of uRace would still be suitable, perhaps even to a greater extent. The part that would change is the generation of new configurations based on previous measurements: the resampling. The high degree of synergy in our categorical parameter space induced our choice for Biased Sampling Distributions to generate child configurations. If the search space allows it, more sophisticated sequential measurement techniques such as the ones described in Section 2.2.2 can be used instead, which potentially improves the effectiveness of uRace. As long as the sequential measurement technique adheres to the idea of parallel evolution of multiple local optima and an increasingly localized search around the most promising areas of the solution space, uRace would not structurally change with a different child-generation method.

6.1.3 Further Research

In this section, we propose some suggestions to further develop uRace. Recall from Section 1.4 that we decided early on to exclude the SA part of OPLB from our tuning effort. We hypothesized that this would both radically speed up the measurements and increase the efficiency of the learning process (by removing any stochastic noise from our measurements), while achieving similar performance. Indeed, the results confirm that the quality of a single construction solution is a good indication of the quality of the final solution, such that we sacrifice little in terms of performance by our exclusion of SA. Still, it may be the case that some parameter configurations are more susceptible for improvement through SA than others, which we can exploit, at the expense of longer tuning time, by including the SA in our tuning procedure. In that case we would have a stochastic parameterized algorithm, which, as discussed

in Section 6.1.1, requires slight modification of uRace.

Whereas we expect the inclusion of SA in the offline tuning process to lead to modest performance gains, we suspect that the inclusion of SA in an additional *online* tuning process may greatly benefit the performance. This should be seen as an extension to, rather than a modification of uRace. In particular, the best-few application of uRace could be extended with an additional online tuning effort. We can use uRace to drastically reduce the parameter search space of OPLB, identifying a few well-performing configurations (and therefore construction heuristics) for a typical problem instance. Subsequently, we can use this from a hyper-heuristic point of view in which we proceed the offline creation of a well-performing heuristic search space with an online optimization step through iterative heuristic selection. That is, from our small heuristic search space, we aim to select the best heuristics dependent on the specific problem instance and problem state (see Section 2.2.1). This way, we can benefit from the proven idea that some construction heuristics perform better in the first phase of the solution process, whereas others excel as the degree of freedom reduces. We may effectively develop such a hyper-heuristic by employing the already existing SA procedure within OPLB, which surrounds the single construction. Recall that, in this atypical SA procedure, the neighborhood structure is to destruct a random number of iterations, choose a different non-preferred option at random for that iteration, and reconstruct again using *the same* construction heuristic (i.e., parameter configuration). Instead, we could vary, during the solution process of a single instance, the parameter configuration that is used to reconstruct, dependent on which of the heuristics in our small search space is most suitable for the specific problem instance in the current problem state. For example, we can use roulette wheel selection in combination with reinforcement learning as done by Pisinger and Ropke (2007) to continuously select the most suitable reconstruction heuristics based on previous performance.

6.2 Practical Conclusion

In this section, we discuss the practical relevance of our research for ORTEC and its customers. Once OPLB is starting to be utilized by ORTEC's customers, uRace can be used to tune the parameters of OPLB, thereby automatically customizing the algorithmic framework for each specific customer. To this end, a set of representative training instances is required, where, as Section 5.1.3 suggests, the representativeness of the training set benefits the performance of the tuned parameter configuration. Typically, a set of the customer's (representative) historic problem instances can be used as our training set. Alternatively, if such a historic set is not available, a classification of customer types can be used to pinpoint parameter configurations that are known to perform well for such customer types.

Moreover, the relevance of our research is not limited to the implementation phase of OPLB. Instead, we may use uRace to periodically update the parameter configuration at the customer site, based on evaluation of more recent, more representative historic problem instances. In other words, the parameter configuration, and therefore the loading algorithm would automatically adjust to the most recent developments at the customer site. Such a periodic update could be initiated by ORTEC, as part of a maintenance service, but could also be left to the customer. Either way, it would typically occur at a time when computational resources are amply available (e.g., overnight or in the weekends). The periodic update may consist of a complete restart of the tuning process (i.e., start with an LHD as discussed in Section 4.2.3) or uRace can be fed with the best known configurations on that (or a similar) training set. In the latter case, the search for the best parameter configuration will typically be less explorative, and is therefore more suitable for shorter tuning times. Furthermore, the use of uRace is not limited to the tuning of one type of parameter configuration per customer. Ideally, the customer would anticipate the type of problem instances they will face in the future (e.g., tomorrow), examine their database for

historic problem instances that are similar (e.g., orders with similar products from a similar season last year), and let uRace run, overnight, on that training set. In the morning there will be a brand new, automatically customized algorithm, ready to facilitate an efficient, customized loading process for ORTEC's customer.

Since the tuning time of uRace can easily be regulated, customers are free to decide how much computation time they want to invest (i.e., the offline running time-performance tradeoff). Similarly, customers may make their own online running time-performance tradeoff by choosing how many of the best parameter configurations to assess on the online problem instance (i.e., single-best or best-few).

Finally, there are other software solutions at ORTEC, covering other OR challenges, that possess a similar parameterized algorithmic framework. Although further research should go into the robustness of parameter configurations on those problem spaces (and therefore the suitability of problem family tuning), and the nature of the parameter space (and therefore the applicability of our resampling method), they can potentially also be efficiently configured with uRace.

Bibliography

Adenso-Diaz, B. and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54:99–114, 2006.

Ai, M., H. Yanzhen, and S. Liu. Some new classes of orthogonal latin hypercube designs. *Journal of Statistical Planning and Inference*, 142, 2012.

Balaprakas, P., M. Birattari, and T. Stutzle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. *Lecture Notes in Computer Science*, 183:108–122, 2007.

Barr, R., B. Golden, J. Kelly, M. Rescende, and W. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1:9–32, 1995.

Bechhofer, R.E. A single-sample multiple decision procedure for ranking means of normal populations with known variances. *The Annuals of Mathematical Statistics*, 25(1):16–39, 1954.

Bilgin, B., E. Ozcan, and E.E. Korkmaz. An experimental study on hyper-heuristics and final exam scheduling. In *Proceedings of the International Conference on the Practice and Theory of Automated Timetabling (PATAT'06)*, pages 123–140, 2006.

Birattari, M. *Tuning metaheuristics: A machine learning perspective*, volume 197 of *Studies in Computational Intelligence,*. Springer, 2009.

Birattari, M., T. Stutzle, L. Paquette, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann, 2002.

Bischoff, E.E. and M.S.W. Ratcliff. Issues in the development of approaches to container loading. *Omega International Journal of Management Sciences*, 23:377–390, 1995.

Bischoff, E.E., F. Janetz, and M.S.W. Ratcliff. Loading pallets with non-identical items. *European Journal of Operational Research*, 84:681–692, 1995.

Bortfeldt, A. and H. Gehring. A hybrid genetic algorithm for the container loading problem. *European Journal of Operations Research*, 131:143–161, 2001.

Bortfeldt, A., H. Gehring, and D. Mack. Constraints in container loading - a state-of-the-art review. *European Journal of Operations Research*, 229:1–20, 2013.

Burke, E.K., M.R. Hyde, G. Kendall, and J.R. Woodward. Automating heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 07)*. Springer, 2007.

Burke, E.K., M.R. Hyde, G. Kendall, and J.R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In *Collaborative Computational Intelligence*. Springer, 2009.

Burke, E.K., M.R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J.R. Woodward. *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*. Springer US, 2010.

Buttrey, S.E. Nearest-neighbor classification with categorical variables. *Computational Statistics & Data Analysis*, 28:157–169, 1998.

Chang, H., M. Fu, J. Hu, and S. Marcus. *Simulation-Based Algorithms for Markov Decision Process*. Springer, Berlin, 2007.

Conover, W.J. *Practical Nonparametric Statistics*. John Wiley & Sons, 1999.

Cowling, P., G. Kendall, and E. Soubeiga. Third national conference on the practice and theory of automated timetabling (patat 2000). In *Lecture Notes in Computer Science*, pages 176–190. Springer, 2000.

Das, S. and D. Ghosh. Binary knapsack problems with random budgets. *Journal of the Operations Research Society*, 54:970–983, 2003.

Dowsland, K.A., E. Soubeiga, and E.K. Burke. A simulated annealing hyper-heuristic for determining shipper sizes. *European Journal of Operations Research*, 179(3):759–774, 2007.

Dyckhoff, H. A typology of cutting and packing problems. *European Journal of Operations Research*, 44:145–159, 1990.

Eiben, A.E. and S.K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1:19–31, 2011.

Epstein, L. and R. Steevan . Online bin packing with resource augmentation. *Discrete Optimization*, 4: 322–333, 2007.

Fanslau, T. and A. Bortfeldt. A tree-search algorithm for solving the container loading problem. *INFORMS Journal on Computing*, 22:222–235, 2010.

Frazier, P.I., W. Powell, and S. Dayanik. A knowledge-gradient policy for sequential information collection. *SIAM Journal on Control and Optimization*, 47(5):2410–2439, 2008.

George, J.A. and D.F. Robinson. A heuristic for packing boxes into a container. *Computers & Operations Research*, 7:147–156, 1980.

Gonçalves, J.F. and M.G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research*, 39:179–190, 2012.

Greffenstette, J.J. Optimization of control parameters for genetic algorithms. *IEEE Transactions Systems, Man, and Cybernetics*, 16:122–128, 1986.

Horst, R. and P.M. Pardalos. *Handbook of Global Optimization*. Kluwer Academic Publishers, 1995.

Jones, D.R., M. Schonlau, and W.J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

Juraitis, M., T. Stonys, A. Starinskas, D. Jankauskas, and D. Rubliauskas. A randomized heuristic for the container loading problem: Further investigations. *Information Technology and Control*, 35(1): 7–12, 2006.

Keller, R.E. and R. Poli. Cost-benefit investigation of a genetic-programming hyperheuristic. In *Proceedings of Artificial Evolution (EA'07)*, pages 13–24, 2007.

Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science, New Series*, 220:671–680, 1983.

Law, A.M. and D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, 2000.

Likert, R. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, 1932.

Lizotte, D.J. *Practical Bayesian Optimization*. PhD thesis, University of Alberta, 2008.

Maron, O. and A. Moore. The racing algorithm: model selection for lazy learners. *Artificial Inteligence Review*, 11:193–225, 1997.

McKay, M.D., W.J. Conover, and R.J. Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 221:239–245, 1979.

Mercer, R.E. and J.R. Sampson. Adaptive search using a reproductive metaplan. *Kybernetes (The International Journal of Systems and Cybernetics)*, 7:215–228, 1978.

Mes, M.R.K., W.B. Powell, and P.I. Frazier. Hierarchical knowledge gradient for sequential sampling. *Journal of Machine Learning Research*, 12:2931–2974, 2011.

Moura, A. and J.F. Oliviera. A grasp approach to the container-loading problem. *IEEE Intelligent Systems*, 20:50–57, 2005.

Nannen, V. and A.E. Eiben. A method for parameter calibration and relevance estimation in evolutionary algorithms. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 06)*. Morgan Kaufman, 2006.

Ngoi, B.K.A., M.L. Tay, and E.S. Chua. Applying spatial representation techniques to the container packing problem. *International Journal of Production Research*, 32:111–123, 1994.

Pisinger, D. Heuristics for the container loading problem. *European Journal of Operations Research*, 141:382–392, 2002.

Pisinger, D. and S. Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34:2403–2435, 2007.

Powell, W.B. The knowledge gradient for optimal learning. *Wiley Encyclopedia of Operations Research and Management Science*, pages 2931–2974, 2010.

Ross, P., S. Schulenburg, J.G. Marin-Blázquez, and E. Hart. Hyper-heuristics: learning to combine simple heuristics in bin-packing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pages 176–190. Morgan-Kauffman, 2002.

Sani, F. and J. Todman. *Experimental Design and Statistics for Psychology: A First Course; Appendix 1*, volume 30. Wiley, 2006.

Terno, J., G. Scheithauer, and U. Sommerweiß. An efficient approach for the multi-pallet loading problem. *European Journal of Operations Research*, 123:372–381, 2000.

Wäscher, G., H. HauBner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operations Research*, 183:1109–1130, 2007.

Wilcoxon, F. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1:80–83, 1945.

Zhang, D., Y. Peng, and S.C.H. Leung. A heuristic block-loading algorithm based on multi-layer search for the container loading problem. *Computers & Operations Research*, 39:2267–2276, 2012.