Contextual Hyperparameter Optimization for the Train Unit Shunting Problem

Leon van der Knaap

Technische Universiteit Delft

TU Delft

Contextual Hyperparameter Optimization for the Train Unit Shunting Problem

by

Leon van der Knaap

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday June 30, 2021 at 11:00 AM.

Student number:	4972376	
Project duration:	September 10, 2020 – June 30, 2021	
Thesis commitee:	Dr. M. M. de Weerdt,	TU Delft, supervisor
	Dr. ir. L. Bliek,	TU Eindhoven, supervisor
	Dr. ir. S. E. Verwer,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Cover photo: NS Merkplaats

TU Delft

Abstract

One of the planning problems encountered by the Dutch Railways (NS) at shunting yards is the train unit shunting problem (TUSP). This problem considers idle train units that have to be parked, be cleaned, undergo regular maintenance, and be reconfigured into the scheduled departing train compositions. A local search algorithm is being developed to solve this problem at all shunting yards in the Netherlands.

This research considers the optimization of the hyperparameters of this local search algorithm. We take contextual information, based on features of problem instances, into account to construct a mapping from contexts to hyperparameter configurations. General difficulties when considering hyperparameter optimization are the expensive function evaluations of the target algorithm that place a tight budget on the number of possible configurations to query; and noisy observations due to the stochasticity of this algorithm. Therefore, we resort to using Bayesian optimization to construct a surrogate that models our belief of the unobservable objective function. Additional encountered difficulties for this specific problem are heterogeneous problem instances and the underlying problem to be a constraint satisfaction problem (instead of the more commonly studied optimization problem). We show that these difficulties prevent an instance-specific approach and result in a poor generalization from the performance on the training set to test data. We also show that we can improve the performance of the default hyperparameter configuration by proposing strategies to construct portfolios consisting of different configurations.

Preface

This master's thesis project has been written as the final fulfillment of the requirements of the degree of Master of Science in computer science at the Delft University of Technology. The project describes my research of the previous 10 months on the application of contextual hyperparameter optimization for the train unit shunting problem in collaboration with the Nederlandse Spoorwegen.

First of all, I would like to thank Mathijs de Weerdt and Laurens Bliek for all the time that they invested and for the valuable guidance they provided as my supervisors during this project. Laurens, I want to thank you for guiding me with your expertise in the field of surrogate modeling and for helping me with the technical details, while allowing me the freedom to explore my own directions. Mathijs, thank you for continuously encouraging me to think outside the box by drawing parallels with other fields of research and for helping me in clearly formulating my thoughts in a structural manner. Also, I would like to thank Sicco Verwer for being on my thesis committee.

I want to thank Bob Huisman for giving me the opportunity to combine my final academic project with an internship at the Nederlandse Spoorwegen and for helping me set realistic expectations. In addition, I would like to thank all the colleagues of the R&D department for their interest and cooperation.

Finally, I want to show my appreciation to my roommates and my girlfriend for keeping me sane in a time where we are forced to stay home for an excessive amount of time. Also, thanks to my fellow students who showed enthusiasm in my work and who helped me with the issues that I encountered. I especially want to express my gratitude to my parents for all their support and motivation.

Leon van der Knaap
Rotterdam, June 2021

Contents

1 Introduction

The NS (Nederlandse Spoorwegen; Dutch Railways) posses a large fleet of trains that can be deployed to transport a great number of daily passengers. This number of passengers peaked at 1.3 million in 2019. To meet the commuters' demand, most trains are deployed during the morning and evening rush, while only a few trains are used at night. Idle trains are relocated from the railroad network to shunting yards that also function as service sites. Here, multiple tasks need to be performed before the following shift. The arriving train units have to be parked, be cleaned, undergo regular maintenance, and be reconfigured into the scheduled departing train compositions. The resulting planning problem is referred to as the train unit shunting problem (TUSP). Every day, an instance of the TUSP needs to be solved at every one of the 35 service sites in the Netherlands. These encountered problems are currently being solved by human planners at their respective sites. To aid these planners, algorithms are being developed to solve varying mathematical formulations of the TUSP. In this thesis, we consider a local search method that was introduced by van den Broek (2016) as the heuristic method to solve the TUSP. An implementation of this local search method is currently in a far stage of development. This local search algorithm aims to solve a satisfaction problem version of the TUSP by finding any feasible solution. That is, a complete shunting plan of movements from arriving train units to departing train compositions, that performs all scheduled tasks, while not conflicting with a predefined set of constraints.

Every large algorithm has its set of hyperparameters: configurations that control the 'behavior' of the algorithm. For example, the hyperparameters of a convolutional neural network consist of the number and the shape of the filters, the number of hidden layers, the learning rate, and the weight initialization. When we restrict to local search algorithms, we can consider what neighborhoods to include, the initial solution, and the technique of local search to use, such as simulated annealing, variable neighborhood search or tabu search. Subsequently, we can select the initial temperature and the decreasing factor for simulated annealing; the neighborhood sizes and allowed fitness reduction for variable neighborhood search; or the tabu tenure for tabu search.

All these hyperparameters are fixed during the execution of the algorithm, but can potentially influence both the run-time and the outcome of the calculations. Therefore, the overall performance of an algorithm is dependent on the joint configuration of its hyperparameters. The problem of finding an optimal configuration of hyperparameters is especially popular within the machine learning domain (e.g. Bergstra et al., 2011; Snoek et al., 2012; Kotthoff et al., 2019; Hinz et al., 2018). However, it has also been successfully applied within fields such as reinforcement learning (Brochu et al., 2010); robotics (Martinez-Cantin et al., 2007); lasers (Duris et al., 2020); and state of the art solvers for Mixed Integer Programming (MIP) problems (Hutter et al., 2011; Wang et al., 2013).

The first problem of finding a good configuration of hyperparameters for a target algorithm is that there is no known representation of the objective function that defines the performance of the algorithm as a function of the configuration. This is often called a *black-box* function, and we can only evaluate this function by querying a certain configuration and obtaining a noisy output value. The second problem is the time-intensive evaluation of every single queried configuration. The target algorithm, of which the computation time can range from a couple of seconds up to multiple days,

must be executed in order to obtain a single evaluation. Early stopping might be possible in some cases, but affects the reliability of the evaluation (Falkner et al., 2018).

As a result, we have a tight budget on the number of evaluations of hyperparameter configurations by the target algorithm. Consequently, intuitive methods such as grid search become intractable for even small numbers of hyperparameters. Also, many popular learning algorithms such as artificial neural networks or regression models cannot be properly trained.

A popular method that can overcome these problems is Bayesian optimization (BO). This technique makes use of a surrogate model: an evolving representation of our current belief of the unknown black-box function. This surrogate is used in an iterative process, where we use the surrogate model to find a new promising hyperparameter configuration in the search space; and update the surrogate based on the newly received evaluation of the queried configuration. A common application of using BO for hyperparameter optimization is to consider instance-specific problems. For example: we want to find the hyperparameter configuration of a machine learning classification algorithm that maximizes the performance on a single dataset. The classifier can be executed for every given configuration and the performance on the particular dataset can be measured by a loss function. This allows for a simple setup to find this optimal configuration. However, this does not guarantee whatsoever that this configuration works well on other datasets (Bardenet et al., 2013). In general, it is not efficient to use the same obtained hyperparameter configuration for all problem instances, given a wide variety of possible instances. Unfortunately, the time limits prevent us from finding an optimal configuration for every newly encountered problem instance.

Therefore we aim to find hyperparameter configurations based on problem features, also referred to as *contextual information*, or simply *contexts*. For the train unit shunting problem, we can consider the number of train units, the number of scheduled tasks, or the number of required train reconfigurations in a problem instance as examples of such contexts. By considering different problem instances with similar problem features, we search for configurations that perform well on certain contexts. We then construct a mapping from contexts to optimal hyperparameter configurations. When we want to efficiently solve a problem instance that we have never encountered before, we can observe the context of this problem and obtain a corresponding configuration from this mapping.

One of the main distinctions between our studied problem setting and problems that are commonly studied in this research field is that we consider the underlying problem version to be a satisfaction problem. The target algorithm that solves this problem consists of an artificial objective function, where the weight coefficients in this function do not represent a 'ground truth'. However, these weights do strongly affect the outcome of the algorithm. In this thesis, we focus on the weight coefficients as the hyperparameters to optimize. We show that the commonly studied instance-specific approach is not attainable for the satisfaction problems in general. Therefore, we resort to a contextual optimization as a more general approach by considering similar problem instances. We formulate the main research question as follows:

1. How can we use a contextual approach to find optimal hyperparameter configurations for the target algorithm for different instances of a realistic satisfaction problem based on the context of the problem instances?

We mentioned that Bayesian optimization is a method that can model a black-box function well with only a small number of possibly by noise corrupted observations of the unobservable function. Consequently, this method is deemed to be the most promising application for the hyperparameter optimization problem according to the majority of literature. Therefore, in this research, we limit the methods to tackle this problem to Bayesian optimization. We consider satisfaction problem formulations of two different problems: the train unit shunting problem and the bin packing problem. The bin packing problem is a subproblem of the TUSP that is defined by a more concise formulation that structurally still resembles the formulation of the TUSP. The instances of this problem are more basic which makes the bin packing problem easier to manipulate.

The goal of this thesis is to research the application of Bayesian optimization to find hyperparameter configurations for the local search algorithm that solves the train unit shunting problem, based on the context of problem instances, that optimize the performance over all problem instances. Here, we define the performance as the number of instances for which the local search algorithm can find a

feasible solution. We aim to answer the presented main research question by considering the following sub-questions:

2. Does Bayesian optimization in a contextual approach find hyperparameter configurations for different contexts that obtain a better performance compared to a context-free approach?
3. How does the performance of hyperparameter configurations that are obtained by a context-free Bayesian optimization approach compare to the performance with the default configuration in a realistic satisfaction problem?
4. How much can the performance of optimal hyperparameter configurations be improved when considering a contextual Bayesian optimization approach compared to a context-free approach in a realistic satisfaction problem?

We can positively answer the first sub-question if we can obtain a better performance for a contextual approach on any problem. Therefore, we opt for the more basic bin packing problem that we have more control over. Since we manipulate the bin packing problem, we cannot categorize it as a *realistic* satisfaction problem. The remaining sub-questions focus on a quantification of the difference in performance. For such a quantification, we are only interested in realistic problem instances. Hence, we restrict ourselves to the train unit shunting problem. To answer the last question, we consider multiple components of contextual Bayesian optimization, such as the choice on how much and which contexts to include, the strategy to select problem instances to be used when evaluating configurations, and the distribution of the available budget over multiple instances, the number of iterations and the stopping conditions.

Finally, we consider the hyperparameter optimization problem for the target algorithm that solves instances of the TUSP from a more practical point of view. We are interested in whether we can use the knowledge that we acquired from this research in practice to improve the performance of the algorithm in its current deployment state. Therefore, we tackle the final question:

5. Can we find hyperparameter configurations that lead to a better performance of the local search target algorithm that solves the TUSP in the current strategy that is deployed by the NS?

The contribution of this thesis is threefold:

- we address difficulties of using Bayesian optimization when the underlying problem is a satisfaction problem instead of an optimization problem;
- we present a contextual approach to find hyperparameter configurations for the local search algorithm and compare its performance to a context-free approach and to the default configuration;
- we propose to extend the currently existing strategy of solving a single problem instance with multiple instantiations of the local search algorithm in parallel by introducing a *portfolio approach*. A portfolio denotes a set of hyperparameter configurations such that different configurations are considered simultaneously when solving a single problem instance.

The outline of the remainder of this thesis is as follows. In Chapter 2, we present a literature review on the subjects of hyperparameter optimization, surrogate models and contextual Bayesian optimization. Chapter 3 contains a formal problem formulation and background information on Bayesian optimization using Gaussian processes. In Chapter 4, we discuss the application of Bayesian optimization to satisfaction problems. Next, in Chapter 5, we introduce the search space as a subspace of the configurations space and we present the performance metric to evaluate the performance of a hyperparameter configuration. Then, in Chapter 6, we introduce the setup and results of the experiments on the bin packing problem. The experiments on the train unit shunting problem are discussed in Chapter 7. Finally, we summarize the main conclusions of this thesis and suggest directions of further research in Chapter 8.

2

Literature review

In this chapter, we present a brief history of approaches that have been used for hyperparameter configuration. This is followed by an overview of existing literature regarding surrogate modelling, (contextual) Bayesian optimization and hyperparameter optimization for algorithms that solve satisfaction problems.

2.1. Strategies for hyperparameter optimization

Deciding on suitable hyperparameter configurations for your algorithm has been a well-known problem that has been limited by the expensive evaluation of single configurations. Therefore the standard has been a combination of manual search to find the most promising region of configurations, extended by small-sized grid search for some hyperparameter tuning (Larochelle et al., 2007). Bergstra et al. (2013b) describe the importance of finding hyperparameter configurations by claiming that posing the question of "how good is this model on that dataset?" is imprecise. Instead, you should ask about the quality of this model with the best configuration that can realistically be found on a dataset. In addition, they describe the benefits of automated hyperparameter optimization approaches over manual or grid search. Due to the continuously increasing computing power, this optimization problem has become a more realistic direction to explore. The tight budget on the number of possible function evaluations due to the expensive evaluation of each configuration, however, will remain an issue of this problem. Simple approaches, such as random search (Bergstra and Bengio, 2012) can avoid this problem and are still being used as a benchmark for more sophisticated methods. Other quite simple techniques that have been introduced include the local search algorithm ParamILS (Hutter et al., 2009) or the racing algorithm F-Race (Birattari et al., 2010). These are examples of model-free approaches that do not incorporate the full observation history. Therefore, these more simple approaches might not be the most suitable to handle the tight budget on the number of observations.

A popular strategy that does not require a large number of observations is Bayesian optimization (Brochu et al., 2010). Bayesian optimization describes a technique that places a prior distribution, that captures the current belief, over the unknown black-box function. It then uses this prior to construct a posterior distribution over the function after every new obtained function evaluation. This method thus iteratively updates a surrogate model that represents our belief of the unknown function based on the observed data. We discuss Bayesian optimization in more detail in Section 3.2. There are multiple methods that can be used to construct these posterior distributions. The most common method is based on Kriging models (first introduced by Krige (1951); later popularized by Metheron (1971)) that uses a Gaussian Process (GP; Williams and Rasmussen, 2006) as the surrogate model. The attractiveness of a GP is due to the consistency requirement; it is *closed under sampling.* Simply put: if we use a GP as a prior on f, then the posterior distribution of f given a set of samples of $\{(x_i, f(x_i)), i = 1, \ldots, N\}$ is still a GP. In every step, the updated posterior distribution is then used to find a new region in the search space to use as the next query, by optimizing an acquisition function (also called *infill criterion*). The second advantage of GPs is that their statistical definition can be used to analytically derive such acquisition functions. Some of the most popular and widely used acquisition functions are the probability of improvement (PI; Kushner, 1964) the expected improvement (EI; Jones et al., 1998), or

the Gaussian Process Upper Confidence Bound (GP-UCB; Srinivas et al., 2009), which all have been shown to perform well (Bull, 2011), even when combined in a portfolio (Hoffman et al., 2011). These acquisition functions balance the notions of exploitation and exploration over the search space. Hence, the optima of these functions are located where the prediction of the surrogate model is both large and uncertain. We provide more details on Gaussian processes and acquisition functions in Section 3.3.

One of the first popular Bayesian optimization based algorithms that was aimed to solve black-box optimization problems with limited function evaluations was the Efficient Global Optimization (EGO) algorithm from Jones et al. (1998). This algorithm uses a GP process and EI as the acquisition function. Bartz-Beielstein et al. (2005) are the first to use an approach that closely resembles EGO to the hyperparameter configuration problem by introducing the Sequential Parameter Optimization (SPO) approach. The application of the EGO algorithm, however, is limited to algorithms that consist of only continuous hyperparameters and that have a noise-free function. Subsequently, Huang et al. (2006) introduce the Sequential Kriging Optimization (SKO) method that can also account for noisy observations. Following this, a lot of research has been able to show that Bayesian optimization can successfully be applied to the hyperparameter configuration problem (e.g. Hutter et al., 2010; Snoek et al., 2012; Bergstra et al., 2013b; Berkenkamp et al., 2016).

Another strategy of Bayesian optimization that we describe is the work of Bergstra et al. (2011). The authors introduce a tree of Parzen estimators (TPE), that divides samples with large and small loss over two different distributions from which to sample. Whereas Gaussian processes are surrogate models that model the belief of $p(y|x)$, TPE models and samples from $p(x|y)$ and $p(y)$ which is similar given Bayes' theorem. TPE is formulated to handle more general search spaces than GPs, including categorical and conditional spaces. The publicly available library for TPE, *Hyperopt* (Bergstra et al., 2013a), has often been used to find suitable hyperparameter configurations for many (machine learning) models (e.g. Yoo et al., 2017; Lv et al., 2020). For an extensive survey on the topic of Bayesian optimization, we refer to Shahriari et al. (2015).

The technique of Bayesian optimization in surrogate modelling can be generalized to sequential model-based optimization (SMBO). The two main ingredients of an SMBO technique are a surrogate model that has an exact mathematical formulation that allows for cheap optimizations; and an acquisition function. We discussed Bayesian optimization using Gaussian processes and TPE as some of the popular techniques. Alternatively, successful surrogate modelling techniques include piece-wise linear surrogate models (Bliek et al., 2020a,b) and random forest models (Hutter et al., 2010). The piece-wise linear surrogate models use local optimization and perturbation as acquisition function. Their main advantages are the possibility to scale over many samples (Bliek et al., 2020a) and the ability to handle a high dimensional and mixed-variable space while still maintaining interactions between all variables, as demonstrated by Bliek et al. (2020b). The Sequential Model-based Algorithm Configuration (SMAC) method, introduced by Hutter et al. (2010), uses the flexibility of decision trees that can handle discrete, categorical and conditional variables, by using a random forest surrogate model. This surrogate modelling technique is specifically designed for the hyperparameter configuration (or *algorithm configuration*) problem and yields good performances on a wide range of algorithms (Hutter et al., 2013). Due to the discontinuity and the indifferentiability of the response surface of trees and forests, SMAC has to rely on local and random search to maximize its acquisition function.

2.2. Contextual Bayesian optimization

The field of hyperparameter configuration optimization by using surrogate models has evolved by, not only considering a wide range of surrogate models and acquisition functions, but also by making distinctions over the input space. It is often possible to identify different contextual information about the problems to solve. The correlation of different problems with similar contextual information, or simply *context* can be used by adding the context dimensions to the input space of the surrogate model. Terms that are also frequently used in literature include: *profiles*, *functions*, *features* and especially *tasks*.

One of the early works of extending Gaussian processes to a contextual setting has been done by Bonilla et al. (2008). Since then, Krause and Ong (2011) used these Gaussian processes in a contextual bandit problem. The aim here is to optimize over a (possibly finite) time horizon in an online setting by choosing an action, after observing a context, per time step. Swersky et al. (2013) extended contextual Gaussian processes to the framework of Bayesian optimization. This research also considers the problem of hyperparameter tuning for machine learning models, and hence they focus on the per-

formance of the best obtained configuration in their history of samples, instead of the summation of performance over a time horizon. They show that differentiating over different contexts and taking the correlation between these contexts into account, can significantly speed up the process compared to a context-free approach. However, their research was aimed to find a configuration that performed well on all contexts, instead of finding a mapping from individual contexts to suitable configurations. A similar setup is also considered by Hutter et al. (2010), who introduced SMAC, which differentiates between different problem instances, but aims to find a single algorithm configuration for all instances.

The concept of contextual Bayesian optimization has been applied in several closely related optimization problems. In multi-fidelity optimization problems (Huang et al., 2006; Forrester et al., 2007), one aims to find the optimum of a high fidelity model, from which it is expensive to sample. Cheaper samples can be obtained from lower-fidelity models that, in combination with the correlation between the fidelity models, can be used to construct a surrogate model of the high fidelity model. Most notably, Klein et al. (2017) combine multi-fidelity with the contextual Bayesian optimization approach of Swersky et al. (2013) by considering the percentage of the number of datasets that are used in the subset of instances over which the performance is evaluated as the context dimension. The goal is to find the optimal configuration for the complete number of datasets. Moss et al. (2020) derive a contextual entropy search acquisition function for a Gaussian process over the joint space of variables and fidelities.

Furthermore, contextual Bayesian optimization can be used when considering different contexts sequentially. Bardenet et al. (2013) consider the problem of finding different configurations for a given classification algorithm when applied to different datasets. They introduce a ranking procedure that uses the pairwise preferences over configurations for previously considered datasets, in order to quickly find a similar configuration for a new dataset. Continuing on this, Yogatama and Mann (2014) avoid the construction of the rankings and manage to come up with a smaller time complexity by assuming an online approach for encountering new datasets. Perrone et al. (2018) use Gaussian processes and introduce a model with a linear time complexity to *warm-start* a new GP, by making use of previously trained GPs. This procedure avoids the cubic run-time that is required in Bayesian optimization, which makes it unattractive to initialize the model when there is already available data.

Finally, we turn to methods that include the combination of finding distinct configurations for every context, but where the contexts do not need to be considered sequentially. Bardenet et al. (2013) also introduce the surrogate-based collaborative tuning (SCoT) algorithm. Here, they apply their ranking procedure to multiple datasets simultaneously, by spending one iteration on each problem in turn. Dai et al. (2020) use a similar approach and extend the GP-UCB acquisition function to a Multi-Task GP-UCB (MT-GP-UCB). This closely resembles the contextual GP-UCB (CGP-UCB) from Krause and Ong (2011). However, Dai et al. (2020) find a configuration for every context, and do so by considering all contexts simultaneously in a round-robin fashion.

To conclude, we consider methods that consider the contexts simultaneously, but not even necessarily in turn. Ginsbourger et al. (2014) are the first to include the context dimensions of the joint space in the acquisition function when finding a conditional mapping from contexts to optimal actions. They then extend the popular EI acquisition function to the profile EI (PEI) criterion, where they assume a uniform distribution over the contexts. Pearce and Branke (2018) describe an optimization framework that does not require this uniform distribution assumption and introduce the REVI and CLEVI sampling procedures. Finally, Char et al. (2019) present a contextual variant of Thompson sampling in an offline setting for discrete variables with a single Gaussian process over the joint space of contexts and actions. However, in their subsequent research (Chung et al., 2020) where their method is applied to a nuclear fission problem, they only consider a small finite set of contexts and model each context with an independent GP. This problem description is the most similar to the problem that we consider in this research.

2.3. Hyperparameter optimization for algorithms that solve satisfaction problems

We conclude this chapter with a discussion of research on the optimization of hyperparameters for algorithms that solve constraint satisfaction problems (CSP). The underlying problem that is solved by the target algorithms being a CSP forms the biggest difference between the problem that we study

and any research that we discussed so far in this chapter.

The main application of research on hyperparameter optimization of algorithms that solve satisfaction problems using Bayesian optimization or similar techniques are on solvers for the Boolean Satisfiability problem (SAT) (e.g. Xu et al., 2008; Lindauer et al., 2017). The methods in this research are validated on SAT instances that have been ensured to be able to be solved in similar ranges of computation times. This implies that heterogeneity in the difficulty over the training instances is not an issue. The study that introduces SMAC (Hutter et al., 2010) experimentally validates their new algorithm by optimizing the hyperparameter configurations of a local search method that solves SAT instances. This local search algorithm contains only 4 hyperparameters of which none are part of the objective function. More notably, they randomly sample a subset of problem instances per iteration of the optimization procedure, also implying that there is no heterogeneity in the difficulty of the training instances. These studies consider satisfaction problems that closely resemble optimization problem instances by restricting to satisfiable instances and measuring the performance of configurations based on the time until the SAT solver finds a feasible solution. Our research differs by being limited to realistic problem instances that may not be satisfiable and strongly vary in the required time to find a feasible solution, if such a solution exists, due to the heterogeneity between the instances. We are interested in finding hyperparameter configurations that maximize the number of problem instances that can be solved, whereas the aforementioned research aims to find configurations that find feasible solutions in the least amount of time.

The only research that we were able to find that considers the tuning of hyperparameters in the objective function of a local search target algorithm that solves instances of a satisfaction problem is the breakout method from Morris (1993). This method iteratively changes the weight coefficients in the objective function of a local search heuristic that is applied to a CSP. The objective function is defined as a weighted sum of constraint violations. The local search algorithm is iteratively called with a restart in a randomly chosen point of the search space. Every time the algorithm is executed and gets stuck in a local optimum, all the weights that correspond to violated constraints are increased. The objective values of the obtained local optima are never compared to actively search for configurations of weight coefficients that perform well. This method can hence be categorized as an intuitive model-free approach for the hyperparameter optimization of a local search problem for a CSP. Such model-free approaches have been shown to consistently be outperformed by model-based approaches. Therefore, we consider Bayesian optimization as a model-based approach to study the optimization of hyperparameters in the objective function of a local search target algorithm.

3

Problem description

In this chapter, we formally present the problem at hand and introduce the techniques that we use to solve this problem. These techniques are the main ingredients in the method that we use to answer the research questions about the hyperparameter configuration problem.

We first present the mathematical problem formulation in Section 3.1. We follow by providing some background information on the topic of Bayesian optimization using a Gaussian process as a surrogate model in Sections 3.2 and 3.3.

3.1. Problem formulation

We consider the problem of finding a hyperparameter configuration for a target algorithm $\mathcal{A}$ with a configuration space Θ that consists of d tuneable hyperparameters. We are given a set of problem instances $\mathcal{I}$, where each instance $i \in \mathcal{I}$ has an observable context $z \in Z$, and where $z_i = (z_{i1}, \ldots, z_{im})$ denotes the context values of instance i's m context features.

We want to find a configuration $\theta = (\theta^1, \ldots, \theta^d) \in \Theta$ that optimizes the performance of $\mathcal{A}$ on instances with context z, for every $z \in Z$, by optimizing the function $f : \Theta \times Z \to \mathbb{R}$ that represents the performance of target algorithm $\mathcal{A}$. The output of this function f can represent any metric on the performance, such as the time to find the obtained solution or some measure on the quality of the final solution.

We assume that no mathematical expression is known for the objective function $f(\theta, z)$ and that we can only access this function through function evaluations. Every evaluation of function f requires some expensive resource. In this research, we consider this resource to be time in the form of long required computation times of the target algorithm $\mathcal{A}$. As a result, we are limited to a small number of function evaluations. The evaluations of function f are corrupted by noise due to stochasticity in algorithm $\mathcal{A}$ and observed as $y = f(\theta, z) + \varepsilon$. Here, $\varepsilon \in \mathbb{R}$ is assumed to be an independent and identically distributed zero mean random noise variable $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$, with $\mathcal{N}$ denoting a normal distribution.

The aim is to find a mapping $h : Z \to \Theta$ from every context to the configuration for which $\mathcal{A}$ yields the optimal performance. The performance of this mapping can be defined as

$$\sum_{z \in Z} f(h(z), z)\, w(z). \tag{3.1}$$

Here, $w(z)$ denotes some positive weight over context z such as the density function over the context space. In this research, we consider a uniform density function such that $w(z)$ is equal for all z. The definition for the performance using this equation with a uniform density function simply results in the average performance of the configuration per context, over the set of contexts. Furthermore, we only consider a discrete set of contexts. In case the set of contexts is continuous, the summation in Equation (3.1) is replaced with an integral.

We can sample from function f iteratively such that we can choose a new configuration θ_{j+1} to query after having observed $y_j = f(\theta_j, z_j) + \varepsilon_j$. We keep track of all n so far queried configurations and contexts, and their received function evaluations in an observation history $\mathcal{D}_n = \{(\theta_1, z_1, y_1), \ldots, (\theta_n, z_n, y_n)\}$.

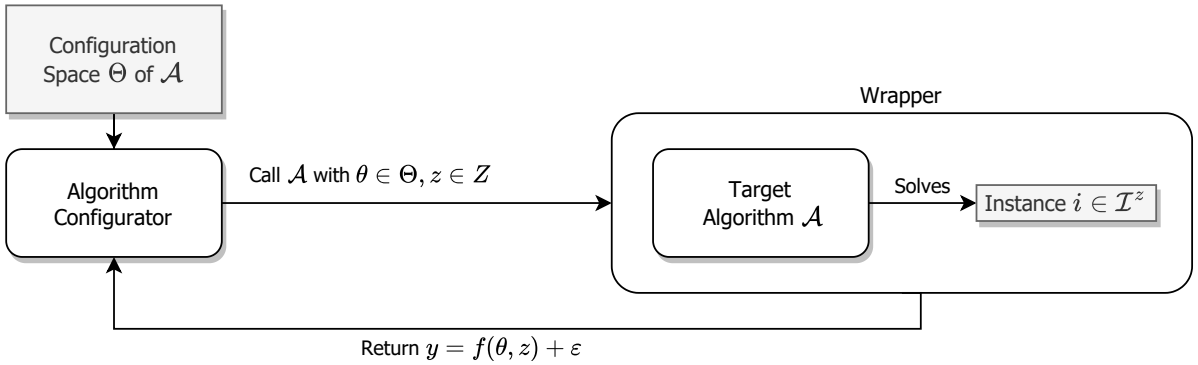


Figure 3.1: Specification of the workflow of contextual algorithm configuration (Eggensperger et al., 2019).

This set of observations can be used in a model-based approach to determine new configurations to query.

In Figure 3.1, we visualize an outline of the procedure to find these configurations. This is a specification of the general workflow of algorithm configuration as presented by Eggensperger et al. (2019). Here, target algorithm $\mathcal{A}$ and the underlying problem that it solves are placed inside a wrapper. This wrapper is queried to call the target algorithm given some configuration θ and some context z. Since the context of every problem instance is observable, we can divide the set of problem instances $\mathcal{I}$ over sets $\mathcal{I}^z$ that correspond to their context such that $\cup_{z \in Z} \mathcal{I}^z = \mathcal{I}$, and $\mathcal{I}^{z_i} \cap \mathcal{I}^{z_j} = \emptyset$ if $z_i \neq z_j$. The target algorithm solves an instance from the subset of instances that is compatible with the received context. It then returns function evaluation y to the configurator.

In an online approach, the configurator first observes a context and then decides upon a configuration to query. In an offline approach, the configurator simultaneously decides upon a context and a configuration. Any queried configuration is some point in the d-dimensional configuration space Θ. The configuration and contexts to query in any iteration are decided on by the algorithm configurator. This configurator can be instantiated by many different approaches: ranging from manual search to model-based methods.

We assume that we have a budget of samples N that is known in advance. After N samples, we can construct our mapping h by either returning the best incumbent configuration for every context $z \in Z$ over the observation history $\hat{\theta} = \arg\max_{\theta_i} \{y_i \in \mathcal{D}_N \mid z_i = z\}$; or returning the optimal expected configuration $\hat{\theta}$ for every context. Note that the first approach is only possible for a finite number of contexts. The special case with $|Z| = 1$ corresponds to a standard context-free approach, for which mapping h reduces to solving a standard global optimization problem:

$$\min_{\theta \in \Theta} f(\theta).$$

Next, we provide more details on Bayesian optimization and discuss various (surrogate-based) modeling techniques that can instantiate the algorithm configurator in Figure 3.1 to optimize $f(\theta, z)$. In order to make the derivations more tractable, we consider the notation of such a context-free approach for the remainder of this chapter. As a result, we reduce $f(\theta, z)$ to $f(\theta)$.

3.2. Bayesian optimization

Bayesian optimization is a strategy that aims to find the optima of a (non-convex) objective function in situations where no mathematical expression for the objective function is known. Instead, the observer can query input values and can obtain (possibly by noise corrupted) function evaluations based on this input. Bayesian optimization has become a popular technique due to its efficiency regarding the number of function evaluations it requires (for early work see, e.g. Mockus, 1994; Jones et al., 1998). The technique makes use of Bayesian statistics: a field of statistics that is mainly based around inference using Bayes' theorem. Bayes' theorem is used to update probabilities after observing new observations. Given events A and B, the conditional probability of A given B is defined as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

In the context of Bayesian optimization applied to the problem of hyperparameter optimization, this can be interpreted as follows. Let $y_j = f(\theta_j) + \varepsilon_j$ denote an observation of some objective function at θ_j. We can define a set of the first n observations as $\mathcal{D}_n = \{(\theta_1, y_1), \ldots, (\theta_n, y_n)\}$. $P(f)$ denotes the prior distribution: our current belief of the unknown objective function $f(\theta)$. The likelihood function $P(\mathcal{D}_n|f)$ denotes our belief on the probability of the received observations given this prior distribution. We can then calculate the posterior distribution as follows:

$$P(f|\mathcal{D}_n) = \frac{P(\mathcal{D}_n|f)P(f)}{P(\mathcal{D}_n)}.$$

In many scenario's, $P(\mathcal{D}_n)$ is difficult to calculate, and we can remove this normalizing constant from the above equation:

$$P(f|\mathcal{D}_n) \propto P(\mathcal{D}_n|f)P(f) \tag{3.2}$$

This equation describes that the posterior distribution of f given our current history of observations is proportionally equal to the prior distribution of f, multiplied by the likelihood of the observations given this distribution. Equation (3.2) is used in Bayesian optimization after every newly obtained observation to update our current belief of objective function f. Hereby, $P(f|\mathcal{D}_n)$ functions as our surrogate model of the unknown function $f(\theta)$, in which we are interested.

The general application of Bayesian optimization is summarized in Algorithm 1. We have an acquisition function α, an initial prior distribution S, a known budget on the number of samples N, and the unobservable objective function f. Every iteration n consists of the following steps: first, a promising new configuration θ is selected by acquisition function α; the objective function f of the target algorithm is queried with this configuration as the input; a possibly by noise-corrupted evaluation of this function is received; the surrogate model S is updated by taking this new evaluation into account.

This very generally formulated Bayesian optimization framework has two key ingredients. The first ingredient is the surrogate model S: a probabilistic distribution that captures our current beliefs of the unknown objective function. This surrogate model is initialized as the prior distribution. After every observation, we compute the posterior distribution according to Equation (3.2). The surrogate model S is then set to this posterior, which is used as the prior distribution in the subsequent iteration.

The second key ingredient is the acquisition function $\alpha : \Theta \to \mathbb{R}$. This function evaluates the utility of candidate configurations by combining both exploration and exploitation given our observation history. A new configuration to query can be determined by maximizing this function.

Algorithm 1 Bayesian optimization

Input: acquisition function α, prior distribution S, N, f.

1: **for** $n = 1, 2, \ldots, N$: **do**
2: select $\theta_{n+1} = \arg\max_\theta \alpha(\theta; \mathcal{D}_n)$
3: query objective function $y_{n+1} = f(\theta_{n+1}) + \varepsilon_{n+1}$
4: augment data $\mathcal{D}_{n+1} = \{\mathcal{D}_n, (\theta_{n+1}, y_{n+1})\}$
5: update surrogate model S based on $S, \mathcal{D}_{n+1}$
6: **end for**

3.2.1. Surrogate models in Bayesian optimization

We briefly discussed some popular surrogate models, including Gaussian processes, SMAC, TPE, and piece-wise linear models, and their respective advantages in Chapter 2. These are different types of models that have been used in a Bayesian optimization framework. To recap: both TPE and SMAC make use of decision trees in their surrogate model. As a result, these methods are internally capable of handling more general search spaces, including discrete, categorical, and conditional spaces. They can also scale to relatively large evaluation budgets. This property is even more favorable for the piece-wise linear model which does not grow in size over the number of observations. In addition, this method can also handle discrete and mixed-variable search spaces. Furthermore, these methods have all been shown to be able to handle high dimensional search spaces with up to 238 dimensions for the piece-wise linear model (Bliek et al., 2020b).

Gaussian processes have their disadvantages: a lot of modelling tricks are required for it to be able to handle high dimensional, discrete, categorical, or conditional search spaces. Even then, they generally perform worse compared to the alternative surrogate models on these spaces. Also, the updating step of a Gaussian process has a cubic time complexity. Hence, the computation time increases per iteration, possibly catching up to or even exceeding the time of the target algorithm.

In this research, however, the largest search space that we consider has a medium dimensionality (11 dimensions) that consists of only continuous hyperparameters. That is, the space is not discrete, not categorical, and not conditional. We provide more details on this search space in Chapter 7. The target algorithms that we consider require a computation time that is relatively large compared to the updating step of the surrogate model. As a result, in practice, the cubic time complexity never becomes a bottleneck of the search process.

Simply put: none of the difficulties of the problem that we consider are tackled by any of these alternative methods. On the other hand, these discussed methods have their drawbacks. The piece-wise linear surrogate model encounters difficulties with exploration due to the simplicity of its acquisition function that is a result of the surrogate not being smooth. TPE uses a more basic tree structure compared to the random forest of SMAC and generally gets outperformed by this alternative model (Thornton et al., 2013). SMAC still requires a relatively large number of observations by pooling multiple configurations in its acquisition function and sampling evaluations for every queried configuration multiple times. Furthermore, its surrogate model does not predict well for points that are distant from its observations. This has been shown to result in little exploration during training (Greenhill et al., 2020) and poor extrapolation (Shahriari et al., 2015).

Since these alternatives to using Gaussian processes tackle problems that we do not encounter, while exposing their drawbacks, the use of the popular Gaussian process as a surrogate model seems to be the most suitable. Next, we provide some more details on Gaussian processes and their advantage over the discussed alternatives.

3.3. Gaussian processes

We use a Gaussian process regression model as the surrogate model in our described Bayesian optimization framework. A Gaussian process is a stochastic process for an infinite collection Θ of random variables such that any finite subset of them is distributed jointly as a multivariate normal distribution. A Gaussian process $GP(\mu, k)$ is specified by a mean function $\mu : \Theta \to \mathbb{R}$ and a kernel (or covariance) function $k : \Theta \times \Theta \to \mathbb{R}$ that specifies the similarity between any pair of hyperparameter configurations.

One of the reasons for the popularity of Gaussian processes is its conjugate likelihood-prior pair property. A likelihood-prior pair is conjugate if the resulting posterior distribution is of the same form as the prior distribution. A combination of a Gaussian prior distribution and a Gaussian likelihood function is such a pair. Since we assumed that $\varepsilon_1, \ldots, \varepsilon_n$ are independent and identically distributed according to $\varepsilon_j \sim \mathcal{N}(0, \sigma_\varepsilon^2)$, we have likelihood function $p(\mathcal{D}_n|f) = \mathcal{N}(f, \sigma_\varepsilon^2 I_n)$. Because this likelihood function is Gaussian, and we decided to use a Gaussian process as our prior distribution, the resulting posterior distribution is also a Gaussian process. Since the shape of the posterior is now known, this distribution can be derived in closed form using Equation (3.2).

For a more detailed derivation, we consider a prior distribution $GP(\mu, k)$, an observation history $\mathcal{D}_n = \{(\theta_1, y_1), \ldots, (\theta_n, y_n)\}$ and a set of L unqueried points $\hat{\theta}$. We use the kernel function k to compute a $n \times n$ covariance matrix C for the observations in $\mathcal{D}_n$; a $n \times L$ cross-covariance matrix R; a $L \times L$ covariance matrix $\hat{C}$ over the pairs of points in $\hat{\theta}$. Using the normal distribution of $\hat{f}$ conditional on f, the posterior distribution $P(\hat{f}|\mathcal{D}_n)$ becomes $GP(\hat{\mu}, \hat{k})$, with

$$\hat{\mu} = R^T (C + \sigma_\varepsilon^2 I_n)^{-1} y, \tag{3.3}$$

$$\hat{k} = \hat{C} - R^T (C + \sigma_\varepsilon^2 I_n)^{-1} R, \tag{3.4}$$

where $y = [y_1, \ldots, y_n]^T$. For an extensive derivation of these formulas, we refer to Rogers and Girolami (2016), Section 8.2.

In the Bayesian optimization framework as presented in Algorithm 1, we compute the posterior distribution after every new observation. In this case, $|L| = 1$ and $\hat{\theta}$ denotes observation θ_{n+1}. Consequently, R denotes a vector of size n and $\hat{C}$ is a scalar that contains the variance at θ_{n+1}. The cubic time complexity of updating a Gaussian process as we previously mentioned, is the result of the matrix

inversions in Equations (3.3) and (3.4).

A second advantage of Gaussian processes is that we can use the properties of Gaussian distributions to easily calculate statistical values that can be used in the acquisition function. Acquisition functions aim to maximize some combination of exploitation and exploration over the search space. Given the Gaussian distribution of the posterior distribution, we can use configurations with a large mean value (Equation (3.3)) for exploitation, and configurations with a large variance (Equation (3.4)) for exploration. Or we can incorporate the best observed incumbent performance $y^* = \max\{y_n \in \mathcal{D}_N, n = 1,\ldots,N\}$ to improve upon by analytically computing the probability $\Phi\left(\frac{\mu(\theta)-y^*}{\sigma(\theta)}\right)$. Here, Φ denotes the standard normal cumulative distribution function.

For the remainder of this thesis, we consider the Gaussian Process Upper Confidence Bound (GP-UCB) acquisition function as introduced by Srinivas et al. (2009). This acquisition function is defined as:

$$\alpha(\theta, \mathcal{D}_n) = \mu_n(\theta) + \beta_n^{1/2}\sigma_n(\theta), \tag{3.5}$$

where $\sigma_n(\theta)$ denotes the standard deviation at point θ, which is equal to the covariance $\sqrt{k(\theta,\theta)}$. The first term of the equation aims to maximize the exploitation, while the second term aims to maximize the exploration over the search space. The constant value β denotes the trade-off between the two notions. Altogether, this acquisition function prefers configurations with high expected performance and large uncertainty. The acquisition function is then used at iteration n to determine the next configuration to query by finding:

$$\theta_{n+1} = \arg\max_{\theta\in\Theta} \alpha(\theta, \mathcal{D}_n). \tag{3.6}$$

A final advantage of Gaussian processes is that their statistical properties and general applicability make them the most widely used and researched statistical model within the field of Bayesian optimization. As a result, a lot of extensions have been proposed to account for contextual optimization. We already discussed several examples of related research in Chapter 2.

We conclude this chapter with a visualization of the application of Bayesian optimization using a Gaussian process to optimize an unknown function that is presented in Figure 3.2. This example shows how the Gaussian process surrogate model is updated over three iterations. The goal is to find the maximum of the objective function. Every iteration, the configuration that maximizes the acquisition function is queried. For illustrative purposes, the true objective function is shown in this figure. In practice, this function is unknown.

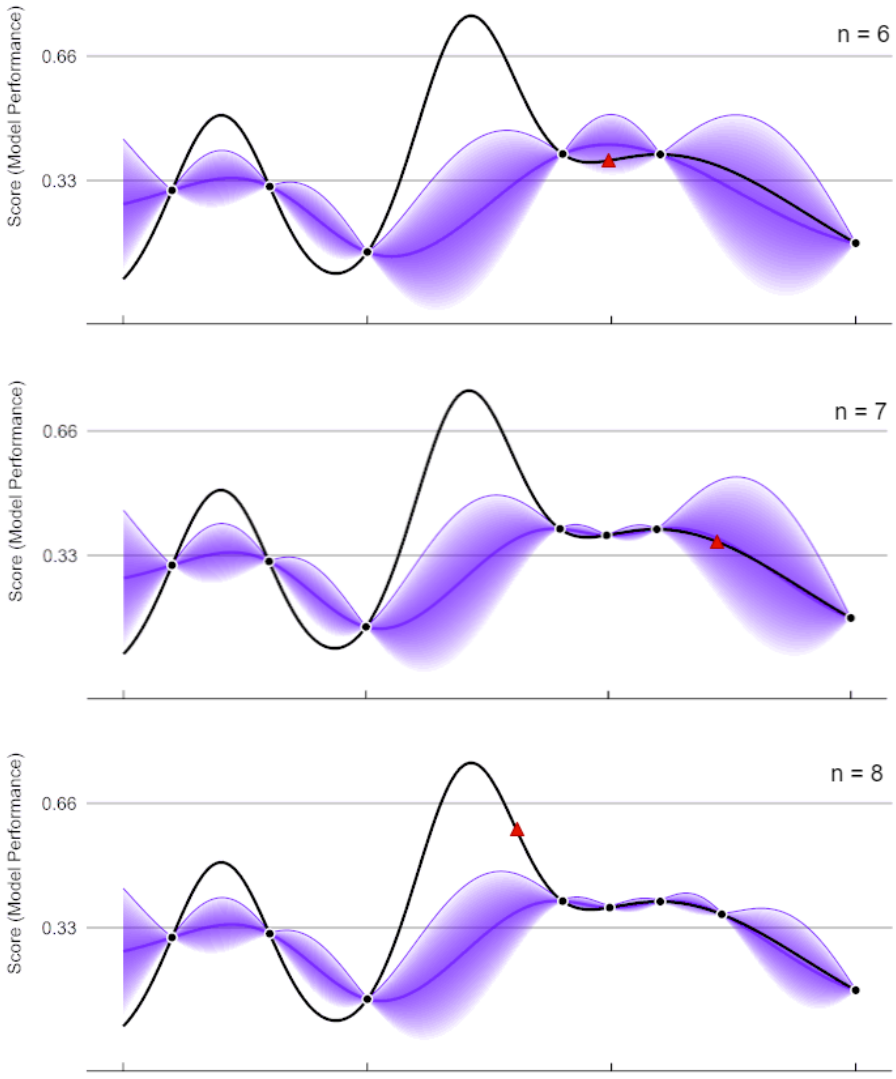


Figure 3.2: Example of three iterations of the Bayesian optimization procedure with a Gaussian process surrogate model on a 1D function. The black line denotes the (unknown) true objective function. The black dots mark empirical observations of this function. The dark purple line represents the current mean of the GP with the shaded area around it denoting its confidence interval. The red triangles mark the point that currently maximizes the acquisition function, and hence denote the configuration that is to be queried in the next iteration. This figure is an adaptation of work by Sam Wilson, distributed under a CC-BY-SA 4.0 license.

4

Local search on satisfaction problems

In this chapter, we discuss local search heuristics as the target algorithm of which we aim to optimize its set of hyperparameters and the problem version that the target algorithm solves. This problem version is a constraint satisfaction problem (CSP). A local search heuristic, however, can only solve optimization problems. Therefore, we first present an approach to transform general satisfaction problems into an optimization problem formulation such that they can be solved using local search. We then describe how this approach is applied to the specific problems that we consider in this research: the train unit shunting problem (TUSP) and the bin packing problem.

We first provide some background information on local search heuristics and, in particular, variable neighborhood search (VNS) in Section 4.1. In Section 4.2, we show how CSPs can be transformed to optimization problem formulations using the so-called *penalty method*. We then give details on the TUSP and the bin packing problem in Sections 4.3 and 4.4. We conclude this chapter with a remark on the application of contextual hyperparameter optimization on satisfaction problems in Section 4.5.

4.1. Local search heuristics

Local search is a heuristic method that aims to solve optimization problems by moving over the search space. This method iteratively moves from a current solution to a candidate solution that is located in the neighborhood of the search space of this current solution. The likelihood of moving from a solution to such a candidate solution generally depends on the difference between the fitness values of the respective solutions. A basic local search method is the hill climbing method that continues to move to the best candidate solution over the entire neighborhood. It terminates when it finds itself stuck in a local optimum. We are generally interested in finding the global optimum over the search space. Assuming that the objective function to optimize is non-convex over the search space; a local search heuristic should avoid getting stuck in a local optimum.

Metaheuristics are heuristics that depend on stochastic methods to be able to escape these local optima. Metaheuristics cannot guarantee that a global optimum will be found. However, local search metaheuristics have been shown to be able to yield good solutions for a wide range of combinatorial optimization problems (e.g. Lin and Kernighan, 1973; Benlic and Hao, 2013; Xu and Cai, 2018). Examples include simulated annealing, tabu search and variable neighborhood search.

4.1.1. Variable neighborhood search

In this thesis we focus on finding hyperparameter configurations for a local search target algorithm that solves the train unit shunting problem. More specifically, we consider VNS, a metaheuristic introduced by Mladenović and Hansen (1997), as the specific target algorithm.

VNS is an extension of iterated local search (ILS) that moves to every considered candidate solution with certainty if it has a more optimal objective value. Therefore it quickly descents to a local optimum. Once the current solution has no candidate solution in its neighborhood that has a better fitness value, and is thus in a local optimum, VNS moves to a perturbation phase. In this phase, a series of random walks is performed with the aim to leave the basin of this local optimum. After a number of walks, the descending phase is re-entered. If the local optimum of the new descending phase does not have an

improved fitness value compared to the previous local optimum, the number of walks is incremented. This effectively increases the size of the neighborhood that is considered in the perturbation phase, hence the name. If the fitness of a new local optimum is an improvement, the size of the neighborhood is reset to the defined minimum. VNS consists of hyperparameters that denote the minimum and the maximum number of random walks in a perturbation phase; and a limit on the maximum increase in objective value per step in the perturbation phase The approaches that we consider throughout this work can be readily applied to any local search (meta)heuristic.

4.2. Penalty method for satisfaction problems

In the previous section, we discussed the general concept of local search heuristics to solve optimization problems. Local search can only be applied to optimization problems since it requires an objective function to compare the fitness of two candidate solutions. In this research, however, we consider a problem that is formulated as a CSP. That is, the overall goal is not to find the optimal solution to the problem, but any feasible solution: a solution that satisfies a defined set of constraints. In this section, we show how a satisfaction problem can be transformed into an optimization problem such that it can be solved by a local search heuristic.

The only goal of a satisfaction problem is to find any solution that satisfies the set of constraints. Therefore, in contrast to optimization problems, there is no preference between pairs of such feasible solutions. If we want to apply a local search method on our underlying satisfaction problem in order to find such a feasible solution, we need to be able to compare some fitness value of encountered candidate solutions that are not feasible. Therefore, the satisfaction problem needs to be reformulated as an optimization problem by introducing an objective function that measures the fitness of solutions that do not satisfy all constraints. Hence, the fitness of a solution should represent some notion on the level of infeasibility of a solution. The common practice in a boolean satisfiability problem (SAT) is to use the sum of the violated clauses as an evaluation (if the objective is to minimize). In more general CSP problems that consist of different constraints, weights are usually assigned to the violations of different constraint types. The amount of violation per constraint can be measured using slack variables. By introducing such slack variables in an optimization problem formulation, infeasible solutions for the original CSP, have a corresponding feasible solution in this optimization problem. However, a feasible solution to the optimization problem formulation is only feasible for the CSP if all slack variables are equal to zero. In that case, there is a total of zero constraint violations.

The need for the use of any slack variables to make a solution feasible can then be penalized in the objective function. This method of transforming constraint optimization problems into unconstraint problems by adding a penalty term to the objective function is referred to as a *penalty method*. Assuming that all weights in the objective function are positive, the objective must be larger than zero if we have at least one constraint violation. The converse then means that if we have a solution to the optimization problem with an objective value equal to zero, this solution is a feasible solution to the original CSP. When solving the optimization problem using a local search method, the search can be terminated once we have found such a solution.

We discussed in Section 3.2 how Bayesian optimization updates its belief of some unobservable function $f(\theta)$ based on new observations in the form of function evaluations $y = f(\theta) + \varepsilon$. Commonly, function $f(\theta)$ is the objective function of some optimization problem. An optimization problem describes the general problem of finding the solution from all feasible solution that attains the most extreme objective value and may be formulated as follows:

$$\text{(Opt)} \quad \max_{x} \quad f(x), \tag{4.1}$$

$$\text{s.t.} \quad g_i(x) \leq c_i \quad \text{for } i = 1, \ldots, n. \tag{4.2}$$

Here, Equation (4.2) defines the set of constraints that are required to be satisfied. Given some solution x, the result of the formulation is either the objective value $f(x)$ or that solution x is infeasible since at least one of the n constraints is violated.

We can, however, apply the penalty method to reformulate this optimization problem formulation such that every solution x is feasible. We do so by introducing a slack variable to every constraint to transform the inequality constraints to equality constraints. We then extend the objective function by

adding a term that penalizes the values of the positive slack variables. This term denotes a penalty that is incurred based on the degree of the violation of some constraint; and the severeness of this violation. Let p_i denote the slack variable for constraint i and w_i its respective weight. The transformed optimization problem with penalty variables (Opt-P) is then formulated as follows:

$$\begin{aligned} \text{(Opt-P)} \quad \max_x \quad & f(x) - \sum_{i=1}^{n} w_i \cdot \max\{0, p_i\}, \\ \text{s.t.} \quad & g_i(x) = c_i + p_i \quad \text{for } i = 1, \ldots, n, \\ & p_i \in \mathbb{R} \quad \text{for } i = 1, \ldots, n. \end{aligned}$$

The weight coefficients per constraint type w_i in the objective function are hyperparameters that can be tuned. They can be specified to allow for a trade-off between any constraint violations and the value of the original objective function. Also, the original objective function can effectively be retrieved by configuring sufficiently large values for these weight coefficients, denoted by $w_i = M$. In that case, if a feasible solution for (Opt) exists, such a solution will be feasible for (Opt-P) and yield an objective value smaller than M. On the other hand, if an optimal solution to problem (Opt-P) has an objective value larger than M, at least one constraint must be violated, and hence no feasible solution to problem (Opt) exists. It should be clear that a feasible solution for (Opt-P) is infeasible for (Opt) in case any slack variable p_i is larger than zero.

We now consider a transformation of CSP problems to allow for infeasible solutions. A CSP describes the problem of finding any feasible solution x such that:

$$\text{(Sat)} \quad g_i(x) \leq c_i \quad \text{for } i = 1, \ldots, n.$$

The only difference of (Sat) compared to (Opt) is that there is no objective function $f(x)$ and therefore there is no preference over any of the feasible solutions. In optimization problem (Opt) we can use the objective function to compare which non-optimal solution is closer to the optimum. Due to the lack of this function in (Sat), we can not compare a pair of infeasible solutions and decide which solution is 'closer' to feasibility. Generally speaking, however, optimization problems are more difficult to solve than satisfaction problems since any optimal solution to an optimization problem is a (possibly non-unique) solution to the corresponding satisfaction problem. As a result, you could solve any satisfaction problem by transforming it into an optimization problem formulation and using the optimal solution of this obtained problem formulation.

We perform this transformation by introducing slack variables to the satisfaction problem formulation in a similar fashion as our transformation from (Opt) to (Opt-P). The set of inequalities that define the constraint set of problems (Opt) and (Sat) are identical. As a result, they remain identical after the addition of the slack variables. Where, in (Opt), these slack variables were added as an extra term to the existing objective function, in problem (Sat), they introduce a new objective function. After transforming it to an optimization formulation, the satisfaction problem with penalty variables (Sat-P) looks as follows:

$$\begin{aligned} \text{(Sat-P)} \quad \min_x \quad & \sum_{i=1}^{n} w_i \cdot \max\{0, p_i\}, \\ \text{s.t.} \quad & g_i(x) = c_i + p_i \quad \text{for } i = 1, \ldots, n, \\ & p_i \in \mathbb{R} \quad \text{for } i = 1, \ldots, n. \end{aligned}$$

Please note that the maximization of a negative function is equivalent to a minimization problem. We can now solve this problem to optimality using some heuristic method that relies on an objective function, such as local search. By only using hyperparameter configurations for which $w_i > 0$ for all $i = 1, \ldots, n$, and only incurring a penalty for slack variables $p_i \geq 0$ due to the maximization term in the objective function, the following relation holds:

$$\sum_{i=1}^{n} w_i \max\{0, p_i\} > 0 \iff \exists\ p_i > 0. \tag{4.3}$$

That is, if a solution x to (Sat-P) has an objective value that is larger than zero, then there must exist a constraint that is violated. As a result, solution x is not a feasible solution to (Sat). On the other hand, if solution x has an objective value equal to zero, this solution satisfies (Sat).

The weight coefficients w_i in the newly constructed objective function denote the severeness of the violation of constraint i. Different weight values per constraint i can be used to suggest the relative importance of satisfying certain types of constraints. This also allows for a possible ranking of solutions that do not satisfy problem (Sat). There is no *ground truth* to these weight coefficients. Hence, they can be regarded as hyperparameters instead of model parameters. Recall that within local search heuristics in general, and particularly within variable neighborhood search, we tend to move from a current solution to a candidate solution if that candidate solution has a better fitness value. By considering different configurations of the weight coefficients, we can construct different preferences over the candidate solutions. This can result in different paths over the search space for local search heuristics when executed using different configurations of the weight coefficients.

This leads to the following claim: by using different weight configurations, a local search heuristic moves differently over the search space. This allows for the possibility to avoid, or even escape, local optima in which the local search would get stuck when executed with a different configuration. Consequently, there is a possibility to find a feasible solution to the underlying satisfaction problem given some configuration of hyperparameters where this is not possible given a different hyperparameter configuration.

We now translate these general problem formulations to the problem of optimizing the hyperparameter configuration θ of some target algorithm $\mathcal{A}$. To remain consistent with the notation introduced in Chapter 3, we let θ_k denote the weight coefficient of the penalty p_k on violating constraint k, for $k = 1, \ldots, d$. Here, d denotes the number of dimensions of the configuration space and thus the number of hyperparameters. Let $g_k(\theta)$ denote the degree of violation of constraint k of the underlying problem after running $\mathcal{A}$ with configuration θ. Since we want $\mathcal{A}$ to find solutions that violate none of the constraints we set $c_k = 0$ for all $i = 1, \ldots, d$. Since, by definition, $g_k(\theta) \geq 0$, we can restrict the domain of the slack variables to only positive real values. Therefore, the maximization term over the slack variables in the objective function of (Sat-P) becomes redundant and can be removed. If the original problem is a satisfaction problem of the form (SAT), the resulting optimization formulation within the framework of hyperparameter optimization (HO-P) becomes the following:

$$\text{(HO-P)} \quad \min_{\theta} \quad \sum_{k=1}^{d} \theta_k p_k, \tag{4.4}$$

$$\text{s.t.} \quad g_k(\theta) = p_k \quad \text{for } k = 1, \ldots, d, \tag{4.5}$$

$$p_k \geq 0 \quad \text{for } k = 1, \ldots, d. \tag{4.6}$$

In case the original problem is an optimization problem, the resulting formulation looks the same, but the objective function (4.4) is replaced by:

$$\max_{\theta} f(\theta) - \sum_{k=1}^{d} \theta_k p_k, \tag{4.7}$$

to remain consistent with problem formulation (Opt-P).

Within the problem of hyperparameter configuration, the weights w in the objective function have been replaced by our general notation for hyperparameters θ in objective functions (4.4) and (4.7). As a result, a pair of solutions with exactly the same set of variables that have been obtained by the target algorithm with different queried hyperparameter configurations, can have different objective values due to different hyperparameter configurations used by $\mathcal{A}$. This has the consequence that we cannot use the objective value in the evaluation of the performance of $\mathcal{A}$ for Bayesian optimization. This will be explained in more detail in Section 5.3.

4.3. Train unit shunting problem

In this section, we discuss the first target algorithm of which we aim to optimize its set of hyperparameters; and the underlying problem that it solves. First, we give a brief introduction to the train unit shunting problem. For an extensive description of this problem, we refer to van den Broek (2016).

The TUSP considers the problem of parking, cleaning, regularly maintaining and reconfiguring the compositions of arriving train units into the scheduled departing train composition. These duties are performed at shunting yards that also function as service sites. A shunting yard contains multiple tracks over which trains can pass or on which they can be parked. These tracks are connected to each other by switches that restrict the movement of a train between different tracks. Train tracks are either free tracks, which are accessible from both sides, or tracks where one side of the train track is blocked by a bumper. On the latter, the train leaves from the same side as it has accessed the track. If the train consists of multiple train units, the order of the departing units is the reversed order of the arriving units. Every problem instance of the train unit shunting problem consist of a problem location and a problem scenario. The location of an instance denotes the infrastructure of a shunting yard. It represents:

- the topology of the tracks, the switches, and the bumpers;
- the length of the tracks;
- the gateways: tracks over which trains arrive to and depart from the shunting yard.

Besides the problem location, every problem instance has its problem scenario. This scenario consists of:

- the sequence and composition of arriving train;
- the sequence and train types of departing trains;
- the timetables of arriving and departing trains.

The information from both the problem scenario and the problem location define the parameter values that are used in the constraints of the TUSP.

The TUSP has been described by van den Broek (2016) to be a combination and generalization of four problems: a matching problem, a bin packing problem, a rush hour problem, and an open shop scheduling problem. First, there can be differences between the compositions of the arriving and the departing trains as specified in the problem scenario. In this case, a subset of the arriving trains has to be split and reconfigured to match the required composition of the departing trains. The matching problem specifies the assigning of arriving train units to departing train units.

When a train is not moving over the tracks, it must be parked on one of the available tracks. The number of trains that can be parked on certain tracks depends on both the length of the track and that of the train units. This subproblem resembles the bin packing problem, which we discuss in more detail in the next section.

The rush hour problem describes the simultaneous movements of all trains over the shunting yard. Its solution is defined as a series of movements for every train over the tracks of the shunting yard. There are a lot of constraints here to consider. First, the series of movements of each train must start and end when the respective train arrives and departs such that the departing schedule is met on time. The trains should not block each other's movements nor simultaneously cross over any switches.

Finally, this series of movements should consider any service tasks for the train units if these are contained in the problem scenario. Examples of these tasks are the cleaning and inspection of the train units, and they must be performed at designated tracks of the shunting yard. For instance, the cleaning task must be performed on tracks with a cleaning station. This describes the open shop scheduling problem.

In this thesis, we limit ourselves to instances of the TUSP that do not contain any required service tasks. This way, we reduce the number of subproblems by ignoring the scheduling problem. This relaxation reduces the problem difficulty and the sizes of the configuration space and the search space, which we discuss in Section 5.2. For all methods that we describe in this thesis, these service tasks can readily be included for any subsequent research.

4.3.1. HIP

The problem version of the TUSP that we consider is a CSP. A feasible solution to this satisfaction problem is a representation of a complete shunting plan of movements from arriving trains to departing train compositions that do not conflict with the set of constraints. This CSP is solved by an implementation of a local search heuristic called the hybrid integral planning method (HIP). At the beginning of this chapter, we mentioned that local search is a heuristic that solves optimization problems by optimizing an objective function. HIP finds feasible solutions for the satisfaction problem by optimizing an objective function that consists of a weighted sum of the constraint violations. The problem formulation of the TUSP is a specification of the generally defined satisfaction problem (Sat) and HIP solves the transformed problem version (Sat-P).

The objective function of the TUSP that is considered by HIP is defined as a weighted sum of the constraint violations, also referred to as *conflicts*. The formulation of this objective function meets the criteria such that Equation (4.3) holds. Therefore, HIP is terminated once a candidate solution is evaluated that has an objective value equal to zero. The implementation of HIP contains multiple hyperparameters that we discuss later. We refer to the values of these hyperparameters that are currently used in deployment as the *default configuration*. This default configuration is currently used for every considered instance of the TUSP. The default type of local search that is used by HIP is a VNS metaheuristic. The weights in the objective function of HIP of the default configuration are chosen to denote the difference in the severeness of these specific types of conflicts. However, if we are only interested in solutions with zero conflicts, and we make reasonable assumptions about the objective function (i.e. the number of conflicts is non-negative and the weight per conflict is positive), then it holds that a solution is feasible if and only if its objective value is equal to zero, regardless of the chosen weights.

4.4. Bin packing problem

In the previous section, we introduced the train unit shunting problem and described it as a combination of four problems. One of these subproblems is the bin packing problem. The bin packing subproblem comes into play for the shunting problem when we have to park trains of varying lengths on the available tracks.

In this section, we present a problem formulation for the bin packing problem and a local search specification to solve this problem formulation. The bin packing problem is a widely studied problem in combinatorial optimization; it has a simple problem description and formulation; and it structurally resembles the TUSP. Therefore, this problem is mainly presented to exemplify how the CSP version of the TUSP can be reformulated as an optimization problem to be solved by a local search algorithm and how hyperparameters in the formulation can impact the local search. The motivation for all design choices within this section is to closely resemble the formulation of the TUSP that is solved by the target algorithm.

4.4.1. Problem formulations

We first introduce the satisfaction problem formulation of the bin packing problem. Since the available tracks at a shunting yard have varying lengths, we consider the bin packing problem with heterogeneous bin capacities. For this problem formulation, let I denote the set of items and J the set of available bins. Let b_j be the size of bin j and s_i the size of item i. The binary decision variable x_{ij} is equal to 1 if item i is allocated to bin j, 0 otherwise. The satisfaction problem version of the bin packing problem is then formulated as:

$$\text{(Bin)} \quad \sum_{i \in I} s_i x_{ij} \leq b_j \qquad \forall j \in J, \tag{4.8}$$

$$\sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I, \tag{4.9}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in I, j \in J, \tag{4.10}$$

Constraint (4.8) guarantees that the capacity of every bin is not exceeded by the set of assigned items. Constraint (4.9) assures that every item is allocated to exactly one bin. Problem (Bin) aims to find any solution x that satisfies the set of constraints. Therefore, the problem formulation does not have an objective function.

We want to solve this problem formulation with a local search heuristic. In order to do so, we need to assign a value to any (infeasible) solution that represents the quality of the solution such that we can compare any pair of candidate solutions. Therefore, we transform problem (Bin) into an alternative problem formulation that includes an objective function. This formulation must allow for violations of the Constraints (4.8) and (4.9) such that these constraint violations can be included as weighted penalties in the objective function. Also, we should be able to retrieve a feasible solution to the original problem if it exists based on the solution to this alternative problem formulation.

For this transformation, we use the penalty method from Section 4.2 and define (Bin) as a specification of the generally defined transformation from (Sat) to (Sat-P). We can, for example, consider Constraint (4.8) and define $g_j(x)$ to be the (possibly negative) degree of violations of this constraints. That is:

$$g_j(x) = \sum_{i \in I} s_i x_{ij} - b_j$$

In order to obtain a feasible solution, we do not want any violations. Therefore we choose $g_j(x) \leq c_j = 0$. By adding slack variables to construct equality constraints we obtain:

$$\sum_{i \in I} s_i x_{ij} - b_j = p_j$$

In the objective function of (Sat-P), by optimizing $\min \sum_j w_j \max\{0, p_j\}$, we only penalize positive values for the slack variables. Since $g_j(x)$ is defined as the (possibly negative) degree of constraints violations, we can equivalently define the slack variables as follows:

$$p_j = \max\{0, \sum_{i \in I} s_i x_{ij} - b_j\} \tag{4.11}$$

Using this definition for the penalty of the violation of constraint j, we can directly optimize $\min \sum_j w_j p_j$.

The next step in our transformation is to construct a set of constraints $k = 1, \ldots, d$, such that we can define a distinct penalty variable p_k for every constraint type k. We consider two approaches for each constraint in the original satisfaction problem: we penalize whether a constraint is violated; and also the level at which a constraint is violated. In the bin packing problem formulation (Bin), when considering the bin capacity constraint (4.8), this can be formulated as the following two distinct penalty variables relating to bin j:

$$p_{2j} = \max\{0, \sum_{i \in I} s_i x_{ij} - b_j\} \qquad \forall j \in J, \tag{4.12}$$

$$p_{3j} = \mathbb{I}\{\sum_{i \in I} s_i x_{ij} > b_j\} \qquad \forall j \in J, \tag{4.13}$$

where $\mathbb{I}\{\cdot\}$ denotes an indicator function. Here, Equation (4.13) considers the question: *"is the capacity of bin j exceeded?"*, while Equation (4.12) denotes the level of violation of the capacity constraint, which can be interpreted as *"by how much is the capacity of bin j exceeded?"*.

Next, we split the equality from Constraint (4.9) in two inequality constraints. That is, every item can be assigned to at most one bin, formulated as:

$$\sum_{j \in J} x_{ij} \leq 1 \qquad \forall i \in I; \tag{4.14}$$

and every item must be assigned to at least one bin. This second inequality can be formulated as Equation (4.14), with a a flipped inequality sign. It can then be subsequently rewritten to an equality constraint using the general form of Equation (4.11):

$$p_{1i} = \max\{0, 1 - \sum_{j \in J} x_{ij}\} \qquad \forall i \in I, \tag{4.15}$$

This equation defines the penalty variable p_{1i} to be equal to 1 if item i is not assigned to any bin, and 0 otherwise. If we derive formulations for this inequality for both approaches as described above (see Equations (4.12) and (4.13)), we would yield the exact same constraint formulation twice. If that

is the case, we only include the derived penalty variable once, to avoid unnecessarily increasing the number of variables, and with it, the dimensions of the configuration space. This also frequently occurs in the transformation of the TUSP formulation.

The inequality constraint (4.14) is the only constraint from the bin packing satisfaction problem that is not allowed to be violated in our problem formulation. The correctness of this is a result of our chosen initial solution and neighborhood relations, which in combination implicitly guarantee that this constraint is satisfied for every candidate solution throughout the search process. Since it is always satisfied, we can avoid the need to have a penalty variable and a weight for every constraint. The initial solution and the neighborhood relations will be described in more detail in the following subsection. This general motivation is also applied within the TUSP formulation that is considered by HIP.

Finally, the objective of the problem is to minimize a weighted sum over the three penalty terms. The transformation results in the following optimization problem formulation for the bin packing problem:

$$\begin{aligned}
\text{(Bin-P)} \quad \min_{x,p} \quad & \theta_1 \sum_{i \in I} p_{1i} + \theta_2 \sum_{j \in J} p_{2j} + \theta_3 \sum_{j \in J} p_{3j}, && && (4.16)\\
\text{s.t.} \quad & x_{ij} \in \{0,1\} && \forall i \in I, j \in J, && (4.17)\\
& (4.12), (4.13), (4.14), (4.15).
\end{aligned}$$

We refer to the summations $\sum_{i \in I} p_{1i}$, $\sum_{j \in J} p_{2j}$, $\sum_{j \in J} p_{3j}$ as p_1, p_2, p_3, respectively, such that the objective function (4.16) can be rewritten as:

$$\min_{x,p} \sum_{k=1}^{3} \theta_k p_k \tag{4.18}$$

The three penalty variables can now be interpreted as follows. The penalty variable p_1 denotes the number of items that are not assigned to any bin. Variable p_2 yields a penalty based on the total amount of exceeded capacities over all bins, whereas p_3 represents the number of bins of which the capacity is exceeded.

To motivate the simultaneous use of p_2 and p_3, we can consider two intermediate infeasible candidate solutions. The first candidate allocates a subset of the items in such a way that the capacity of many bins is slightly exceeded. The second candidate allocates the same subset of items to a single bin, thereby heavily exceeding its capacity but not that of the other bins. A relatively large value of θ_2 compared to θ_3 would prefer the first candidate over the second, and vice versa. Since we consider intermediate candidate solutions, the candidate that is more probable to lead to a feasible solution might be dependent on the subset of unassigned items, the sizes of the items and the capacities of the remaining bins. As a result, differentiating between these two penalties can result in a better evaluation of the intermediate solutions.

We have shown for the general case that we can find whether the optimal solution to (Sat-P) yields a feasible solution to (Sat) by using Equation (4.3). For the specification (Bin-P) we have, by definition, $p_k \geq 0$ for all k. Then, by only using hyperparameter configurations for which $\theta_k > 0$ for all k, the following relation holds:

$$\sum_{k} \theta_k p_k > 0 \iff \exists\ p_k > 0. \tag{4.19}$$

This can be interpreted as follows: if we find an optimal solution to problem (Bin-P) with an objective value larger than zero, this solution does not satisfy (Bin). On the other hand, if the obtained solution has an objective value that is equal to zero, this solution satisfies the original satisfaction problem version of our bin packing problem.

4.4.2. Local search procedure

In this section, we introduce details on the local search target algorithm to solve problem (Bin-P). To remain consistent with HIP, the local search algorithm that we use is a VNS metaheuristic. We show that every solution that is considered as a candidate solution during the search process of this local search algorithm is a feasible solution to this problem formulation. To do so, we show that every

constraint of the bin packing problem must be satisfied, possibly by using positive values for the slack variables.

Initial solution
The local search algorithm must start with an initial solution that satisfies all constraints in problem (Bin-P) that cannot be violated using penalties. Hence, we start with an assignment $\{X \mid \sum_{j=1}^{n} x_{ij} \leq 1 \ \forall i \in I, \ X \ \in \{0,1\}^{I \times J}\}$ to satisfy Constraints (4.14, 4.17). We simply use $x_{ij} = 0 \ \forall \ i \in I, j \in J$. That is, every item starts unassigned.

For local search in general, the chosen initial solution can have a great influence on the obtained solution, although this relation is usually not very obvious in advance. The choice of the initial solution impacts the values of the weights that are the most suitable to find feasible solutions. A clear example is the following: if we start by having all items unassigned, we likely want a relatively large penalty on the unassigned items (large θ_1). However, if we start by assigning all items to a single bin, we likely want a relatively large penalty on the violated capacity of a bin (large θ_2). This way, we start with a large total penalty, and we quickly move to less penalized solutions.

Neighborhood relations
A local search algorithm iteratively considers a candidate solution that is the neighbor of the current solution. The neighbors of a solution are all solutions within the defined neighborhood, which is a subspace of the search space. This subspace is defined by neighborhood relations. We consider three actions that individually make up the neighborhood of a solution.

1. **Move:** take one item and one bin; and move the item to the bin. This action yields a maximum of $|I| \times |J|$ candidate solutions.
2. **Swap:** take two items; and swap the bins that they are allocated to. This action yields a maximum of $|I|^2$ candidate solutions.
3. **Merge:** take two bins; and assign items from one bin to the other bin. This action yields a maximum of $|J|^2$ candidate solutions.

In these actions, we also consider an auxiliary bin to represent the items that are not assigned to a real bin. For example, the move action can remove an allocated item from a bin by moving it to this auxiliary bin. The three actions are performed in random order. When an action is considered, as long as no newly constructed candidate solution has been accepted, all possible candidates in the respective neighborhood are considered, before the next action is considered.

We have motivated in the previous subsection that the combination of the initial solution and the neighborhood relations can guarantee that a subset of the constraints is always satisfied throughout the search process.

In our case, all items start unassigned in the initial solution. All three actions can construct candidate solutions where an unassigned item gets assigned to a single bin. However, none of the introduced actions can construct a candidate solution where an item gets assigned to an additional bin if it was already assigned to a bin. Therefore, at no point can a single item be assigned to multiple bins simultaneously. Thus we can state that Constraints (4.14) and (4.17) will be satisfied for every considered candidate solution.

4.5. Contextual optimization on satisfaction problems

Within the problem of the optimization of hyperparameter configurations, we aim to optimize the performance of the target algorithm on multiple problem instances or datasets. In this research, these are different instances of the train unit shunting problem or the bin packing problem. One approach to tackle this problem is the so-called *instance-specific* hyperparameter optimization approach (Kadioglu et al., 2010). This approach aims to find an optimal configuration for the target algorithm for every distinct problem instance or dataset. As an example, we can consider one of the most popular applications of Bayesian optimization: the optimization of the hyperparameter configuration of a machine learning classification problem. In this setting, we have a classification model and a dataset and we want to find hyperparameters that minimize the loss function on this dataset. If we have a solution, it

does not necessarily mean that this configuration also works for the same classification model on different datasets. Since we want to optimize the performance of our classifier on this particular dataset, we want to find a configuration that should then only be applied to this dataset. If the goal is to find configurations that optimize the performance on a larger number of datasets, this instance-specific approach still works; it just requires additional time to consider every individual dataset to find the corresponding optimal configuration.

However, if the problem at hand is a satisfaction problem, such an instance-specific approach is not attainable. If we consider a satisfaction problem instead of an optimization problem, we are only interested in finding some configuration for which the target algorithm finds any feasible solution for the specific instance. If, at some point in the configuration procedure, we query a configuration that yields a feasible solution, there is no need to continue searching. If the queried configuration yields no feasible solution, the only thing we learned is that this particular configuration is not suitable. After several unsuccessful attempts of querying hyperparameter configurations, we have not learned which configuration is now promising to query. Hence, we are effectively random sampling configurations from the search space; we are not training based on the received observations.

To emphasize this, we consider the problem from the perspective of the NS. A feasible solution must be found for every instance of the TUSP daily at every shunting yard. An instance-specific approach to this problem setting would be to first search for a hyperparameter configuration for every problem instance by trying to solve the specific instance; and then to solve the instance. Again, assuming that the instance is never solved in the training phase we are just trying to solve the instance with a randomly chosen configuration. If the instance is solved with such a configuration, we can immediately stop the process. This strategy is not attainable due to the tight limit on the available time. This describes exactly why an uninformed search over the configuration space should be prevented.

As an alternative, we can consider a generalized approach for satisfaction problems by considering the number of instances that we can solve given a configuration. Or we can consider another performance metric that turns the problem into an optimization problem. We can train on multiple problem instances by finding the configuration that optimizes this performance metric and use this configuration for new instances. But an instance-specific approach to find a configuration for a satisfaction problem is not attainable.

5

Evaluating the performance of hyperparameter configurations

As described in Chapter 3, Bayesian optimization receives the performance of a configuration in the form $y = f(\theta) + \varepsilon$. So far, we have not provided details on the actual interpretation of this function f, only that we aimed to optimize this function. Furthermore, we showed in Section 4.5 that we cannot use an instance-specific approach when solving satisfaction problems. To counteract this, we resort to evaluating the performance of a configuration on multiple instances, which we call a *batch* of training instances. In this chapter, we discuss the performance metric that we use to evaluate the performance of the target algorithm given hyperparameter configurations on these batches. This performance metric defines the objective function f that we aim to optimize.

The performance metric measures the performance of a configuration over a batch of training instances. The definition of the performance metric and the decisions regarding the construction of batches form the strategy to evaluate the performance of hyperparameter configurations. The configurations to be evaluated are those queried by the algorithm configurator over the search space. The search space, the search strategy of the configurator and the performance evaluation strategy are the three main components of the hyperparameter configuration problem (Li and Talwalkar, 2020). We discuss these components in Section 5.1. In the remainder of the chapter, we present the implemented design of the components that we use to solve the specific problems that we described in Chapters 3 and 4. The choices regarding the application of these components that we present in this chapter are motivated based on the observations and analysis of preliminary experiments that we do not discuss in detail in the remaining sections.

In Section 5.2, we present the configuration space that we use for the definition of the search space. Then, in Sections 5.3 and 5.4, we discuss the component of evaluating the performance of hyperparameter configurations and present difficulties that are the result of a satisfaction problem. In Section 5.5, we present the performance evaluation strategy that we use to estimate the performance of a configuration. We conclude this chapter in Section 5.6 with a revisitation of the workflow diagram that we presented in Figure 3.1.

5.1. Components of the hyperparameter optimization problem

The hyperparameter optimization problem is a widely studied area that is applicable to many different types of problems. Numerous strategies have been introduced that focus on different aspects of the problem at hand. In general, existing approaches that tackle the hyperparameter optimization problem rely on three components (Elsken et al., 2019; Li and Talwalkar, 2020):

1. the search space,
2. the search strategy,
3. the performance evaluation strategy.

The search space defines the set of possible hyperparameter configurations that can be queried by the algorithm configurator. The search strategy is the method that is used by the configurator to select new configurations to query. The performance evaluation strategy describes the method in which the estimation of the performance of a configuration is returned to the configurator.

Bayesian optimization and alternatives that we discussed in Chapter 2 are examples of search strategies. The surrogate model and the acquisition function that are used within the Bayesian optimization framework jointly determine the search over the configuration space. This configuration space is a part of the input of the Bayesian optimization algorithm (see Algorithm 1 and Figure 3.1) and defines the search space of the algorithm. In Section 5.2, we discuss the construction of the search space that we use for the problems that we introduced in Chapter 4.

The evaluation of the performance is the observation that the Bayesian optimization search strategy receives from the target algorithm after querying it with some configuration, see Figure 3.1. Whereas the search space is a part of the input to the Bayesian optimization algorithm that is received at the beginning and then remains unchanged throughout the procedure, the evaluation of the performance that is observed, is received as new input to the configurator in an iterative manner. These received observations, denoted by $y_1, \ldots, y_N$, are used to construct a surrogate model over the configuration space.

The evaluation of the performance of the hyperparameter configuration for the target algorithm can be defined in a variety of ways, using different interpretations of the outcomes of the target algorithms. Here we have to consider two factors. We want to evaluate a configuration with a performance metric that closely relates to our overall objective. On the other hand, we want this evaluation to provide informative observations. In many optimization problems, a suitable and intuitive evaluation of the performance of an algorithm is the value of the objective function that the algorithm optimizes. For example, for a machine learning classification model, this may be the outcome of the loss function; for a MIP solver, this can be the objective value.

In a satisfaction problem, we do not have such an objective function. We only get a binary response whether or not a feasible solution has been found. However, a surrogate model can naturally fit better as a response surface model over a set of real-valued numbers than over binary observations. In Chapter 4, we showed how to transform a satisfaction problem into an optimization problem by constructing an objective function. A logical response to the problem of binary evaluations would be to evaluate the performance of a configuration based on this newly constructed objective function. However, this is not a reliable method since this objective function contains a subset of the hyperparameters that we change over iterations. We discuss this in more detail in Section 5.3. Next, in Section 5.4, we present additional obstacles regarding evaluating the configurations. Finally, in Section 5.5, we present the performance metric that we use for the performance evaluation strategy.

5.2. Defining the search space

In this section, we elaborate on the first of the previously discussed components: the search space. We introduce the subspace of the full configuration space of the target algorithm that we then define as our search space. This search space is the input for the Bayesian optimization procedure, see Algorithm 1 and Figure 3.1. We aim to reduce the size of the search space by incorporating prior knowledge about the problem and the target algorithm at hand. This allows for a simplified search process. However, as mentioned by Elsken et al. (2019), we have to be aware that this also creates a human bias due to our beliefs of the configuration space. As a result, this can potentially prevent optimal configurations that are actually located outside of the reduced search space from being found.

5.2.1. Subspace of the complete configuration space

We denote the complete configuration space of the target algorithm $\mathcal{A}$ by Θ'. The dimensions of this complete space consist of every tune-able hyperparameter that exists within $\mathcal{A}$. The space Θ' is then composed over the full domain of every hyperparameter such that Θ' contains every possible hyperparameter configuration. For HIP this complete configuration space includes:

- boolean values that denote the use of every defined neighborhood relation;
- all configurable settings of the MIP-solvers that are used to compute the initial solutions;

- additional hyperparameters used in constraints and weight coefficients in the objective function of the problem formulations that are solved by the MIP-solvers;
- a linear weight and a shape value for every cost component in the objective function of the local search heuristic;
- a setting to decide on the type of local search heuristic to use;
- respective hyperparameters of this chosen local search heuristic, such as values for the perturbation phase of VNS and hyperparameters related to the temperature of simulated annealing.

This complete configuration space includes discrete, categorical, and conditional hyperparameters.

For the bin packing problem the complete configuration space only consists of the weight coefficients $\{\theta_1, \theta_2, \theta_3\}$ from objective function (4.16) and the hyperparameters from the variable neighborhood search metaheuristic.

Every hyperparameter that exists in some target algorithm adds an extra dimension to the configuration space. For example, the addition of a discrete hyperparameter with a domain consisting of K values to the configuration space increases the total size of the configuration space by a factor K. It is unlikely that every hyperparameter has an equally strong impact on the performance of the target algorithm. Several researchers, such as Bergstra and Bengio (2012), have shown that for certain problems, most dimensions do not change the objective function significantly. Multiple techniques have been proposed to exploit this low effective dimensionality property to find the most important dimensions, also within the Bayesian optimization framework using Gaussian processes (Wang et al., 2013). However, we do not know the effective dimensionality of the complete configuration space of HIP. Since the drawbacks of GPs include high dimensionality and complex search spaces, we decide to reduce the configuration space by only considering a subset of the hyperparameters.

5.2.2. Dimensions of the search space

In the complete description of all hyperparameters defined in HIP that we listed above, we already differentiated between multiple types of hyperparameters that introduce natural subsets (in the bullet points). We identify several interesting subsets of hyperparameters to consider in the configuration space. However, due to the limitation on available resources, we only consider a single subset for the remainder of this research. The hyperparameters from the subset of the complete configuration space that we choose to consider then become the dimensions that span the search space.

Hyperparameters related to the MIP-solver

First, a clear motivation for the reduction of the configuration space is the empirical impact of several hyperparameters on the performance. The list of hyperparameters presented above includes many hyperparameters that belong to the MIP-solver. This solver is only used to find a solution to a subproblem of the TUSP. The solution to this subproblem is then used as the initial solution of the local search heuristic.

We want to emphasize that, in theory, the position of this starting point on the search space can have a large influence on whether and when the global optimum is found. However, it is generally not known beforehand which initial solution will lead to a global optimum. Additionally, for all encountered problems, only a minor fraction of the overhead computation time is used to find this initial solution. As a result, this set of hyperparameters that correspond to the MIP-solver, which makes up the majority of the configuration space, has a very limited potential on the increase in performance that its configuration can yield. Therefore, we choose to not consider this large number of hyperparameters in our search space.

Hyperparameters related to the local search heuristic

We also identified multiple types of hyperparameters for the local search heuristic. This includes the type of heuristic to be used, the standard hyperparameters of the particular local search heuristic, and the use of the distinct neighborhood rules. These are standard hyperparameters that are usually tuned when optimizing the general performance of a particular type of local search algorithm on a variety of problems. However, we consider the specific problem of the TUSP (and bin packing as its subproblem) and a particular satisfaction problem formulation. Although we hypothesize that the optimization

of these hyperparameters related to the local search heuristic can result in an improvement of the performance of HIP, we expect that this potential improvement will be marginal. In particular, we expect that the optimization of this subset of hyperparameters will mainly result in finding feasible solutions more quickly compared to the default configuration. However, we think that is very unlikely that optimized configurations of these hyperparameters will yield feasible solutions that will not be found by using the default or any alternative configurations. Since the main goal is to find feasible solutions for as many problem instances as possible, we prefer to focus on hyperparameters for which we think that alternative configurations can actually yield feasible solutions for problem instances that so far have remained unsolved.

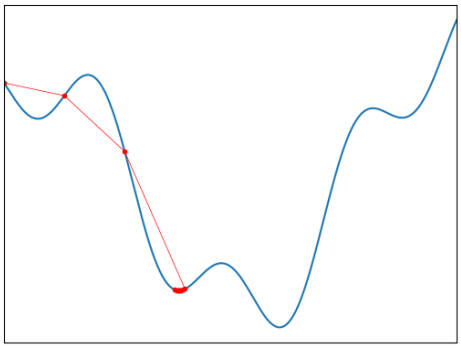
Hyperparameters related to the objective function
Finally, we consider the subset of the hyperparameters that denote the weight coefficients and shapes for the cost components in the objective function of the local search heuristic. In Chapter 4, we already mentioned that local search heuristics, when executed with different configurations of the weight coefficients, can result in different paths over the search space. This is a result of different preferences between candidate solutions due to the different objective functions. It then becomes possible to avoid, or even escape, local optima in which the local search would get stuck when executed with a different configuration. Consequently, there is a possibility to find a feasible solution to some problem given some configuration of weight coefficients in the objective function where this is not possible given a different set of coefficients.

In Figure 5.1, we visually show for every subset of hyperparameters that we discussed, that a different configuration can theoretically change the optimum in which the local search procedure gets stuck. The displayed local search heuristic is a standard hill climbing approach that considers candidates at a fixed step size away from the current position, and moves to the candidate solution that attains the smallest fitness value. The step size decreases if none of the candidate solutions have a smaller fitness value than that of the current solution in order to converge to a (local) optimum. The starting position is at the far left for the default configuration.

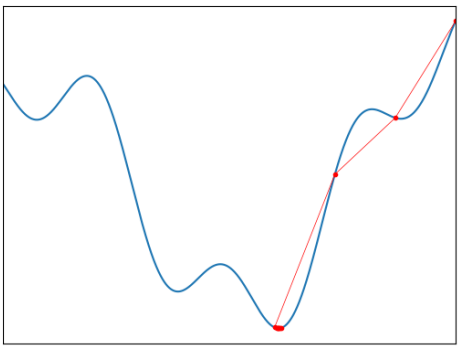
In the provided example, the local search heuristic with the default hyperparameter configuration gets stuck in a local optimum (Figure 5.1a). However, the global optimum is reached when using any of the alternative configurations (Figures 5.1b to 5.1d). These alternative configurations denote: a different configuration for the subproblem that defines the initial position; a different configuration of the local search hyperparameters (in particular: a larger initial step size); and different weights in the objective function. The figures that are presented here are two-dimensional representations to make it visually clear. They can also be interpreted as the objective values over a one-dimensional slice of a multi-dimensional search space. The horizontal axis then represents the value of one of the multiple hyperparameters. The visualization in Figure 5.1d then depicts a transformation of the objective function over the search space that would be obtained by increasing the weight of this particular hyperparameter, relatively to the other hyperparameters. As a result, the local search will prefer candidate solutions with larger values on the horizontal axis compared to the default hyperparameters.

We showed for the first two discussed subsets that it is possible to find the global optimum instead of the defaults local optimum (Figures 5.1b and 5.1c). According to our reasoning, this makes it theoretically possible to find feasible solutions to problem instances that have not yet been solved by using the default hyperparameter configuration. However, in practice, when considering a larger and more complex search space than shown in this figure, we expect this to not frequently occur. We merely expect that the optimization of these subsets can only yield a marginal improvement in performance compared to the default configuration by finding the feasible solution in less time. The reason for this is that we expect there to be less consistency between optimal initial positions in the search space and optimal local search hyperparameters for different problem instances. This makes it more difficult to generalize optimal hyperparameter configurations for a set of problem instances. Recall that we argued in Section 4.5 that an instance-specific approach for hyperparameter configuration is not possible for our particular problem. Since we consider a contextual approach, we focus on a subset of hyperparameters for which we expect there to be a strong correlation between optimal configurations for problem instances with the same contexts.

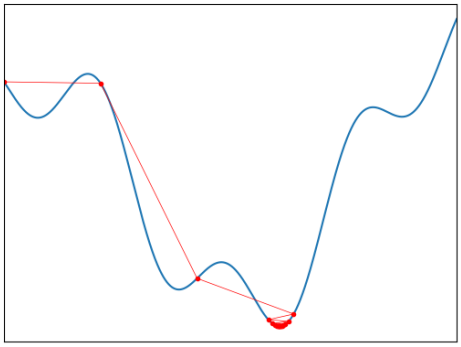
Therefore, we choose to limit the number of hyperparameters in the optimization to the set of hyper-



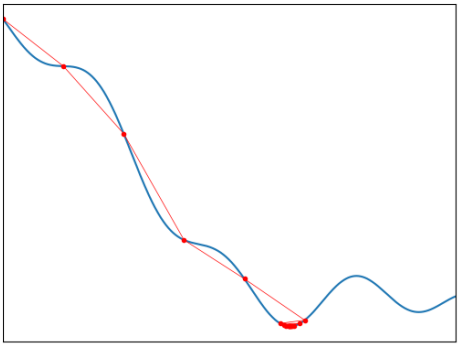
(a) Default hyperparameters.



(b) Alternative hyperparameters that result in a different initial position.



(c) Alternative local search hyperparameters (i.e. a larger step size).



(d) Alternative hyperparameters in the objective function.

Figure 5.1: Examples of runs of a hill climbing local search heuristic with different hyperparameter configurations over a fixed search space. The run with the default hyperparameters ends up in a local minimum; all alternatives in the global optimum.

parameters in the objective function of the target algorithm. We only consider the hyperparameters denoting the weight coefficients for every cost component in the objective function, not the shape-defining hyperparameters. We expect that including two hyperparameters per cost component would excessively increase the number of dimensions of the search space. We focus on the weight coefficients since it allows for an easier comparison between the relative differences of pairs of constraint violations. Especially after a logarithmic transformation of the search space, which we discuss in Section 5.2.4. In addition, we include the hyperparameter of the VNS that denotes the limit on the maximum increase in objective value per step in the perturbation phase. We consider this value since it is directly related to the value of the objective function. If we would omit this hyperparameter, a larger weight coefficient for some penalty variable would have two simultaneous effects. It increases the penalty variables relative importance compared to other penalties and it would reduce the number of additional penalties that are allowed between a step in the perturbation phase. We want to isolate these two effects, which can be done by including this hyperparameter that belongs to the VNS heuristic.

An additional advantage of this particular subset of hyperparameters is that it is generally applicable to different types of heuristics. For example, the hyperparameters in the objective function are not dependent on the actual type of local search heuristic that is used. This function is also applicable for global search heuristics such as evolutionary algorithms. This is not the case for the alternative subsets of the hyperparameters. A third advantage is that the chosen subset of hyperparameters consists of only continuous hyperparameters. By omitting all discrete, conditional and categorical hyperparameters, the resulting configuration space defines a continuous search space. As previously discussed,

such a space makes it easier to apply Bayesian optimization on, and is especially suitable for using a Gaussian process as a surrogate model.

Fixing one dimension of the search space
The set of hyperparameters that we now use to define the search space consists of the weight coefficients in the objective function and the hyperparameter of the VNS that denotes the limit on the maximum increase in objective value per step in the perturbation phase. The values of the set of hyperparameters represent a linear transformation of the objective function over the search space. This implies that, if we multiply all the hyperparameters with a constant value, nothing has changed. This is redundant and allows us to keep the value of one of the hyperparameter fixed, thereby effectively reducing the number of dimensions of the search space by one.

We choose the VNS hyperparameter as the dimension for which the value is fixed. If we multiply all configurable hyperparameters with a constant value, while keeping this VNS setting fixed, we decrease the weighted number of additional penalties over a random walk step in the perturbation phase by this constant factor. Therefore, the actual configuration subspace that we consider and that defines the search space consists only of the set of the weight coefficients in the objective function of the problem formulation.

5.2.3. Bounding the search space

In order to fit a Gaussian process as a surrogate model over the search space, this space must be defined as a finite space. Hence, every hyperparameter, represented as a dimension of the search spaces, must have a finite domain. We mentioned that the subset of hyperparameters of the target algorithm that we choose to define our search space consists of only continuous hyperparameters. Therefore, every hyperparameter must have a corresponding lower and upper bound. So far we only restricted the weight on all penalties to be strictly positive.

As mentioned in Section 4.3.1, HIP currently has a default hyperparameter configuration. Anastacio et al. (2019) have demonstrated how such default configurations can be exploited to construct a search space of reduced size, and thereby improving the efficiency of configuration algorithms. The authors propose three different reduction techniques that can be applied to continuous and discrete variables. They apply these to any continuous variable with a minimum current domain range of 1. We use the first of their proposed techniques since that is the only technique that does not require any initial bound on the domains. Here, we define the domain of every hyperparameter to begin at one-tenth of the default value and end at ten times the default value:

$$D_k = [0.1 \cdot d_k, 10 \cdot d_k] \tag{5.1}$$

Since this domain reduction technique requires a default hyperparameter configuration, we can only apply it to the TUSP. For the weight coefficients of problem (BinPen), we choose domains that we think are sufficiently large to contain the global optimum. Here, we are aware of the existing human bias and we do not guarantee that this is actually the case.

5.2.4. Logarithmic transformation of the search space

We now have a finite search space that is a subspace of the configuration space of the target algorithm. In order to improve the search over this space, we apply a logarithmic transformation to every dimension. We use a log-scale with base 2 for the weights θ_i. Hence, instead of directly optimizing the objective function $\sum_{i \in I} \theta_i p_i$, the local search heuristic optimizes $\sum_{i \in I} 2^{\theta_i} p_i$ and we recompute the actual values for θ_i. Such techniques are often applied to properly scale the differences between values of hyperparameters, see for example Dauphin et al. (2015). As an example, a domain of $[1, 100]$ is transformed to $[0, 6.644]$ such that the effective difference that is observed by the search strategy is equally large between 1 and 2 as between 50 and 100. We choose to use base 2 for this transformation since the default hyperparameter configuration consists of many numerical values that are multiples of two of other hyperparameter values.

The logarithmic scaling transforms the domain around the default configuration for every hyperparameter k from Equation (5.1) to

$$D'_k = [d_k - \log_2 10, d_k + \log_2 10]. \tag{5.2}$$

This means that every hyperparameter has a domain length equal to $2 \cdot \log_2 10$ and, as a result, it never subceeds the minimum domain range of 1 for continuous variables that Anastacio et al. (2019) proposed. Therefore, the technique of defining the domain of a hyperparameter around the default configuration can safely be applied to every hyperparameter.

5.3. On the use of the hyperparameterized objective function in the evaluation of configurations

So far we have discussed the first two of the three identified components of the hyperparameter optimization problem. Please recall that the second component, the search strategy, has been discussed in Chapter 3 in the form of Bayesian optimization. For the remainder of this chapter, we consider the third component: the performance evaluation strategy.

In Chapter 4, we showed that we can transform the satisfaction problem formulations that we consider for the bin packing and the TUSP to optimization problem formulations by constructing objective functions using the penalty method. We motivated that the binary evaluations that denote whether or not a satisfaction problem has been solved make it difficult to fit a response surface model. A logical response to this issue would be to evaluate the performance of a configuration based on this newly constructed objective function. In this section, we show that this is not a reliable method since we would then evaluate the performance for a configuration of hyperparameters based on varying hyperparameter values.

To verify that we cannot use the objective function that we constructed in our transformation, we first assume that this actually is possible. That is, we want to find the configuration that minimizes the function $\min_\theta \sum_{k=1}^{d} \theta_k p_k$, where the penalty variables p are the result of running target algorithm $\mathcal{A}$ with configuration θ. Since, by definition, the penalties are non-negative, and we only assign strictly positive weights to such penalties, we have the bounds $p_k \geq 0$, $\theta_k > 0$ for $k = 1, \ldots, d,$. Clearly, we find the minimum to the function stated above for $\theta \to 0$. In this scenario, we would completely ignore the outcome of the target algorithm and with that, whether we can actually find a feasible solution.

Alternatively, we can remove the varying values for θ from our objective function, and aim to find the configuration that yields the smallest value $\min \sum_{k=1}^{d} p_k$. This objective function denotes the (unweighted) total sum of the penalty variables. It can be interpreted as treating every penalty variable as if they are of equal importance. Since we simultaneously vary the values of hyperparameters that denote the relative importance of the different constraint penalties, we arrive at a contradiction. Consider for example a simple problem on a two-dimensional configuration space. We assume that the search space is sufficiently small that the target algorithm finds and returns the optimal solution from the search space according to the objective function. The problem formulation consists of two penalty variables: $p = (p_1, p_2)$ and we want to find the configuration $\theta^* = (\theta_1^*, \theta_2^*)$ with which the target algorithm finds a solution that minimizes the function $\min_\theta p_1 + p_2$. The first queried hyperparameter configuration is $\theta^A = (100, 1)$. The optimal solution found by the target algorithm is $p^A = (1, 0)$. The second hyperparameter configuration that is queried is $\theta^B = (1000, 1)$. The optimal solution found by the target algorithm is $p^B = (0, 999)$. The objective function under consideration deems the first hyperparameter configuration to yield a better performance, since $p_1 + p_2 = 1 < 999$. However, by calling the target algorithm with configuration $\theta = (1000, 1)$, we explicitly define the target algorithm to value the solution $(0, 999)$ over $(1, 0)$.

This basic example that considers the sum of penalties is simply a specification of the more general function $\min \sum_{k=1}^{d} w_k p_k$, with $w_k = 1$ for all $k = 1, \ldots, d$. Any specification of this function with values w_k to evaluate the outcome of θ yields a bias for configurations that have the same relative differences for θ_k as w_k. In the example above, there is a bias to configurations with similar values for θ_1 and θ_2, that results in a preference of θ^A over θ^B.

This shows that we can not use the artificial objective function that we constructed with the penalty method, or any variation that assigns a relative preference to different penalty variables, in the performance metric of the performance evaluation strategy.

5.4. Evaluating on batches of problem instances

The hyperparameter optimization problem considers the optimization of the performance of the target algorithm on multiple problem instances or datasets. In this research, these are varying problem in-

stances of the TUSP. In general, one can either opt for instance-specific hyperparameter optimization, or for generalized hyperparameter optimization. The goal of instance-specific optimization is to find a specific configuration for every problem instance. However, we showed in Section 4.5 that we should not use an instance-specific approach when considering satisfaction problems. The aim of generalized hyperparameter optimization is to find a configuration that works for all instances of a specific problem. Generally speaking, it is not a good idea to use a single hyperparameter configuration for every instance or dataset of a given problem. This should be fairly obvious: the reason that hyperparameters exist in the form of tune-able control parameters within the *programming by optimization* paradigm is that different values yield a better performance for different problems. Otherwise, the issue of optimizing hyperparameters would not be such a problem.

We consider the contextual approach where we want to find configurations for particular subsets of the problem instances and do so by grouping similar problem instances that share the same context into these subsets. In order to properly evaluate the performance of a configuration on instances of these subsets, we need to have multiple instances with equal contexts to train on. We also need instances that will never be used during the training to allow for an unbiased evaluation of the final performance.

Hence, we need to split the available problem instances per context into a training test and a test set. Next, we consider the set of problem instances to actually train on. We refer to the set of problem instances that we use to evaluate the performance of the target algorithm with some configuration during the training phase as a *batch* of problem instances. This batch of instances is a subset of the available training instances.

5.4.1. Selecting the batches

After dividing the available problem instances for every context value over a training set and a set test, we define the batch of problem instances that is used to evaluate a hyperparameter configuration. We can make several decisions regarding the composition of the training instances in this batch, possibly varying per call to the target algorithm.

The batch size

We first decide on the number of training instances K that defines the size of the batch. A common pitfall in the field of machine learning is to choose a too small batch size. This introduces a possibility to overfit (or *over-tune*) on this small number of training instances, leading to a poor generalization to unseen instances. In addition, the use of small batches resembles instance-specific tuning. Clearly, a generalized approach that uses batches that consist of only a single problem instance is equivalent to instance-specific optimization.

On the other hand, there are also disadvantages to choosing a large batch size. It is possible to define the batch as the set of all problem instances in the training set to evaluate the performance on all training instances. This, however, results in an M-fold increase in computation time, with M denoting the available number of training instances. Recall that the expensive evaluation of the target algorithm is the bottleneck of the optimization procedure. This M-fold increase of the computation time could also have been used to increase the stopping condition T of the target algorithm; or to increase the number of iterations N of the BO procedure (and hence, the number of sampled configurations).

We can interpret the size of a batch as a parameter with domain $[0, 1]$ that represents the fraction of available training instances that become part of the batch. This (hyper)parameter is generally ignored by experimenters by setting it to 1. In this problem setting, we are restricted to a smaller value for this parameter, but it cannot be too small. Research has been done by Klein et al. (2017) that combines the topics of multi-fidelity and contextual Bayesian optimization by considering the batch size as the percentage of the number of available instances as the context. This is related to multi-fidelity since smaller percentages are cheaper to evaluate, but provide less accurate measures of the performance. The goal is then to find the optimal configuration for the complete number of datasets. We have chosen not to explore this approach since it requires small subsets to be very representative of the complete set of problem instances. This puts more emphasis on the heterogeneity of the training set which we discuss next. The problem of heterogeneity seems to be not an existing issue within the experiments (on the MNIST datasets) that are considered in this research.

The batch sampling strategy
Next, we need to choose a strategy to select the batch as the subset of the available training instances. A common approach is to sample a new subset of training instances per iteration (Hutter et al., 2013). Thereby, we evaluate the performance of the target algorithm using different configurations on different problem instances. The advantage of this strategy is that the training process will not overfit on a particular small set of problem instances, without the need for a large batch size.

However, preliminary results have shown that this approach provides very poor results on our particular problem. This is due to the heterogeneity in the problem instances. This heterogeneity denotes a large difference in the level of difficulty of problem instances, even after dividing all instances based on a variety of different contexts. As a result, evaluating a configuration multiple times yields very different observations of the performance evaluation. These differences due to the different sampled batches cannot be properly explained by the noise variable ε. This makes it nonsensical to try to fit a response surface model over the received observations. When applying this strategy of sampling new batches per queried configuration, we observed the evaluation of a configuration to be more dependent on the general difficulty of the sampled problem instances than the influence of the queried configuration on the target algorithm. It has become apparent that, at the end of the Bayesian optimization procedure, the optimal configuration θ^* does not represent the most promising configuration on a new set of instances. The search strategy instead prefers some queried configuration that has been evaluated on a drawn sample that contains relatively many training instances of low difficulty.

In general, if the instance set is shown to be heterogeneous, an instance-specific approach should be used (Eggensperger et al., 2019). However, we have shown that this is not possible. Therefore, in order for this sampling strategy to work, we need to address the heterogeneity. A possibility is to split into more contexts by considering a multitude of problem features simultaneously. This can lead to more homogeneous problem instances per split. However, it also increases the number of contexts that need to be considered. As a result, this either yields another M-fold increase of the number of configurations that we need to find; or an addition of M dimensions to the joint search space.

Additionally, a larger number of considered contexts restricts the number of available problem instances for each training set. This affects the possible batch size K. In the scenario that the chosen features to split on result in a context that is only met by a single scenario, we again have to resort to an instance-specific approach. If not even a single of such instances exists in the training set, we do not even train a configuration for this context. Then, the mapping h is incomplete and we cannot map an instance from the test set with this context to a configuration.

We use the alternative strategy of sampling a single subset of the available training instances as our batch; and keeping this set fixed during the training procedure. Thereby we ignore much of the available training data, and we have the possibility to overfit on this data, especially on small batch sizes. But it allows for a consistent evaluation of the performance of the configurations, which is essential during training to find well-performing configurations.

Selecting the instances
Finally, we decide on which available training instances become part of the fixed batch that we use to evaluate the performance. We mentioned above, as a pitfall to random sampling batches of training instances per iteration, that there is a lot of heterogeneity in the training set. If we only choose a subset of the available set of training instances to actually train on, we can benefit from having more *informative* problem instances in this batch. For example, if a training instance is really easy to solve, the local search heuristic can find a feasible solution within a very small number of steps, regardless of the hyperparameter configuration that is used. Such an instance will have an equal contribution to the evaluation of the performance for every queried configuration. It becomes difficult to differentiate between poor and well-performing configurations based on these problem instances. Therefore, we want to avoid the appearance of too easy problem instances in our subset of training instances.

We remark that the presence of too difficult instances are also disadvantageous in the training process. However, finding a feasible solution for a problem instance that had been deemed to be difficult should be awarded in the evaluation of the configuration that was used to solve the instance. Therefore, we do not aim to filter any difficult problem instances.

We perform a preprocessing step in which we consider every available training instance and flag those that we consider to be too easy to properly train on. We deem a problem instance to be too

easy if a feasible solution is found with a very small number of local search steps, regardless of the hyperparameter configuration that is used. In this case, it has become apparent that the configuration is not determinative to quickly find such a feasible solution. We first generate a set of random hyperparameter configurations by uniformly sampling random points from the search space. We then apply the local search heuristic to every training instance with every configuration from this set. Here, we use a very small stopping condition such that this preprocessing step does not have a large effect on the overhead time. If, for a certain training instance, a feasible solution has been found within this stopping condition for every configuration, we flag this particular instance. We construct the set of training instances from which we sample the batches to train on per run, using only unflagged problem instances. All flagged instances are moved from the training set to the test set. The pseudocode for the procedure of selecting the problem instances in the training set is presented in Algorithm 2. Here, $U(\Theta)$ denotes the uniform distribution over the search space. The unflagged problem instances that are a part of the training set, but do *not* belong to the selected batch, are not moved to the test set. This way, the test set remains unchanged between different runs such that we can fairly compare the respective performances over the test set.

The flagged instances that are moved from the training to the test set have thus been shown to be quickly solved using multiple configurations. This implies that they are very likely to be quickly solved by any configuration from the search space. However, this is not a guarantee. Therefore, we still want to include these instances in the test set. In the unlikely scenario that a configuration does not result in quickly finding a feasible solution, this is then still affected in the evaluation of the performance of this particular configuration. At the same time, the addition of these instances to the test set does not strongly impact the total computation time. It is important to point out that the test set is no longer a true representative of the population of problem instances. By considering an oversampled number of easy instances, the evaluation of the performance on the test is biased. We argue that this is not an issue since the training set is also no longer a representation of the population after flagging and filtering of easy instances. Hence, the training and the test set do not follow the same probability distribution. Furthermore, all problem instances that we consider are generated and do not represent a set of real-life TUSP instances. Therefore, the absolute values of the performance on the test set have no real meaning; the test performances should mainly be regarded in comparisons. The effect of overrepresenting the number of easy instances in the test set on the percentage of solved instances is that the relative differences in performance between different configurations are reduced, but the absolute differences in the number of instances that have been solved remain unchanged.

Algorithm 2 Instance selection

Input: stopping condition T, number of configurations C, $\mathcal{A}$,$\mathcal{I}^{train}$,$\mathcal{I}^{test}$

generate $\mathcal{C} = \{\theta_1, \ldots, \theta_C\} \overset{i.i.d.}{\sim} U(\Theta)$
for $i \in \mathcal{I}^{train}$ **do**
  for $\theta_c \in \mathcal{C}$ **do**
    call $\mathcal{A}$ with θ_c, observe time to terminate $t_i(\theta_c)$
  end for
  if $t_i(\theta_c) < T \ \forall \theta_c \in \mathcal{C}$ **then**
    move i from $\mathcal{I}^{train}$ to $\mathcal{I}^{test}$
  end if
end for
return $\mathcal{I}^{train}$, $\mathcal{I}^{test}$

5.5. The performance evaluation metric

In this section, we introduce the performance metric that we use to evaluate the quality of a hyperparameter configuration θ on the target algorithm $\mathcal{A}$. This performance metric defines the function $f(\theta, z)$ that is being optimized for every z. It also denotes the value that is observed by the Bayesian optimization framework in the form of $y = f(\theta, z) + \varepsilon$ to update the surrogate model.

We mentioned in Section 3.1 that the output of this function f can represent any metric on the performance, such as the time to find the obtained solution or some measure on the quality of the final solution. It is important to choose a suitable performance metric in order for the Bayesian optimization

procedure to properly train to find optimal configurations. Since we consider a satisfaction problem, the primary objective of the target algorithm given a problem instance is to find a feasible solution. The secondary objective would be to find this solution within a reasonably small computation time. The most straightforward evaluation of a configuration would then be to use a metric that describes whether a feasible solution has been found for a particular problem instance within reasonable time. However, this performance metric would be binary which is hard to deal with. In Section 5.4, we motivated the use of a batch of instances during the evaluation of a configuration. This makes it possible to add more variation over the received evaluations.

We first introduce a performance metric $\tilde{f}(\theta, i)$ that measures the performance of a configuration on a specific instance i. We then measure this performance for every instance i in batch I^z. The batches of problem instances are composed of instances that have the same context such that the batch of instances I^z is a subset of the set $\mathcal{I}^z$. We then define the performance metric for a context z by taking the average of the performance metric over all instances in the considered batch that attain this context:

$$f(\theta, z) = \frac{\sum_{i \in I^z} \tilde{f}(\theta, i)}{K}. \tag{5.3}$$

Here, K denotes the batch size, which is equal to $|I^z|$. This way, we obtain a performance metric f for context z that we can use in Equation (3.1).

We now define the performance metric on a specific instance $\tilde{f}(\theta, i)$. First, for the main factor of the performance of a single instance, we consider the primary objective. Let $obj_i(\theta)$ denote the objective value of problem (Sat-P) that is obtained by target algorithm $\mathcal{A}$ when using configuration θ on instance i. The term $\mathbb{I}\{obj_i(\theta) = 0\}$ then represents whether a feasible solution has been found to problem (Sat) using θ for instance i.

Second, we add a relatively small penalty based on the time it takes the target algorithm to find the (possibly feasible) solution. Let T denote the stopping condition of the target algorithm for a single problem instance. This condition is either defined as the maximum allowed time in seconds or as the maximum number of local search steps. The run-time of the target algorithm using configuration θ is represented by $t_i(\theta)$ for instance i and is defined with the same unit of measurement as the stopping condition T. Then $t_i(\theta)/T$ denotes the percentage of the stopping condition that is used by the target algorithm with configuration θ to solve instance i. Since $t_i \leq T$, $\forall i \in \mathcal{I}$, this fraction attains a theoretical minimum of 0 and a value of 1 if no feasible solution is found within T. We only want this term to give a relatively small penalty compared to the first term. In the worst case, it still should not exceed the impact of solving one additional instance in the batch. Therefore, we divide this term by K to limit its maximum value. Then, we deduct this term from the indicator function to obtain the following performance metric for a single instance:

$$\tilde{f}(\theta, i) = \mathbb{I}\{obj_i(\theta) = 0\} - \frac{t_i(\theta)}{K \cdot T}. \tag{5.4}$$

Finally, we take the average of this performance metric over all instances in the batch to arrive at the final performance metric:

$$f(\theta, z) = \frac{\sum_{i \in I^z} \left(\mathbb{I}\{obj_i(\theta) = 0\} - \frac{t(\theta)_i}{K \cdot T} \right)}{K}. \tag{5.5}$$

5.6. Specification of the workflow revisited

To conclude this chapter, we briefly revisit the specification of the workflow that we presented in Figure 3.1. The revised workflow is visualized in Figure 5.2. The describes changes are also applicable to the workflow of the bin packing problem by replacing the TUSP for the bin packing problem and HIP for the VNS local search algorithm from Section 4.4.2.

First, in Chapter 3, we introduced that the algorithm configurator that we consider is a Bayesian optimization strategy with a Gaussian process as its surrogate model. In Chapter 4, we introduced HIP and the TUSP as the target algorithm and the problem that it solves. In this chapter, we specified the following changes that are now apparent in the revised workflow:

- The search space is defined as a subspace of the complete configuration space of HIP.
- A call of Bayesian optimization to the wrapper includes a stopping condition T

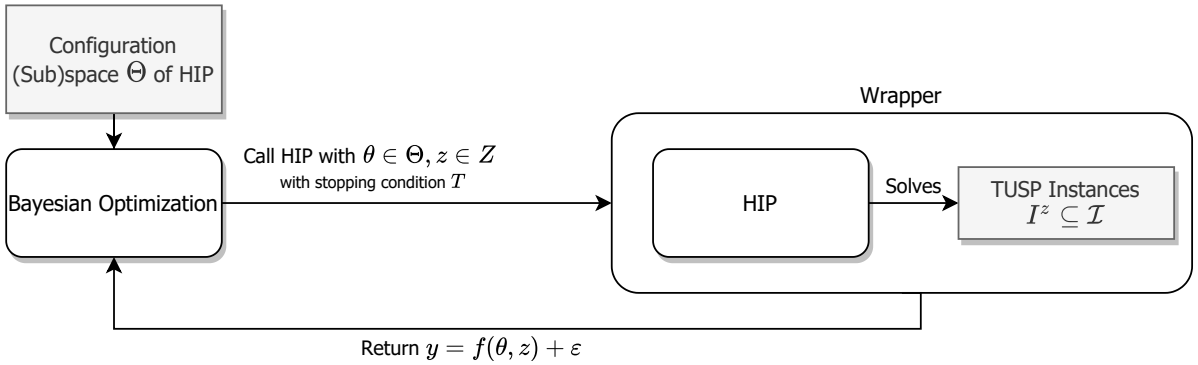


Figure 5.2: Specification of the workflow of contextual algorithm configuration (Eggensperger et al., 2019).

- HIP solves a batch of problem instances that attain the queried context z, instead of a single instance
- The value that is returned by the wrapper is the summation of the performance metrics for all instances in the batch.

6

Experiments on the bin packing problem

In Chapter 4, we introduced the problems including their respective problem formulations that we study in this research. Afterwards, in Chapter 5, we defined the search space and the performance metric to evaluate the performance of queried configurations. In this chapter, we introduce the setup and results of the experiments on the bin packing problem. These experiments regard the problems from Chapter 4 and include all design choices that we motivated in Chapter 5. The experiments on the train unit shunting problem are discussed in the next chapter.

Each proposed experiment is designed to tackle (a sub-question of) one of the research questions, see Chapter 1. The experiments on the bin packing problem focus on the first sub-question regarding whether Bayesian optimization in a contextual approach finds hyperparameter configurations for different contexts that obtain a better performance compared to a context-free approach. To answer this question, we compare two different strategies: a context-free approach and a contextual approach. The context-free approach is the most standard strategy: when encountering a new problem instance, we do not consider its problem features. Thus, we simply aim to find a single hyperparameter configuration that yields a generally well performance on all problem instances. To obtain this configuration, we train on any set of problem instances regardless of their contextual information. In the contextual approach, we observe the problem features of a newly encountered problem instance and we aim to use a specific configuration based on the observed context. To find these configurations, we train on sets of problem instances where each set only contains instances with the same contexts; and we aim to optimize the configuration for every such set in isolation.

We first introduce the experiments applied to the bin packing formulations in Section 6.1. Then, we discuss the results of the experiments on the bin packing problem in Section 6.2.

6.1. Experimental setup

We first consider the bin packing problem that we discussed in Section 4.4. We introduced this particular problem since it is one of the subproblems of the train unit shunting problem that occurs when we have to park trains on tracks of the shunting yard. In this setting, both the bins and the items can have varying sizes to represent the trains and the tracks that have varying lengths. We solve this problem by optimizing problem formulation (Bin-P), a transformed version of the satisfaction problem formulation (Bin). To find the optimal solution to instances of (Bin-P), we use variable neighborhood search (VNS) as a local search heuristic. This local search algorithm is designed to resemble HIP as closely as possible. We provided details on the local search that we use in Section 4.4.2.

We previously mentioned that we can reduce the number of dimensions in the search space by one by setting the limit on the maximum increase in objective value per step of the VNS algorithm to a fixed value. In this set of experiments, we consider a simplified problem where we aim to investigate the proposed first research sub-question in a more isolated setting, and not to find the best attainable performance. Therefore, we choose to reduce the dimensions of the search space by two. We do so by fixing $\theta_2 = 0$, or $\theta_2 = 1$ prior to the logarithmic transformation. We choose this particular weight coefficient since θ_2 and θ_3 are closely related. Also, the limit on the maximum increase in objective value per step of the VNS algorithm hyperparameter is fixed at 10. The configuration θ_n from Algorithm 1

can then be represented as $\theta_n = [\theta_{1n}, \theta_{3n}]$. The bin packing problem and the local search heuristic are specifically designed to tackle the research question and therefore no default hyperparameter configuration exists. Hence, we can not construct a finite domain around a default configuration d_k as defined in Equation (5.2). We define the following bounds on the search space: $[2^{-5}, 2^5]$ and $[2^{-3}, 2^3]$, which yields domains of $\theta_1 \in [-5, 5]$, $\theta_3 \in [-3, 3]$, respectively, after the logarithmic transformation. For this set of experiments, we use the computation time in seconds as the stopping condition per problem instance. The target algorithm is called with stopping condition $T = 0.2$ seconds per instance.

Let N denote the budget on the number of iterations of the Bayesian optimization procedure, see line 1 of Algorithm 1. The total budget on the number of queried configurations to target algorithm $\mathcal{A}$ is equal to N plus the number of initial points. The initial points to construct the prior Gaussian process are uniformly sampled from the search space. The number of initial points is equal to $2 \cdot d$, with d denoting the number of dimensions of the search space. The number of dimensions is equal to the number of tune-able hyperparameters such that $d = 2$ for the bin packing problem and we only use 4 configurations uniformly sampled over the search space to construct the initial surrogate model. We use $N = 200 - 2 \cdot d$ for the context-free approach. For the contextual approach, we divide the available budget on the number of iterations evenly over the two surrogate models. The optimal configuration θ^* over the training procedure is defined as the configuration that yielded the largest observed value of the performance metric, as defined in Equation (5.5), over the batch of training instances. That is,

$$\theta^* = \arg\max_{\theta_i} \{y_i \in \mathcal{D}_N\}. \tag{6.1}$$

We refer to the observed performance metric of the optimal configuration as the *training score*.

We then evaluate the performance of this optimal configuration by running the target algorithm with this configuration on all available test instances. For this evaluation, the test performance of the configuration is defined as the percentage of instances for which a feasible solution has been found within the stopping conditions, also referred to as the *test score*. We no longer account for the number of local search steps in which such a feasible solution was found, as in the performance metric on the training set. This factor was added to the definition of $f(\theta, z)$ from Equation (5.5) to differentiate between configurations that resulted in the same number of solved instances in a constant batch. For the overall performance, however, we are only interested in the total percentage of instances for which a feasible solution is found, given a hyperparameter configuration. The stopping condition per instance T that is used to evaluate the performance on the training set is equal to that during training.

We perform 5 different runs of the Bayesian optimization algorithm, where each run consists of the training procedure followed by an evaluation of the optimal configuration over the test set. The runs vary by training different surrogate models due to the stochasticity in the observations of the performance of the target algorithm. We report on the average test performance, including the standard deviations. We report the values of the one-sided paired t-test ($\alpha = 0.05$) to test the statistical significance of the differences in performance between the configurations obtained by the context-free approach and the contextual approach.

6.1.1. Context: heterogeneous sizes

To be able to study whether we can use contextual information when searching for optimal hyperparameter configurations to obtain a better overall performance, we first need to define a context dimension such that we can assign every problem instance to a corresponding context. For the bin packing problem, we first introduce two distinct problem versions where we differentiate between heterogeneity and homogeneity of the item and bin sizes. We then show how these problem versions can be used in the definition of the context dimension.

We introduce two distinct problem versions that instantiate the (Bin-P) problem. Version A contains $|I|$ items and $|J| = |I|$ bins. The sizes of items s_i and bins b_j are equal for all $i \in I, j \in J$. There are multiple feasible solutions to instances of this problem version. If we represent a solution by the allocation of items to bins as a binary matrix X, with respect to variable x_{ij} as defined in Constraint (4.17), then we can state that every permutation matrix of size $|I| \times |I|$ represents a feasible solution to problem version A with $|I|$ items. That is, every problem instance of version A has $|I|!$ feasible solutions. Version B also contains $|I|$ items and $|J| = |I|$ bins. All items and bins are now heterogeneous with unique pairs of item sizes and bin capacities: $b_1 = s_1 = 1$, $b_i = s_i = b_{i-1} + 1$, $\forall i \in \{2, \ldots, |I|\}$. Problem instances

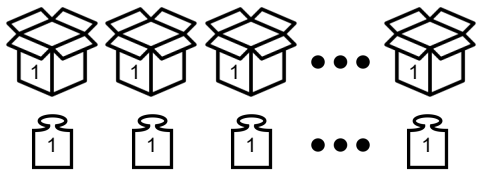
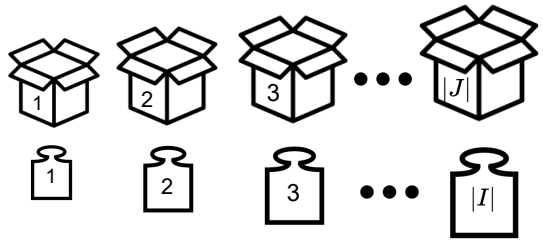


Figure 6.1: (a) Representation of problem version A of the bin packing problem. The sizes of the bins and items are homogeneous. Any one to one matching of items to bins is an optimal solution.



(b) Representation of problem version B of the bin packing problem. The sizes of the bins and items are heterogeneous and step-wise increasing by one. The unique optimal solution is a matching of items to the bin of equal size.

of this version have a single feasible solution, with every item assigned to the unique bin of equal size. It then holds that the unique feasible solution to this instance is the $|I| \times |I|$ identity matrix $I_{|I|}$. This means that all considered instances of the bin packing problem are satisfiable. That is, a feasible solution exists for every problem instance. The differences between problem version A and version B are illustrated in Figure 6.1.

To illustrate, if we have $|I| = 3$. Then the instance of version A has bin sizes $b = [1,1,1]$ and item sizes $s = [1,1,1]$. The set of $|I|! = 6$ feasible solutions X consists of:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}, \begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}, \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}, \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}, \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}, \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}$$

The instance of version B has bins of sizes $b = [1,2,3]$ and item sizes $s = [1,2,3]$. The unique feasible solution is I_3, the first matrix listed above.

We now define the context dimension to be the absence or presence of heterogeneity of the sizes of items and bins. The context space is thus defined by a single dimension with two context values. We can observe this contextual information for every encountered problem instance. Therefore, this feature can be used to optimize different hyperparameter configurations. We construct separate surrogate models for the different contexts and train them according to the Bayesian optimization procedure in isolation. For every newly encountered problem instance, we then use the best configuration that has been obtained by the Bayesian optimization search strategy for the corresponding observed context.

The batches of problem instances that we use during the training are constructed by using step-wise increasing values for $|I|$ for both problem versions. We use a batch of size $K = 10$ with a minimum $|I| = 5$ such that the batch contains instances of sizes $|I| = \{5,6,\ldots,13,14\}$. For the contextual approach, the batch contains one instance of every size of the respective problem version. For the context-free approach, we randomly sample one of the problem versions per size. The sizes of the considered instances have been determined by preliminary testing such that none of the problem instances can be solved in a very small number of local search steps, regardless of the configuration that is used. Therefore, the preprocessing step that filters easy problem instances, as described in Section 5.4, is redundant for this experiment.

To evaluate the performance of the obtained hyperparameter configurations on the test set, we again consider two different sets. The first contains instances of version A, the other of problem version B. Each set contains problem instances of 5 different sizes, with $|I| = \{9,10,11,12,13\}$ for version A and $|I| = \{6,7,8,9,10\}$ for version B. The local search procedure is applied to every batch 200 times, yielding a total of $1,000$ instances per problem version. We also consider $1,000$ instances per approach for every size $|I|$ in isolation.

6.1.2. Test environment

The Bayesian optimization framework and corresponding Gaussian process implementation that we use to execute the Bayesian optimization procedure from Algorithm 1 is available online, see Nogueira

(2014). We use the reported default settings from version 1.2.0 of this module. That is, the Gaussian process uses a Matérn kernel with $\nu = 2.5$. This is a widely used kernel function for Gaussian processes that was used by, for example, Hutter et al. (2013) and Klein et al. (2017) for the hyperparameter optimization problem. We use the Gaussian Process Upper Confidence Bound (GP-UCB) acquisition function over the current Gaussian process to iteratively determine the next configuration to query, as defined in Equation (3.5). The hyperparameter β from this equation that balances exploration and exploitation is set to 2.576. This is the default acquisition function of the available library. The control parameter that determines the level of flexibility of the Gaussian process to handle noise is set to 0.1. We use the default method of this model of uniformly sampling the initial points over the search space to construct the prior Gaussian process.

The target algorithm that is used to solve any problem instance is a variable neighborhood search heuristic with a minimum and a maximum number of random walks, which define the possible sizes of the neighborhoods, of 3 and 6, respectively. The maximum increase in objective value per step is 10 for the bin packing problem. The target algorithm can be called with a specified seed to control the stochasticity of the algorithm. We always call the target algorithm with the same seed. This ensures that the number of steps of the local search to find a feasible solution for some hyperparameter configuration on a problem instance can be exactly replicated by calling the target algorithm with the same configuration. The source code for the experiments on the bin packing problem is available at `github.com/LvdKnaap/BinPacking`. All bin packing experiments were conducted on an HP laptop with Intel Core i7-7700HQ CPU and 16GB of RAM.

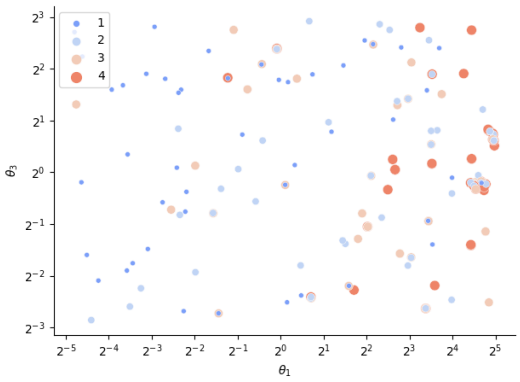
6.2. Results

In this section, we present the results of the experiments that we described in this chapter. The focus of this section is to identify whether we can use available contextual information when searching for optimal hyperparameter configurations to obtain a better overall performance. We aim to do so by considering the formulation of the bin packing problem that we presented in Section 4.4. We compare the performance of the target algorithm with configurations obtained by Bayesian optimization in a context-free approach with the performance of configurations from a contextual approach. For the remainder of this section, we refer to these obtained hyperparameter configurations values as $\theta^{Combined}$, $\theta^{Isolated-A}$ and $\theta^{Isolated-B}$ for the context-free approach, and the contextual approach for problem versions A and B, respectively.

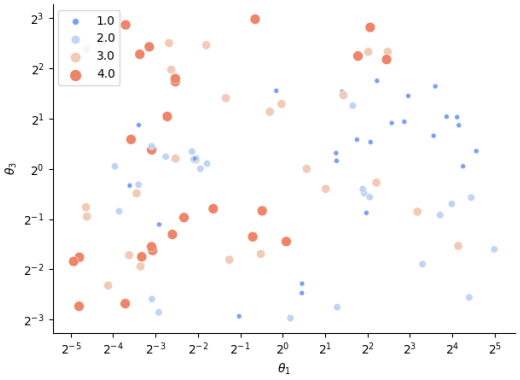
6.2.1. Performance on the training set

We first consider the performance of the different approaches on the training set. We start by visualizing the process of finding the optimal hyperparameter configurations for the context-free and the contextual approach. At the end of the Bayesian optimization algorithm, we have an observation history $\mathcal{D}_N = \{(\theta_{11}, \theta_{31}, y_1), \ldots, (\theta_{1N}, \theta_{3N}, y_N)\}$. Recall that the value used for θ_2 is fixed to simplify the optimization problem by reducing the search space. In order to clearly visualize these tuples of length 3, we present the observation history as a two-dimensional visualization where the obtained values y are discretized and visualized by markers of different colors and sizes. Scatter plots that denote the observation histories of both approaches are presented in Figure 6.2. The three observation histories visualized together denote the queried configurations one of the 5 runs of the Bayesian optimization algorithm from Algorithm 1. In this figure, all queried values of θ_1 are plotted on the x-axis and the corresponding θ_3 on the y-axis. Every marker represents a queried hyperparameter configuration. The legend differentiates between the hyperparameter configurations based on the observations of the performance metric $y = f(\theta, z) + \varepsilon$. The 25% of configurations that yielded the smallest performance are visualized with small, blue markers, and those configurations with the 25% largest values are denoted by large, red markers. The acquisition function for the Gaussian combines the notions of exploration and exploitation. As a result, the regions with a high density of markers generally represent the queries that *exploit* previously obtained queries that attained a large value of the performance metric. Regions of low density are queries that aimed to *explore* unobserved regions of the search space.

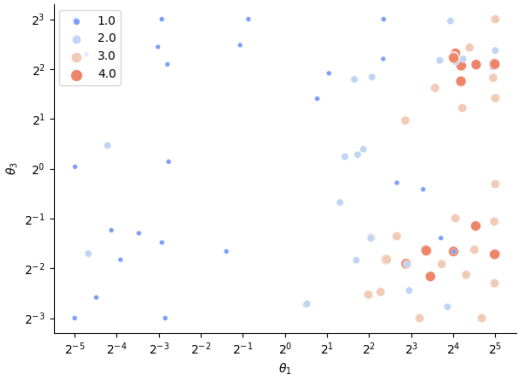
Figure 6.2a displays all values for (θ_1, θ_3) that have been queried in training the context-free approach. We can observe from this figure that a better performance seems to be attained for hyperparameter configurations that consist of relatively large values for θ_1 and, to a lesser extent, values for θ_3 that are quite close to 1. However, it is not directly clear what specific region of the search space yields



(a) Context-free approach



(b) Contextual approach for problem version A



(c) Contextual approach for problem version B

Figure 6.2: Sampled weight values for the context-free and the contextual approach. The queried values for θ_1 and θ_3 are shown on the horizontal and the vertical axis, respectively. The large, red (small, blue) markers denote the 25% of the queries that attain the largest (smallest) performance metric.

a good performance. The queried configurations for the contextual surrogate models are visualized in Figures 6.2b and 6.2c. To account for a constant budget of function evaluations per approach, we have divided the available budget evenly over the isolated problem versions. We clearly see different patterns for the queried configurations between problem versions A and B. Most of the queries that attain a large value of the performance metric are now located in different regions of the search space. The markers for problem B are located densely in two regions with large values for θ_1. Most noticeably is that Figure 6.2c still closely resembles Figure 6.2a, while Figure 6.2b shows different performances over the search space, with good performances for small values for θ_1. This implies that we can expect that the contextual approach can yield a better performance, especially by considering different configurations for instances of version A.

6.2.2. Performance on the test set

We evaluate the configurations that yielded the largest performance on the training set for every run over the instances in the test set. The observed test scores of the local search algorithm when using the optimal hyperparameter values are presented in Table 6.1. The reported values are averages over the 5 runs (standard deviations between brackets). The best attained performances over the hyperparameter configurations obtained by the different approaches per problem version are displayed in bold.

To re-iterate, in a context-free version, we would always choose to use weights $\theta^{Combined}$ regardless of the problem version at hand. In a contextual setting, we first observe the problem version and then choose the configuration that is obtained by training on the corresponding problem version. We see in Table 6.1 that for both problem versions, this yields a better individual performance than the configuration that was obtained for the context-free approach. The poor performances of the contextual

Table 6.1: Percentage of solved instances of problem versions A and B for all different hyperparameter configurations. The reported values are averages over 5 runs (standard deviations between brackets).

	Isolated A	Isolated B
$\theta^{Combined}$	49.13% (3.62)	64.46% (4.60)
$\theta^{Isolated-A}$	**73.75%** (2.18)	11.96% (1.85)
$\theta^{Isolated-B}$	47.52% (3.83)	**66.83%** (6.98)

Table 6.2: Percentage of solved instances of problem versions A and B of different sizes for the hyperparameter configurations of the context-free and contextual approach. The reported values are averages over 5 runs (standard deviations between brackets).

size $\|I\|$	5	6	7	8	9	10	11	12	13	14	15
	Problem version A										
$\theta^{Combined}$ (avg.)				100	99.76	96.9	72.7	15.28	0.86	0.02	0
$\theta^{Combined}$ (st.dev.)					(0.5)	(5.8)	(13.7)	(6.2)	(0.6)	(0.04)	
$\theta^{Isolated-A}$ (avg.)					**100**	**99.88**	**93.14**	**76.98**	**24.74**	**2.82**	**0.28**
$\theta^{Isolated-A}$ (st.dev.)						(0.2)	(13.7)	(25.7)	(10.6)	(1.4)	(0.3)
	Problem version B										
$\theta^{Combined}$ (avg.)	100	99.76	96.46	76.08	48.82	22.88	7.74	**1.38**	0.02	0	
$\theta^{Combined}$ (st.dev.)		(0.4)	(2.4)	(8.9)	(6.1)	(5.9)	(2.8)	(0.8)	(0.04)		
$\theta^{Isolated-B}$ (avg.)	100	**99.98**	**96.98**	**81.08**	**49.76**	**24.44**	**7.76**	0.98	**0.06**	0	
$\theta^{Isolated-B}$ (st.dev.)		(0.04)	(0.9)	(3.0)	(6.0)	(3.7)	(2.6)	(0.7)	(0.05)		

approach reported in the table on the instances of the problem version it was not trained on are thus not relevant. They are merely included in the table to emphasize that the configurations obtained by the contextual approach perform well only on the instances that they were trained on. Therefore, in a setting where we can first observe the context and then choose a suitable configuration, the contextual approach outperforms the context-free setup.

For the next performance analysis, we compare the performance between the configurations obtained by the context-free approach and the best configurations obtained by the corresponding contextual approach on problem versions A and B per instance size. We consider $1,000$ instances for every combination of problem version, instance size and approach per run. The average percentages of obtained feasible solutions over the 5 runs are presented in Table 6.2. The left column shows the weight configuration that is chosen for both approaches for either problem version A or B. The weight configurations with the best performance for every problem version and size are displayed in bold. Blank values in the table denote either 0% or 100%.

We observe from Table 6.2 that the configuration from the contextual approach that was trained on problem version A, yields a better performance than the context-free configuration for every considered problem size. In addition, it finds a feasible solution for all $1,000$ runs for one input size larger than the other configurations (for $|I| = 9$ instead of 8). Also, these weights are the only ones for which the local search procedure can find any feasible solutions for instances of sizes 15. The configuration that was obtained by the contextual approach by training on instances of version B also yield a better average performance than the context-free approach for all but one instance size. This difference in performance, however, is less significant compared to the increase on problem version A. This is in line with our expectations due to the weights of the context-free being relatively closer to the weights of Isolated B, as observed in Figure 6.2.

We summarize the potential increase in performance when encountering new problem instances in Figure 6.3. This figure visualizes the expected performance for both approaches on unencountered problem instances for every instance size. Here, we assumed that unencountered problem instances are equally likely to be either of problem version A or of problem version B. The score is defined as the average percentage of instances for which a feasible solution has been found. The error bars denote the standard deviations over the 5 runs around the average performance. We see that for all sizes, the contextual approach, on average, solves a larger percentage of instances. This increase in performance is statistically significant according to the one-sided paired t-test with a p-value of 0.0001. Therefore,

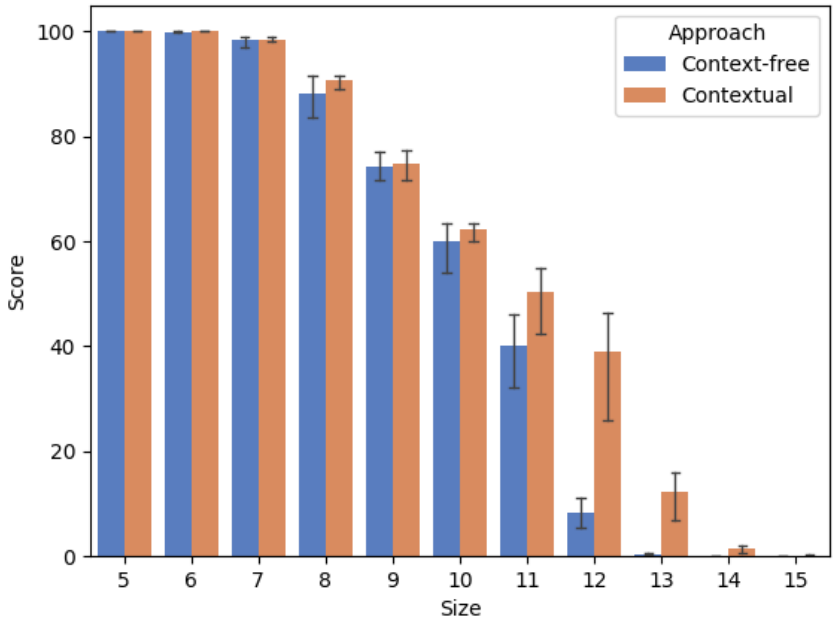


Figure 6.3: The performance for both approaches on unencountered problem instances for every problem size $|I|$. Here, newly encountered problem instances have an equal probability to belong to either problem version. The score is defined as the average percentage of instances for which a feasible solution has been found. The error bars denote the standard deviations around the average scores over 5 runs.

the null hypothesis of equal performance can be rejected with a significance level of $\alpha = 0.05$.

For the results presented in this figure, we assumed a probability of 0.5 for a new unencountered instance to belong to either problem version. Since the contextual approach outperforms the context-free approach as seen in Table 6.2, the contextual approach will, on average, yield a better performance compared to the context-free approach regardless of this probability.[1] It should be clear that a larger probability of instances to belong to problem version A makes the increase in performance more significant.

We conclude this set of experiments by stating that, when solving instances of the bin packing problem with a variable neighborhood search heuristic, a better performance is obtained if we use hyperparameter configurations that have been found when training on instances of the same problem version. New encountered problem instances of version B can, on average, be solved more frequently, while instances of problem version A will be solved statistically significantly more often. Therefore, the use of Bayesian optimization in a contextual approach finds hyperparameter configurations that obtain a better performance if we make use of the available contextual information.

[1] With the exception of instances of size $|I| = 12$ if newly encountered instances have a probability of at least 99.4% to belong to version B.

7

Experiments on the train unit shunting problem

In Chapter 6, we introduced the experiments on the bin packing problems and discussed the results of these experiments. This set of experiments focused on the first sub-question that we formulated in Chapter 1. We concluded this chapter with the observation that the use of Bayesian optimization in a contextual approach finds hyperparameter configurations that obtain a better performance if we make use of the available contextual information of the bin packing problem instances. Thereby we can positively answer the corresponding sub-question. For the remaining research questions, we are interested if this contextual approach can also improve the performance compared to a context-free approach on the more realistic train unit shunting problem.

To remain consistent with the structure from Chapter 6, we first introduce the experiments applied to the train unit shunting problem (TUSP) in Section 7.1. We study whether we can improve the performance of HIP, by finding hyperparameter configurations using Bayesian optimization, compared to the (context-free) default configuration. Then, we discuss the results of the experiments on the train unit shunting problem in Section 7.2.

7.1. Experimental setup

In this section, we introduce the set of experiments on the train unit shunting problem. Here, we are interested in how the performance of hyperparameter configurations that are obtained by the Bayesian optimization approach compares to the performance with the default configuration that is currently used by HIP. Therefore, we use the performance of HIP when using this default configuration as a benchmark for comparison against other configurations for all experiments on the train unit shunting problem.

In addition, we use this default configuration to define the domain for every hyperparameter in the search space using Equation (5.2). With the exception of the hyperparameter denoting the limit on the maximum increase in objective value per step in the perturbation phase of the VNS algorithm. This value is fixed at 50.

Recall that N denotes the number of iterations of the Bayesian optimization procedure, see line 1 of Algorithm 1, and that the total budget on the number of queried configurations to target algorithm $\mathcal{A}$ is equal to N plus the number of initial points. Similar to the experiments on the bin packing problem, the initial points to construct the prior Gaussian process are uniformly sampled from the search space. The number of initial points is equal to $2 \cdot d + 1$, with d denoting the number of hyperparameters. The additional initial point is the default configuration of HIP that we use as the first queried configuration. There are 11 weight coefficients in the objective function of the TUSP such that the definition of the search space consists of $d = 11$ dimensions. Unless mentioned otherwise, we use $N = 200 - (2 \cdot d + 1) = 177$ iterations of the Bayesian optimization procedure (Algorithm 1) to iteratively determine a configuration to query by optimizing the acquisition function. We list the names, the default values used in HIP, and the bounds on the constructed domains of the hyperparameters that we

include in the search space in Appendix C. For a detailed explanation of these hyperparameters, we refer to van den Broek (2016) and van den Broek et al. (2018)

We use the same definition for the training score for the train unit shunting problem as for the bin packing problem, see Equation (6.1). By including the default hyperparameter configuration that is available as the first of the initial queried configurations, the training score is always at least as large as the performance metric that is obtained by the default configuration. This should be clear since by definition $\mathcal{D}_1 \subseteq \mathcal{D}_N$, such that

$$\max_{i=1,\ldots,N}\{y_i \in \mathcal{D}_N\} \geq \max_{i=1}\{y_i \in \mathcal{D}_1\} = y_1, \tag{7.1}$$

where y_1 is the observed value of the performance metric of the first queried configuration, being the default configuration.

We then evaluate the performance of this optimal configuration by running the target algorithm with this configuration on all available test instances. We use the same definition for the test performance for the experiments on the train unit shunting problem as for the bin packing problem, which is the percentage of instances for which a feasible solution has been found within the stopping condition T. For all experiments that are related to the TUSP, we use a stopping condition on the maximum allowed number of steps of the local search algorithm per training instance. This is equivalent to the number of evaluated candidate solutions. The target algorithm is called with stopping condition $T = 10,000$ local search steps. We opt for such a stopping condition, instead of time in seconds, to better cope with the execution of HIP on multiple instances in parallel. We make use of parallel computing to simultaneously solve multiple instances in a batch by constructing a new local search algorithm on a thread for individual instances. We only consider parallel solving of instances in the same batch of problem instances. In addition, the execution of HIP on small instances has a relatively heavy I/O load, of which the effect on the overhead time is further enhanced when using multiple threads. The experiments on the bin packing problem are not performed in parallel. There, we prefer the use of time in seconds since it is more representative of the actual goal of the practitioner, which is to find a feasible solution for a problem instance within a controllable time frame.

7.1.1. Problem instances

In Section 4.3, we introduced the train unit shunting problem and we mentioned that instances of the TUSP consist of a problem location and a problem scenario. We now present the fixed problem location and the varying scenarios that we consider in the experiments on the train unit shunting problem. The problem instances of the TUSP that we consider in the experiments are generated instances. They are artificial instances that do not represent a real problem scenario or a real shunting yard. It is unknown whether an instance of the train unit shunting problem is satisfiable. That is, it might be the case that no feasible solution exists. All generated problem instances have a unique problem scenario, but share the same location. By only considering a single location, we reduce the number of possible features that can be considered for the contexts. We can only differentiate on contexts that relate to the different problem scenarios.

Problem location

In Figure 7.1, we visualize the topology of the shunting yard that we use for all TUSP experiments. The yard consists of four railway tracks on which trains can be parked, marked by the grey tracks in the figure. These four tracks all have a length of 334 meters and are accessible from only one side. As a result, the parking tracks follow a last in - first out scheme if multiple train units are parked on the same track. The shunting yard contains a single gateway-track (at the top left) for all trains to enter and exit the site. Finally, the shunting yard contains two different switches. These switches prevent any direct movement of train units between the gateway track and the top-right parking track; the bottom left track and any of the two bottom right tracks; and between the two bottom right parking tracks. These movements are still possible with so-called *saw-moves*: by first moving to an auxiliary track. Such a movement switches the direction of the train units in a train composition.

Problem scenarios

We consider four different types of train units in these experiments, shown in Figure 7.2. This set of different train units consists of two different train types: SLT and VIRM, with both types having train

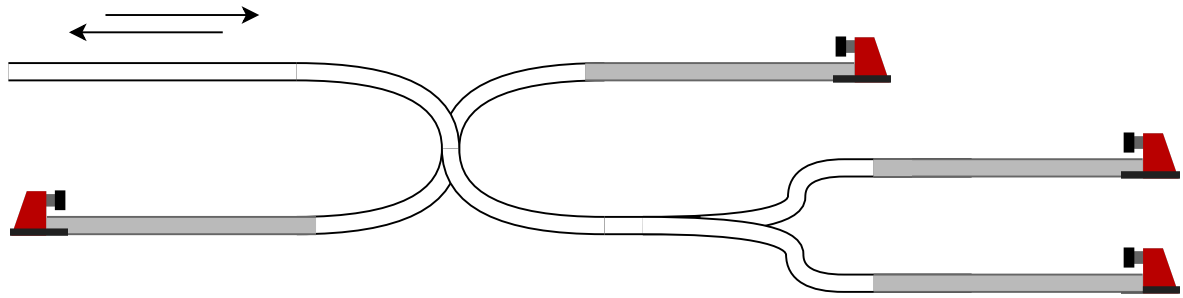


Figure 7.1: Visualization of the shunting yard used in the TUSP experiments. The yard contains a single gateway track to enter and exit the site. There are four parking tracks of equal length that are only accessible from one side.

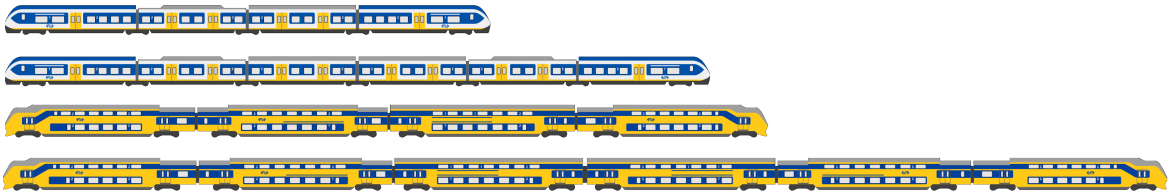


Figure 7.2: The four different train units that are considered in the TUSP experiments. Top to bottom: SLT-4, SLT-6, VIRM-4, VIRM-6. The relative lengths of the displayed train units are true to scale.

units of two different lengths: either having 4 or 6 carriages. A train is composed of one or two train units of the same train type. For example, an SLT-4 can be combined with an SLT-6 to form a train of 10 carriages. The lengths of the four different train units are 70, 101, 109 and 162 meters for the SLT-4, SLT-6, VIRM-4 and VIRM-6, respectively. Hence, the longest possible train is composed of two VIRM-6 units and has a length of 324 meters.

The composition of the arriving and departing trains is dependent on the total number of train units in a problem instance. That is, every problem with an equal total number of train units has the same set of arriving and departing train configurations in the problem scenario. The sequences of both incoming and outgoing trains are random. The arriving and departing times are also random.

Preliminary results show that this composition of the arriving and departing trains has a strong impact on the general difficulty of finding feasible solutions for a problem instance. Certain compositions of arriving and departing train units might allow for suitable reconfigurations. As a result, it is possible that instances with a smaller number of train units become more difficult to solve. However, this seems to be an exception. In general, a larger number of train units increases the number of required reconfigurations and movements, while at the same time crowding the shunting yard. There seems to be a strong positive correlation between the number of train units and the difficulty of a problem instance.

7.1.2. Context: the number of train units

In Section 5.4, we discussed the heterogeneity of the difficulty of the problem instances. This heterogeneity resulted in heavily varying performances for a fixed configuration when evaluating on different subsets of problem instances. We aim to reduce this heterogeneity by considering suitable contexts to split the instances on. We can identify many features that have been shown to have a significant impact on the difficulty of the TUSP instances in general, see for example Dai (2018). Examples of promising features that relate to the location of problem instances include: the number and length of the tracks of the shunting yard; the topology of the shunting yard; the presence or absence of bumpers that restrict the open sides of tracks. Examples of features that relate to the problem scenarios are: the total number of train units; the number of arriving and departing trains; the different types of train units; the time between arriving and departing train units; the compositions of departing and arriving trains.

From a practical point of view, it is more tractable to consider the features of the problem scenario. To isolate the effect of the problem feature under consideration as much as possible, you want the remaining problem features to undergo little variation. The topology of the shunting yard of a problem

instance puts many restrictions on the possible problem scenarios that can be considered on the location. These restrictions do not apply the other way around. Hence, it is more practical to consider a set of problem instances that include a variety of problem scenarios on a fixed problem location; than a fixed scenario on different locations. Therefore, it is easier to isolate the effect of problem features related to the scenario of a problem instance.

Within the set of features of the problem scenarios, the number of train units in a problem instance seems to be an obvious choice to consider as the first context to take into account. A larger number of train units generally yields more required reconfigurations of the train compositions; more trains to be parked on the tracks; and more movements over all tracks and switches. Therefore, a larger number of train units is likely to increase the difficulty of every subproblem of the train unit shunting problem. By dividing the problem instances based on the context of the number of train units, it seems reasonable to expect that we can reduce the heterogeneity within each context.

All problem instances that we consider in this research consist of a number of train units ranging from 5 to 11. As previously mentioned, a train can be composed of either one or two train units. Therefore, the number of arriving trains of a problem instance can be different than the number of departing trains, but the number of train units is constant between the arrivals and the departures. In the contextual approach, we construct distinct surrogate models for the different contexts and train them according to the Bayesian optimization procedure in isolation. Since the number of train units ranges from 5 to 11, we train 7 models for a single run.

We expect that the contextual approach reduces the heterogeneity of the problem instance within a certain training set by splitting the training sets based on the number of train units. As a result, we expect that the contextual approach leads to a better generalization to test instances, which in turn yields to a better performance, than the context-free approach.

7.1.3. Training and test set

For our experiments on the TUSP we have 700 problem instances with unique problem scenarios. The location of every problem instance corresponds to the shunting yard that is visualized in Figure 7.1. There are exactly 100 problem instances for every considered number of train units. We divide the available problem instances over a training and a test set for every context that we consider. First, we withhold 20 problem instances for every number of train units to initialize the test set. Then we preprocess the remaining instances by flagging the instances that we deem to be too easy and filtering them from the training set, as described in Section 5.4.1. We do so by performing Algorithm 2 with input: $C = 20$ configurations, a stopping condition of $T = 1{,}000$ (one-tenth of the default stopping condition) local search steps, HIP as the target algorithm $\mathcal{A}$, and the current training and test set per context. This implies that we flag every remaining training instance i for which:

$$\max\{t_i(\theta_1), \ldots, t_i(\theta_{20})\} < 1{,}000, \tag{7.2}$$

where $t_i(\theta)$ denotes the number of local search steps before the target algorithm $\mathcal{A}$ terminates on instance i when using configuration θ. The local search target algorithm only terminates within the stopping condition if a feasible solution has been found within this number of local search steps. Hence, we can safely stop the execution of the target algorithm after $T = 1{,}000$ local search steps for each configuration.

Figure 7.3 shows the maximum number of local search steps to find a feasible solution by considering all 20 configurations for all problem instances. This value is the left-hand side of Equation (7.2). The problem instances are separated by the number of train units. The rightmost bin contains all instances for which the target algorithm did not find a feasible solution for at least one configuration within the small stopping condition. The leftmost bin, however, shows that there are a relatively large number of problem instances for which we find feasible solutions within only a small number of local search steps for every configuration $\theta_1, \ldots, \theta_{20}$. This bin represents the portion of the available training instances that are not useful to be part of a batch that is used during the evaluation of the performance of a hyperparameter configuration. We can also see that there is only a small number of instances for which a feasible solution has been found with every considered configuration, where at least one configuration resulted in more than 50, but less than $1{,}000$ local search steps. This small number of problem instances is also flagged. Since this is such a small percentage of instances, we conclude that we can generally claim that a problem instance is either easy to solve or not, and we can safely filter the easy instances.

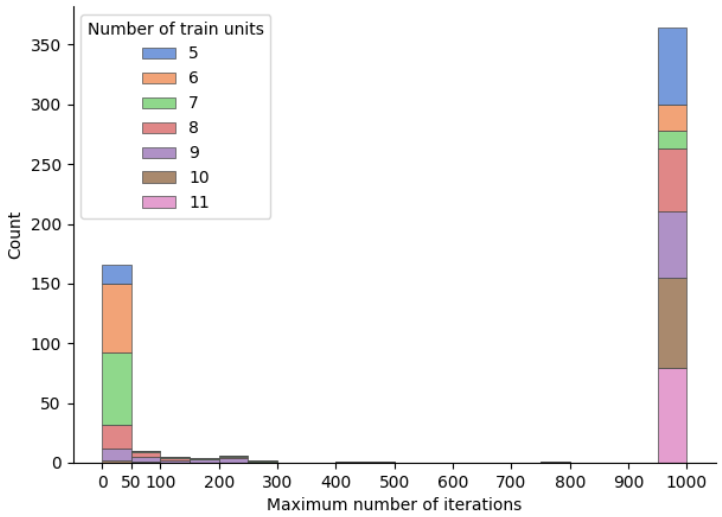


Figure 7.3: Stacked histogram of the maximum number of local search steps to find a feasible solution for every problem instance, over 20 hyperparameter configurations. The histogram is stacked over the context of the number of train units per instance. The stopping condition equals 1,000 steps of the local search algorithm.

From the remaining set of unflagged instances, we sample a random subset of 24 instances to use as the training set.[1] All problem instances that do not become a part of the training set are moved to the test set such that the test set contains 76 problem instances for every context. Finally, the batch of problem instances that is used to evaluate the performance of a configuration is a randomly drawn subset of this obtained training set. Unless mentioned otherwise, we use $K = 12$ as the default value for the batch size for all TUSP experiments. By using a batch size that is smaller than the number of instances in the training set, we can perform multiple runs per context where we only vary the batches of problem instances. The test set between these runs then remains unchanged, which allows for a fair comparison of the performance on the test set between different batches. To evaluate the performance of the obtained hyperparameter configurations on the test set, we run every test instance of the constructed set of 76 problem instances per context three times. We vary over these three times by using different seeds for the target algorithm. Similar to during the training, we use a stopping condition $T = 10,000$ steps of the target algorithm per test instance. For every experiment, we indicate the performance of the hyperparameter configuration over all tested instances that correspond to the same context by reporting the test score. This test score is defined as the percentage of these 228 instances for which a feasible solution has been found.

To capture the statistical variability of this evaluation protocol, we repeat this procedure 8 times for the experiments that consider a contextual approach. Every distinct run considers a different subset of the training set as the batch of instances to evaluate the performance of a configuration during training. We report the average test scores over these 8 runs along with the standard deviations. We compare the performance obtained by the Bayesian optimization procedure with the performance of the default configuration. Since the default configuration does not train on the different batches per run, the test score does not change when using different batches per run. We do however calculate and report the training score of the default configuration for the different runs. We report the p-values of the one-sided paired t-test ($\alpha = 0.05$) to test the statistical significance of the improvement on the performance of the configurations obtained by Bayesian optimization compared to the default configurations.

For the experiments that consider a context-free approach, we define a single training set that is the union of the training sets per context after filtering. We also evaluate the performance on the set that contains all test instances. The performance on the test set in the context-free approach is also

[1] For the contexts with 6 or 7 train units, we sample only 12 unflagged instances and add 12 flagged instances. Instances with these numbers of train units are easier to solve, which results in a limited number of non-easy instances. This is also visible in Figure 7.3. We also need a sufficient number of non-easy instances in the test set to fairly evaluate the performance of some configuration.

measured by running all test instances with the optimal configuration three times with different seeds. Since we only need to train a single surrogate model per run, we increase the number of runs of the Bayesian optimization procedure from 8 to 20 runs.

7.1.4. Parallel portfolio approach for the TUSP

For the last set of experiments on the train unit shunting problem, we consider the hyperparameter optimization problem for the target algorithm that solves instances of the TUSP from a more practical point of view. We are interested in whether we can use the knowledge that we acquire from this research in practice to improve the performance of the considered local search algorithm in its current deployment state.

In the current deployment of HIP, it is part of the default procedure to run multiple processes on a single problem instance in parallel. Every process instantiates a new version of the local search algorithm to solve the same instance. By running these processes simultaneously, all processes can be stopped once any of the processes find a feasible solution to the problem instance. The different processes do not communicate with each other while solving a problem instance and do not have any form of shared memory. This makes the setup different from parallel computation. In the current procedure, the only variation between the processes is a different seed that is passed to the random number generator for the stochasticity of the local search. This variation over the seeds makes it possible for different executions of the local search target algorithm to consider different sequences of candidate solutions. As a result, different runs over the search space are considered by the local search algorithm. This can reduce the computation time to find a feasible solution, if one exists, compared to a single execution of the algorithm.

Besides these different seeds, it is possible to provide more variation to the different executions of the algorithm. One of the possibilities is to consider a set of different hyperparameter configurations, where every process is executed with a different configuration from this set. We refer to this set of different configurations as a *portfolio* of hyperparameter configurations. We expect that the use of such a portfolio over the different processes makes it likely to create even more variation between different executions of the local search algorithm than by only using different seeds. Therefore, we expect that the use of a portfolio reduces the computation time even further compared to only considering the use of different seeds. We also expect that larger degrees of variation within these portfolio result in a larger variation of the computation times of the different local search executions. This variation can be unwelcome if we only have a limited number of available processes by introducing risk. Since we are only interested in the process that finds the feasible solution the fastest, we expect that the level of improvement in performance increases when a larger number of processes is available.

We compare the performance of multiple strategies to construct the portfolios of hyperparameter configurations. Here, we differentiate between the default portfolio, random portfolios and trained portfolios. Within these different strategies, we consider different methods to construct portfolios.

Default portfolio

First, we consider a *default portfolio* that contains only the default hyperparameter configuration. Hence, every process uses the default configuration and the processes only differ from each other by considering different seeds. To remain consistent and allow for a fair comparison, every process is issued its own seed for all alternative strategies. The use of this default portfolio describes the current strategy that is used in HIP. Therefore, the performance of the default portfolio is equal to the performance that is obtained by running HIP with the default settings. This provides a suitable benchmark for comparison against alternative portfolios.

Random portfolios

The second strategy is to construct a *random portfolio* by drawing random configurations from the configuration space. To sample these random configurations, we consider the following methods. We uniformly sample a configuration from the search space: $\theta_p \overset{i.i.d.}{\sim} U(\Theta)$ for processes $p = 2,\ldots,P$. Here, P denotes the number of available processes. Process $p = 1$ receives the default configuration of hyperparameters.

Or we sample a configuration θ_p according to a normal distribution around the default configuration: $\theta_{pk} \overset{i.i.d.}{\sim} \mathcal{N}(d_k, \sigma_k^2)$ for every hyperparameter k and process $p = 2,\ldots,P$. For this method, process $p = 1$ receives the default configuration as well.

A statistical property of the normal distribution is that the probability to sample a value according to this distribution that is in the confidence interval defined as three standard deviations around the mean is 0.997. We define the value for the standard deviation $\sigma_k = 1.1073$ for every hyperparameter k, which is equal to 1/6th of the domain length of every hyperparameter after the logarithmic transformation as defined in Equation (5.2). As a result, the probability of drawing a value θ_k from this particular normal distribution has a probability of 0.997 to be located in this defined domain. Since the configuration space is spanned by the d dimensions of all hyperparameters, the probability to sample a configuration that is located inside the search space Θ is roughly equal to $1 - 0.997^d$. We defined the search space as a subspace of the complete configuration space. Therefore, it is not an issue if the sampled configuration θ is located outside of this space. To research the effect of the standard deviation, we also consider a method where we sample the configurations according to a normal distribution where the value of the standard deviation is twice as small: $\sigma_k = 0.55365$.

Trained portfolios

Next, we consider strategies that construct a portfolio of configurations by training over a set of problem instances. We first consider a portfolio that is constructed by random search. Here, we define a simple training procedure where we randomly sample a single configuration from the search space for every iteration. After the training procedure, the portfolio is defined as the set of P configurations that received the largest observations y of the performance metric. We consider the same distributions for the sampling strategies as in the random portfolios: a uniform distribution and a normal distribution around the default configurations with standard deviations $\sigma = 1.1073$ and $\sigma = 0.55365$.

Finally, we consider a portfolio that is constructed by Bayesian optimization. For this strategy, we use the same training setup as the portfolio constructed by random search, but the configurations are queried by the Bayesian optimization procedure. The training procedure is set up the same as the previously discussed experiments that use Algorithm 1. The only difference occurs by replacing the single optimal configuration θ^* as defined in Equation (6.1) by the set of P configurations that yielded the best observed performance metric. The random search portfolio can be considered as a specification of this training procedure where the acquisition function is replaced by a random sample from the uniform distribution $\theta_n \overset{i.i.d.}{\sim} U(\Theta)$ or from the normal distribution $\theta_n \overset{i.i.d.}{\sim} \mathcal{N}(d, \sigma^2)$ in iterations $n = 1, \ldots, N$ and where the surrogate model S is completely ignored.

We want to emphasize to the reader that the Bayesian optimization procedure that we use in this research has been constructed with the specific aim to find the single most optimal hyperparameter configuration. Not a portfolio of the best P configurations. Due to the exploitation aspect of the acquisition function, it is very likely that most of the best P configurations are located very close to each other in the configuration space. It is more promising to include well-performing configurations that are spread over the configuration space in the portfolio. This should be taken into account. We do so by defining a greedy rule that iteratively adds the configuration with the best observed performance from the observation history to the portfolio. In this iterative process, we ignore configurations that are within a certain distance to any configuration that is already included in the portfolio. We use the Manhattan distance over the search space to define this minimum distance. The pseudocode of this greedy configuration selection rule is presented in Algorithm 3. Here, we assume that the number of configurations N in the observation history is larger than the number of processes P. The Manhattan distance in line 4 is calculated between two configurations that have been logarithmically transformed. Considered configurations are not included if the Manhattan distance to any already included configuration is smaller than 1. The exception is that we add a subset of the configurations with the largest values for y if not a sufficient number of configurations exist that meet this distance-based criterion (see lines 11-17). This ensures that the number of configurations in the portfolio that is returned, is equal to the number of available processes P.

Comparing the performance of portfolios

We measure the performance of the different strategies using two metrics. First, we define the test score as the percentage of instances that is solved within the stopping conditions. However, we now consider a feasible solution to be solved if a feasible solution is found by *at least one* of the processes. Secondly, we measure the number of local search steps of the process that finds a feasible solution within the smallest number of steps. This is defined as:

Algorithm 3 Greedy configuration selection

Input: observation history $\mathcal{D}_N = \{(\theta_1, \theta_N), \dots (\theta_N, y_N)\}$, number of processes P.
Output: portfolio Π.

```
Sort D_N based on descending y
Initialize portfolio Π = {θ_1}
for i = 2,...,N: do
    calculate Manhattan distance between θ_i and all θ ∈ Π
    if all distances > 1 then
        add θ_i to Π
    end if
    if |Π| = P then return Π
    end if
end for
for i = 2,...,N: do
    if θ_i ∉ Π then
        add θ_i to Π
    end if
    if |Π| = P then return Π
    end if
end for
```

$$\min\{t_i(\theta_p), p = 1, \dots, P\}, \tag{7.3}$$

for instance $i \in \mathcal{I}$. If this value is equal to the stopping condition T on the number of local search steps, none of the processes have found a feasible solution.

During the performance evaluation, we run every instance i from the test set in isolation. We increase the stopping condition when solving the test instances to $T = 100,000$ local search steps. This larger stopping condition allows for more disparity in both performance metrics between the different portfolios such that we can better compare the relative differences in performance. The stopping condition when evaluating configurations during the training of the trained portfolios remains at $T = 10,000$.

The trained portfolios are the only strategies that can potentially benefit from a contextual approach. However, due to time restrictions, we only consider the context-free approach for all portfolios. We also consider a limited number of 4 runs for the trained portfolios. In each run, we train the portfolios on different sampled batches of training instances and evaluate the portfolio once on every test instance. For the random portfolios, we generate a new portfolio of configurations for every single test instance. This way, we can obtain a reliable estimate of the performance of these strategies without having to resort to multiple runs.

Every process that instantiates a new version of HIP is executed on a single thread. For this set of experiments we consider two different numbers of threads: $P = 6$ and $P = 24$. However, since we do not measure the performance of the portfolio approach based on time in seconds, the availability of this number of threads is not actually required. The experiments can be reproduced on a single thread by sequentially running P executions of the target algorithm with different configurations from the portfolio. After all the runs are completed, the smallest number of local search steps over the runs can be computed using Equation (7.3). This method is not only more time-consuming since you use a single thread, but also since you cannot terminate all open threads once of the processes finds a feasible solution.

We already formulated the expectation that the use of portfolios of different hyperparameter configurations improves the performance compared to the default portfolio. We expect that the trained portfolios yield the best performance since they are already evaluated during a training procedure. At the very least, the randomly sampled configurations that perform the worst on the training instances, will be disregarded.

7.1.5. Test environment

The Bayesian optimization framework and corresponding Gaussian process implementation that we use for the TUSP is the same as for the bin packing experiments as introduced in Section 6.1.2. The settings of the variable neighborhood search target algorithm that is used to solve any problem instance are also the same, with the exception of the maximum increase in objective value per step. This value is set to 50 for the TUSP experiments, whereas it was 10 for the bin packing problem. This adjustment is made to account for the default configuration values that are used in this set of experiments.

The target algorithm that is used to solve instances of the train unit shunting problem is HIP, see Section 4.3.1. The functionality of the version of HIP that we use can be retrieved.[1] The setup of the Bayesian optimization procedure for all experiments can also be found.[2] This repository includes directories containing all generated TUSP instances divided over the (pre-processed) training and test sets per context. The values of the default configuration that are used are reported in Appendix C. All TUSP experiments were performed on a Microsoft corporation virtual machine with Intel Xeon E5-2673 CPU and 16GB of RAM. The TUSP experiments are solved in parallel on a maximum of 6 threads. Since we only use parallel solving over instances in a batch of problem instances, the actual number of threads that can be used is bounded by the batch size K.

7.2. Results

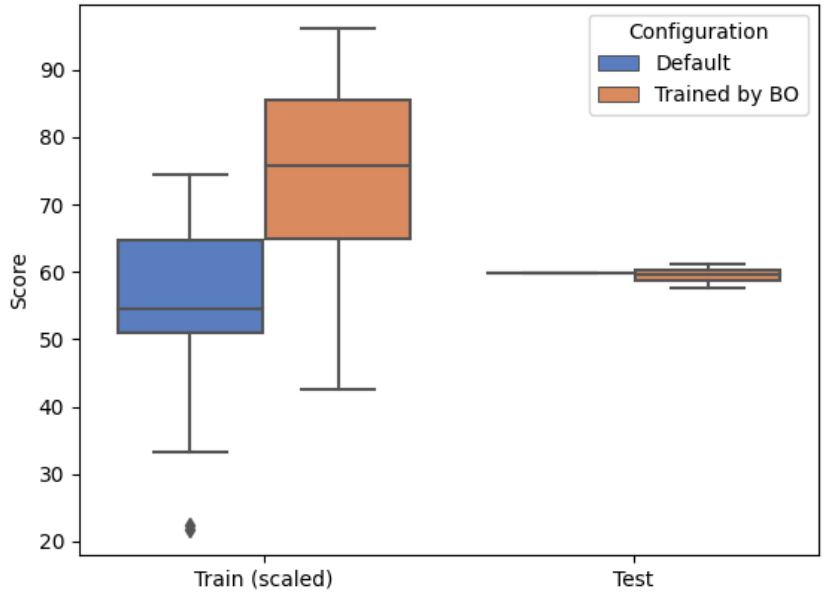
In this section, we present the results of the experiments on the train unit shunting problem. We first compare the performance of HIP with hyperparameter configurations that are obtained by a context-free Bayesian optimization approach to the performance with the default configuration of HIP. Then, we compare the performance of the default configuration against the performance of configurations obtained by Bayesian optimization in a contextual approach. Finally, we present the results on the use of different portfolios of hyperparameter configurations in the parallel portfolio approach.

7.2.1. Context-free approach

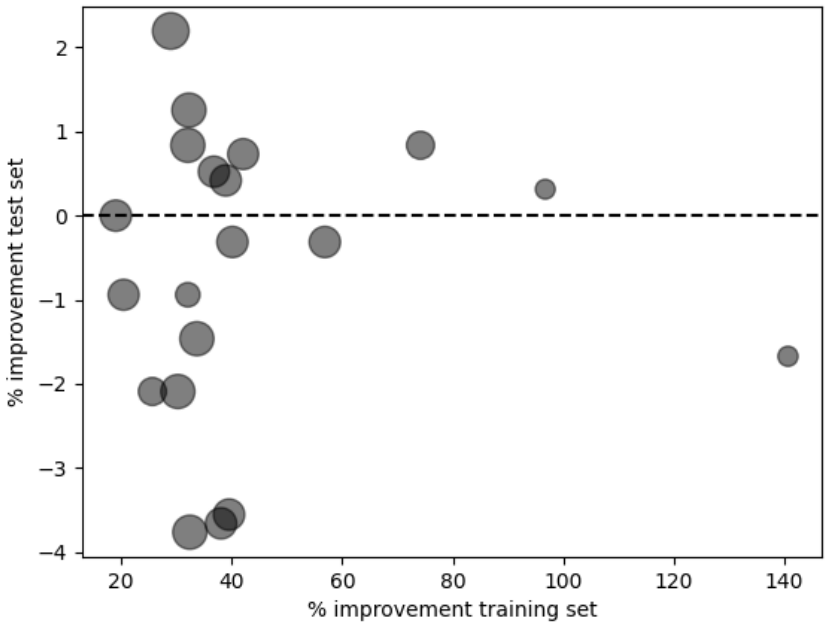
In the context-free approach, we ignore any contextual information. We compare the performance of the configuration obtained by Bayesian optimization against the default configuration. The default configuration is an example of a context-free approach since it is used for every considered problem instance. In Figure 7.4, we visualize the performance of the configurations obtained by Bayesian optimization versus the performance of the default configuration on the training batches and on the test set. Figure 7.4a contains boxplots that denote the different median scores. These boxplots also show the first and third quartiles and the minimum and maximum values, excluding outliers based on the interquartile range (IQR), over the 20 runs. These statistical measures represent the deviation over the average scores. The default configuration is identical over the 20 runs, but since the subset of the training set varies, the corresponding training score varies over the different runs. We scaled the training scores such that all the boxplots for the training and the test scores fit in the same figure. Since the instances in the test set and the three different seeds do not change between the different runs, the test score of the default configuration does not change either. As a result, the corresponding boxplot is a horizontal line. Figure 7.4b shows the improvement in performance of the 20 different configurations obtained by the context-free Bayesian optimization approaches, compared to the performance of the default configuration. The horizontal axis denotes the percentage improvement of the configuration obtained by BO against the default configuration on the training set. The vertical axis denotes the percentage improvement on the test set.

We observe from Figure 7.4a that, while the configurations found by Bayesian optimization yield larger training scores, the median test score is slightly smaller than that of the default configuration. Equation (7.1) shows that the training score of Bayesian optimization is at least as large as the training score obtained by the default configuration, since we include the default configuration as one of the initial points in the Bayesian optimization procedure. The large deviations in the training scores between the different runs are a result of the different sampled batches of training instances per run. The IQR of the test scores obtained by Bayesian optimization, on the other hand, is relatively small. More importantly, the constant test score of the default configuration is almost as large as the first quartile of the Bayesian optimization scores. This means that the Bayesian optimization procedure yields a test

[1] Hash ID of the git commit: c38e2f13af22eebbce6c5198852d5be9be01d81a.
[2] Hash ID of the git commit: 4aeaad88990818a36d799354a7588f5bea94fd3a.



(a) Boxplots of the median training and test scores over 20 runs for the different configurations. The top and the bottom of the box denote the first and third quartile. The whiskers denote the minimum and maximum scores, excluding outliers based on the IQR.



(b) Scatter plot of the relative differences in performance between the different configurations. The horizontal/vertical axis denotes the percentage improvement of the configuration obtained by BO against the default configuration on the training/test set. The marker sizes represent the training score obtained by the default configuration.

Figure 7.4: The training scores and test scores of the configurations obtained by Bayesian optimization in a context-free approach and the default configuration over 20 runs. Each run considers a different batch of training instances.

score that is smaller than the default configuration for more than half of the 20 runs.

The differences in the scores of the individual runs are emphasized in Figure 7.4b. Here, we see a confirmation that every run yields an improvement on the performance of the configurations obtained by Bayesian optimization on the training set compared to the default configuration. The figure shows that the configurations obtained by Bayesian optimization yield a better performance on the test set compared to the default configuration for only 8 out of 20 runs. Since the average test score is larger for the default configuration, the performance obtained by Bayesian optimization is clearly not a statistically significant improvement.[1] Also, there does not seem to be a strong correlation between a large improvement on the training set and a positive increase in performance on the test set, with a Pearson's correlation coefficient of only +0.017.[2]

The smallest markers denote runs with batches on which the default configuration yields relatively small training scores. We observe from the figure that the Bayesian optimization obtains the largest percentage in improvement on these batches. However, this improvement in performance does not generalize to the test set. This implies that the performance that is attained after training on the training instances, can not be reproduced on the new, unencountered test instances. The level of improvement of the performance on the training set heavily varies based on different batches of training instances, with increases in performance of up to 150%. This large variation can be caused by the context-free approach. This approach has a maximum level of heterogeneity over the full training set, such that there can be large variations in difficulty and usefulness between different sampled batches of problem instances. In a contextual approach, we expect the heterogeneity within training sets to be smaller, such that the training scores of the default configuration and the attained improvement by BO become more balanced. For the following experiments, we consider the contextual approach to see whether this is true and whether this leads to a better generalization to the test set.

7.2.2. Contextual approach

In the contextual approach, we differentiate on the context regarding the number of train units of every problem instance. We compare the performance of the configuration obtained by Bayesian optimization against the default configuration for 8 runs per number of train units. Similar to the previous set of experiments, the runs differ by considering a different batch as the subset of the training sets.

In Figure 7.5, we present bar plots that show the performance of the default configuration and the best configurations found by Bayesian optimization, per number of train units. The error bars denote the standard deviations over the 8 runs around the average performance. Figure 7.5a shows the performance on the training set for the two approaches; Figure 7.5b displays the performance on the test set, for every number of train units. To re-iterate: the default configuration is fixed for every run. The training score of the default configuration still varies based on the different batches of training instances between the different runs. However, since the test set is fixed, the performance of the default configuration on the test set is constant. As a result, Figure 7.5b shows no error bars for the default configuration.

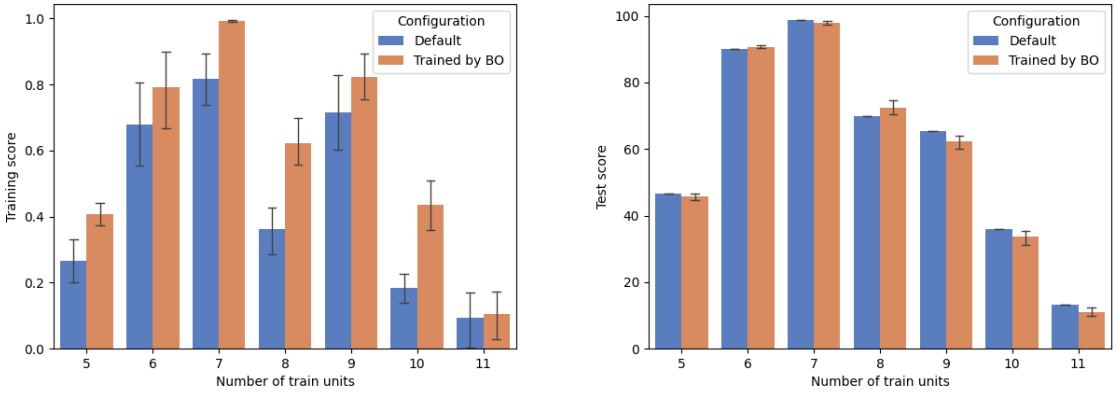
First, we again observe from Figure 7.5a that the configurations obtained by Bayesian optimization always yield a better training score than the default configuration, see Equation (7.1). However, these trained configurations do not yield a better improvement on the test set, as shown in Figure 7.5b. Regarding this second figure, we first note that smaller test scores are achieved for the contexts of 5 and 6 train units than for 7 train units. We mentioned in Section 7.1.1 that this is possible due to fixed arriving and departing train compositions per context; and that certain compositions of arriving and departing train units might allow for more suitable reconfigurations.[3] The test scores become smaller when considering more than 7 train units. We can expect that trend to continue if we would consider even larger number of train units.

Next, we observe that it differs per run whether the configuration trained by BO yields a slightly better or slightly worse performance on the test set compared to the default configuration. The ab-

[1] The obtained p-value from the one-sided paired t-test is $0.9414 > 0.05$. However, the performance of the BO configurations is also not statistically significantly worse. The obtained p-value from the one-sided paired t-test over the training scores is 0.0079. Therefore, the BO does yield a significant improvement for the training scores at $\alpha = 0.05$.

[2] This yields a p-value of 0.941957 which is not significant with a sample size $n = 20$ at $\alpha = 0.05$.

[3] An example: the composition of the arriving trains for 7 train units are: SLT-6, SLT-4, VIRM-4, SLT-4+SLT-6, VIRM-6+VIRM-6. This is equal to the compositions of departing trains and thus, reconfigurations are never required. However, they can be helpful to reduce the number of movements. e.g. 3 SLT units can be parked on the same track and then re-configured to switch the direction of the train that consists of 2 units.



(a) Score on the training set, per number of train units.

(b) Score on the test set, per number of train units.

Figure 7.5: Performances of the default configuration and the best configuration found by Bayesian optimization per number of train units. The error bars denote the standard deviations around the average scores over 8 runs.

solute difference in performance between the configurations of the two approaches on the test set is quite difficult to read in Figure 7.5b. We emphasize this difference in performance in Figure 7.6. Here, the difference is calculated as the performance of the configuration that is obtained by BO minus the performance of the default configuration, for every number of train units. The boxplot at the far right denotes the difference in performance of the context-free approach from Figure 7.4a. The default performance is represented by a horizontal dashed line at $y = 0$.

From this figure, we observe that the horizontal line that represents the default performance cuts through the box including its whiskers for every number of train units. This implies that the default performance would not be considered as an outlier in the performances for any number of train units. Thus, there never is a significant improvement in the performance of the configurations obtained by Bayesian optimization for any number of train units. The median performance is smaller than the default performance for 5 of the 7 different number of train units. The performance obtained by Bayesian optimization is clearly not a statistically significant improvement over the performance of the default configuration.[1]

We mentioned that we expected the performance of the contextual approach to be at least better than that of the context-free approach, since the contextual approach should reduce the amount of heterogeneity between different batches of a fixed context. This is not immediately clear from Figure 7.6. The observations between the context-free and the contextual performance are no longer pair-wise comparable since they are trained and evaluated on different test sets. However, the test set for the context-free approach is the union of all test sets with a fixed number of train units. This allows us to compare the overall percentage of solved test instances between the context-free and the contextual approaches. The context-free approach solves 59.335% of the encountered test instances, whereas the contextual approach finds a feasible solution for merely 59.179% of the test instances. Even though this comparison favors the context-free approach, this comparison is biased in favor of the contextual approach since we did not split the budget on the number of function evaluations over the Bayesian optimization procedure for every context. Therefore, the contextual approach used 7 times the number of function evaluations that the context-free approach used.

This counter-intuitive observation makes it important to research whether the budget on the number of function evaluations can be better distributed; and whether additional function evaluations even yield better results.

7.2.3. Alternative distributions of the function evaluations

We consider an optimization problem where the bottleneck is the tight budget on the number of function evaluations. In this research, we consider the expensive resource to be time due to the long com-

[1]The obtained p-value from the one-sided paired t-test is $0.9815 > 0.05$. In fact, a two-sided paired t-test with $\alpha = 0.05$ would reject the null hypothesis that the performance is equal, with a p-value of 0.0371, hence favoring the default configuration.

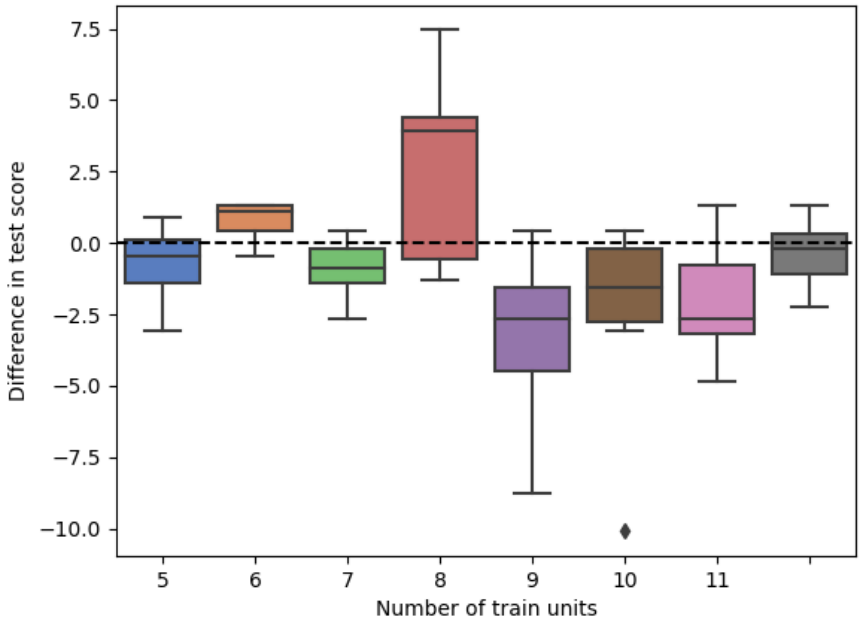
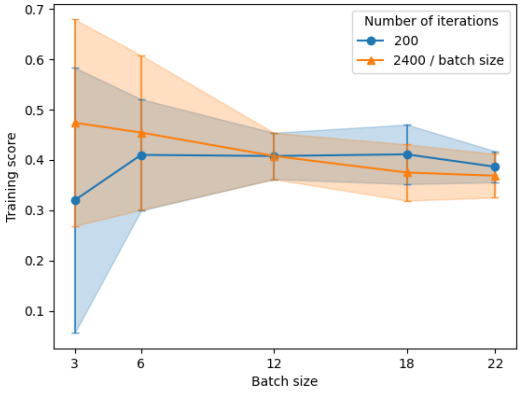


Figure 7.6: Boxplots of the median difference between the percentage of solved problem instances over the test set for the best configuration found by BO and the default configuration per number of train units. The top and the bottom of the box denote the first and third quartile. The whiskers denote the minimum and maximum scores, excluding outliers based on the IQR. The boxplot at the far right denotes the difference in performance of the context-free approach. The default performance on the test set is represented by a horizontal dashed line.

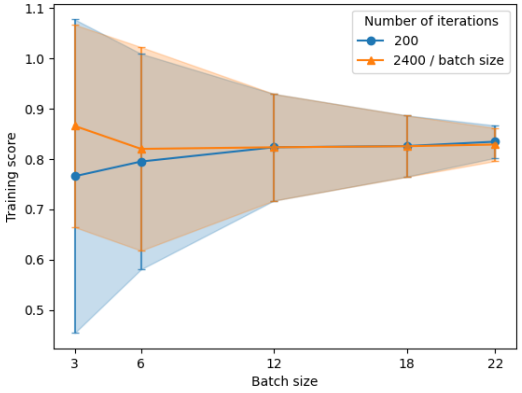
putation times of the target algorithm. This target algorithm is queried for every problem instance in the batch of training instances for every configuration that is considered during an iteration of the Bayesian optimization procedure. Therefore, the total computation time heavily depends on the stopping condition T; the number of iterations of the Bayesian optimization procedure N; and the batch size of considered problem instances K. For example, if you would multiply one of these values by two, it would result in double the total computation time. We also remark that it is possible to run every configuration multiple times on a single instance to account for the stochasticity of the target algorithm, which would also contribute to the total computation time. We choose not to do this. In the experiments that we discussed so far, we considered the default values of a stopping condition of $T = 10,000$ steps of the target algorithm, $N = 200$ iterations of the Bayesian optimization procedure and a batch of size $K = 12$ training instances for every number of train units.

For the next set of experiments, we test the hypotheses that the use of a small training batch results in inconsistent training; and that the use of a small training batch leads to a poor generalization to test instances. It is important to test these hypotheses: if we find that they can be rejected, then it is possible to train the Bayesian optimization procedure with smaller batch sizes. This allows for an increase in either the stopping condition T or the number of training iterations N, without affecting the total number of function evaluations. To test these hypotheses, we consider batch sizes K of varying sizes during the training procedure. We expect that the configurations that are trained on small batches show a poor generalization to the test set since the configurations might be over-tuned on this small set of training instances. In addition, we expect that small batches show a large variation in the yielded performance since there is a large variation in the usefulness and heterogeneity of the sampled batch of training instances. On the other hand, we expect these effects to diminish when considering larger batches of training instances. Therefore, the batch size K should definitely not be too small, but it is also not desired to take a very large batch size due to its impact on the required number of function evaluations.

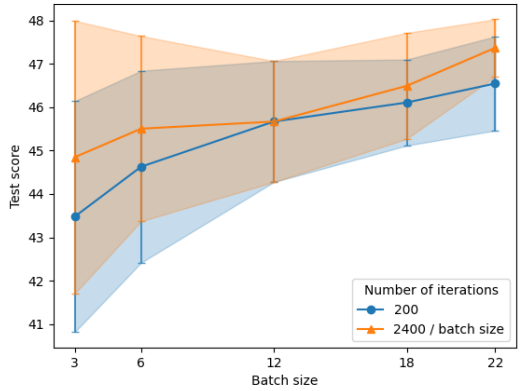
For this experiment, we consider batches of sizes $K \in \{3, 6, 12, 18, 22\}$. The maximum batch size that we consider is 22 since the training set per context consists of 24 instances and we still want to be able



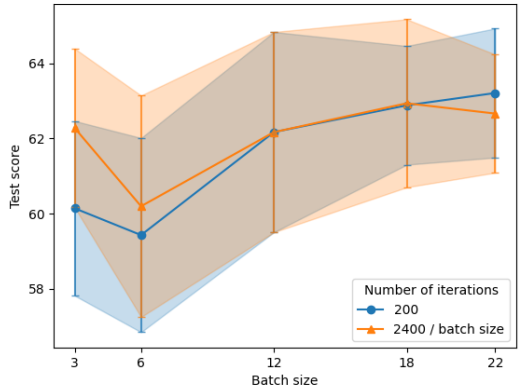
(a) The training score for the context of 5 train units.



(b) The training score for the context of 9 train units.



(c) The test score for the context of 5 train units.



(d) The test score for the context of 9 train units.

Figure 7.7: Performance of the configurations obtained by BO on the training and the test set for the contexts of 5 and 9 train units with different distributions of the function evaluations. The blue line isolates the effect of different batch sizes, while the orange line represents a fixed budget on the number of function evaluations. The error bars denote the standard deviations around the reported average scores over 8 runs.

to sample different batches of training instances to measure the variability over different runs. We perform the Bayesian optimization procedure for the different batch sizes with both: $N = 200$ iterations; and $N = 2{,}400/K$ iterations. The first approach isolates the effect of using different batch sizes. The second approach considers a fixed budget on the number of function evaluations by dividing the budget on the number of iterations by the considered batch size. The budget on the number of iterations of the two approaches coincides with the default $K = 12$ at $N = 200$. We perform the experiments in the contextual approach on the instances of 5 and 9 train units. We choose these particular contexts since the training sets corresponding to the contexts of 6 and 7 train units consist of only 12 unflagged training instances. Furthermore, the instances with 10 or 11 train units are relatively more difficult to solve, which results in more possible subsets containing only instances that cannot be solved. These subsets do not allow for proper training. We run every combination of batch size, number of train units and number of iterations 8 times. In Figure 7.7, we show the average training and test scores for the different batch sizes and number of iterations. The error bars denote the standard deviations around the reported average scores over the 8 runs.

We have previously shown in Equation (7.1), and confirmed in the results of previous experiments, that the training score obtained by Bayesian optimization is at least as large as the training score of the default configuration. The reason for this was that the default configuration was the first queried configuration in the BO procedure and that we define the training score to be the configuration that yields the largest performance metric. We can generalize this by saying that we obtain an equal or larger training score when using a larger number of iterations in the Bayesian optimization procedure.

This can be observed in Figures 7.7a and 7.7b as follows: the orange markers denote the performance after 800 and 400 iterations for batch sizes 3 and 6, respectively. For these batch sizes, the obtained training scores are larger than those of the blue markers with 200 iterations. At batch sizes 18 and 22, the blue line denotes a larger number of iterations and thus a larger training score. More importantly, we observe more variation in the training score over the different runs for small batch sizes. This is in line with our expectation that the large variation between the usefulness and heterogeneity of the instances in combination with a small sampled batch of training instances yields a large variation in the performance on the training set.

Looking at the performance on the test set in Figures 7.7c and 7.7d, we observe an increase in average performance for larger batch sizes. This implies that the increase in performance from the training set does not generalize to the test instances, which suggests that the trained configurations over-tune on the small number of training instances when training on a small batch. These observations imply that we cannot reject the hypotheses stated above. Therefore, we should not use too small batch sizes.

Another important observation from this figure is that a larger number of iterations not necessarily seems to yield a better performance on the test set. The average performance on the test set for $K = 18$ in Figure 7.7d and for both $K = 18$, $K = 22$ in Figure 7.7c is smaller when performing 200 iterations of the Bayesian optimization procedure. This means that for these experiments, the performance decreases when we allow for a larger number of iterations. If we pair-wise compare every run with a larger number of iterations against the corresponding run with a smaller number of iterations, the obtained improvement of the test score for a larger number of iterations is not statistically significant.[1] If we only include the performance with $K > 12$, this difference is negative, implying that a larger number of iterations results in a worse performance.

7.2.4. Performance of the incumbent configurations over iterations

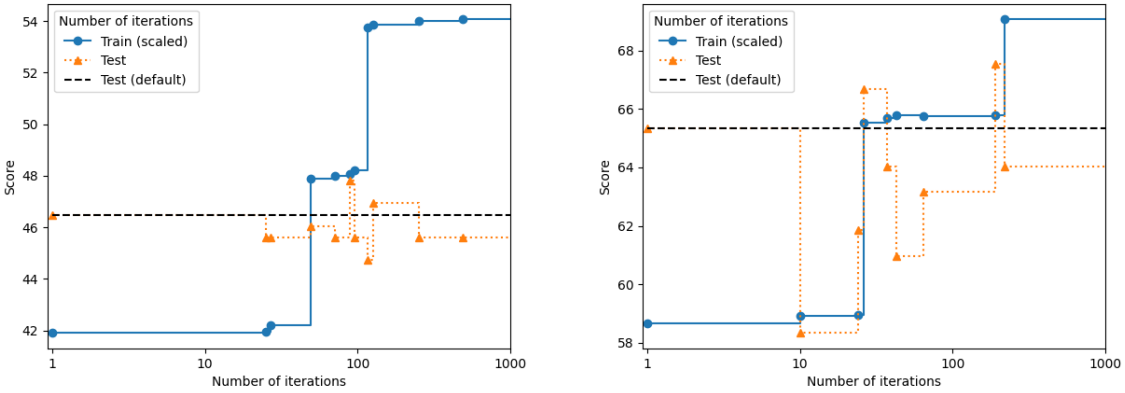
We concluded the previous experiment with the observation that a larger number of iterations of the training procedure does not always result in a better performance on the test set. This begs the question of whether the performance generally improves a larger number of function evaluations is available. In this set of experiments, we want to investigate the performance of the configurations obtained by Bayesian optimization on the test set when allowing for increasing numbers of iterations of the training procedure. We do so by measuring the performance of the best incumbent hyperparameter configuration over a large number of training iterations.

The best incumbent hyperparameter configuration during the Bayesian optimization procedure is defined as the queried configuration that so far yielded the best observation of the performance metric, as defined in Equation (6.1). The best incumbent configuration is saved and can be extracted at any iteration n. To show the performance of the best configuration at every iteration of the training procedure, we only need to measure the performance on the test set for every queried configuration during the BO procedure that becomes the incumbent configuration.

We measure this performance of the best configuration over the iterations for the contexts of 5 and 9 train units. We consider a single run such that we can consider the maximum batch size $K = 24$. We show the performance on the training and test set for $N = 1,000$ iterations in Figure 7.8. The markers denote every iteration in which a new best configuration is found. Only at these iterations, the training score of this configuration and its corresponding test score are updated. We scaled the training scores such that the scores on the training set and the test set fit on the same axis. The figure also contains the performance of the default configuration on the test set as a baseline for the performance. The number of iterations on the horizontal axis is plotted on a logarithmic scale. The number of iterations until a new incumbent configuration is found quickly increases.

We observe that the training score is monotonically increasing over the number of iterations. The corresponding test score, however, seems to fluctuate without an apparent upwards trend. It seems to be random whether a newly incumbent configuration will yield a better performance than the default configuration and than the previous incumbent configuration. Based on these results, we can conclude that the performance on the test set does not necessarily improve when we allow for a larger number of iterations. This seems to be a very clear indication that the performance of trained configurations on the training set does not generalize to newly encountered problem instances in the test set.

[1] The obtained p-value from the one-sided paired t-test is $0.2657 > 0.05 = \alpha$.



(a) The context of 5 train units.

(b) The context of 9 train units.

Figure 7.8: The performance on the training and the test set for every newly incumbent hyperparameter configuration over a large number of iterations of the Bayesian optimization procedure.

7.2.5. Results on the parallel portfolio approach

The last set of results that we discuss are the experiments on the parallel portfolio approach for the train unit shunting problem. Here, we run multiple processes of HIP on the same instance in parallel with different hyperparameter configurations for every process. We compare the performance, based on the thread of the process that finds a feasible solution in the smallest number of local search steps, for various strategies to construct portfolios. These include a default portfolio, random portfolios and trained portfolios. For the random and trained portfolios, we consider configurations that are sampled according to a normal distribution around the default configuration d using $\theta \sim \mathcal{N}(d, \sigma^2)$ with standard deviations $\sigma = 1.1073$ and $\sigma = 0.55365$, which we refer to as *normal large*, and *normal small*, respectively. In addition, we consider a method where the configurations are sampled from a uniform distribution $\theta \sim U(\Theta)$. For the trained portfolios, we also consider a portfolio that is obtained by the best P configurations from the Bayesian optimization training procedure after filtering configurations that are located close to each other in the search space using Algorithm 3.

We present the performance of the different strategies to construct portfolios in Table 7.1. The left column contains the name of the method, divided by default, random or trained portfolios. The performance is represented by the percentage of solved instances and the average number of local search steps to find the feasible solution. We compare the performance for $P = 6$ and $P = 24$ processes. For the trained portfolios, we report the average performances and corresponding standard deviations over 4 runs. In every run, we evaluate the portfolio of configurations that is obtained from the training procedure on all available test instances. For the random portfolios, however, we generate new portfolios for every test instance and consider every single test instance only once. Therefore, we can not compute standard deviations and only report the average performance of the portfolios.

We first observe that, for all portfolios, the average number of local search steps of the process that finds the feasible solution the quickest does not strongly vary over the different methods. Therefore, the application of random or trained portfolios does not result in significantly larger computation times compared to the default portfolio.

Out of the random portfolios, the only method that achieves a better performance based on the percentage of solved instances when $P = 6$ is the method of sampling configurations according to a normal distribution with a small standard deviation. This performance is emphasized in bold in the second column as the best performance over all portfolios. This is the random strategy that yields the least variation over the configuration space. However, the performance of the normal distributions with both a small and a large standard deviation is better compared to the performance of the default configuration when $P = 24$. In this scenario, the best performance is obtained for the method with relatively more variation over the configuration space. In addition, the relative decrease in performance of the uniform configuration compared to the default configuration diminishes for a larger number of processes. Based on the performance of the random portfolios, we conclude that the potential increase in performance for portfolio approaches increases for larger number of available processes. Further-

Table 7.1: Performance of different strategies to construct portfolios, in terms of the percentage of solved test instances and the average number of local search steps. The reported values of the trained portfolios are averages over 4 runs (standard deviations between brackets).

Method	% solved instances		local search steps	
	P = 6	P = 24	P = 6	P = 24
Default	76.32	79.70	**27,068**	23,605
Normal small σ (random)	**77.07**	79.89	27,171	22,997
Normal large σ (random)	75.00	80.08	28,863	23,269
Uniform (random)	73.68	78.76	29,909	24,649
Normal small σ (trained)	76.22 (0.50)	80.08 (0.49)	27,729 (447)	23,075 (530)
Normal large σ (trained)	76.74 (1.47)	**80.55** (0.45)	27,380 (1,098)	**22,912** (528)
Uniform (trained)	74.62 (2.43)	79.84 (0.42)	28,809 (2,469)	23,531 (424)
BO (trained)	75.61 (2.54)	78.43 (2.26)	27,710 (2,101)	24,519 (1844)

more, a larger P results in a larger optimal variation over the configuration space of the sampling method.

Three out of the four trained portfolios yield a better average performance compared to the default configuration when $P = 24$, and only the large normal trained portfolio when $P = 6$. When comparing the trained portfolios against their corresponding random portfolio, we observe that on average the performance does improve when choosing the configurations based on the observed performance from the training procedure. The p-value of the one-sided sign test for the best portfolio against the default portfolios are 0.186 for $P = 6$ and 0.013 for $P = 24$. This implies that the random normal portfolio with a small standard deviation does not achieve a significant improvement for $P = 6$ at significance level $\alpha = 0.05$. However, the trained normal portfolio with a large standard deviation does achieve a statistically significant improvement over the default portfolio when $P = 24$. We emphasize that the reported performance is based on a limited number of 4 runs. We encounter quite large differences in performances over these runs, as can be seen by the large standard deviations, especially for a smaller number of processes. Therefore, we have to be careful when drawing conclusions based on these performances.

The Bayesian optimization portfolio does not yield the best performance, but it appears to be competitive strategy. It is the method that yields the largest standard deviations over the 4 runs. One of the constructed portfolios that was trained during a run of Bayesian optimization solved 81.02% of the test instances, which is the maximum achieved performance over all evaluated portfolios. This shows that the application of Bayesian optimization is a promising strategy to construct portfolios. We already emphasized that the Bayesian optimization that we use has not been created with the aim to find a portfolio of the best P configurations. Therefore the achieved performance is still sub-optimal and leaves ample room for improvement.

We conclude that the application of portfolio approaches can improve the performance compared to a portfolio of default configurations. To obtain a significant increase in performance, it is important to choose the search space and sampling strategy in a suitable manner to allow for a sufficient, but not excessive, level of variation between the different configurations in the portfolio. The optimal level of variation also depends on the number of available processes. A larger number of available processes increases the potential increase in performance.

8

Conclusion

The main goal of this thesis was to research how we can use a contextual approach to find optimal hyperparameter configurations for target algorithms that solve different instances of a realistic satisfaction problem based on the context of problem instances. We considered Bayesian optimization as the method to find optimal hyperparameter configurations. We focused on HIP, a variable neighborhood search heuristic that solves instances of the train unit shunting problem as the target algorithm.

We start this chapter by stating the conclusions of the experiments that tackle the research questions that we presented in Chapter 1. We then discuss the encountered difficulties of the problem and reflect on the impact of this research. Finally, we conclude this thesis with some suggestions on directions of further research that can be explored in Section 8.1.

We first performed experiments on the bin packing problem as one of the subproblems of the TUSP. We observed that, when solving instances of the bin packing problem with a variable neighborhood search heuristic, a statistically significantly better performance was obtained if we use hyperparameter configurations that have been found when training in a contextual approach compared to a context-free approach.

We then compared the performances of configurations found by Bayesian optimization against the performance of the default configurations of HIP. We did not observe an improvement in performance from the experiments on the more realistic instances of the TUSP. For the context-free approach, the Bayesian optimization procedure resulted in worse test scores than the default configuration for more than half of the runs. We expected this poor performance to be due to the heterogeneity of the training instances, which we hoped to reduce by using a contextual approach. However, dividing the instances based on the context of the number of train units did not result in a significant improvement in the performance for any number of train units compared to the default configuration. Both approaches were able to find configurations that yielded great improvements on the training score, but the performance of these configurations did not generalize to new problem instances in the test set. This implies that the performance that is attained after training on the training instances, can not be reproduced on the so far unencountered test instances.

We discussed in Chapter 5 that the hyperparameter optimization problem relies on three main components: the search space, the search strategy, and the performance evaluation strategy. Hence, the overall performance should be dedicated to the combination of the implementation of these components. Therefore, we conclude that the satisfaction problem and the heterogeneous problem instances do not allow for a proper evaluation of the hyperparameter configurations to be used in Bayesian optimization on this particular search space for the TUSP. This, however, does not mean that it is not possible to use a contextual approach to find optimal hyperparameter configurations for different instances of a satisfaction problem based on the contexts using an alternative application of these components. There are many alternatives left to research before the main research question can be decisively answered. We strongly believe that (a contextual approach to) the optimization of the hyperparameters of HIP is still a promising direction of research to increase the performance of this local search algorithm.

We did manage to show successes in the application of a contextual approach on satisfaction problems. We first showed that the use of batches of problem instances in the training procedure allows for a more representative evaluation of the performance of configurations. We also showed that a performance metric that is composed of multiple factors can make this evaluation more closely resemble a continuous optimization problem. In addition, we observed that the filtering of too easy training instances reduces the percentage of redundant problem instances in the evaluation over the batch, which effectively reduces the number of required function evaluations while training. For problems that consist of instances with less heterogeneity, these techniques become even more promising. In this scenario, it would also be feasible to randomly sample batches between different iterations of the Bayesian optimization procedure to reduce the risk of over-tuning. Finally, the achieved significant increase in performance on the bin packing problem shows that a contextual approach to the hyperparameter optimization problem can improve the performance on problems of lesser complexity or heterogeneity than the TUSP.

The train unit shunting problem is a complex problem and HIP makes use of many different definitions of local search relations that define the neighborhood of candidate solutions. Therefore, there are many possible paths over the search space. As a result, it might be the case that, even when considering reasonably large batches of training instances, the search strategy still over-tunes on these instances. This is possible if there is little similarity in the optimal configurations for different problem instances. Given that the performance metric is defined over the batch, the surrogate model can overfit on a small number of instances in this batch as long as it does not negatively affect the remaining instances. We observed that splitting the problem instances based on the context of the number of train units does not remove all heterogeneity per context. To the best of our knowledge, no reasonably small subset of context dimensions can significantly reduce this heterogeneity, while still having a sufficient number of problem instances per context. The TUSP as the problem at hand is too complex and contains too many features on which you would need to split to completely remove this heterogeneity. We discuss this in the subsequent section regarding further research.

The common strategy in this field of research to handle such a heterogeneous instance set is to use an instance-specific approach, which we have shown to not be possible for satisfaction problems. Alternatively, we can consider a portfolio approach with multiple configurations such that different problem instances in the same context can be solved by different hyperparameter configurations.

As a final research question, we were interested in whether we can find hyperparameter configurations that lead to a better performance of the local search target algorithm that solves the train unit shunting problem in the current strategy that is deployed by the NS. Here, we focused on the parallel approach in the current procedure of HIP that solves the same problem instance of the TUSP in a multi-threaded fashion. We concluded that the application of random and trained portfolios can improve the performance compared to a portfolio of default configurations. It is important to choose a suitable search space and sampling strategy that depends on the number of available processes to create a sufficient, but not excessive, level of variation between the different configurations in the portfolio.

The portfolio constructed by Bayesian optimization did not yield the best performance on average. However, the portfolio that achieved the largest performance on the test instances was constructed using this strategy. We include propositions to improve the portfolio approach as the most promising suggestion for further research.

8.1. Further research

We conclude this thesis with some suggestions on directions of further research that can be explored. We mainly discuss a further examination of the portfolio approach, which we deem to be the most promising direction.

Portfolio approaches

We mentioned in Section 7.1.4 that the Bayesian optimization procedure that we use in this research is constructed with the aim to find the single most optimal hyperparameter configuration, instead of a portfolio of the best P configurations. The exploitation aspect makes it likely that the best P configurations are located close to each other in the configuration space. We naively avoided this by defining a greedy rule that avoids these very similar configurations. However, this greedy rule makes the efficiency of the search strategy sub-optimal. We think that a better representative portfolio

of trained configurations can be obtained if the search strategy takes a degree of diversification into account. This can be done by considering a joint configuration space for the different processes or by including an alternative acquisition function.

The portfolio approach also offers benefits by considering approaches that can avoid issues that we encountered during this research, such as the heterogeneity of the problem instances and the poor generalization of the performance from the training set to test instances. We already show the latter by not considering a training procedure for the random portfolios. If a training procedure is not required, it becomes easier to consider a larger configuration space. The practitioner has more control over, for example, the hyperparameters to include in alternative configurations. In addition, you can control the level of risk aversion by specifying the method to sample new configurations over the chosen configuration space. It is also possible to consider a diversification of the selected algorithms by simultaneously considering alternatives to the local search approach of HIP such as an evolutionary algorithm approach or a multi-agent path finding approach. This can for example be done by dividing the different algorithms over subsets of processes.

Furthermore, the use of a portfolio approach allows for alternative optimization strategies. When every problem instance is solved with a variety of hyperparameter configurations, it becomes easy to gather data on the performance of HIP given different configurations. This data can be used to train models to learn which hyperparameters are the most important in the classification of well-performing configurations. A wide range of machine learning techniques can be considered for such a problem.

Another possibility that arises when solving problem instances in parallel is to consider a combination of portfolios. It is especially interesting to combine one of the trained portfolios with the default portfolio by reserving a percentage of the available threads for the default configuration. The portfolio approach with a reserved number of threads for an incumbent configuration, such as the default, allows for a safe online training procedure. When HIP is in full deployment and daily solves instances for every shunting yard, data can be gathered on the actual performance of the deployed algorithm. When only a single configuration is considered by the algorithm, it is *dangerous* to use an online procedure since the exploration of new configurations makes it possible that no solution is found on some days. The portfolio approach can resolve this problem by always reserving threads for safe execution and assigning the remaining threads for an online learning procedure.

Service tasks

One of the subproblems of the TUSP is an open shop scheduling problem where service tasks must be performed on train units that arrive at the shunting yard. We relaxed this subproblem to reduce the problem difficulty and the size of the search space. The complete configuration space of HIP contains hyperparameters that are related to constraint violations regarding these service tasks. Although the inclusion of these hyperparameters increases the number of dimensions, their optimal values might differ heavily over the various optimal configurations in a contextual approach. More importantly, features on the tasks in a problem instance introduce promising new contexts. Therefore, it might be possible that the inclusion of these service tasks introduces potential contexts to split on.

Alternative context dimensions

We mentioned earlier in this chapter that, to the best of our knowledge, no reasonably small subset of context dimensions can be constructed that significantly reduces the heterogeneity of problem instances per context. Here, we considered the definition of an isolated problem feature as a context dimension. An alternative strategy would be to combine many problem features in a small number of dimensions using feature reduction techniques. One example would be the use of principal components analysis (PCA) which considers an aggregation of problem features to a smaller set of dimensions by maximizing the variation over these new dimensions. Such a resulting set of dimensions (possibly only a single dimension) with a large variation in difficulty implies the existence of promising dimensions to use as context dimensions to split the problem instances on. These splits are likely to reduce the heterogeneity of the resulting contexts.

Joint search space

In Chapter 2, we discussed work on contextual Bayesian optimization by considering a joint search space that combines the configuration space and the context space. Such a search space can be used to train a single surrogate model by taking the correlation over the context dimensions into account. This

allows for more efficiency regarding the number of function evaluations since you no longer need to train a distinct Gaussian process for every context. Therefore, in case a setting with distinct GPs provides a good performance, a strategy that fits a single GP on the joint space might perform even better since it has a larger budget on the number of function evaluations for this single surrogate model. Since we did not consistently obtain a good performance on distinct surrogate models for our considered problem, we did not consider training a single GP on the joint search space. This alternative strategy will not resolve the problem of the poor generalization of the performance to the test set. However, if the application of PCA or an alternative feature reduction approach can successfully reduce the heterogeneity per context, it might be beneficial to consider the approach of training a single GP over the resulting joint search space.

To conclude, there are multiple research directions that can be explored to find hyperparameter configurations that improve the performance of the target algorithm on instances of satisfaction problems. The suggested directions that we discussed here can benefit from the research and methodology provided in this research by altering a single component of the problem at hand. Furthermore, the procedures that we described in this research might prove to achieve a significant increase in performance if only one direction of one of these components is shown to provide promising results.

Bibliography

Anastacio, M., Luo, C., and Hoos, H. H. (2019). Exploitation of default parameter values in automated algorithm configuration. In *Workshop Data Science meets Optimisation (DSO), IJCAI*.

Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013). Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207.

Bartz-Beielstein, T., Lasarczyk, C. W., and Preuß, M. (2005). Sequential parameter optimization. In *2005 IEEE congress on evolutionary computation*, pages 773–780.

Benlic, U. and Hao, J.-K. (2013). Breakout local search for the max-cutproblem. *Engineering Applications of Artificial Intelligence*, 26(3):1162–1173.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554.

Bergstra, J. S. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305.

Bergstra, J. S., Yamins, D., and Cox, D. D. (2013a). Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, page 20.

Bergstra, J. S., Yamins, D., and Cox, D. D. (2013b). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123.

Berkenkamp, F., Krause, A., and Schoellig, A. P. (2016). Bayesian optimization with safety constraints: Safe and automatic parameter tuning in robotics. In *arXiv preprint arXiv:1602.04450*.

Birattari, M., Yuan, Z., Balaprakash, P., and Stützle, T. (2010). Empirical methods for the analysis of optimization algorithms, chapter f-race and iterated f-race: An overview.

Bliek, L., Verwer, S., and de Weerdt, M. (2020a). Black-box combinatorial optimization using models with integer-valued minima. *Annals of Mathematics and Artificial Intelligence*, pages 1–15.

Bliek, L., Verwer, S., and de Weerdt, M. (2020b). Black-box mixed-variable optimisation using a surrogate model that satisfies integer constraints. In *arXiv preprint arXiv:2006.04508*.

Bonilla, E. V., Chai, K. M., and Williams, C. (2008). Multi-task Gaussian process prediction. In *Advances in neural information processing systems*, pages 153–160.

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. In *arXiv preprint arXiv:1012.2599*.

Bull, A. D. (2011). Convergence rates of efficient global optimization algorithms. *Journal of Machine Learning Research*, 12(10).

Char, I., Chung, Y., Neiswanger, W., Kandasamy, K., Nelson, A. O., Boyer, M., Kolemen, E., and Schneider, J. (2019). Offline contextual Bayesian optimization. In *Advances in Neural Information Processing Systems*, pages 4627–4638.

Chung, Y., Char, I., Neiswanger, W., Kandasamy, K., Nelson, A. O., Boyer, M. D., Kolemen, E., and Schneider, J. (2020). Offline contextual Bayesian optimization for nuclear fusion. In *arXiv preprint arXiv:2001.01793*.

Dai, L. (2018). A machine learning approach for optimisation in railway planning. Master's thesis, Delft University of Technology.

Dai, S., Song, J., and Yue, Y. (2020). Multi-task Bayesian optimization via Gaussian process upper confidence bound. In *ICML 2020 Workshop on Real World Experiment Design and Active Learning*.

Dauphin, Y. N., De Vries, H., and Bengio, Y. (2015). Equilibrated adaptive learning rates for non-convex optimization. In *arXiv preprint arXiv:1502.04390*.

Duris, J., Kennedy, D., Hanuka, A., Shtalenkova, J., Edelen, A., Baxevanis, P., Egger, A., Cope, T., McIntire, M., Ermon, S., and Ratner, D. (2020). Bayesian optimization of a free-electron laser. *Physical review letters*, 124(12):124801.

Eggensperger, K., Lindauer, M., and Hutter, F. (2019). Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 64:861–893.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55):1–21.

Falkner, S., Klein, A., and Hutter, F. (2018). Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR.

Forrester, A. I., Sóbester, A., and Keane, A. J. (2007). Multi-fidelity optimization via surrogate modelling. *Proceedings of the royal society a: mathematical, physical and engineering sciences*, 463(2088):3251–3269.

Ginsbourger, D., Baccou, J., Chevalier, C., Perales, F., Garland, N., and Monerie, Y. (2014). Bayesian adaptive reconstruction of profile optima and optimizers. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):490–510.

Greenhill, S., Rana, S., Gupta, S., Vellanki, P., and Venkatesh, S. (2020). Bayesian optimization for adaptive experimental design: A review. *IEEE Access*, 8:13937–13948.

Hinz, T., Navarro-Guerrero, N., Magg, S., and Wermter, S. (2018). Speeding up the hyperparameter optimization of deep convolutional neural networks. *International Journal of Computational Intelligence and Applications*, 17(02):1850008.

Hoffman, M. D., Brochu, E., and de Freitas, N. (2011). Portfolio allocation for Bayesian optimization. In *UAI*, pages 327–336. Citeseer.

Huang, D., Allen, T. T., Notz, W. I., and Miller, R. A. (2006). Sequential kriging optimization using multiple-fidelity evaluations. *Structural and Multidisciplinary Optimization*, 32(5):369–382.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2010). Sequential model-based optimization for general algorithm configuration (extended version). In *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.*

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). An evaluation of sequential model-based optimization for expensive blackbox functions. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1209–1216.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492.

Kadioglu, S., Malitsky, Y., Sellmann, M., and Tierney, K. (2010). Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756. Citeseer.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR.

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. (2019). Auto-weka: Automatic model selection and hyperparameter optimization in weka. In *Automated Machine Learning*, pages 81–95. Springer, Cham.

Krause, A. and Ong, C. S. (2011). Contextual Gaussian process bandit optimization. In *Advances in neural information processing systems*, pages 2447–2455.

Krige, D. G. (1951). A statistical approach to some basic mine valuation problems on the witwatersrand. *Journal of the Southern African Institute of Mining and Metallurgy*, 52(6):119–139.

Kushner, H. J. (1964). A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Fluids Engineering*, 86.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J. S., and Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480.

Li, L. and Talwalkar, A. (2020). Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR.

Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516.

Lindauer, M., Hoos, H. H., Leyton-Brown, K., and Schaub, T. (2017). Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence*, 244:272–290.

Lv, Q., Wang, Y., Zhang, B., and Jin, Q. (2020). Rv-ml: An effective rumor verification scheme based on multi-task learning model. In *IEEE Communications Letters*. IEEE.

Martinez-Cantin, R., de Freitas, N., Doucet, A., and Castellanos, J. A. (2007). Active policy learning for robot planning and exploration under uncertainty. In *Robotics: Science and systems*, pages 321–328.

Metheron, G. (1971). Theory of regionalized variables and its applications. *Cah. Centre Morrphol. Math.*, 5:211.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100.

Mockus, J. (1994). Application of Bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365.

Morris, P. (1993). The breakout method for escaping from local minima. In *AAAI*, pages 40–45.

Moss, H. B., Leslie, D. S., and Rayson, P. (2020). Mumbo: Multi-task max-value Bayesian optimization. In *arXiv preprint arXiv:2006.12093*.

Nogueira, F. (2014). Bayesian Optimization: Open source constrained global optimization tool for Python. `https://github.com/fmfn/BayesianOptimization`.

Pearce, M. and Branke, J. (2018). Continuous multi-task Bayesian optimisation with correlation. *European Journal of Operational Research*, 270(3):1074–1085.

Perrone, V., Jenatton, R., Seeger, M. W., and Archambeau, C. (2018). Scalable hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 6845–6855.

Rogers, S. and Girolami, M. (2016). *A first course in machine learning*. CRC press.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. (2015). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. W. (2009). Gaussian process optimization in the bandit setting: No regret and experimental design. In *arXiv preprint arXiv:0912.3995*.

Swersky, K., Snoek, J., and Adams, R. P. (2013). Multi-task Bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855.

van den Broek, R. W. (2016). Train shunting and service scheduling: An integrated local search approach. Master's thesis, Utrecht University.

van den Broek, R. W., Hoogeveen, H., van den Akker, M., and Huisman, B. (2018). A local search algorithm for train unit shunting with service scheduling. In *Transportation Science, submitted*.

Wang, Z., Zoghi, M., Hutter, F., Matheson, D., and De Freitas, N. (2013). Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, pages 1778–1784.

Williams, C. K. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606.

Xu, Z. and Cai, Y. (2018). Variable neighborhood search for consistent vehicle routing problem. *Expert Systems with Applications*, 113:66–76.

Yogatama, D. and Mann, G. (2014). Efficient transfer learning method for automatic hyperparameter tuning. In *Artificial intelligence and statistics*, pages 1077–1085.

Yoo, S., Haider, M. A., and Khalvati, F. (2017). Estimating optimal depth of vgg net with tree-structured Parzen estimators. *Journal of Computational Vision and Imaging Systems*, 3(1).

A

List of symbols

α	acquisition function
β	exploration and exploitation balance parameter
ε	noise variable
Θ	configuration space
Θ'	complete configuration space
$\theta \in \Theta$	a configuration of hyperparameters
θ^*	optimal hyperparameter configuration
Π	portfolio of hyperparameter configurations
$\mathcal{A}$	target algorithm
b_j	the size of item j for the bin packing problem
$\mathcal{D}_n$	observation history at iteration n
D_k	domain of hyperparameter k
d	number of dimensions of the configuration space
d_k	default configuration value of hyperparameter k
$f(\theta, i)$	performance metric of configuration θ for instance i
$f(\theta, z)$	performance metric of configuration θ for context z
g_j	the degree of violations of constraint j
h	mapping from contexts to configurations
$\mathcal{I}$	set of problem instances
I	set of items in the bin packing
$I \subseteq \mathcal{I}$	batch of problem instances
$I^Z \subseteq \mathcal{I}^Z$	batch of problem instances from a set of instances with context z
$i \in \mathcal{I}$	problem instance
J	set of bins in the bin packing problem
k	kernel function
K	batch size
m	number of dimensions in the context space = number of contexts
$\mathcal{N}$	normal distribution
N	budget of samples
P	number of processes
p	penalty variable
S	surrogate model
s_i	the size of item i in the bin packing problem
T	stopping condition
$t_i(\theta)$	time (or local search steps) before $\mathcal{A}$ terminates on instance i given θ
U	uniform distribution
w	weight coefficient
$w(z)$	probability distribution over z
x	decision variable
Z	context space
$z \in Z$	a context in the context space

B

List of acronyms

BO	Bayesian optimization
CGP-UCB	Contextual Gaussian process upper confidence bound
CLEVI	Convolutional local expected value of improvement
CSP	Constraint satisfaction problem
EGO	Efficient global optimization
EI	Expected improvement
GP	Gaussian process
GP-UCB	Gaussian process upper confidence bound
GRF	Gaussian random field
HIP	Hybrid integration planning method
HO	Hyperparameter optimization
ILS	Iterated local search
IQR	Interquartile range
MIP	Mixed-integer programming
MT-GP-UCB	Multi-task Gaussian process upper confidence bound
NS	Nederlandse spoorwegen (Dutch railways)
PCA	Principal component analysis
PEI	Profile expected improvement
PI	Probability of improvement
REMBO	Random embedding Bayesian optimization
REVI	Regional expected value of improvement
SA	Simulated annealing
SAT	Boolean satisfiability problem
SCoT	Surrogate-based collaborative tuning
SKO	Sequential kriging optimization
SLT	Sprinter lighttrain
SMAC	Sequential model algorithm configuration
SMBO	Sequential model-based optimization
SPO	Sequential parameter optimization
TPE	Tree-structured Parzen estimator
TUSP	Train unit shunting problem
UCB	Upper confidence bound
VIRM	Verlengd interregio materieel
VNS	Variable neighborhood search

C

Hyperparameters in HIP

Table C.1: The hyperparameters in HIP that we include in the search space. For every hyperparameter we list its name, its default value that is used in the current version of HIP, and its (un)transformed domain in the search space. For an explanation of the hyperparameters, we refer to van den Broek (2016) and van den Broek et al. (2018).

Name of hyperparameter	Default	Lower bound	Upper bound	Transformed lower bound	Transformed upper bound
Train unit crossings	10	1	100	0	6.643856
Arrival delays (number)	15	1.5	150	0.584963	7.228819
Arrival delays (per second)	0.002	0.0002	0.02	-12.2877	-5.64386
Departure delays (number)	25	2.5	250	1.321928	7.965784
Departure delays (per second)	0.005	0.0005	0.05	-10.9658	-4.32193
Track length violations	12	1.2	120	0.263034	6.906891
Track length violations (per meter)	0.01	0.001	0.1	-9.96578	-3.32193
Track length violations (per second)	0.0002	0.00002	0.002	-15.6096	-8.96578
Shunt moves	0.02	0.002	0.2	-8.96578	-2.32193
Combine on departure	15	1.5	150	0.584963	7.228819
Illegal parking time	0.01	0.001	0.1	-9.96578	-3.32193
Maximum objective value increase per step	**50**	**50**	**50**	x	x